

# ***SUAVE: VHDL Extensions for System-Level Modeling***

---

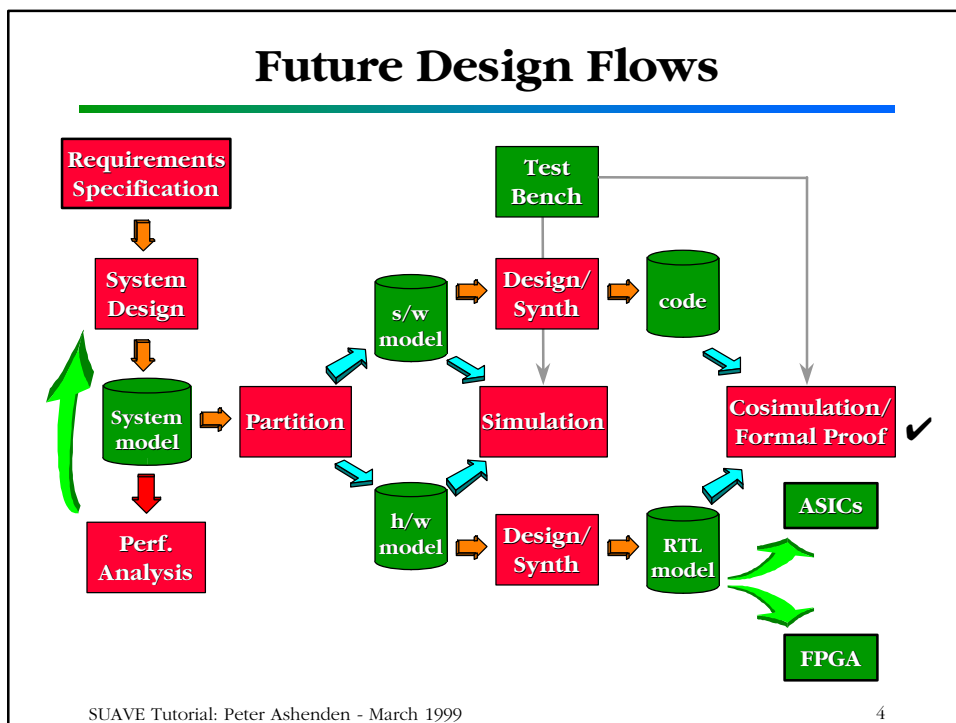
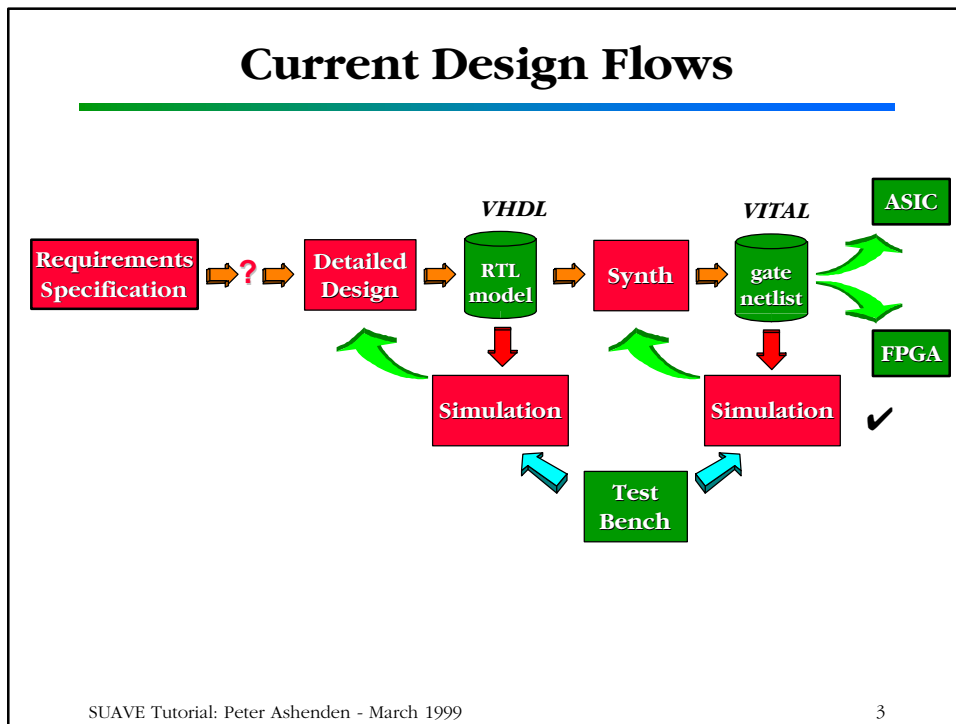
Peter J. Ashenden  
Dept Computer Science  
University of Adelaide, Australia  
petera@cs.adelaide.edu.au  
www.cs.adelaide.edu.au/~petera

March 1999

## **In a Nutshell ...**

---

- VHDL is a standard hardware description language, and is good for
  - register transfer level and behavioral modeling
  - structural modeling
- SUAVE makes it better!
  - at what it's already good for
  - and for modeling large complex systems
- **S**AVANT and **U**niversity of **A**delaide  
**V**HDL **E**xtensions
  - SAVANT: **S**tandard **A**nalyzer of **V**HDL **A**pplications  
for **N**ext-generation **T**echnology
    - Phil Wilsey, U. Cincinnati



## High-Level Modeling

- System-level models
  - describe behavior of complex systems at a high level of abstraction
- Test benches
  - manage complex data sets and test sequences
- It's really software engineering!

## High-Level Modeling Support

- Need abstract data types (ADTs) to manage complexity
  - VHDL has poor encapsulation
- Need inheritance to classify abstractions and for re-use
  - object-oriented methods
  - VHDL has no form of inheritance
- Need late binding to support evolution
  - VHDL only has ad-hoc polymorphism
    - subprogram overloading

## High-Level Modeling Support

---

- Need type-based genericity for re-use
  - VHDL only has constant-based genericity
- Need dynamic process creation to model dynamic reactive systems
  - VHDL only has static process creation
- Need abstract communication
  - VHDL communication is via “hardware” signals

## SUAVE Extensions

---

- SUAVE adds
  - secure ADTs: private types and private parts in packages
  - OO features: tagged types, type extension, inheritance, class-wide polymorphic types
  - type generics
  - generic packages and subprograms
  - channels and message passing
  - process declarations and instantiation
- The ADT, OO and generic extensions are based on ADA-95
  - integrated cleanly into VHDL

## Language Design Principles

---

- Simplicity of mechanism
- Orthogonality of mechanism
  - with clearly defined interactions
- Integration with existing
  - semantic mechanisms
  - syntax
  - language philosophies

## Agenda

---

- ADTs
  - private types and private parts in packages
- OO features
  - inheritance, tagged types, type extension, class-wide polymorphic types
- Type generics
- Generic packages and subprograms
- Communication
  - channels and message passing
- Processes
  - declarations and instantiation

## Abstract Data Types (ADTs)

- Key mechanism for managing complexity
- Consists of
  - encapsulated data type
  - visible set of operations
- ADT user can only manipulate data with the public operations
  - secure, prevents inadvertent corruption of state
- VHDL mechanism
  - type and operations declared in a package
  - operations implemented in the package body
  - *type details can't be hidden!*

## Package Example

```

package test_queues is
  use work.tests.all;
  constant max_size : positive := 100;
  type queue_array is array (0 to max_size - 1) of test;
  type queue is record
    head, tail : natural range 0 to max_size - 1;
    size : natural range 0 to max_size;
    the_buffer : queue_array;
  end record queue;
  function is_empty ( Q : queue ) return boolean;
  function is_full ( Q : queue ) return boolean;
  procedure add ( Q : inout queue; item : in test );
  procedure remove ( Q : inout queue; item : out test );
end package test_queues;

```

## Package Example

```
use work.test_queues.all;
variable init_test_queue : test_queues.queue;
...
if not is_full ( init_test_queue ) then
    add ( init_test_queue, next_test );
end if;
...
init_test_queue.size := 42;
```

## Private Types/Private Parts

- SUAVE strengthens encapsulation for ADTs
- Private types in packages
  - ADT user knows type name, but not details
  - declared using reserved word **private**
- Private part in package declarations
  - delimits publicly visible from hidden aspects
  - separated by reserved word **private**
  - allows analyzer to know sufficient details of private types to allocate storage

## Secure ADT Example

```

package test_queues is
  use work.tests.all;
  type queue is private;
  function is_empty ( Q : queue ) return boolean;
  ...
private
  constant max_size : positive := 100;
  type queue_array is array (0 to max_size - 1) of test;
  type queue is record
    head, tail : natural range 0 to max_size - 1;
    size : natural range 0 to max_size;
    the_buffer : queue_array;
  end record queue;
end package test_queues;

```

SUAVE Tutorial: Peter Ashenden - March 1999

15

## Limited Private Types

- Normal private types have predefined assignment and (in)equality operations
  - element-wise copy/comparison
  - may not be appropriate, eg for pointer-based types
- A *limited private type*
  - cannot be assigned by ADT user
  - does not have predefined (in)equality
  - ADT can define them if appropriate
- Private type containing a pointer must say so
  - use reserved word **access** in declaration
  - prevents use as type of signals

SUAVE Tutorial: Peter Ashenden - March 1999

16



## Limited Private Type Example

```

package test_queues is
  use work.tests.all;
  type queue is limited access private;
  constant empty_queue : queue;
  procedure copy ( from_Q : in queue; to_Q : inout queue );
  impure function "=" ( L, R : in queue ) return boolean;
  impure function "/=" ( L, R : in queue ) return boolean;
  ...
private
  ...
end package test_queues;

```

SUAVE Tutorial: Peter Ashenden - March 1999

17

## Limited Private Type Example

```

package test_queues is
  ...
private
  type queue_element;
  type queue_ptr is access queue_element;
  type queue_element is record
    the_test : test;
    next_element, prev_element : queue_ptr;
  end record queue_element;
  type queue is record
    head, tail : queue_ptr;
  end record queue;
  constant empty_queue : queue := queue'(null, null);
end package test_queues;

```

SUAVE Tutorial: Peter Ashenden - March 1999

18

## Limited Private Type Example

```
use work.test_queues.all;  
variable test_queue_1, test_queue_2 : test_queues.queue;  
...  
if test_queue_1 /= empty_queue then  
    copy ( from_Q => test_queue_1, to_Q => test_queue_2 );  
end if;
```

## Lightweight Packages

- Currently packages are design units in VHDL
- SUAVE allows packages to be declared in any declarative part
  - eg, inside entities, architectures, process, subprograms, other packages
- Package declaration and body must appear in the same declarative region
- Allows local ADTs
- Like class definitions in C++

## Package Example

---

```

architecture behavioral of microcontroller is
  package words is
    type word is range 0 to 2**16 - 1;
    function "+" ( L, R : in word ) return word;
    ...
  end package words;
  package body words is
    function "+" ( L, R : in word ) return word is
      begin
        return (L + R) mod 2**16;
      end function "+";
    ...
  end package body words;
  ...

```

SUAVE Tutorial: Peter Ashenden - March 1999

21

## Package Example

---

```

  ...
  use words.all;
  signal s : word;
  ...
begin
  ...
end architecture behavioral;

```

SUAVE Tutorial: Peter Ashenden - March 1999

22

## Agenda

---

- ADTs
  - private types and private parts in packages
- OO features
  - inheritance, tagged types, type extension, class-wide polymorphic types
- Type generics
- Generic packages and subprograms
- Communication
  - channels and message passing
- Processes
  - declarations and instantiation

## Primitive Operations & Inheritance

---

- For a type declared in a package
  - subprograms with parameters of the type are called *primitive operations*
- SUAVE allows new types to be *derived*
  - use the reserved word **new**
  - eg, **type event\_count is new natural**;
  - distinct type from the parent (*cf* subtypes)
  - derived type inherits primitive operations of the parent
- If a type is derived within a package
  - may augment and override inherited primitive operations

## Tagged Record Types

- SUAVE adopts tagged record types from Ada-95
  - use the reserved word **tagged**
  - tagged record object includes a runtime tag
  - identifies the specific type of the object
- Defining a tagged type and operations in a package is like defining a class

## Tagged Record Example

```

package instructions is
  ...
  type instruction is
    tagged record
      opcode : opcode_type;
    end record instruction;
  function privileged ( instr : instruction;
                       mode : protection_mode ) return boolean;
  procedure disassemble ( instr : instruction; L : inout line );
  ...

variable current_instruction : instruction
           := instruction'(opcode => nop);

```

## Type Extension

---

- OO involves refinement of data types
- A tagged type can be extended on derivation
- Defining a derived type in a package is like defining a subclass
  - inherits record elements and operations from parent
  - can add new record elements
  - can override and augment operations

## Type Extension Example

---

```

type ALU_instruction is
  new instruction with record
    destination, source_1, source_2 : register_number;
  end record ALU_instruction;
procedure disassemble ( instr : ALU_instruction; L : inout line );

```

---

- ALU\_instruction includes
  - opcode, destination, source\_1 and source\_2
  - inherits privileged operation
  - overrides disassemble operation

## Abstract Types and Operations

- If type is to be used only as parent for derivation, declare it to be *abstract*
  - use reserved word **abstract**
  - cannot declare objects to be of an abstract type
- If an operation is required for all derived types, but not for parent type, declare it as an *abstract operation* of the parent type
  - no body for the abstract operation
  - must declare overriding operations for derived types
  - like C++ pure virtual functions

SUAVE Tutorial: Peter Ashenden - March 1999

29

## Abstract Type/Op Example

```

type memory_instruction is
  abstract new instruction with record
    base : register_number;
    offset : integer;
  end record memory_instruction;
function effective_address_of ( instr : memory_instruction )
  return integer;

procedure perform_memory_instruction
  ( instr : memory_instruction ) is abstract;

```

---

```

function effective_address_of ( instr : memory_instruction )
  return integer is
begin
  return read_register( instr.base ) + instr.offset;
end function effective_address_of;

```

SUAVE Tutorial: Peter Ashenden - March 1999

30

## Abstract Type/Op Example

```

type load_instruction is
  new memory_instruction with record
    destination : register_number;
  end record load_instruction;

procedure perform_memory_instruction
  ( instr : load_instruction );

procedure disassemble ( instr : load_instruction; L : inout line );

procedure perform_memory_instruction
  ( instr : load_instruction ) is
  variable ...
begin
  MDR := read_memory ( effective_address_of ( instr ) );
  ...
end procedure perform_memory_instruction;

```

SUAVE Tutorial: Peter Ashenden - March 1999

31

## Class-wide Types

- A *class-wide type*, T'class, includes
  - the type T (must be a tagged type)
  - all types derived directly or indirectly from T
- For example, instruction'class includes
  - instruction, ALU\_instruction,
  - memory\_instruction, load\_instruction, ...
- SUAVE allows constants, variables and signals to be of class-wide types
  - such objects are *polymorphic*
  - can take on values of different specific types during their lifetimes

SUAVE Tutorial: Peter Ashenden - March 1999

32



## Dynamic Dispatch

- When an operation is applied to a polymorphic object
  - tag used to determine the specific type
  - appropriate version of the operation is invoked
  - *dynamic dispatch*, or *late binding*

```

signal fetched_instruction : instruction'class;
...
variable L : line;
...
disassemble ( fetched_instruction, L );

```

SUAVE Tutorial: Peter Ashenden - March 1999

33

## Class-wide Operations

- A subprogram can have parameters of class-wide types
- An entity can have ports of class-wide types
- Dynamic dispatch may be required

```

procedure execute ( instr : instruction'class ) is
...
begin
  if privileged ( instr, current_mode ) then
    raise_exception ( privileged_instruction );
  else
    disassemble ( instr, L );
  ...
  end if;
end procedure execute;

```

SUAVE Tutorial: Peter Ashenden - March 1999

34

## Example: Instruction Register

---

```

use work.instructions.all;
entity instruction_reg is
  port ( clk : in bit; jam_nop : in bit;
         instr_in : in instruction'class;
         instr_out : out instruction'class );
end entity instruction_reg;

```

SUAVE Tutorial: Peter Ashenden - March 1999

35

## Example: Instruction Register

---

```

architecture behavioral of instruction_reg is
begin
  store : process ( clk ) is
    constant nop_instruction : instruction
      := instruction'(opcode => nop);
    begin
      if clk = '1' then
        if jam_nop = '1' then
          instr_out <= nop_instruction;
        else
          instr_out <= instr_in;
        end if;
      end if;
    end process store;
end architecture behavioral;

```

SUAVE Tutorial: Peter Ashenden - March 1999

36

## Type Conversion

---



---

- Invoking a parent-type operation
  - convert object to the parent type

---

```

procedure disassemble ( instr : ALU_instruction; L : inout line ) is
  ...
begin
  disassemble ( instruction(instr), L ); -- disassemble opcode
  write ( L, ' ');
  disassemble_reg ( instr.destination, L );
  ...
end procedure execute;
```

## Extension Aggregates

---



---

- Converting to a derived type
  - need to provide values for the extended elements
  - use an *extension aggregate*

---

```

variable potential_load : load_instruction;
constant nop_instruction : instruction
           := instruction'(opcode => nop);
  ...
potential_load := load_instruction'( nop_instruction with
                                   base => 0, offset => 0,
                                   destination => 0 );
```

## Encapsulation and Extension

- SUAVE provides mechanisms for integrating private types and extensible types
- A private type can be tagged
  - can be extended, but details of parent are hidden
  - can also be abstract, limited, ...
- A tagged type can be extended with a *private extension*
  - details of extension are hidden
- Combination
  - a private tagged type can be extended with a private extension

SUAVE Tutorial: Peter Ashenden - March 1999

39

## Private Extension Example

```

package MAC_level is
  ...
  type MAC_packet is abstract tagged private;
  procedure set_MAC_dest ( pkt : inout MAC_packet;
                          dest : in MAC_level_address );
  ...
private
  type MAC_packet is
    tagged record
      source, dest : MAC_level_address;
    end record MAC_packet;
end package MAC_level;

```

SUAVE Tutorial: Peter Ashenden - March 1999

40

## Private Extension Example

---

```

package body MAC_level is
  procedure set_MAC_dest ( pkt : inout MAC_packet;
                           dest : in MAC_level_address ) is
    begin
      pkt.dest := dest;
    end procedure set_MAC_dest;
  ...
end package body MAC_level;

```

SUAVE Tutorial: Peter Ashenden - March 1999

41

## Private Extension Example

---

```

package network_level is
  use work.MAC_level.MAC_packet;
  ...
  type network_packet is
    new MAC_packet with private;
  procedure set_network_dest ( pkt : inout network_packet;
                               dest : in network_level_address );
  ...
  private
    type network_packet is
      new MAC_packet with record
        source, dest : network_level_address;
        control : control_type;
        time_to_live : hop_count;
      end record network_packet;
  end package network_level;

```

SUAVE Tutorial: Peter Ashenden - March 1999

42

## Private Extension Example

---

```
package body network_level is  
  procedure set_network_dest ( pkt : inout network_packet;  
                               dest : in network_level_address ) is  
    begin  
      pkt.dest := dest;  
    end procedure set_network_dest;  
  ...  
end package body network_level;
```

---

- The **source** and **dest** elements in the private extension are distinct from the **source** and **dest** elements in the parent
  - preserves information hiding

## Agenda

---

- ADTs
  - private types and private parts in packages
- OO features
  - inheritance, tagged types, type extension, class-wide polymorphic types
- Type generics
- Generic packages and subprograms
- Communication
  - channels and message passing
- Processes
  - declarations and instantiation

## Type Generics

- VHDL only allows generic constants
  - used to specify timing parameters, operational parameters, port index bounds
- SUAVE allows generic types
  - used for type of ports, internal data, ...
- Significantly enhances reuse

## Example: Generic Mux

```
entity mux is  
  generic ( type data_type is private;  
            T_pd : time );  
  port ( sel : in bit;  
         d0, d1 : in data_type; d_out : out data_type );  
end entity mux;
```

```
architecture data_flow of mux is  
begin  
  mux_flow :  
    d_out <= d0 after T_pd when sel = '0' else  
            d1 after T_pd;  
end architecture data_flow;
```

## Example: Generic Mux

```

signal sel : bit;
signal s0, s1, s_out : bit;
signal i0, i1, i_out : integer;
use work.network_level.all;
signal pkt0, pkt1, pkt_out : network_packet;

bit_mux : entity work.mux(data_flow)
  generic map ( data_type => bit, T_pd => 2 ns )
  port map ( sel => sel, d0 => s0, d1 => s1, d_out => s_out );
int_mux : entity work.mux(data_flow)
  generic map ( data_type => integer, T_pd => 5 ns )
  port map ( sel => sel, d0 => i0, d1 => i1, d_out => i_out );
pkt_mux : entity work.mux(data_flow)
  generic map ( data_type => network_packet, T_pd => 12 us )
  port map ( sel => sel, d0 => pkt0, d1 => pkt1, d_out => pkt_out );

```

SUAVE Tutorial: Peter Ashenden - March 1999

47

## Generic Type Definitions

- Different kinds of formal type definitions
  - restrict the actual type that can be associated on instantiation
  - imply information about the formal type that can be used in the generic unit
- Example
  - **generic ( type index is (<>) );**
  - actual type must be a discrete type
  - unit can use `index'succ`, "`<`", ...

SUAVE Tutorial: Peter Ashenden - March 1999

48



## Generic Type Definitions

---

<b>[ [ abstract ] tagged ]</b> <b>[ limited ] [ access ] private</b>	Formal is private, with specified restrictions. Actual is any type that meets the restrictions.
<b>[ abstract ] new <i>type</i></b> <b>[ with [ access ] private ]</b>	Formal is a derived type. Actual must be derived from the specified type. If <b>with private</b> is specified, actual must be a tagged type.
<b>( &lt;&gt; )</b>	Formal is discrete. Actual must be discrete.
<b>range &lt;&gt;</b>	Formal is an integer type. Actual must be an integer type.
<b>range &lt;&gt; . &lt;&gt;</b>	Formal is a floating-point type. Actual must be a floating-point type.
<b>units &lt;&gt;</b>	Formal is a physical type. Actual must be a physical type.

## Generic Type Definitions

---

<b>array ( <i>index_def'n</i> )</b> <b>of <i>element_def'n</i></b>	Formal is an array type. Actual must be an array type with the same index and element types.
<b>access <i>type</i></b>	Formal is an access type. Actual must be an access type with the same designated type.
<b>file of <i>type</i></b>	Formal is a file type. Actual must be a file type with the same element type.
<i>formal subprogram</i>	Formal is a subprogram. Actual must be a subprogram with the same signature.
<i>formal package</i>	Formal is an instance of a specified package. Actual must be a similar instance of the specified package.

## Example: Generic Counter

```

entity counter is
  generic ( type count_type is ( <> ) );
  port ( clk : in bit; data : out count_type );
end entity counter;

architecture behavioral of counter is
begin
  count_behavior : process is
    variable count : count_type := count_type'low;
  begin
    data <= count;
    wait until clk = '1';
    if count = count_type'high then
      count := count_type'low;
    else
      count := count_type'succ( count );
    end if;
  end process count_behavior;
end architecture behavioral;

```

SUAVE Tutorial: Peter Ashenden - March 1999

51

## Example: Generic Counter

```

subtype short_natural is natural range 0 to 255;
type state_type is ( idle, receiving, processing, replying );
...

short_natural_counter : entity work.counter(behavioral)
  generic map ( count_type => short_natural )
  port map ( clk => master_clk, data => short_data );
state_counter : entity work.counter(behavioral)
  generic map ( count_type => state_type )
  port map ( clk => master_clk, data => state_data );

```

SUAVE Tutorial: Peter Ashenden - March 1999

52

## Example: Generic Shift Register

```

entity shift_register is
  generic ( type index_type is ( <> );
            type element_type is private;
            type vector is array ( index_type range <> )
              of element_type );

  port ( clk : in bit;
         data_in : in element_type; data_out : out vector );
end entity shift_register;

```

SUAVE Tutorial: Peter Ashenden - March 1999

53

## Example: Generic Shift Register

```

architecture behavioral of shift_register is
begin
  shift_behavior : process is
    constant data_low : index_type := data_out'low;
    constant data_high : index_type := data_out'high;
    subtype ascending_vector is
      vector ( data_low to data_high );
    variable stored_data : ascending_vector;

    begin
      data_out <= stored_data;
      wait until clk = '1';
      stored_data( data_low to index_type'pred(data_high) )
        := stored_data( index_type'succ(data_low) to data_high );
      stored_data( data_high ) := data_in;
    end process shift_behavior;
end architecture behavioral;

```

SUAVE Tutorial: Peter Ashenden - March 1999

54

## Example: Generic Shift Register

```
signal master_clk, carry_in : bit;
signal result : bit_vector(15 downto 8);

bit_vector_shifter : entity work.shift_register(behavioral)
  generic map ( index_type => natural,
                element_type => bit,
                vector => bit_vector )
  port map ( clk => master_clk,
             data_in => carry_in, data_out => result );
```

## Agenda

- ADTs
  - private types and private parts in packages
- OO features
  - inheritance, tagged types, type extension, class-wide polymorphic types
- Type generics
- Generic packages and subprograms
- Communication
  - channels and message passing
- Processes
  - declarations and instantiation

## Generic Packages/Subprograms

- SUAVE also allows generic clauses in
  - package declarations
  - subprogram declarations
- Generic packages and subprograms
  - cannot be used or called directly
  - must be instantiated first
- Adopted directly from Ada
  - adapted to integrate with VHDL syntax
  - similar to C++ templates

SUAVE Tutorial: Peter Ashenden - March 1999

57

## Example: Generic Queue

```

package queues is
  generic ( type element_type is private );
  type queue is access private;
  impure function new_queue return queue;
  impure function is_empty ( Q : in queue ) return boolean;
  procedure append ( Q : inout queue; E : in element_type );
  procedure extract_head ( Q : inout queue; E : out element_type );
private
  type element_node;
  type element_ptr is access element_node;
  type element_node is record
    next_element : element_ptr;
    value : element_type;
  end record element_node;
  type queue is record
    head, tail : element_ptr;
  end record queue;
end package queues;

```

SUAVE Tutorial: Peter Ashenden - March 1999

58

## Example: Generic Queue

```

type test_vector is ...;
variable generated_test : test_vector;
package test_queues is new queues
  generic map ( element_type => test_vector );
variable tests_pending : test_queues.queue
  := test_queues.new_queue;
...

test_queues.append ( tests_pending, generated_test );

```

## Example: Generic Swap

```

procedure swap
  generic ( type data_type is private )
    ( a, b : inout data_type ) is
    variable temp : data_type;
  begin
    temp := a; a := b; b := temp;
  end procedure swap;

procedure swap_times is new swap
  generic map ( data_type => time );
variable old_time, new_time : time;

swap_times ( old_time, new_time );

```

## Example: Generic Collection

```

package ordered_collection_adt is
  generic ( type element_type is private;
            type key_type is private;
            function key_of ( e : element_type ) return key_type;
            function "<" ( l, r : key_type ) return boolean is <> );
  type ordered_collection is access private;
  function new_ordered_collection return ordered_collection;
  procedure insert ( c : inout ordered_collection;
                    e : in element_type );
  procedure traverse
    generic ( procedure action ( element : in element_type ) )
      ( c : in ordered_collection );
  ...

```

SUAVE Tutorial: Peter Ashenden - March 1999

61

## Example: Generic Collection

```

  ...
private
  -- Concrete representation is a doubly linked list
  -- with a sentinel record pointing to head and tail.
  type ordered_collection_object;
  type ordered_collection_ptr is
    access ordered_collection_object;
  type ordered_collection_object is record
    next_element, prev_element : ordered_collection_ptr;
    element : element_type;
  end record ordered_collection_object;
  type ordered_collection is new ordered_collection_ptr;
end package ordered_collection_adt;

```

SUAVE Tutorial: Peter Ashenden - March 1999

62

## Example: Generic Collection

```

package body ordered_collection_adt is
  function new_ordered_collection return ordered_collection is
    variable result : ordered_collection_ptr
      := new ordered_collection_object;
  begin
    result.next_element := result;
    result.prev_element := result;
    return ordered_collection(result);
  end function new_ordered_collection;
  ...

```

SUAVE Tutorial: Peter Ashenden - March 1999

63

## Example: Generic Collection

```

...
procedure insert ( c : inout ordered_collection;
  e : in element_type ) is
  variable current_element : ordered_collection_ptr
    := ordered_collection_ptr(c).next_element;
  variable new_element : ordered_collection_ptr;
begin
  while current_element /= ordered_collection_ptr(c)
    and key_of(current_element.element) < key_of(e) loop
    current_element := current_element.next_element;
  end loop;
  -- insert new element before current_element
  new_element := new ordered_collection_object'(
    element => e,
    next_element => current_element,
    prev_element => current_element.prev_element );
  new_element.next_element.prev_element := new_element;
  new_element.prev_element.next_element := new_element;
end procedure insert;

```

SUAVE Tutorial: Peter Ashenden - March 1999

64



## Example: Generic Collection

```

...
procedure traverse
  generic ( procedure action ( element : in element_type ) )
    ( c : in ordered_collection ) is
    variable current_element : ordered_collection_ptr
      := ordered_collection_ptr(c).next_element;
  begin
    while current_element /= ordered_collection_ptr(c) loop
      action ( current_element.element );
      current_element := current_element.next_element;
    end loop;
  end procedure traverse;
end package body ordered_collection_adt;

```

SUAVE Tutorial: Peter Ashenden - March 1999

65

## Example: Generic Collection

```

type stimulus_element is
  record
    application_time : delay_length;
    pattern : std_logic_vector (0 to stimulus_vector_length - 1);
  end record stimulus_element;
function stimulus_key ( stimulus : stimulus_element ) return delay_length is
begin
  return stimulus.application_time;
end function stimulus_key;
package ordered_stimulus_collection_adt is
  new ordered_collection_adt
    generic map ( element_type => stimulus_element,
      key_type => delay_length,
      key_of => stimulus_key,
      "<" => std.standard."<");
  signal dut_inputs : std_logic_vector (0 to stimulus_vector_length - 1);

```

SUAVE Tutorial: Peter Ashenden - March 1999

66

## Example: Generic Collection

```

stimulus_generator : process is
  use ordered_stimulus_collection_adt.all;
  variable dut_stimuli : ordered_collection := new_ordered_collection;
  procedure apply_stimulus ( stimulus : stimulus_element ) is
  begin
    wait for stimulus.application_time - now;
    dut_inputs <= stimulus.pattern;
  end procedure apply_stimulus;

  procedure apply_all_stimuli is new traverse
    generic map ( action => apply_stimulus );

begin
  apply_all_stimuli ( dut_stimuli );
  wait;
end process stimulus_generator;

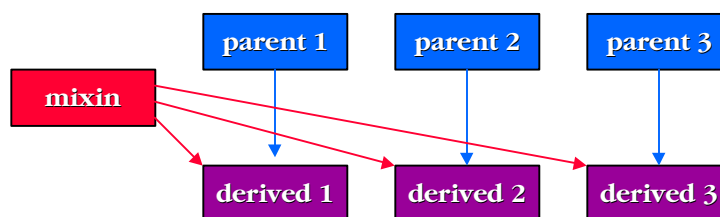
```

SUAVE Tutorial: Peter Ashenden - March 1999

67

## Mixin Inheritance

- Specification of elements and operations to be inherited by several different derived types
  - *mixed-in* with the parent types
- In SUAVE, combine OO features with formal generic derived types
  - *cf* Java interfaces: but with bodies
  - *cf* C++: requires multiple inheritance



SUAVE Tutorial: Peter Ashenden - March 1999

68

## Example: Addressing Modes

- Given type `instruction` and operations as defined earlier
  - define load and store instructions with different addressing modes
  - indexed, displacement, ...

```

type load_instruction is
  abstract new instruction with record
    destination : register_number;
  end record load_instruction;

type store_instruction is
  abstract new instruction with record
    source : register_number;
  end record store_instruction;

```

SUAVE Tutorial: Peter Ashenden - March 1999

69

## Example: Addressing Modes

```

package indexed_addressing_mixin is
  generic ( type parent_instruction is new instruction with private );
  type indexed_instruction is
    new parent_instruction with record
      index_base, index_offset : register_number;
    end record indexed_instruction;
  function effective_address ( instr : indexed_instruction )
    return integer;
end package indexed_addressing_mixin;

package indexed_loads is new indexed_addressing_mixin
  generic map ( parent_instruction => load_instruction );
alias indexed_load_instruction is indexed_loads.indexed_instruction;

package indexed_stores is new indexed_addressing_mixin
  generic map ( parent_instruction => store_instruction );
alias indexed_store_instruction is indexed_stores.indexed_instruction;

```

SUAVE Tutorial: Peter Ashenden - March 1999

70

## Agenda

---

- ADTs
  - private types and private parts in packages
- OO features
  - inheritance, tagged types, type extension, class-wide polymorphic types
- Type generics
- Generic packages and subprograms
- Communication
  - channels and message passing
- Processes
  - declarations and instantiation

## Communication

---

- VHDL signals closely mirror electrical signals
  - assignment to drivers involves inertial or transport delay
  - event times and delays specified exactly
  - multiple drivers require resolution
  - receiver must sense signal values at the correct time, otherwise values are lost
    - require signalling protocol
- High-level modeling
  - prefer more abstract communication model
  - Causality-based synchronization
  - Queuing of values between processes

## Abstract Communication

---

- SUAVE provides
  - named, typed communication channels
  - message-passing
    - synchronous, bounded asynchronous, or unbounded asynchronous
    - messages queued on channel until received
    - send, receive and select statements
  - point-to-point and multicast connection

## Channel Types

---

- Channel types
  - types representing channels with a specified message types
  - null channel: empty message type
    - used for synchronization without data transfer
- Unbounded channel types
- Bounded channel types
  - specified buffer capacity for a channel
  - may be 0,  $\Rightarrow$  synchronous communication
- Unconstrained bounded channel types
  - buffer capacity unspecified

## Channel Objects

- Channels
  - declared objects of channel types
  - may be declared in the same places as signals
- Interface channels
  - ports of entities, components and blocks
  - parameters of subprograms
  - **in** or **out** mode

## Channel Example

```

type request_message is ...;
type result_message is ...;
type acknowledgment_channel is null channel;
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;
type result_channel is channel buffer <> of result_message;
subtype blocking_result_channel is result_channel buffer 2;

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

```

## Interface Channel Conformance

- To connect a formal channel port to an actual channel...
- if formal is unbounded
  - actual must be unbounded
- if formal is constrained bounded
  - actual must be constrained bounded
  - buffer sizes must match
- if formal is unconstrained bounded
  - actual must be constrained bounded
  - buffer size of formal inferred from actual

SUAVE Tutorial: Peter Ashenden - March 1999

77

## Interface Channel Example 1

```

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 source_image, intermediate_image : image_channel;
  ...
begin
  filter : component image_filter
    port map ( raw_image => source_image,
               filtered_image => intermediate_image );
  ...
end architecture performance_modeling;

```

SUAVE Tutorial: Peter Ashenden - March 1999

78

## Interface Channel Example 2

```

type pipe_link is channel buffer <=> of link_data;

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

channel link1, link2 : pipe_link buffer link_buffer_size;
...

stage1 : component pipe_stage
  generic map ( size => link_buffer_size )
  port map ( link_in => link1, link_out => link2 );

```

## Message Passing: Send

- Send statement
  - may block if channel message queue is full
    - never if channel is unbounded
  - adds message to tail of channel message queue
  - sending process then continues execution

```

send result_message'( ... ) to result;
send to request_ack;

```

- Concurrent sends from different processes
  - all messages added to queue, order undefined



## Message Passing: Receive

- Receive statement
  - remove a message from head of channel message queue and assign the value to a variable
  - receiving process blocks if queue is empty

```
receive next_request from request;
receive from result_ack;
```
- Multiple processes receiving from a channel
  - all processes receive each message sent to the channel, and in the same order

SUAVE Tutorial: Peter Ashenden - March 1999

81

## Send & Receive Example

```
type image_token is ...;
type image_channel is channel of image_token;
channel source_image, intermediate_image, ... : image_channel;
```

```
camera_controller : process is
  variable acquired_image : image_token;
begin
  acquired_image := ... ;    -- acquire next image
  send acquired_image to source_image;
end process camera_controller;

first_filter : process is
  variable raw_image, filtered_image : image_token;
begin
  receive raw_image from source_image;
  filtered_image := ... ;    -- filter the raw image
  send filtered_image to intermediate_image;
end process first_filter;
```

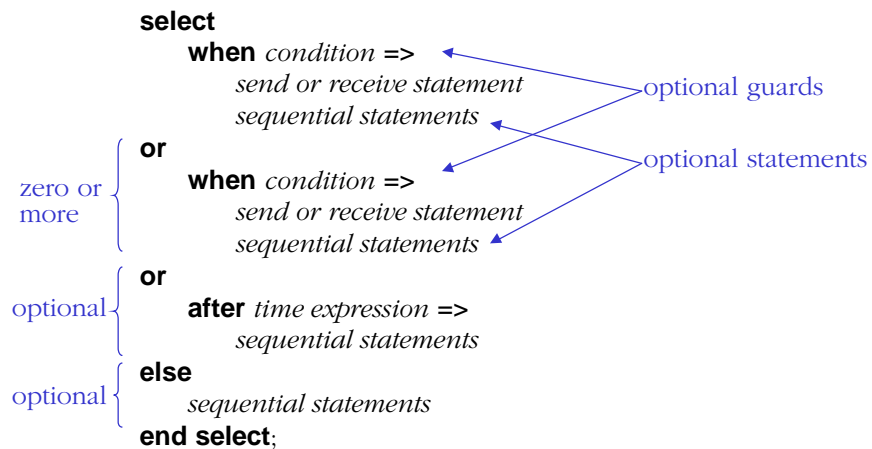
...

SUAVE Tutorial: Peter Ashenden - March 1999

82

## Message Passing: Select

- Non-deterministic choice between alternative sources for message reception



SUAVE Tutorial: Peter Ashenden - March 1999

83

## Select Example

- Readers and writers arbiter
  - reader requests access to resource, waits for acknowledgment, reads resource, then sends “finished” message
  - writer obeys similar protocol
  - multiple readers allowed when no writers are active
  - at most one writer allowed, but only when no readers are active

SUAVE Tutorial: Peter Ashenden - March 1999

84

## Select Example

```

type read_request_channel is channel of ... ;    -- includes reader id
type read_finished_channel is null channel;

type write_request_channel is channel of ... ;    -- includes writer id
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;
    variable read_request_info : ... ;
    variable write_request_info : ... ;

...

```

SUAVE Tutorial: Peter Ashenden - March 1999

85

## Select Example 1

```

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;

```

SUAVE Tutorial: Peter Ashenden - March 1999

86

## Select Example 2

---

```

type bounded_channel is
    channel buffer max_size of message_type;
channel message_stream : bounded_channel;
...
message_source : process is
    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;

```

SUAVE Tutorial: Peter Ashenden - March 1999

87

## Dynamically Created Channels

---

- In some cases, need to create channels dynamically (during simulation)
  - Use *access-to-channel* type
    - access type designating a channel type
  - Use **new** to allocate a channel
  - Use **deallocate** to delete a channel
- 

```

type result_ref is access result_channel;
variable result : result_ref := new result_channel;
...
deallocate ( result );

```

SUAVE Tutorial: Peter Ashenden - March 1999

88

## Agenda

- ADTs
  - private types and private parts in packages
- OO features
  - inheritance, tagged types, type extension, class-wide polymorphic types
- Type generics
- Generic packages and subprograms
- Communication
  - channels and message passing
- Processes
  - declarations and instantiation

## Abstraction of Concurrency

- VHDL allows statically defined processes
- If system being modeled has a dynamically varying number of active objects
  - VHDL is inadequate
- Eg, client/server system
  - dynamically created agents to handle requests
- SUAVE allows dynamically created processes
  - instances of declared processes
  - process declaration includes generic & port lists
  - instances include generic and port maps
  - static or dynamic instances

## Process Declarations

- May have separate declaration and body
  - or just body

```

process identifier is
  generic ( ... );
  port ( ... );
end process identifier;

```

omit if empty

```

process identifier is
  generic ( ... );
  port ( ... );
  declarations
begin
  sequential statements
end process identifier;

```

omit if empty

## Static Process Instantiation

- Process instantiation statement
  - concurrent statement, like a process statement
  - makes an instance of a declared process
  - created at elaboration-time

```

label : process process_name
  generic map ( ... )
  port map ( ... );

```

omit if empty

## Dynamic Process Instantiation

- Sequential process instantiation statement
  - sequential statement, occurs in a process or subprogram body
  - makes an instance of a declared process
    - *depends* on the process containing the declaration of the instantiated process
  - created at simulation-time

```
label : process process_name
       generic map ( ... )
       port map ( ... );
```

← omit if empty

## Process Termination

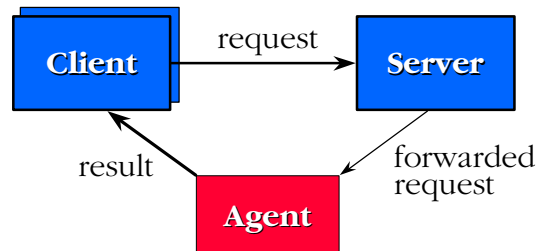
- A process may execute a *terminate statement*

**terminate;**

- Termination involves
  - waiting until all dependent processes terminate
  - disconnecting drivers
  - disassociating actual signals and channels from ports

## Example: Client/Server System

- Client-server system
  - number of clients not known *a priori*
  - multi-threaded server
  - creates new process to handle a request



## Example: Client/Server System

**architecture** system\_level of client\_server\_system is

```

type result_value is ... ;
type result_channel is channel of 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 of request_info;
type request_ref is access request_channel;

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

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

channel server_request : request_channel;

  ...
    
```



## Example: Client/Server System

```

begin
  the_server : process server
    port map ( request => server_request );
  client_pool : for client_index in 1 to number_of_clients generate
    a_client : process client
      port map ( request => server_request );
  end generate client_pool;
end architecture system_level;

```

```

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;

```

SUAVE Tutorial: Peter Ashenden - March 1999

97

## Example: Client/Server System

```

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 ( request => new_agent_request.all );
  send info to new_agent_request.all;
end process server;

```

SUAVE Tutorial: Peter Ashenden - March 1999

98

## Conclusion

---

- Complexity management  $\Rightarrow$  abstraction
- SUAVE: abstraction for
  - data modeling
  - communication & timing
  - concurrency
- System-level modeling
  - pre- hardware/software partitioning
- Further info
  - <http://www.cs.adelaide.edu.au/~petera/>