VLSI DESIGN
1999, Vol. 9, No. 2, pp. 181–201
Reprints available directly from the publisher
Photocopying permitted by license only

© 1999 OPA (Overseas Publishers Association) N.V.
Published by license under
the Gordon and Breach Science
Publishers imprint.
Printed in Malaysia.

Abstract Architecture Representation Using VSPEC

PHILLIP BARAONA and PERRY ALEXANDER*

Department of Electrical and Computer Engineering and Computer Science, The University of Cincinnati, Cincinnati, OH

(Received 10 March 1996; In final form 2 April 1997)

Complex digital systems are often decomposed into architectures very early in the design process. Unfortunately, traditional simulation based languages such as VHDL do not allow the impact of these architectural decisions to be evaluated until a complete, simulatable design of the system is available. After a complete design is available, architectural errors are time-consuming and expensive to correct. However, there is an alternative to simulation based techniques: formal analysis of abstract architectures at the requirements level. This paper describes VSPEC's approach for defining and analyzing abstract architectures. VSPEC is a Larch interface language for VHDL that allows a designer to specify the requirements of a VHDL entity using the canonical Larch approach. VHDL structural architectures that instantiate VSPEC entities define abstract architectures. These abstract architectures can be evaluated at the requirements level to determine the impact of architectural decisions. This paper briefly introduces VSPEC, provides a formal definition of VSPEC abstract architectures and presents two examples that illustrate the architectural definition capabilities of the language.

Keywords: Requirements specification, VHDL, abstract architecture, Larch, interface specification, formal methods

INTRODUCTION

Architectural design decisions made early in a system's design profoundly affect overall design quality. Unfortunately, architecture decisions are rarely evaluated until late in the design process. Simulation-based design languages such as VHDL [5, 12] do not allow evaluation until complete models exist. For large systems, simulatable models appear late in the design process, driving up the cost of error correction. These models include not only architectural decisions, but also component design decisions. The ability to analyze architectural decisions as they are made would significantly reduce this cost.

A solution to late architecture evaluation is the formal analysis of abstract architectures at the requirements level. An abstract architecture is an interconnected collection of components where the requirements of each component are specified

*Corresponding author. e-mail: {pbaraona, alex}@ececs.uc.edu

without defining their implementation. Thus, an abstract architecture describes a class of solutions with a common structure rather than a single instance from that class. Formally described abstract architectures can be evaluated early in the design process when architecture decisions are made before component designs exist.

VSPEC [7], a Larch interface language [10] for VHDL [12], is a requirements specification language that includes formal architecture definition support. VSPEC describes the requirements of digital system components using the canonical Larch approach. Each VHDL entity is annotated with a pre- and post-condition to specify the entity's functional requirements. VSPEC-annotated entities can be connected together using a VHDL structural architecture to form abstract architectures. The VHDL architecture indicates interconnection in the traditional manner, but the VSPEC specification defines the requirements of each component instead of a specific design.

The description of a sorting component illustrates the difference between VHDL and VSPEC. In VHDL, the simplest way to describe the function of a sorting component is a behavioral architecture that implements a quicksort, bubble sort or some other sorting algorithm. This is actually a description of how the sorting component behaves. In contrast, a VSPEC specification of this component explicitly describes what the device must do without defining how it is done. A VSPEC description of a sorting component is shown in Figure 1. It states the output has all the same elements as the input **(permutation(output'post, input)) and the output is in order (ordered(output'post))**. Any sorting algorithm may be used to implement these requirements, but VSPEC allows this algorithm to be chosen later in the design process.

```
entity sort is
  port (input: in element_array;
        output: out element_array);
  includes SortPredicates;
  modifies output;
  sensitive to input'event;
  ensures
    permutation(output'post,input);
    ordered(output'post);
end sort;
```

FIGURE 1 VSPEC description of a sort entity.

Larch interface languages have been developed for a variety of programming languages including C [9], C++ [15] and Modula-3 [14]. At the single component level, VSPEC differs very little from other interface languages. However, defining a Larch interface language for VHDL presents a problem not found in these other languages. In traditional programming languages, a language construct executes after the construct immediately preceding it terminates. In VHDL, there is no implicit execution order among process level constructs and thus no means of determining when a component's pre-condition should hold. VSPEC addresses this problem by allowing a user to define an activation condition in addition to the pre- and post-condition for an entity. When an entity's state satisfies its activation condition, its pre-condition must hold and the entity must perform its specified transformation.

This paper describes VSPEC, concentrating on the language's facilities for describing abstract architectures. The next section provides a brief summary of the VSPEC language. After this, we describe VSPEC abstract architectures, including a definition of the VSPEC state model and a description of how a process algebra (CSP [11]) is used to provide a semantics for the VSPEC activation condition. Next, two example VSPEC specifications are presented that illustrate the abstract architecture representation capabilities of the language. Finally, the paper concludes with a discussion of related work and a brief summary.

VSPEC

VSPEC is used to describe what a digital system should do. It adds a requirements definition capability to VHDL entities analogous to the requirements definition capability that Larch interface languages add to traditional procedure

and function signatures. As shown in Figure 2, the requirements of a VHDL entity can be defined by describing a relationship from the current inputs and state of the system to the outputs and the next state. This section describes how $F(x,s)$ and s are defined in VSPEC and contrasts these definitions with VHDL definitions of $F(x,s)$ and s.

As shown in the **find** entity of Figure 3, a VHDL entity defines an interface. The output of **find** should be the element from the **input** array with the same key as the **key** input. A VHDL **entity** does not describe functional information such as this. The **entity** only defines the component's interface.

The VHDL **architecture** construct describes the function of a component by associating behavior and/or structure with an **entity**. Figure 4 is a behavioral VHDL description of the **find** component's function. In terms of the state model in Figure 2, this architecture describes $F(x,s)$ as a linear search algorithm. This looks very similar to a C or Pascal function describing how the system behaves. Unfortunately, this operational description biases the system towards a particular implementation. Since VSPEC's purpose is requirements specification, it is undesirable to bias the system to a particular implementation this early in the design process.

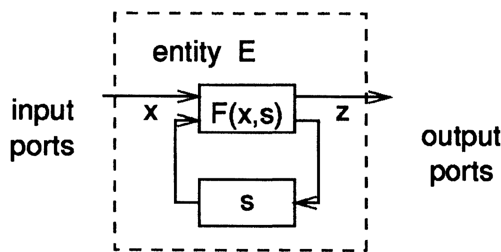


FIGURE 2 State-based specification model.

```
architecture behavior of find is
begin
  process (input,key)
  begin
    for i in input'range loop
      if key = input(i).key then
        output <= input(i);
        exit;
      end if;
    end loop;
  end process;
end behavior;
```

FIGURE 4 A behavioral VHDL **architecture** defining the **find** component's behavior.

VSPEC eliminates this problem by allowing a user to declaratively specify the requirements of a digital system. Seven clauses annotate the VHDL entity construct to allow the specification of what a component should do instead of VHDL's description of how the component performs this function. The **requires**, **ensures** and **sensitive to** clauses are used to specify the device's functional requirements. Non-functional constraints are described in the **constrained by** and **modifies** clauses. The component's internal state is declared in the **state** clause and the **includes** clause is used to make types and operators from a Larch shared language description visible in a VSPEC component. The remainder of this section briefly summarizes these clauses. For a more complete description of the VSPEC clauses, see one of the other VSPEC references [1, 7].

Component function is described in the **requires** and **ensures** clauses. The **requires** clause defines a pre-condition over inputs and state variables while the **ensures** clause defines a post-condition over inputs, outputs and state variables. The **ensures**

```
entity find is port
  (input: in element_array;
   key: in keytype;
   output: out element);
end find;
```

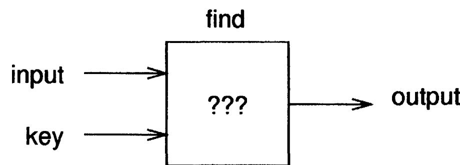


FIGURE 3 A VHDL **entity** defining the interface for a **find** component.

clause defines legal outputs and the next state when the **requires** clause is satisfied. A component's user is responsible for making certain the **requires** clause is satisfied whenever the component is in use. When the **requires** clause is satisfied, the described **entity** is responsible for making the **ensures** clause true.

Let σ be the state of a VSPEC entity as defined by its ports and state variables. If $I(\sigma)$ is the **requires** predicate and $O(\sigma, \sigma')$ is the **ensures** predicate, then the VSPEC annotation defines the following requirements:

$$\forall \sigma \cdot \exists \sigma' \cdot I(\sigma) \Longrightarrow O(\sigma, \sigma') \tag{1}$$

$F(\sigma)$ is an implementation of these requirements if the following condition holds:

$$\forall \sigma \cdot I(\sigma) \Longrightarrow O(\sigma, F(\sigma)) \tag{2}$$

A VSPEC description of a **find** component is shown in Figure 5. Notice that the **requires** clause predicate is **true**, meaning this entity will function correctly for any set of inputs of the proper type. The **ensures** clause predicate states that the **output** element has the same key as the **key** input and **output** is in the input sequence. In terms of the state model in Figure 2, the **ensures** clause predicate defines the requirements of $F(x, s)$, but unlike the VHDL description, it does not describe how to implement the component.

The VSPEC **sensitive to** clause[1] is used to define when a component in an abstract architecture is active. When the **sensitive to** clause predicate is true, a component's pre-condition must hold and an implementation must satisfy the post-condition.

Performance constraints are described in the **constrained by** and **modifies** clauses. Constraints define requirements, such as clock speed or layout area, that are not part of the functional description. The **constrained by** clause defines relations over constraint variables. Currently, the defined constraint variables include power consumption, clock speed, area, pin-to-pin timing, and heat dissipation. Constraint theories written in the Larch Shared Language (LSL) [10] define each constraint type. Users may define their own

```
entity find is port
  (input: in element_array;
   key: in keytype;
   output: out element);
  includes Element(element,keytype,
                   element_array);
  modifies output;
  requires true;
  ensures forall (e : element)
    (output = e implies
       (e.key = key
       and elem_of(e,input)));
  constrained by
    power <= 5 mW
    and key<->output <= 5 ms
    and heat <= 10 mW
    and clock <= 50 MHz;
end search;
```

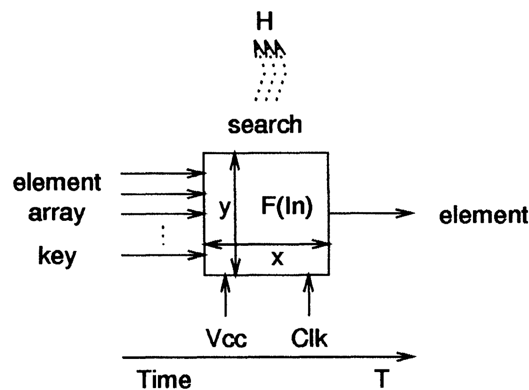


FIGURE 5 The **find** entity annotated with a VSPEC definition.

[1] Previous versions of VSPEC [1, 2, 7] did not have a **sensitive** to clause.

constraints and theories if desired. The **modifies** clause lists variables, ports and signals whose values may be changed by an **architecture** that implements the VSPEC **entity**. This clause is useful when specifying whether an entity modifies a shared variable. The list of objects an entity modifies is not a traditional performance constraint, but this list does restrict the set of potential solutions. Examples of the **constrained by** and **modifies** clauses are shown in Figure 5.

The state of a VSPEC entity is described by the port definition and variables in the **state** clause. In VHDL, ports maintain their values between **entity** invocations so ports are part of a VSPEC entity's state. The **state** clause is used to define internal state variables that are used only in the VSPEC definition. These variables maintain state information that is not recorded in port values. When a VSPEC specification is refined into a VHDL architecture, these internal state variables will be refined into signals or variables that represent the same information. The **state** clause variable declaration represents this information during the requirements specification phase of the entity's design. An example of the state clause can be found in the Move Machine example.

The **includes** clause is the final VSPEC clause.[2] This clause is used to include LSL definitions in a VSPEC description or VHDL package declaration. (See the Move Machine example.) [3] LSL is used to define the types and functions used in a VSPEC specification. An example of the **includes** clause is shown in Figure 5; its syntax is the keyword **includes** followed by a list of trait references. The syntax of a trait reference is similar to a trait reference in LSL. It consists of the trait name followed by an optional parameter list. The parameter list is used to rename LSL names to a name visible in the VSPEC **entity**. Thus, an integer stack is included in a VSPEC specification with this **includes** clause: **includes Stack(integer, int_stack)**.

ARCHITECTURES

The previous section briefly described how VHDL and VSPEC are used to define the requirements of a single device in a digital system. The behavior of a device can also be described by decomposing it into smaller pieces and connecting these pieces together to form an architectural description of the device. This architectural description represents a refinement of the device's behavioral VHDL/VSPEC description. VHDL provides convenient facilities for defining architectural descriptions. This section briefly discusses these facilities and then describes how VSPEC uses them to form an abstract architecture.

VHDL Structural Architectures

VHDL uses structural architectures to represent component composition. A structural architecture describes how sub-components are connected together to form a larger component. Figure 6 shows a structural architecture for **find**. Unlike the behavioral representation in Figure 4, this architecture indicates that a **sort** component connected to a **search** component implements the **find** function. This structural architecture should perform the same function as that specified in the behavioral description.

The VHDL **component** construct defines each component used in a structural architecture. The **structure** architecture of **find** in Figure 6 declares two types of components that are used in this architecture: **sorter** and **searcher**. One instance of each of these components (named **b1** and **b2**) is created in the body of this architecture. The port maps of these component instances are used to indicate how the components are connected together. In the **structure** architecture for find, the system's input array is connected to the **sorter** input and the **sorter** output is connected to internal

[2] Previous versions of VSPEC [1, 2, 7] also contained a **based on** clause. The modified syntax of the **includes** clause described here made the **based on** clause obsolete.

[3] Allowing **includes** clauses in package declarations is a change from previous versions of VSPEC [1, 2, 7].

```
architecture structure of find is
  component sorter
    port (input: in element_array;
          output: out element_array);
  end component;
  component searcher
    port (input: in element_array;
          key: in keytype;
          value: out element);
  end component;
  signal y: element_array;
begin
  b1: sorter port map(input,y);
  b2: searcher port map(y,key,output);
end structure;

entity sort is
  port (input: in element_array;
        output: out element_array);
end sort;

architecture behavior of sort is
begin
  process(input) begin
    -- Behavioral VHDL description
    -- of a bubble sort
  end process;
end behavior;
```

```
entity bin_search is
  port (input: in element_array;
        key: in keytype;
        value: out element);
end bin_search;

architecture behavior of bin_search is
begin
  process (input,key) begin
    -- Binary search algorithm
    -- definition in behavioral VHDL
  end process;
end behavior;

configuration test_struct of find is
  for structure
    for b1:sorter use entity
      work.sort(behavior);
    end for;
    for b2:searcher use entity
      work.bin_search(behavior);
    end for;
  end for;
end test_struct;
```

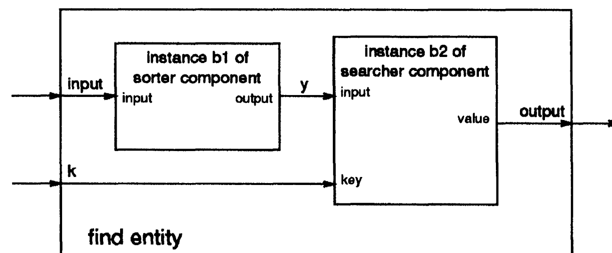


FIGURE 6 A VHDL **architecture** representing the composition of a sorting component and a binary search component implementing the **find** function.

architecture signal **y**. The signal **y** and system input **key** are inputs to the **searcher** component. The output of the **searcher** is connected to the device output.

The VHDL **configuration** construct is used to bind entity-architecture pairs to component instances. In this example, the **test_struct** configuration binds the bubble sort defined by entity **sort** with architecture **behavior** to the **b1** instance of the **sorter** component. Similarly, the binary search defined by entity **bin_search** with architecture **behavior** is bound to the **b2** instance of **searcher**. If there were other architectures for these two entities (such as a structural architecture), a different configuration could have been specified stating that the components in **structure** mapped to these architectures. Entirely different entities could even have been defined.

Since a structural architecture only defines dataflow between components, an additional mechanism must be provided to define when a component activates. VHDL accomplishes this with sensitivity lists and **wait** statements. A sensitivity list is a list of signals. Whenever an event occurs on one of these signals, the process resumes execution. The **behavior** architecture for **sort** is sensitive to its single input, while **bin_search** is sensitive to its input array and key value. This means the sort component sorts its input only when new input arrives. Likewise, a search occurs only when the key value or input array changes. A **wait** statement achieves the same result by waiting on signal events, conditions or for a specific time interval. In this example, **wait** statements could replace sensitivity lists by removing the sensitivity lists and placing **wait** statements referencing the same signals at the end of the process definitions.

These constructs allow VHDL to support architecture representation. Component declarations describe the inputs and outputs of each component type used in the architecture. Instances of these components are created in the architecture body and configurations are used to map component instances to an entity/architecture pair. Net lists indicate signal flow between component instances while sensitivity lists or **wait** statements synchronize component actions.

VSPEC Abstract Architectures

VHDL structural architectures containing VSPEC-annotated components specify abstract architectures. The VHDL architecture remains unchanged, indicating component instantiation and connections. However, a VHDL architecture is not assigned to each component instance in the architecture. Instead, the configuration defines that each component references an entity with an architecture called **VSPEC**. This architecture signifies that at the current point in the design, the requirements of this component are known (*via* the VSPEC description) but no implementation has been defined.[4]

The **structure** architecture of **find**, shown in Figure 6, becomes an abstract architecture by referencing VSPEC definitions of the instantiated components. Figure 7 shows VSPEC **entity** definitions

```
entity sort is
  port (input: in element_array;
        output: out element_array);
  includes SortPredicates;
  modifies output;
  sensitive to input'event;
  ensures
    permutation(output'post,input);
    ordered(output'post);
end sort;

entity bin_search is
  port (input: buffer element_array;
        key: in keytype;
        output: out element);
  includes SortPredicates;
  modifies value;
  sensitive to key'event or input'event;
  requires ordered(input);
  ensures output = e iff (e.key=k and
          element_of(e,input));
end bin_search;

configuration test_vspec of find is
  for structure
    for b1:sorter use entity
      work.sort(VSPEC);
    end for;
    for b2:searcher use entity
      work.bin_search(VSPEC);
    end for;
  end for;
end test_struct;
```

FIGURE 7 VSPEC definitions for the **sort** and **bin_search** components in the **find** architecture.

[4] This is different than leaving the entity **open**. When a VHDL entity is left **open**, the design is being deferred. At the current point in the design, nothing is known about the function of the entity. In contrast, the requirements of a VSPEC entity are known, even though an implementation is not.

for the **sort** and **bin_search** components in Figure 6. A new configuration, **test_vspec**, has been defined for the **find** entity. It specifies that the VSPEC descriptions of **sort** and **bin_search** should be used instead of a specific architecture for these two entities. This configuration describes an abstract architecture for the **find** component. Any implementation satisfying the VSPEC requirements of **sort** and **bin_search** may be associated with the **entity** definitions. The architectures specified in Figure 6 represent one such solution, but there are many others.

The VSPEC description of sort specifies the requirements for a sorting component: the input and output must have all the same elements (*i.e.*, the output is a permutation of the input) and the output must be in order. In a similar fashion, the **bin_search** specification states that whenever the component input is sorted, the component must ensure that the output element contains the same key as the **key** input and this element is an element of the input array. The **requires** and **ensures** clauses of these entities use two predicates (**permutation** and **ordered**) to define these requirements. These predicates are defined in the LSL trait **SortPredicates**, which is included in both VSPEC entities.

Although a VHDL architecture referencing VSPEC definitions defines components and interconnections, additional information must be added to specify when the VSPEC components activate. In traditional sequential programming, a language construct executes following termination of the construct preceding it. For correct execution, a construct's pre-condition must be satisfied when the preceding construct terminates. In hardware systems, components exist simultaneously and behave as independent processes. No predefined execution order exists, thus there is no means for determining when a component's pre-condition should hold. Consider the **find** example. The precondition of **bin_search** need hold only when **sort** has completed its transformation. At all other times, **bin_search** need only maintain its state.

VHDL provides sensitivity lists and **wait** statements to synchronize **entity** execution. VSPEC achieves the same end using the **sensitive to** clause. The **sensitive to** clause contains a predicate called the activation condition that indicates when an entity should begin executing. Effectively, the activation condition defines when a VSPEC annotated **entity**'s pre-condition must hold. When the **sensitive to** predicate is true, the pre-condition must hold and the implementation must satisfy the post-condition. When the **sensitive to** predicate is false, the entity makes no contribution to the next state of the system. Like the **requires** and **ensures** clauses, the **sensitive to** predicate is defined over entity port definitions and variables defined in the **state** clause.

Recall that the structural VHDL architecture for **find** (Fig. 6) specified that the **sort** component should only activate when its input changes and the binary search component activates when one of its inputs changes. Specifying this behavior in VSPEC would not be possible without the **sensitive to** clause. Note the **sensitive to** clauses defined in the VSPEC description of **find** in Figure 7. In VSPEC, a signal's' **event** attribute is true if and only if the signal changed value from the previous state. Thus, both components activate whenever any of their inputs change value.

Architecture Model Semantics

The previous section provided an informal description of how VSPEC can be used to define an abstract architecture. This section provides a more precise, formal definition of the concepts presented above. First, the state of a VSPEC description is defined. After this definition, a precise definition of how the **sensitive to**, **requires** and **ensures** clauses define a transformation over this state is presented. The section concludes with a simple example that illustrates these points.

State Definition

The state definition for an entity is a map from port, signal and variable names to their values.

There are three different views of an entity state: (1) abstract; (2) component; and (3) concrete state. The abstract state is defined by a VSPEC description of an entity. The component state is the state of a single component in an abstract architecture, and the concrete state represents the state of all components of an abstract architecture.

The abstract state includes the ports and state variables of an entity. The VSPEC **sensitive to**, **requires** and **ensures** clause predicates are defined over elements of the abstract state of the entity. The component state applies to an entity included as a component in a structural architecture. The component state is formed by taking the entity's abstract state and subjecting it to the renaming imposed by the signals the component is connected to in the architecture. This component state is used to construct the concrete state of the structural architecture. The concrete state is the union of the component states for all of the components in an architecture. This structural architecture represents a refinement of the VSPEC definition of the entity. There is an abstraction function mapping the concrete state of the structural architecture to the abstract state defined by the VSPEC description of the entity the structural architecture refines.

Consider the VSPEC entity in Figure 8. The abstract state of the three entities in this figure are the inputs, outputs and state variables of the entities. Thus, the abstract states of these entities

```
entity system is
  port (sys_in : in integer;
        sys_out : out integer;);
  state (sys_state : integer;);
end system;

entity comp1 is
  port(in1 : in integer;
       result : out integer;);
  state (c1_state : integer;);
end comp1;

entity comp2 is
  port(in1, in2 : in integer;
       result : out integer;);

  state (c2_state : integer;);
end comp2;
```

```
architecture struct of system is
  component comp1
  port (in1 : in integer;
        result : out integer;);
  end component;

  component comp2
  port (in1, in2 : in integer;
        result : out integer;);
  end component;

  signal x, y : integer;

begin
  A : comp1 port map(sys_in,x);
  B : comp1 port map(x,y);
  C : comp2 port map(x,y,sys_out);
end struct;
```

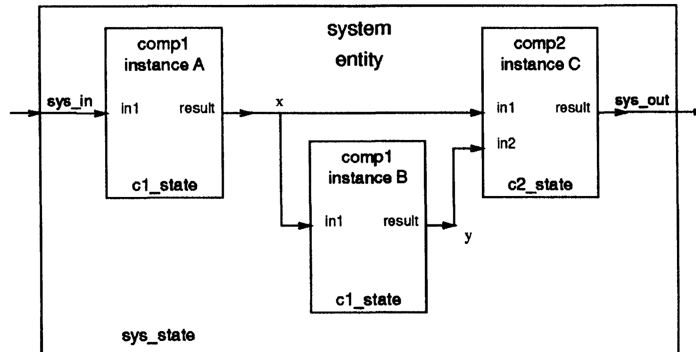


FIGURE 8 Example VSPEC **entity** used to explain the differences between abstract, component and concrete state.

are:

$$
\begin{aligned}
\text{ABSTRACT}_{\text{system}} &= \{\text{syn_in} \mapsto i_0, \\
&\qquad \text{sys_out} \mapsto i_1, \\
&\qquad \text{sys_state} \mapsto i_2\} \\
\text{ABSTRACT}_{\text{comp1}} &= \{\text{in1} \mapsto i_3, \text{result} \mapsto i_4, \\
&\qquad \text{c1_state} \mapsto i_5\} \\
\text{ABSTRACT}_{\text{comp2}} &= \{\text{in1} \mapsto i_6, \text{in2} \mapsto i_7, \\
&\qquad \text{results} \mapsto i_8, \\
&\qquad \text{c2_state} \mapsto i_g\}
\end{aligned}
$$

where $i_0, i_1, \ldots, i_9$ are all integers. As shown, the state is a map from names to values. However, for the purpose of clarity we will show just the names that form the various states throughout the rest of this paper.

Within the **struct** architecture for the **system** entity, **A**'s component state (the first instance of **comp1**) is found by taking **comp1**'s abstract state and performing the renaming defined by the signals the component is connected to. In this case, **in1** is connected to **sys_in** and **result** is connected to signal **x**. Thus, in the context of the **struct** architecture, **in1** of component instance **A** should be replaced by **sys_in** and **result** replaced by **x**. A similar renaming can easily be found for the inputs and outputs of the other components in the **struct** architecture. The renaming for the other components is shown in the definition **A** and **B**'s component states below.

Since the **struct** architecture has more than one instance of the **comp1** entity, the state variables of **comp1** must be renamed to form the component state. This renaming avoids conflicts when forming the concrete state of the **struct** architecture. To simplify matters, we will always rename a component's state variables even if there is only one instance of an entity in the architecture. A number of renaming functions could be chosen, but the one used here is the state variable name in the abstract state subscripted with the instance label from the architecture. The component states of the components in the **struct** architecture are:

$$
\begin{aligned}
\text{COMPONENT}_A &= \text{ABSTRACT}_{\text{comp1}}[\text{in1/sys_in}, \\
&\qquad \text{result}/x, \text{c1_state/c1_state}_A] \\
&= \{\text{sys.in}, x, \text{c1_state}_A\} \\
\text{COMPONENT}_B &= \text{ABSTRACT}_{\text{comp1}}[\text{in1}/x, \\
&\qquad \text{result}/y, \text{c1_state/c1_state}_B] \\
&= \{x, y, \text{c1_state}_B\} \\
\text{COMPONENT}_C &= \text{ABSTRACT}_{\text{comp2}}[\text{in1}/x, \text{in2}/y, \\
&\qquad \text{result/sys_out}, \\
&\qquad \text{c2_state/c2_state}_C] \\
&= \{x, y, \text{sys_out}, \text{c2_state}_C\}
\end{aligned}
$$

We are now ready to form the concrete state of the **struct** architecture for the **system** entity. The concrete state is simply the union of the component states for each component in the architecture:

$$
\begin{aligned}
\text{CONCRETE}_{\text{struct}_{\text{system}}} &= \text{COMPONENT}_A \cup \\
&\quad \cup \text{COMPONENT}_B \cup \\
&\quad \cup \text{COMPONENT}_C \\
&= \{\text{sys_in}, \text{sys_out}, x, y, \\
&\qquad \text{c1_state}_A, \text{c1_state}_B, \\
&\qquad \text{c2_state}_C\}
\end{aligned}
$$

Since an abstract architecture represents a refinement of the requirements specified by VSPEC, an abstraction function can be defined to map the concrete state of the architecture to the abstract state defined by the VSPEC description.

Together, the abstract, component and concrete states represent the state of a VSPEC component. The examples in this paper use these definitions to describe how a VSPEC description behaves.

Transform Definition

The transform performed by a VSPEC architecture is defined by the **sensitive to**, **requires** and **ensures** clauses. The formal definition of the **requires** and **ensures** clauses was discussed in the section titled "VSPEC". This definition is very similar to the transform defined by a traditional Larch interface language. As described in the section titled "VSPEC Abstract Architectures", the **sensitive to** clause is

used to synchronize components and define when the **requires** clause predicate must be satisfied.

Formally, synchronization is easily represented using a traditional process algebra such as CSP [11]. Events are defined as changes in the state of the entity. Assume that $F(St)$ is a function between two states of entity P that implements the requirements specified in P's **requires** and **ensures** clauses (*i.e.*, $F(St)$ satisfies Eq. (2)). The process defined by entity P with a **sensitive to** predicate of $S(St)$ in any state St is:

$$P_{St} = t : \mathrm{SEN} \rightarrow P_{F(St)} \quad (3)$$

where SEN is the set of states that satisfy P's **sensitive to** clause: $\mathrm{SEN} = \{t \mid S(t)\}$. Thus, a process in state St first waits for its **sensitive to** clause to be satisfied and then behaves like the same process in the state defined by applying F to the current state.

Equation 3 defines a CSP process that describes the behavior of a single VSPEC entity. CSP's concurrency operator ($\|$) is used to define a process that describes the behavior of an architecture of VSPEC components. Let $P_0, P_1, \ldots, P_n$ be the processes represented by Eq. (3) for the set of VSPEC component instances in architecture $\mathcal{P}$. The process that represents architecture $\mathcal{P}$ is:

$$\mathcal{P} = P_0 \| P_1 \| \cdots \| P_n \quad (4)$$

Thus, each component in the architecture executes in parallel. Since a component activates only when its **sensitive to** clause predicate is true, this predicate is used to synchronize component execution.

Formal Model Example

This section presents a simple example to explain how the concrete state of a VSPEC architecture changes as its inputs are modified by external components. Consider the architecture shown in Figure 9. The abstract, component and concrete state of the elements of this architecture are:

$$\begin{aligned} \mathrm{ABSTRACT}_{c1} &= \{x, z\} \\ \mathrm{ABSTRACT}_{c2} &= \{x, z\} \\ \mathrm{COMPONENT}_{b1} &= \{i, y\} \\ \mathrm{COMPONENT}_{b2} &= \{y, o\} \\ \mathrm{CONCRETE}_{\mathrm{structural}_{\mathrm{example}}} &= \{i, o, y\} \end{aligned}$$

```
entity example is
  port(i: in integer; o: out integer);
end example;

architecture structural of example is
  component c1
    port(input: in integer;
         output: out integer);
  end component;
  component c2
    port(input: in integer;
    output: out integer);
  end component;
begin
  b1: c1 port map(i,y);
  b2: c2 port map(y,o);
end structural;
```

```
entity c1 is
  port (x: in integer; z: out integer);
  modifies z;
  sensitive to x'event;
  requires I1(x);
  ensures O1(x,z'post);
end c1;

entity c2 is
  port (x: in integer; z: out integer);
  modifies z;
  sensitive to x'event;
  requires I2(x);
  ensures O2(x,z'post);
end c2;
```

FIGURE 9 Specification of two components connected serially.

The transformation performed by an architecture is defined from the components comprising it. Formally, the component requirements for **c1** and **c2** are defined as:

$$\forall x: \text{integer}, \exists z: \text{integer} \cdot I_1(x) \Rightarrow O_1(x, z'\text{post})$$
$$\forall x: \text{integer}, \exists z: \text{integer} \cdot I_2(x) \Rightarrow O_2(x, z'\text{post})$$

The renaming defined by the architecture that is used to create the component state from the abstract state of an architecture can also be applied to these two equations. In this example, this renaming defines the following logical requirements for **b1** and **b2**:

$$\forall i: \text{integer}, \exists y: \text{integer} \cdot I_1(i) \Rightarrow O_1(i, y'\text{post})$$
$$\forall y: \text{integer}, \exists o: \text{integer} \cdot I_2(y) \Rightarrow O_2(y, o'\text{post})$$

The renaming function is also applied to the **modifies**, **state** and **sensitive to** clause of **c1** and **c2**. After this renaming, the logical definitions of each component are expressed in the same name space as the concrete state of the system.

Assume that a, b and c are integer constants and that $f(x)$ and $g(x)$ are functions that satisfy requirements for **c1** and **c2** respectively. Let the initial concrete state of the system be $\mathcal{S}_0 = \{i \mapsto a, y \mapsto b, o \mapsto c\}$ and let i'event be true and y'event be false. This means that **c1**'s **sensitive to** clause is satisfied and **c1**'s pre-condition must hold. **c1** will then make its post-condition hold in the next state. Instantiating the requirements for **c1** gives:

$$\exists z: \text{integer} \cdot I_1(a) \Rightarrow O_1(a, z) \tag{5}$$

Knowing that $f(x)$ satisfies **c1**'s requirements and assuming $I_1(a)$ is true implies that $O_1(a, f(a))$ is also true. Additionally, y'event is known to be false so **c2** maintains its state and o does not change in the next state. Thus, one potential next state for this system is $\mathcal{S}_1 = \{i \mapsto a, y \mapsto f(a), o \mapsto c\}$. Because the function f is one of potentially many functions satisfying **c1**, we cannot claim that this is the only possible next state.

Since y changed values from $\mathcal{S}_0$ to $\mathcal{S}_1$, the predicate y'event is true in $\mathcal{S}_1$. Additionally, i did not change values in $\mathcal{S}_1$ implying that i'event is false in $\mathcal{S}_1$. Thus, only component **c2** activates in state $\mathcal{S}_1$.

Using the same reasoning used for $\mathcal{S}_1$, values for $\mathcal{S}_2$ can be produced. Assuming that $f(a)$ satisfies $I_2(f(a))$ and knowing $g(x)$ satisfies **c2**'s requirements makes $O_2(f(a), g(f(a)))$ true. The input value i has not changed, **c1** maintains its state implying y does not change, and $g(f(a))$ satisfies **c2**'s output condition. Thus, $\mathcal{S}_2 = \{i \mapsto a, y \mapsto f(a), o \mapsto g(f(a))\}$ is a potential next state for the system.

An interesting exercise is defining what happens when the input value i changes between states $\mathcal{S}_0$ and $\mathcal{S}_1$. Assume that i changes value from a to d making $\mathcal{S}_1 = \{i \mapsto d, y \mapsto f(a), o \mapsto c\}$. Now i'event is true in $\mathcal{S}_1$ and both components execute on values from $\mathcal{S}_1$. In this case, $\mathcal{S}_2 = \{i \mapsto d, y \mapsto f(b), o \mapsto g(f(a))\}$. Note the value of o does not change from the previous example because the next state is defined only on variables defined in the current state. Using this model eliminates difficulty caused by instantaneous feedback and "pipelined" update functions. VHDL solves this same problem by allowing an infinite number of delta delays between changes in the modeled simulation time.

Generating Proof Obligations

The VSPEC formal model can be used to verify that a system's abstract architecture description satisfies the requirements described by the VSPEC specification of the system. This verification provides evidence that the abstract architecture description satisfies the abstract VSPEC specification. Finding such evidence depends on: (1) having the system requirements I and O; and (2) relating a concrete state produced by the abstract architecture with the abstract state specified for the system. A system's VSPEC description provides I and O. The abstraction function from the concrete to the abstract state provides the means for comparing the abstract and concrete states.

Weak bisimulation [19] is used as the correctness criteria when attempting to verify that an abstract architecture satisfies a VSPEC description. As shown in Figure 10, weak bisimulation requires that some sequence of state changes in the concrete state of the system result in the correct single state change in the abstract state. Only the first and last of the concrete states are significant. The system may pass through any concrete state as long as the abstraction function applied to the final concrete state results in the correct abstract state as defined by the abstract specification.

In CSP, the sequence of states a VSPEC entity passes through is called a trace. A CSP trace of process P is a finite sequence of symbols representing the events processed by P. VSPEC events are changes in state and they are represented in a trace by the state the entity changes to. Thus, a VSPEC entity satisfies the weak bisimulation criteria if two conditions hold for all traces of the abstract architecture. The first condition is that the abstraction function applied to the initial element of each trace must result in an abstract state that satisfies the abstract pre-condition. The second condition is that the final element of each trace must either have an abstract projection that satisfies the abstract post-condition or there must be some legal sequence of states that can be appended to the trace to form another trace. This ensures that the concrete state eventually reaches a state where the abstract specification is satisfied.

Weak bisimulation is a useful criteria for evaluating an abstract architecture decomposition of many systems. However, weak bisimulation may not be suitable for all systems. Some systems may require more than just the first and last states of the abstract and concrete traces to be equivalent. Defining other correctness criteria for VSPEC abstract architectures is currently an open area of research.

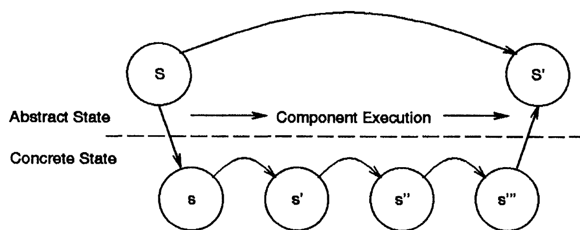


FIGURE 10 Concrete state changes associated with a single abstract state change.

EXAMPLES

This section presents two examples that illustrate how VSPEC can be used to describe an abstract architecture. The first example is a simple tri-state buffer description that is used to define a simple two input multiplexor. This example illustrates what happens when multiple sources drive a single value in a VSPEC abstract architecture. The second example is the description of a simple CPU called the Move Machine. This example illustrates shows a VSPEC description that is decomposed into an abstract architecture.

Buffer and Multiplexor Example

A VSPEC description of a simple buffer is shown in Figure 11. In this example, **input** and **output** are both integers, but the specification could also be used if **input** and **output** were of any other type. When **control** is true, this device passes **input** to **output**. When control is false, the device places no requirements on the value of **output** in the next state. The specification allows for **output** to maintain its current value in the next state, but the specification also allows an external device to

```
entity buffer is
  port (input: in integer;
        control: in boolean;
        output: out integer);
  sensitive to control'event or input'event;
  ensures control implies output'post = input;
end buffer;
```

FIGURE 11 VSPEC description of a simple buffer.

change the value of **output**. Consider using this buffer as a component in the abstract architecture description of the multiplexor in Figure 12.

This figure shows both a VSPEC description of a multiplexor as well as a refinement of this description into an abstract architecture. The VSPEC entity **mux** is a straightforward description of a multiplexor. The **struct** architecture uses two instances of **buffer** and a **not** gate to decompose the multiplexor into an abstract architecture.

Careful examination of this description reveals a very subtle but important point about VSPEC specifications and multiply driven signals. If a component description does not restrict the value of an output signal in the next state, other components in the system can still change the value of this signal without violating the component description. Suppose that the concrete state of the architecture is:

$$\text{CONCRETE}_{\text{struct}_{\text{mux}}} = \{\text{in1} \mapsto 7, \text{in2} \mapsto 3, \text{select} \mapsto \text{true}, \text{output} \mapsto 7, \text{select_inv} \mapsto \text{false}\}$$

so that the abstract state of **buffer** instance **b1** is:

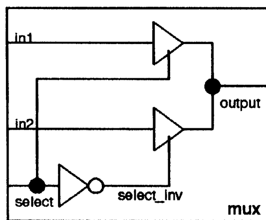
$$\text{ABSTRACT}_{b1} = \{\text{input} \mapsto 7, \text{control} \mapsto \text{true}, \text{output} \mapsto 7\}$$

Assume that some external device changes the **select** input to false. This causes **buffer** instance **b1**'s **control** input to change to false which activates the buffer. This device must now make its **ensures** clause true in the next state. Since **control** is false, the **ensures** clause will be true in the next state for any value of **output**. Thus, **buffer** instance **b2** can change the **output** signal of the architecture to 3 without violating **b1**'s specification. The next state of the device is:

$$\text{CONCRETE}_{\text{struct}_{\text{mux}}} = \{\text{in1} \mapsto 7, \text{in2} \mapsto 3, \text{select} \mapsto \text{false}, \text{output} \mapsto 3, \text{select_inv} \mapsto \text{true}\}$$

Thus, the **output** signal has changed values even though the **b1** buffer instance does not cause it to do so. Even though **b1** does not force a change in state, it does not prohibit one either. An external

```
entity mux is
  port (in1, in2: in integer;
        select: in boolean;
        output: out integer);
  sensitive to in1'event or
    in2'event or select'event;
  ensures
    (select and output'post = in1) or
    (not select and output'post = in2);
end mux;
```




```
architecture struct of mux is
  component buffer
  port (input: in integer;
        control: in boolean;
        output: out integer);
  end component;
  component not
  port (input: in boolean;
        output: out boolean);
  end component;
  signal select_inv : boolean;
begin
  b1: buffer
    port map(in1,select,output);
  b2: buffer
    port map(in2,select_inv,output);
  n1: not
    port map(select,select_inv);
end struct;
```

FIGURE 12 VSPEC and abstract architecture description of a two input mux.

device (buffer instance **b2**) has caused the **output** signal to change values. The specification of **b1** allows this change to occur.

This description may not seem correct to an experienced VHDL user because the output signal is driven by two sources, but no resolution function is specified. Although this is illegal in VHDL, it is allowed in VSPEC. In most cases, the CSP statement that defines a VSPEC entity's contribution to the next state of the system will define a single value for every signal, but a VSPEC description may allow more than one value for a specific signal. This is legal VSPEC because VSPEC is a specification language, not a simulation language like VHDL. This implies that a VSPEC specification does not need to deterministically define a single value for every signal in the system. It is certainly possible to do this with VSPEC by defining the requirements of resolution functions, but a VSPEC specification could allow a signal to be driven to two (or more) different values. In these cases, a designer implementing the specification may chose to drive the signal to any of its allowed values.

The Move Machine

A more complex example is the specification of a Move Machine [22]. The Move Machine is a simple CPU that moves data from one memory location to another. It uses four instructions: jump, load register from memory, store register to memory, and halt, and four addressing modes: absolute, immediate, indirect and relative. Although the Move Machine is a simple device, its structure reflects how a more complex system might be represented.

The first step in specifying the Move Machine is representing it as a simple instruction interpreter (Fig. 13). At this level, only one VSPEC annotated **entity** describes the execution of each instruction and addressing mode. This entity contains state variables to store the current register contents and the value of the instruction pointer. The **sensitive to** clause states that the machine activates when its **start** or **reset** input is on or when the value of the instruction pointer changes. The rather complex **ensures** clause predicate defines how the machine behaves for each instruction and addressing mode. An external entity would use this component by first applying the **reset** signal and then the **start** signal. This causes the machine to begin executing the instruction in memory location 0. The result of each instruction (except **halt**) cause the contents of the instruction pointer to change which activates the machine again in the next state. This continues until a **halt** instruction is processed, causing the machine to stop.

One thing to note about this specification is the **use** clause on the first line. In VHDL, types and functions can be declared in separate packages. These packages are then included in entity and architecture descriptions with the **use** clause. The **mm_types** package referenced in this example is shown in Figure 14. An interesting aspect of this package is the use of incomplete types to specify **address** and **word**. VHDL uses incomplete types to allow references to a type before the type is completely defined (such as in an access type). One use of this is to allow a record to contain a pointer to another record of the same type (*i.e.*, to construct a list).

In VSPEC, incomplete types are used for a slightly different purpose. The type definitions for **address** and **word** are incomplete because no implementation is defined. They are declared to be types, but no additional information is provided. These incomplete types will be given characteristics by the specification, but no specific implementation is implied or mandated. Thus, the designer must select an implementation at a lower abstraction level. Using incomplete types allows the designer to specify a type's characteristics without specifying its implementation.

The characteristics of the **address** and **word** types are defined in the LSL **Instruction** trait. This trait is included in **mm_types** using a VSPEC **includes** clause and the trait is shown in Figure 15. The **Instruction** trait provides definitions for conversion functions that allow instructions, register numbers and addresses to be obtained from memory words. In

```
use work.mm_types.all;
entity mm is
 port (reset,start : in boolean;
       mem: inout memory);
 state (ip : address;
        reg : regfile);
 sensitive to start or reset or
   ip'event;
 ensures

 (reset and ip'post = 0) or

 (not reset and

  ((ins(mem(ip)) = jump and
    ip'post=addr(mem(ip)))

  or (ins(mem(ip)) = load and
  ((am(mem(ip)) = ab and
    reg(rnum(mem(ip)))'post =
     addr(mem(ip))) or
   (am(mem(ip)) = imm and
    reg(rnum(mem(ip)))'post =
     mem(ip +1)) or
   (am(mem(ip)) = ind and
    reg(rnum(mem(ip)))'post =
     mem(addr(mem(ip)))) or
   (am(mem(ip)) = rel and
    reg(rnum(mem(ip)))'post =
     mem(ip + addr(mem(ip))))))

  or (ins(mem(ip)) = store and
  ((am(mem(ip)) = ab and
    mem(addr(mem(ip)))'post =
     reg(rnum(mem(ip)))) or
   (am(mem(ip)) = imm and
    mem(ip +1) =
     reg(rnum(mem(ip)))) or
   (am(mem(ip)) = ind and
    mem(mem(addr(mem(ip)))) =
     reg(rnum(mem(ip)))) or
   (am(mem(ip)) = rel and
    mem(ip + addr(mem(ip))) =
     reg(rnum(mem(ip)))))))

  and ((ins(mem(ip)) = store or
    ins(mem(ip)) = load) and
    ((am(mem(ip)) /= imm and
      ip'post = ip'post+1)
     or (am(mem(ip)) = imm and
      ip'post = ip'post+2))));
end mm;
```

FIGURE 13 The Move Machine requirements represented as an instruction interpreter.

```
package mm_types is
  type address;
  type word;
  includes Instruction(word,address,integer);
  type control is (fetch,decode,execute,halt);
  type memory is array(0 to 256) of word;
  type regfile is array(0 to 15) of word;
end mm_types;
```

FIGURE 14 Package declaring types used in the Move Machine.

the final format of the Move Machine instructions (not shown in this paper), this would be implemented by defining which bits of a memory word encode the instruction, register number and address. However, when specifying the initial requirements of the device, such details should not be considered. All that must be specified is that instructions, register numbers and addresses can be obtained from memory words. This is exactly what the LSL description allows us to say.

Once the Move Machine's initial requirements are defined, the device can be broken up into an abstract architecture and each of the components can be synthesized individually. For a CPU such as the Move Machine, one such architecture is the canonical fetch–decode–execute structure. An instruction is retrieved, the addressing modes are decoded and dereferenced, and the instruction is executed on its operands. Effectively, the Move Machine is now three components that execute in sequence.

Figure 16 shows the fetch–decode–execute architecture for the Move Machine. The signals

```
Instruction(W,A,N): trait

includes
  Natural(N)
  mode enumeration of abs, imm, ind, rel
  instruction enumeration of halt, jump, load, store

introduces
  am: W → mode
  addr: W → A
  ins: W → instruction
  rnum: W → N
```

FIGURE 15 LSL support functions for treating memory contents as instructions. Basic types and conversions are defined.

mem, **reg**, **IP**, **IR**, **EA** and **CNTL** exchange memory, registers and control values between components. The **requires** and **ensures** clauses for each component describe transformations performed on memory and register values while the **sensitive to** clauses uses the control value indicates what component(s) should be active.

Each component's **sensitive to** clause indicates that it should be active when its execution phase begins. As with the instruction interpreter, the machine starts by turning on the **reset** signal. This causes the **fetch** component to activate and sets the instruction pointer to 0. After **reset** turns off, all components are inactive until the **start** signal is asserted. **Fetch's sensitive to** clause is the only **sensitive to** clause satisfied by this action, so **fetch** is the only component that activates. All other components have no affect on the concrete state of the architecture. The **fetch** component retrieves the current instruction from memory and places it in the instruction register (**IR**). It also sets the **cntl** signal to **decode**.

The only component whose **sensitive to** clause is satisfied at this point is **decode**. This component calculates the effective address based on the addressing mode specified by the instruction in the **IR** and sets the **cntl** signal to **execute**. The **execute** component then manipulates the registers and memory based on the current instruction. When a **load**, **store** or **jump** instruction is executed, **execute** sets the **cntl** signal to **fetch** which causes the **fetch** component to activate and the process starts again. If the **halt** instruction is processed, **execute** sets **cntl** to **halt**. This makes all three component's **sensitive to** clauses false and the concrete state of the architecture does not change again until something (such as activating **reset**) outside of **mm** changes it.

RELATED WORK

Software Architecture

The research area most closely related to abstract architecture representation in VSPEC is software architecture [8]. Research in this field has led to the development of several architecture description languages, including UniCon [23], WRIGHT [3, 4] and RAPIDE [16, 17]. Each of these languages allow the definition of components and connectors to define a software architecture. This is similar to the VHDL notion of a structural architecture described in this paper.

Shaw's UniCon language [23] is one example of an architecture description language. A UniCon description consists of component and connector definitions. Each of these definitions gives the type (such as Filter or Process for components and Pipe or FileIO for connectors), association units (component players and connector roles) and an implementation for the component or connector. The primary product of the UniCon compiler is Odinfiles, something similar to makefiles that can be used to construct executables for the described architecture. Thus, one of the main products of a UniCon description is a facility that is used to construct an executable version of the described

```
use work.mm_types.all;
architecture mm_fde of mm is
  component fetch
  port (reset,start : in boolean;
        mem: in memory;
        ip : inout address;
        ir : out word;
        cntl: inout control);
  end component;
  component decode
  port (mem: in memory;
        ip: in address;
        ir: in word;
        ea: out address;
        cntl: inout control);
  end component;
  component execute
  port (mem: inout memory;
        reg: inout registers;
        ea: in address;
        cntl: inout control);
  end component;

  signal CNTL: control;
  signal IP : address;
  signal IR : word;
  signal EA : address;
  signal reg : regfile;

begin
  b1: fetch port map (reset,start,
                      mem,IP.IR,cntl);
  b2: decode port map (mem,IR,EA,CNTL);
  b3: execute port map (mem,reg,EA,CNTL);
end mm_fde;

use work.mm_types.all;
entity fetch is
  port(reset,start : in boolean;
       mem: in memory;
       ip : inout address;
       ir : out word;
       cntl: inout control);
  sensitive to start or reset or
    cntl=fetch;
  modifies ir,cntl;
  requires true;
  ensures
    (reset and ip'post = 0)
    or (not reset and
     ir'post=mem(ip)
     and cntl'post=decode);
end fetch;

use work.mm_types.all;
entity decode is
  port (mem: in memory;
        ip: in address;
        ir: in word;
        ea: out address;
        cntl: inout control);
  sensitive to cntl=decode;
  modifies ea,cntl;
  requires true;
  ensures
   ((am(ir) = ab and
     ea'post=addr(ir)) or
    (am(ir) = imm and
     ea'post=ip+1) or
    (am(ir) = ind and
     ea'post=mem(addr(ir))) or
    (am(ir) = rel and
     ea'post=ip+addr(ir)))
   and cntl'post=execute;
end decode;

use work.mm_types.all;
entity execute is
  port(mem: inout memory;
       ip: inout address;
       ir: in word;
       reg: inout regfile;
       ea: in address;
       cntl: inout control);
  sensitive to cntl=execute;
  modifies mem,reg,ip,cntl;
  requires true;
  ensures
    (ins(ir) = jump and
     ip'post=addr(ir) and
     cntl'post=fetch)
     or (ins(ir) = load and
       reg(rnum(ir))'post=mem(ea) and
       cntl'post=fetch and
       ((am(ir) = imm and
         ip'post = ip+2) or
        (am(ir) /= imm and
         ip'post = ip+1)))
     or (ins(ir) = store and
       mem(ea)'post=reg(rnum(ir)) and
       cntl'post=fetch and
       ((am(ir) = imm and
         ip'post = ip+2) or
        (am(ir) /= imm and
         ip'post = ip+2)))
     or (ins(ir) = halt and
       cntl'post=halt);
end execute;
```

FIGURE 16 High level fetch–decode–execute architecture for the Move Machine CPU.

architecture. This is very different from a VSPEC abstract architecture, which is used to verify that the class of solutions defined by the architecture implements the requirements specified by the VSPEC description of the component.

The WRIGHT architecture description language [3, 4] by Allen and Garlan is of particular interest when discussing abstract architectures in VSPEC. A WRIGHT description consists of a collection of components interacting *via* instances of connector types. Each part of a WRIGHT description is defined using a variant of CSP [11]. Unlike VSPEC's use of CSP to define only communications between components, WRIGHT descriptions use CSP to define the behavior of components as well. WRIGHT's CSP descriptions define the sequence of events that occur in a component or connector. Components and connectors interact when one component/connector *observes* an event *provided* by another. This may cause the second component/connector to *provide* events that cause further interactions. These interactions are all described using CSP.

RAPIDE [16, 17] is an executable architecture description language designed for prototyping architectures of distributed systems. A RAPIDE architecture consists of a set of module specifications (called interfaces), a set of connection rules defining communication between interfaces and a set of formal constraints that define legal patterns of communication. A RAPIDE architecture is executed to produce a partially ordered set of events (poset) that represents the dependencies between events inthe architecture. The RAPIDE tools can then verify this poset does not violate the formal constraints defined in the architecture. A major difference between RAPIDE and VSPEC is that VSPEC descriptions are not executable. They are intended for formal analysis.

Other VHDL-Related Specification Languages

Odyssey Research Associates (ORA) is developing Larch/VHDL, an alternative Larch interface language for VHDL [13]. Larch/VHDL is targeted for formal analysis of a VHDL description and ORA is defining a formal semantics for VHDL using LSL. The LSL representations are used in a traditional theorem prover to verify system correctness. Larch/VHDL annotations are added to a specific VHDL description to represent proof obligations for the verification process. In contrast to this approach, a VSPEC abstract architecture represents the requirements of a class of solutions that satisfy a specification (also given in VSPEC).

Augustin and Luckham's VAL [6] is another attempt to annotate VHDL. The purpose of a VAL annotation to a VHDL description is to document the design for verification. VAL provides mechanisms for mapping a behavioral description to a structural description. Two VAL/VHDL descriptions of a design can be transformed into a self-checking VHDL program that is simulated to verify that the two descriptions implement the same function. This differs from VSPEC because it does not allow the description of a class of solutions that implement a specification. Instead, it allows the verification that a structural description correctly maps to a behavioral description for the entity.

Larch Interface Languages

Larch interface languages have been developed for a variety of programming languages, including LCL [9], Larch/C++ [15] and LM3 [14], interface languages for C,C++ and Modula-3, respectively. Each of these languages allow the description of the pre- and post-conditions for procedures and functions in a sequential programming language. The portions of these languages that allow this type of specification (*i.e.*, **requires**, and **ensures** clauses) are also found in VSPEC, where they are used to specify the transformation performed by a single component. However, since C,C++ and Modula-3 are sequential languages, their Larch interface languages do not have to deal with how the Larch-specified procedures and functions interact when two procedures are executing concurrently as is the case with VSPEC entities. At the present time, we are not aware of other work in the

Larch community where pre- and post-conditions are used to specify the behavior of components in an abstract architecture.

CONCLUSION

Summary

The ability to evaluate architectural decisions early in the design process enhances overall design quality by allowing architectural errors to be discovered when they are less expensive to fix. Unfortunately, VHDL does not allow evaluation until a simulatable model exists. For many complex systems, simulatable models appear late in the design process making architectural errors difficult to correct. An alternative to simulation for evaluating architectural decisions is formal analysis of abstract architectures at the requirements level. An abstract architecture is a set of interconnected components where the requirements of each component are known but the implementation is not. This paper presented VSPEC's support for describing and evaluating abstract architectures during requirements specification.

A VSPEC abstract architecture is formed by instantiating each component in a VHDL structural architecture with a VSPEC entity. The VSPEC description of an entity includes a pre-condition, post-condition and an activation condition that describe the entity's functional requirements. If the current state of the system satisfies the activation condition for one of the components in the abstract architecture, that component's pre-condition must hold and the component must satisfy its post-condition in the next state. A refinement of a VSPEC entity can be compared with the VSPEC specification using weak bisimulation. If some sequence of state changes in the refinement yields the correct single state change in the higher-level description, weak bisimulation holds. This method can be used to formally determine if a VSPEC abstract architecture is a refinement of the VSPEC description of the entity it implements.

Status and Limitations

VSPEC provides a specification capability most appropriate for high levels of abstraction. It is anticipated that designers will represent system requirements with VSPEC, gradually refining requirements into architectures and eventually a VHDL design. During requirements specification when a designer is defining the essential requirements of a system, VSPEC is useful for evaluating the impact of architectural decisions. When design details are available, VHDL simulation is a more suitable analysis activity. Although VSPEC can model design detail, formal analysis is far less pragmatic than VHDL simulation in such situations.

A potential limitation to the VSPEC approach is verifying the refinement of VSPEC requirements into VHDL design representations. Formalizing the tie between VSPEC and VHDL to support verification and comparison with simulation results is the subject of current investigations. In addition, techniques for automatically synthesizing VHDL from VSPEC are currently under development [21, 20]. Studies of error analysis reports for safety-critical software systems suggest that over 90% of safety related errors arise from incorrect or incomplete specifications, not transformation of requirements into implementations [18]. This suggests that the use of techniques such as those proposed here are warranted even before a complete verification path between VSPEC and VHDL exists.

Acknowledgements

Support for this work was provided in part by the Advanced Research Projects Agency and monitored by Wright Labs under the RASSP Technology Program, contract number F33615-93-C-1316. The authors wish to thank our sponsors for their continued support. We would also like to thank the reviewers for their comments.

References

[1] Alexander, P., Baraona, P. and Penix, J. (1994). Using Declarative Specifications and Case-Based Planning for

System Synthesis. *Concurrent Engineering: Research and Applications 2*, **4**.

[2] Alexander, P., Baraona, P. and Penix, J., Application of Software Synthesis Techniques to Composite Systems. In: *Computer in Engineering Symposium of the ASME ETCE* (Houston, TX, January 1995).

[3] Allen, R. and Garlan, D., Formalizing Architectural Connection. In: *Proc. Sixteenth International Conference on Software Engineering* (May 1994), pp. 71–80.

[4] Allen, R. and Garlan, D., The wright architectural specification language. Tech. Rep. CMU-CS-96-TBD, Carnegie Mellon University, September 1996, Draft.

[5] Ashenden, P. (1996). *The Designers Guide to VHDL*, Morgan Kaufmann Publishers, Inc., San Mateo, CA.

[6] Augustin, L., Luckham, D., Gennart, B., Huh, Y. and Stanculescu, A. (1991). *Hardware Design and Simulation in VAL/VHDL*, Kluwer Academic Publishers, Boston, MA.

[7] Baraona, P., Penix, J. and Alexander, P. (1995). VSEAC: A Declarative Requirements Specification Language for VHDL. In: *High-Level System Modeling: Specification Languages*, Berge, J.-M., Levia, O. and Rouillard, J., Eds., Vol. 3 of *Current Issues in Electronic Modeling*. Kluwer Academic Publishers, Boston, MA, Ch. 3, pp. 51–75.

[8] Garlan, D. and Shaw, M. (1993). An Introduction to Software Architecture. In: *Advances in Software Eng. and Knowledge Eng.*, Ambriola, V. and Tortora, G., Eds., Vol. 2, World Scientific, New York, pp. 1–39.

[9] Guttag, J. V. and Horning, J. J., Introduction to LCL, A Larch/C Interface Language, Tech. Rep. 74, Digital Equipment Corporation Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, July 1991.

[10] Guttag, J. V. and Horning, J. J. (1993). *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, New York, NY.

[11] Hoare, C. A. R. (1985). *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs.

[12] *IEEE Standard VHDL Language Reference Manual*, New York, NY (1994).

[13] Jamsek, D. and Bickford, M., Formal Verification of VHDL Models. Technical Report RL-TR-94-3, Rome Laboratory, Griffiss Air Force Base, NY, March 1994.

[14] Jones, K. D., LM3: A Larch Interface Language for Modula-3. A Definition and Introduction. Version 1.0. Technical Report 72, DEC Systems Research Center, June 1991.

[15] Leavens, G. T. (1995). Larch/C++ Reference Manual. Available at: ftp://ftp.cs.iastate.edu/pub/larchc++/lcpp.ps.gz.

[16] Luckham, D., Kenney, J., Augustin, L., Vera, J., Bryan, D. and Mann, W., Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering 21*, **4** (April 1995), 315–355.

[17] Luckham, D. and Vera, J., An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering 21*, **9** (September 1995), 717–734.

[18] Lutz, R. (1992). Analyzing software requirements errors in safety-critical embedded systems. Tech. Rep. 92–27, Department of Computer Science, Iowa State University.

[19] Milner, R., Tofte, M. and Harper, R. (1990). *The Definition of Standard ML*. MIT Press, Cambridge, MA.

[20] Penix, J. and Alexander, P., Design representation for automating software component reuse. In *Proceedings of the First International Workshop on Knowledge-Based Systems for the (re) Use of Program Libraries* (Nov. 1995).

[21] Penix, J., Baraona, P. and Alexander, P., Classification and retrieval of reusable components using semantic features. In: *Proceedings of the 10th Knowledge-Based Software Engineering Conference* (Nov. 1995), pp. 131–138.

[22] Roy, J., Kumar, N., Dutta, R. and Vemuri, R., DSS: A Distributed High-Level Synthesis System. *IEEE Design and Test of Computers* (June 1992), 18–32.

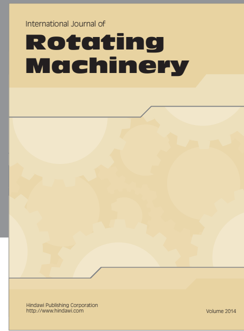
[23] Shaw, M., Deline, R., Klein, D., Ross, T., Young, D. and Zelesnik, G., Abstractions for Software Architecture and Tools to Support them. *IEEE Transactions on Software Engineering 21*, **4** (April 1995), 314–335.

Authors' Biographies

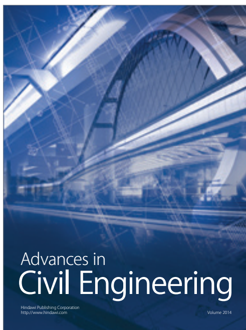
Phillip Baraona is a Computer Engineering Ph.D. candidate at the University of Cincinnati. The topic of his dissertation research is formally defining the semantics of the VSPEC language. This research has led to an interest in topics such as formal specification languages, process algebras, software synthesis and hardware verification. After receiving a B.S. in Computer Engineering from the University of Cincinnati in 1992. Mr. Baraona began working in his current position as a research assistant in UC's Knowledge-Based Software Engineering Lab. Mr. Baraona is a member of IEEE and ACM.

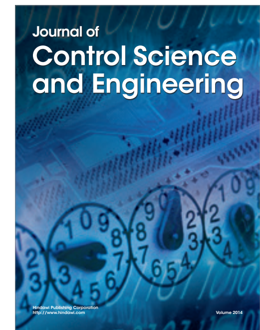
Perry Alexander received the Ph.D. in Electrical and Computer Engineering, the MSEE, BSEE and BSCS from the University of Kansas, Lawrence, KS. He is an assistant professor of Electrical and Computer Engineering and Computer Science and director of the Knowledge-Based Software Engineering Laboratory at the University of Cincinnati. His research interests include design theory, software synthesis, formal specification, formal verification, and software reuse. Dr. Alexander is a member of IEEE, ACM and currently serves as the chair of the IEEE ECBS Technical Committee working group on Systems Evaluation.

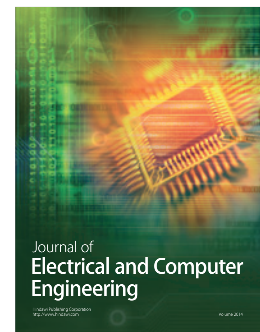
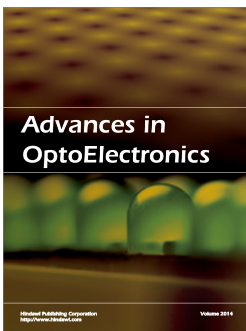
Journal of
Engineering

The Scientific
World Journal

International Journal of
Rotating
Machinery

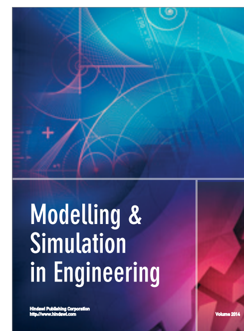
Journal of
Sensors

International Journal of
Distributed
Sensor Networks

Advances in
Civil Engineering

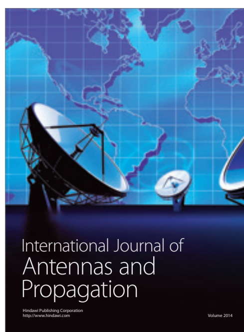
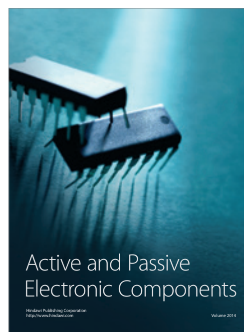
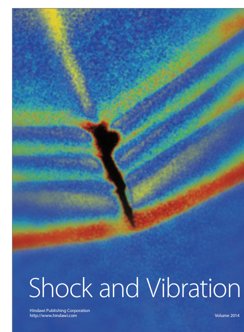
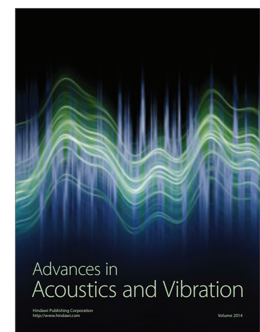
Hindawi
Submit your manuscripts at
http://www.hindawi.com

Journal of
Control Science
and Engineering

Journal of
Robotics

Journal of
Electrical and Computer
Engineering

Advances in
OptoElectronics

VLSI Design

International Journal of
Navigation and
Observation

Modelling &
Simulation
in Engineering

International Journal of
Aerospace
Engineering

International Journal of
Chemical Engineering

International Journal of
Antennas and
Propagation

Active and Passive
Electronic Components

Shock and Vibration

Advances in
Acoustics and Vibration