EVENT PATTERNS: A LANGUAGE CONSTRUCT FOR HIERARCHICAL DESIGN OF CONCURRENT SYSTEMS

David C. Luckham
Bennoit A. Gennart

Technical Report: CSL-TR-90-453

(Program Analysis and Verification Group Report No. 51)

November 1990

Research supported by the Air Force Office of Scientific Research under Grant AFOSR83-0255, and by the Defense Advanced Research Projects Agency/Information Systems Technology Office under the Office of Naval Research, contract N00014-90-J1232.

Event patterns : a language construct for hierarchical design of concurrent systems

by

David C. Luckham

Benoit A. Gennart

Technical Report CSL-TR-90-400
Program Analysis and Verification Group Report No. 50
November 1990

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

Event patterns are a language construct for expressing relationships between specifications at different levels of a hierarchical design of a concurrent system. They provide a facility missing from current hardware design languages such as VHDL, or programming languages with parallel constructs such as Ada. This paper explains the use of event patterns in (1) defining mappings between different levels of a design hierarchy, and (2) automating the comparison of the behavior of different design levels during simulation. It describes the language constructs for defining event patterns and mappings, and shows their use in a design example, a 16-bit CPU.

Key Words and Phrases: Event pat tern mappings, hierarchical design, concurrent systems, discrete event simulation, programming languages, design verification, consistency checking

Contents

List of Figures

1 Hierarchical designs

This paper presents new language features for expressing the hierarchical development of concurrent systems. These features, called event patterns and pattern mappings, can be viewed as extensions of current design and simulation languages, such as VHDL [6]. They provide a basis for implementing new tools for automated analysis of design behavior. While our discussion here is biased towards hardware systems, event patterns and mappings can be used to express hierarchical designs of systems containing both hardware and software components.

The underlying thesis of this paper is that in dealing with designs of concurrent systems, *patterns* are a necessary component of any facility for defining design hierarchy.

We begin with an informal discussion of design hierarchy.

A **specification** of a system (or component of a system) consists of (1) a set of actions (and data parameters) by which the system communicates with other systems, and (2) a definition of behavior by means of relationships between those actions, e. g., causal relationships between actions and functional relationships between their parameters. A typical high level hardware specification, for example, declares input and output ports (changing the values of which are special kinds of actions), and defines a behavior by functions that map input data to output data.

Specifications of a system are made at different levels of abstraction. Levels shown in Figure **1** are commonly referred to in hardware designs. For us, a **level** is determined by a set of concepts — i. e. data types and operations. If a specification (including the specification of its components) assumes only concepts in a set associated with a level, then we say that the specification is made at that level. For example, in Figure 1 at the instruction level, the concepts that are assumed include the data type Integer and the set of arithmetic functions. At the gate level, the concepts are more primitive operations such as the simple boolean functions (nand, nor, not), applied to the data type Bit.

A **design hierarchy** consists of a number of specifications, each made a different levels. Each specification assumes a different set of concepts, although they may have some common concepts. Such specifications are of course related, and definition of those relationships is what concerns us here.

A **design process** includes the following activities :

- **specification** : the activity of specifying a system involves defining its observable behavior by means of specifications. Note we use the word "specification" to describe both the activity of specifying, and result of that activity, i. e. a specification of behavior.

- **refinement** : In top down design methodology, a high level specification of a system is refined step by step, new details being introduced by use of lower level concepts. The result is a design hierarchy.

  Figure 2 shows the structure of the VHDL descriptions corresponding to the instruction and register transfer levels of a CPU design. The instruction level consists of the CPU entity interface, and a behavioral specification. The register transfer level consists of the same CPU entity interface, a structural architecture describing the connections between the CPU components **(REG, PLA, BUF** and **ALU)**, and for each component, a behavioral specification **(bhv).**

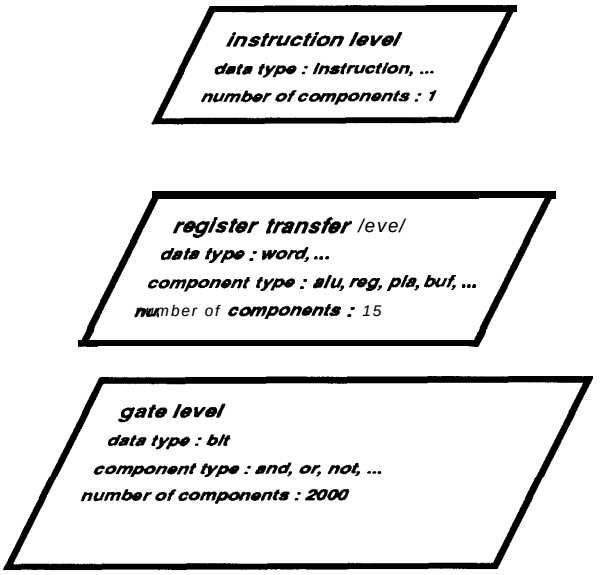

Figure 1: CPU design hierarchy

- **comparative analysis** : the designer must have the ability to analyse the consistency between levels of the design hierarchy, if possible in an automated manner. Analysis methods fall into two categories : *verification* and *validation*. Formal verification is the process of proving the consistency without actually executing a specification. Comparative validation is based on executing specifications at different levels of abstraction and comparing the execution results.

Current high-level languages (hardware description and simulation languages such as VHDL [5] and HSL-FX [3], and general purpose programming languages such as Ada [1]) provide abstraction features that support the specification part of the design process. However none provide features to define hierarchical relationships between specifications — they do not support the refinement part of the design process. Consequently, comparative analysis tools are not provided.

The cost of formally specifying the hierarchical relationships between specifications is greatly outweighed by the benefits : (1) existence of a documentation of the refinement process between specifications ; and (2) the ability to completely automate the comparative analysis step of the design process.

1.1 Structure of the paper

Section 2 describes language constructs *entity*, *action*, *constraint* and *architecture* for specifying concurrent systems at different levels of abstraction. Our proposal for defining mappings between specifications of concurrent systems assumes that a simulation produced by executing a specification is a partially ordered set of events. The partial ordering of events (resulting from executions of actions) is also described informally in section 2. Section 3 describes a mapping construct based on event patterns, and how mappings are used to define design hierarchies. Section 4 shows how to

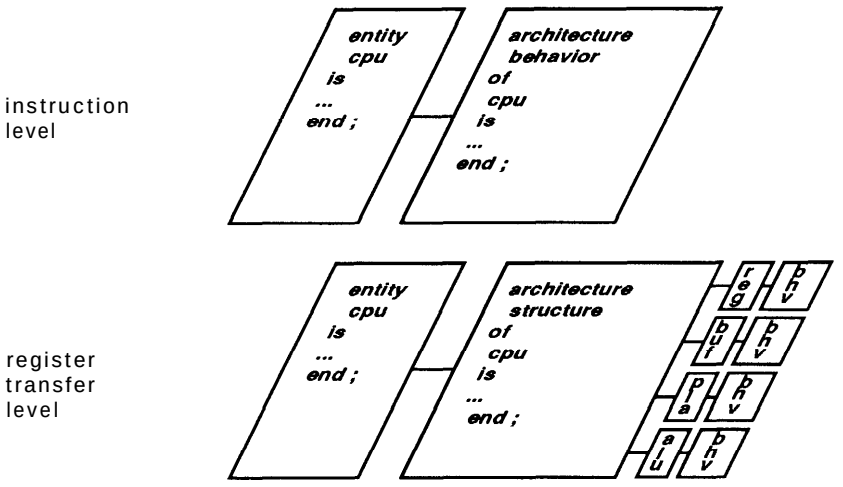

Figure 2: VHDL descriptions

implement comparative validation using event pattern mappings and describes the current status of the research. Section 5 introduces the language for defining patterns of events in a partial ordering. Appendix A presents a complete specification of the CPU example and its components.

The paper assumes some familiarity with VHDL. For the purpose of presenting examples with a minimum of explanation, we use syntax that is based loosely on VHDL and Ada. Language features that are not explained have the same semantics as similar constructs in VHDL. The treatment given here is informal and incomplete, intended as an overview of concepts ; the full details are given in forthcoming reports.

2 Specifying the behavior of concurrent systems

In this section we outline language features for specifying the behavior of a concurrent system. The resulting behaviors are partially ordered sets of events.

In general, a concurrent system specification may have two parts : an *executable* part, and a *constraint bused* part. Executable specifications are usually either programs that simulate the behavior of a design, or networks that define dataflow between other executable specifications of components. Constraint-based specifications define conditions — called **constraints** — that the behavior must satisfy. Often a constraint-based specification can be satisfied by many behaviors. These two parts (executable and constraint based) are separated into a declaration that may contain constraints, and an architecture that is executable (see Figure 2).

The data types and operations that are assumed by a specification are called **concepts** of that specification. Concepts are either imported from software libraries, or else they are predefined types and operations of the formal language. The data type *bit* and the boolean operation "=" are concepts that appear in many specifications.

Our discussion is based on an example, a 16-bit CPU, that will be introduced progressively in the next sections. The example assumes the following concepts :

```
type bit is ('0', '1') ;
type Int2 is integer range 0..3 ;
type WordT is string (1..4) ;
    -- hexadecimal 16 bit
type AddrT is string (1..3) ;
    -- hexadecimal 12 bit
type OpcodeT is (Id, st, ex, id) ;
     -- load, store, execute, idle
type AluOpT is (land, lor, lnot, lxor) ;
type InstrT ( opcode : OpcodeT ) is record
    case opcode is
    when ld | st => register : Int2 ;
                    address : AddrT ;
    when ex      => register1 : Int2 ;
                    register2 : Int2 ;
                    operation : AluOpT ;
    when id      => null ;
    end case ;
end record ;
```

2.1 Design entities, Actions, and Events

- An **entity declaration** is a specification encapsulating (1) a set of actions by which the entity communicates with other entities, and (2) a constraint-based specification of its behavior.

- **actions.** There are three kinds of actions. An *out-action* can be performed by an entity and observed by other entities. An *in-action* of an entity is performed by other entities and observed by the entity. An *internal action* is performed by an entity and observed by that entity. Actions generalize language constructs such as VHDL **ports** and Ada task **entry** declarations for communication between concurrent units.

```
design CPU is
  -- ports
  in action Clk ;
  in action Instr ( ins : InstrT ;
                    din : WordT ) ;
  out action Addr ( d : AddrT ) ;
  out action Re ( b : bit ) ;
  out action We ( b : bit ) ;
  out action Dout ( d : WordT ) ;
      action Load ( addrss : AddrT ;
                    reg : Int2 ;
                    data : WordT ) ;
  -- constraint
  declare
      ?ins : InstrT ;
      ?din : WordT ;
```

```
  pattern
      when Instr (?ins, ?din) where ?ins.opcode = ld then
          Load ( ?ins.address, ?ins. register, ?din) ;
      before Clk -> Clk -> Clk -> Clk ;
  end ;
end CPU ;
```

The CPU declaration has two in-actions, **Clk** and **Instr**, by which other entities can communicate with it. Since **Clk** has no parameters, it can only be used for synchronization. The in-action **Instr** has two parameters (**ins** of type InstrT, and **din** of type WordT). In-action parameters are set by other entities and read by the design entity. Thus **Instr** is used to receive an instruction **ins** and input data **din.**

The CPU can perform the out-actions **Addr,** with one parameter **d** of type AddrT, **Re,** with one parameter **b** of type **bit, We,** with one parameter **b** of type **bit,** and **Dout,** with one parameter **d** of type WordT. **Addr** is the address output of the CPU, **Re** its read-enable output, **We** its write-enable output, and **Dout** its data output. Out-action parameters are set by the design entity and observed by other entities.

The CPU can also perform the internal action **Load** which has parameters : (1) **addrss,** the address in memory from which the CPU loads, (2) **reg,** an integer parameter that represents abstractly the register component of the CPU in which the data is loaded, and (3) **data,** the data word that is loaded. Internal action parameters are set and observed by the design entity. An internal action can be observed only by the entity itself. Intuitively, it indicates an internal change of state of the entity.

- An **event** is an instance of an action. An event results when an entity executes an action.

```
Load ( "9C3", 1 ,  "AFA0")
```

This event is an instance of the action **Load,** with the **addrss** parameter bound to value "9C3", the **reg** parameter bound to value 1, and the **data** parameter bound to value **"AFAO".**

- A **constraint** on behavior may also appear in an entity declaration. A constraint may be an assertion about functional relationships between the parameters of events resulting from in-actions and out-actions, or it may constrain the order in which events can be issued by the design entity. Constraints may be parameterized using *placeholders.* Placeholders are variables that are set when the actions in a constraint are matched to events. *Matching* is the process of replacing placeholders in expressions by values, so that the resulting expressions are identical to event parameters in a simulation. If a placeholder occurs several times in a constraint, all its occurrences must be bound to the same value in the same match. For example, if the action description **Load (?ins.address, ?ins.register, ?din)** is matched to the event **Load** (" 9C3", 1, "AFA0"), the result of the match will bind the placeholder **?din** to value "AFA0", the **address** field of placeholder **?ins** to value "9C3", and the register field of placeholder **?ins** to value 1.

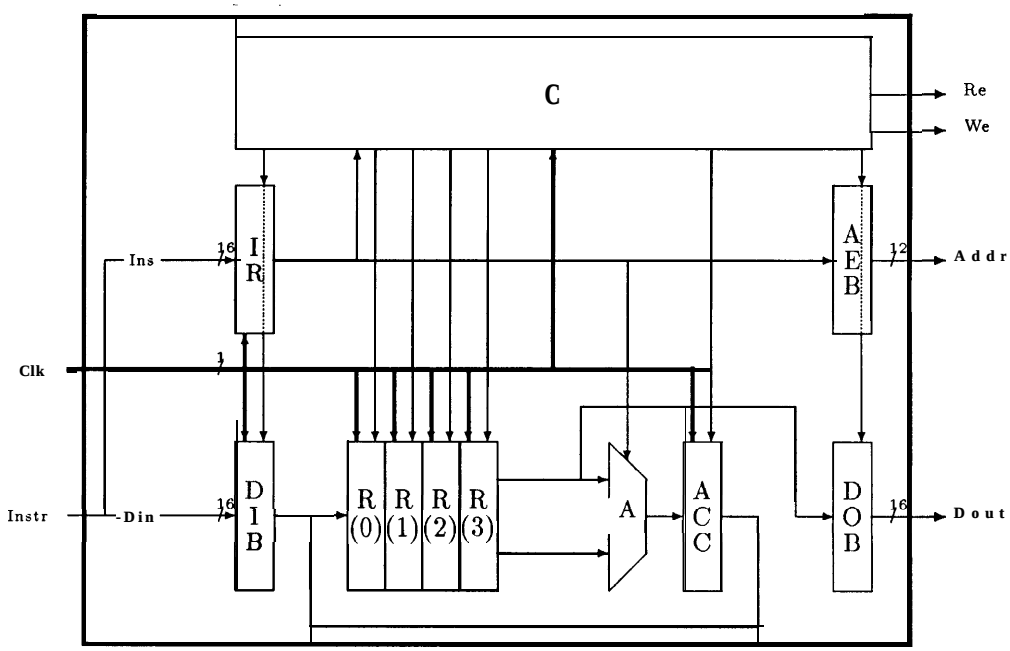

Figure 3: CPU architecture

The constraint in the CPU example above expresses that whenever an event occurs on its in-action **Instr**, with its **ins** parameter bound to a value corresponding to a load instruction **(?ins.opcode = ld)**, then the CPU must perform, within the next four clock events, an internal action **Load**, with its first and second parameters bound to the value taken by the **address** and **register** fields of the pattern variable **?ins**, and its third parameter bound to the value taken by the **?din** pattern variable.

2.2 Entity Architecture

A separate **entity architecture** is associated with an entity declaration. Similarly to VHDL, the architecture contains an executable description of the behavior of an entity. This may be defined by an executable program, or by networks of lower level entities. A network connects an out-action of a component to in-actions of other components, provided the actions have the same parameter profile. If a component executes an out-action with certain parameter values, then events that are instances of the in-actions to which it is connected, occur in other components. Figure 3 gives a graphical representation of the register transfer level CPU architecture. It shows dependencies between components, but for simplicity does not show all of the individual connections.

The CPU architecture is a network of register transfer level components. The CPU, at the register transfer level, consists of five kinds of components : registers, two–output registers, buffers, a logic unit, and a controller. We describe the interface of these components informally ; formal interfaces are given in the Appendix.

- IR, the instruction register, and **ACC**, the accumulator, are **registers.** The register component has four in-actions, **Din** (input data), Ce (clock enable), Oe (output enable), and **Clk** (clock) ;

one out-action, Dout (output data) ; and two internal actions, **Load** and **En.** If the clock is enabled (i. e. the value of the parameter of the last action Ce was '1'), the register loads (stores internally) the value of its data input on each occurrence of the **Clk** action. When the register output is enabled (i. e. the value of the parameter of the last action Oe was '1'), the register outputs its internal state (i. e. the value of the parameter of the last occurrence of the internal action **Load)** through its out-action Dout, and issues an **En** action with its parameter bound to the value of register state.

- R(0), R(1), R(2), R(3) are **two-output registers.** The two-output register component has five in-actions, **Din** (input data), Ce (clock enable), Oe1 (first output enable), Oe2 (second output enable), and **Clk** (clock) ; two out-actions, **Dout1** (output data), and Dout2 (output data) ; and three internal actions, **Load, En1** and **En2.** The behavior of the two-output register is similar to the behavior of the register, with input Oe1 and Oe2 controlling Dout1 and **Dout2** respectively.
- **DIB, DOB** and **AEB** are **buffers.** The buffer component has two in-actions, Oe (output enable) and **Din** (data in) ; one out-action **Dout** (data out) ; and one internal action **En.** When the buffer output is enabled (i. e. the value of the parameter of the last action Oe was true), the buffer outputs its input value (i. e. the value of the parameter of the last occurrence of the in-action **Din).**
- **A** is a **logic unit.** The logic unit component has three in-actions, the two operands Op1 and Op2, and the operation selection **Op** ; and one out-action, Dout. The logic unit can perform one of four operations **(and, or, not, xor)** on the two operands, depending on the operation selection input.
- C is the CPU **controller.** The controller is a state machine. It has two in-actions, **Input** and Clk ; twelve out-actions, **Re** (read enable), **We** (write enable), Roe1 (first register output enable), **Roe2** (second register output enable), **Rce** (register clock enable), Irce (instruction register clock enable), **Irrst** (instruction register reset), Acce (accumulator clock enable), Acoe (Accumulator output enable), **Diboe** (data in buffer clock enable), **Doboe** (data out buffer output enable), and **Aeboe** (address buffer output enable). The controller promotes in-actions of the other CPU components.

The flow of data in the CPU architecture is as follows : the bank of four registers **(R(O), R(l), R(2),** and R(3)) stores operands to be processed by the logic unit **A** ; an accumulator **ACC** stores temporarily the result output by the logic unit, before it is transferred to the register bank ; two buffers, **DIB** (data in buffer) and **DOB** (data out buffer), transfer data from the outside world to the register bank, and vice versa ; an instruction register **IR,** stores an instruction **(ins** parameter of the **Instr** action) for the duration of that instruction ; an address buffer **AE** B, issues addresses to the memory where from the CPU fetches data ; and finally a controller C promotes clock enable (Ce) and output enable (Oe) actions on all the CPU component instances (for example, the controller Acce action promotes the Ce action of the Acc register).

2.3 Simulations

A simulation is a partially ordered set of events resulting from the execution of actions. To express the ordering between events, a **timestamp** consisting of two integers is associated with

each event. The two numbers (*lower_bound* and *upper-bound)* define the time interval during which an event occurs. Timestamp t1 is smaller than timestamp t2 if t1.upper_bound is smaller than t2.lower_bound. The notation for an interval is **[lower-bound, upper-bound]**. If the lower bound is equal to the upper bound, the interval reduces to a single timepoint. An event with a smaller timestamp is said to have occurred *before* another event with a larger timestamp.

The ordering relationship expresses both *causality* (an in-action event causes a design entity to issue another event by executing an out-action), and *clock ordering.* In other words, the existence of an order between two events may indicate that one caused the other, or that one came before the other (or both). The absence of an ordering between two events indicates that they happened independently and concurrently in the simulation.

Notes : (1) Causality and clock order are assumed consistent. If an event causes another, its timestamp cannot be greater than the event it caused. (2) For simplicity of presentation, our simulation model assumes a global clock. Similar partially ordered models of concurrent computation can be given when each entity has a local clock and there is no global clock [2, 4].

Figure 4 shows parts of the simulations produced by executing the CPU at the instruction level and at the register transfer level. In the figure, the numbers to the left of each simulation represent increasing values of time on the global clock. Each event in the simulation is represented by (1) a label (**E** or **F** followed by a number) for future reference (Section 3) ; and **(2)** the actual event description, consisting of an action name and parameter values. To simplify the event descriptions in Figure 4, the timestamps of events with a single timestamp are not shown.

For example, event E1 occurs at time 6, and is an instance of the CPU action **Instr,** with its parameters bound to the values "C333" and "AFA0" ; event E9 covers an interval that goes from time 10 to time **40,** and is an occurrence of action **Load,** with its parameters bound to values "333", 1, and "AFA0". The ordering in the simulations can be determined by the timestamps. For example, event E3 occurred *before* event E5 (E3's timestamp is smaller than E5's timestamp) ; event E4 occurred *while* event E9 was occurring (E4's timestamp lower bound and upper bound both occur within the interval defined by E9's timestamp) ; and event F9 and F10 occurred *simultaneously* (identical timestamp).

These two simulations, although different, are related : they are simulations produced by the same design, for the same test data, but at different levels of the design hierarchy. The next section shows how the relationship between these simulations can be expressed using event patterns.

3 Event pattern mappings

The problem we now address is to provide constructs to express the decisions made during the refinement process — decisions that define relationships between the specifications at the various levels.

3.1 Defining refinements

A mapping consists of an action name and formal parameters followed by an event pattern — the **body** of the mapping. It maps a partially ordered set of events at one level of the design hierarchy into one event at a higher level of the design hierarchy.

A mapping body consists of **event templates,** that will be matched to actual events of a simulation, and relevant ordering relationships among them. Pattern expressions (described in

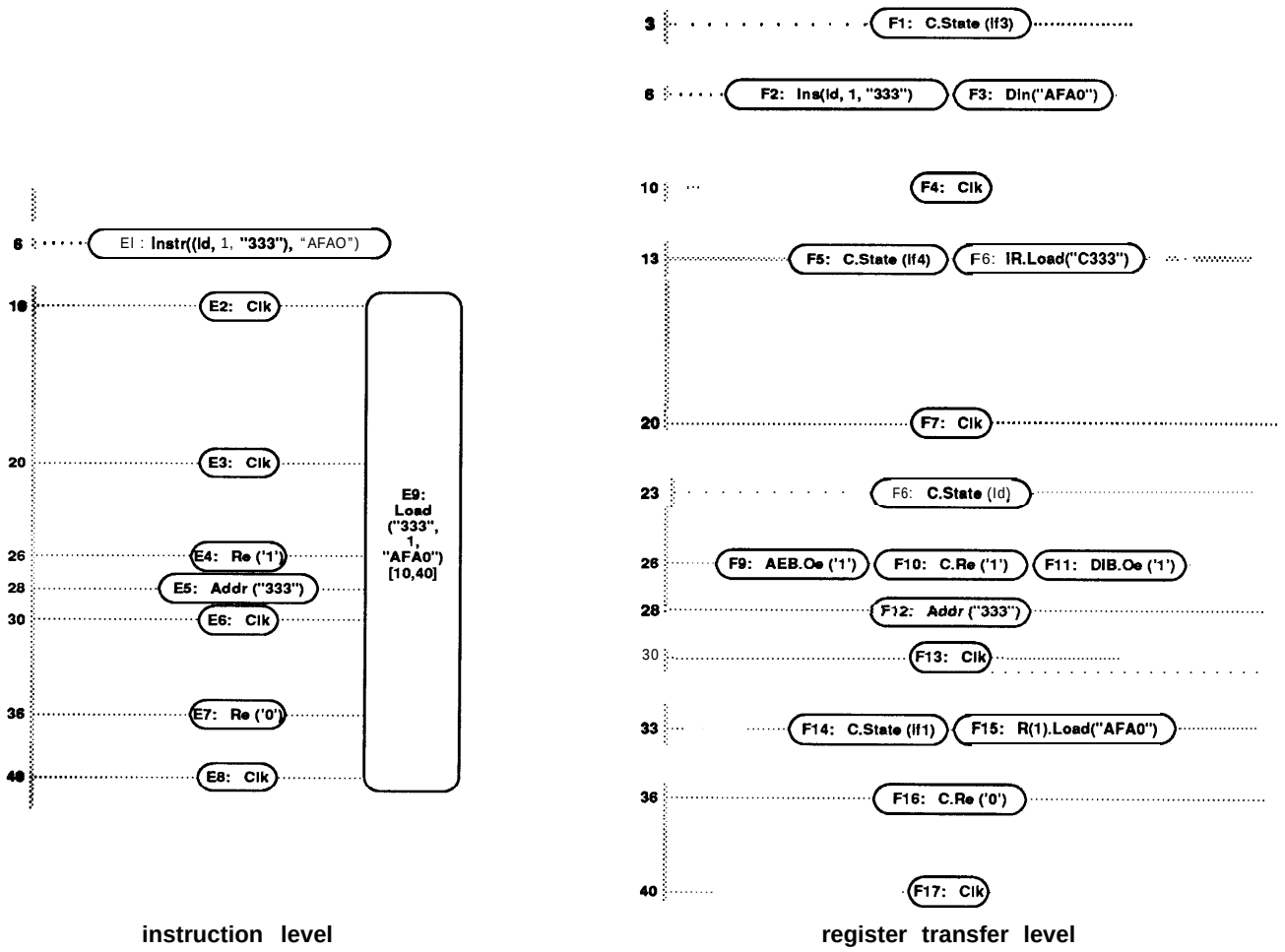

Figure 4: CPU simulations

Section 5) permit ordering and concurrency as part of a mapping definition. A syntactic form of the **mapping** construct is shown below :

```
mapping action name ( formal_parameters )
is
    pattern_body
end action name ;
```

In the CPU example, the **Load** action (instruction level) is refined to a pattern of actions of components at the register transfer level. Between three clock cycles defined by four clock events (intuitively, clock edges), the following events must happen : between the first and the second clock edge (first clock cycle), the controller C changes its state to the value **if4,** and the instruction register loads an instruction ; between the second and the third clock edge (second clock cycle), the controller C changes its state to the value **Id,** executes its read-enable **(Re)** action, the address **(AEB)** and the data input **(DIB)** buffers are enabled (internal action **En)** ; between the third and fourth clock edge (third clock cycle), the controller C changes its state to the value if1, and one of the registers in the CPU register bank loads the value output by the data input buffer. Note that any one of the registers in the CPU register bank may take part in a **Load** instruction. This informal description of the register transfer level pattern can be written as a pattern expression. We begin with a graphical representation.

Conceptually, an event pattern is a graph in which each node is an event template, and directed arcs between nodes indicate ordering relationships. The left part of Figure 5 shows the mapping body defining this refinement of the **Load** action using placeholders. The event templates consist of a label **(P** followed by a number), an action name (e.g. C.**state**) and its parameter values. The **Load** mapping event templates are :

- **Pl, P4,** P9, **P12** : **Clk** ; four clock edges.
- **P2** : C.State(if4) ; the controller C changing its state to value **if4.**
- **P3** : **I R.Load(?ins)** ; the instruction register IR loading an instruction **?ins.**
- P5 : **C.State(Id)** ; the controller C changing its state to value **Id.**
- **P6** : C.Re(**true**) ; the controller C activating its read enable output.
- **P7** : **DIB.En(?data)** ; the data in buffer **DIB** outputting value ?data.
- **P8** : **AEB.En(?addr)** ; the address buffer **AEB** outputting value **?addr.**
- P10 : **C.State(ifl)** ; the controller C changing its state to value if1.
- P11 : **R(?reg).Load(?data)** ; one of the CPU registers loading value ?data.

The ordering relationships between these event templates, as shown in Figure 5, are :

- **Pl** →P2 →P4 : the controller C moves to state **if4** during the first clock cycle **(Pl** before P2, and P2 before P4).
- **Pl** →P3 →P4 : the instruction register **IR** loads an instruction during the first clock cycle.
- P4 →P5 →P9 : the controller C moves to state **Id** during the second clock cycle.
- P4 →P6 →P9 : the controller C activates its **Re** output during the second clock cycle.

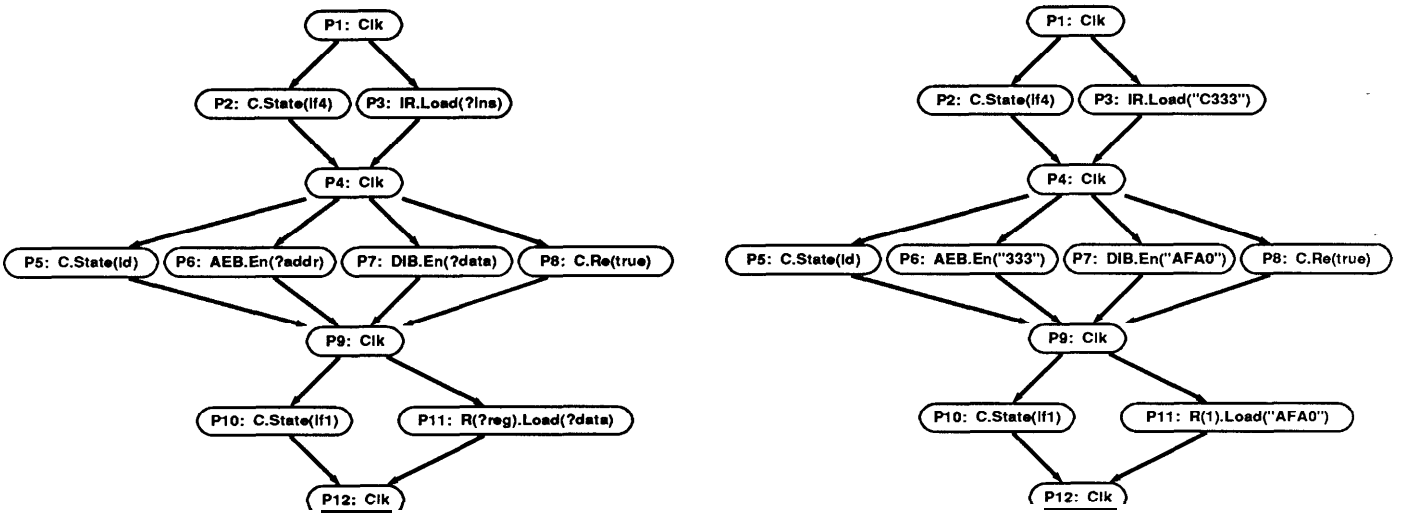

Figure 5: Load action mapping

- P4 →P7 →P9 : the data in buffer **DIB** is enabled and outputs value ?data during the second clock cycle.
- P4 →P8 →P9 : the address buffer **AEB** is enabled and outputs value **?addr** during the second clock cycle.
- P9 →P10 →P11 : the controller C moves to state **if1** during the third clock cycle.
- P9 →P10 →P11 : one of the CPU's registers loads value ?data during the third clock cycle.

The mapping does not specify ordering relationships between P2 and P3, for example. This means that P2 could occur before P3, P3 could occur before P2, or they could occur simultaneously.

The right hand part of Figure 5 shows an instance of the mapping for the CPU action **Load** with pattern variables **?ins, ?addr, ?reg** and ?data bound to the values "C333", "333", 1, and "AFA0" respectively.

Similarly, the **Instr** action (instruction level) is refined to a pattern of actions of components at the register transfer level : both actions **Din** and **Ins** (register transfer level) must occur, but in any order.

3.2 Mapping simulations between levels

Typically, a designer intends a high level action to be implemented by one or more patterns of lower level actions. The designer's intentions can be expressed by mappings, during the top-down development of a design. Mappings, used in this way, are formal assertions about the relationship between different design levels. They can be applied to implement various kinds of comparative analysis. One form of comparative analysis is to map a lower level simulation to a higher level ordering of events, called a *mapped simulation.* The mapped simulation is then checked for consistency with the higher level constraint specifications.

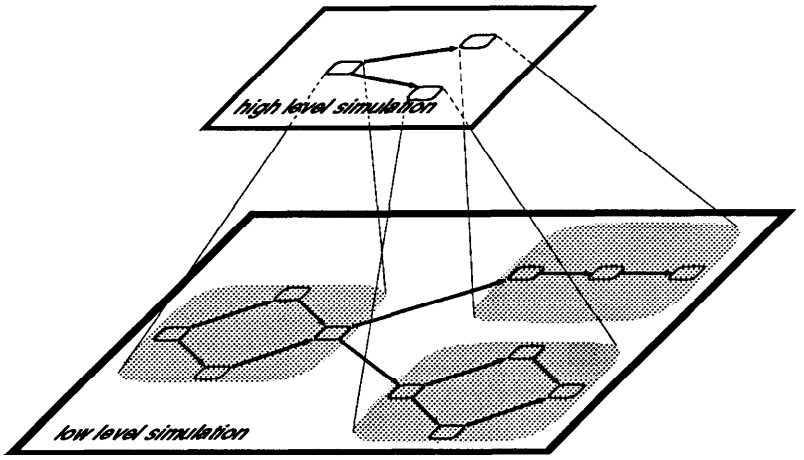

Figure 6: Use of mappings

Figure 6 gives a general idea of the use of mappings to relate simulation events at different hierarchy levels. The top part of the figure shows a high level simulation. The bottom part of the figure shows a simulation of the same entity specified at a lower level of abstraction. Simulation events are represented as circles, and the ordering relationships as arrows. The three high level events are defined as patterns of lower events by mappings. The lower level ordering induces an ordering among the higher level events. Whenever the **event** pat tern body of a mapping matches in the lower level simulation, the higher level action with parameter values (bound in the matching) is mapped to the higher level simulation. In Figure 6, the shaded areas represent suborderings of the lower level simulation that match the mapping body. The mapped high level events are shown, and their correspondence with the lower level simulation is represented by vertical rays.

The ordering of mapped higher level events is determined by the ordering of lower events that match the pattern. A full treatment of mapping–induced simulations is omitted here.

Very simple correspondences between higher and lower level actions, such as one–one equivalence, are obviously expressed by trivial mappings. However, even every–day examples of hierarchical designs cont ain correspondences that involve patterns such as sequences of lower level actions and nondeterministic choices between such sequences. Correspondences jumping over several levels of hierarchical refinement can be quite complex. Pattern mappings provide a convenient and powerful way to define design decisions made during the hierarchical refinement process.

Examples of correspondence between two specifications of the CPU design, are shown in Figure 7. The figure displays CPU simulations at the instruction and register transfer levels. The simulations are exactly the same as in Figure 4. The events are described only by their labels. It also depicts graphically the correspondences defined by the mappings for the instruction level actions **Instr** and **Load.** As explained in section 3.1, the instruction level event **Instr (EI)** corresponds to the occurrence of register transfer level actions **Ins** and **Din (F2** & F3) ; the & operator indicates that the order of events F2 and F3 is not important. In other words, **F2** can occur before F3, F3 can occur before F2, or their time stamp can be identical : all those situations will correspond to

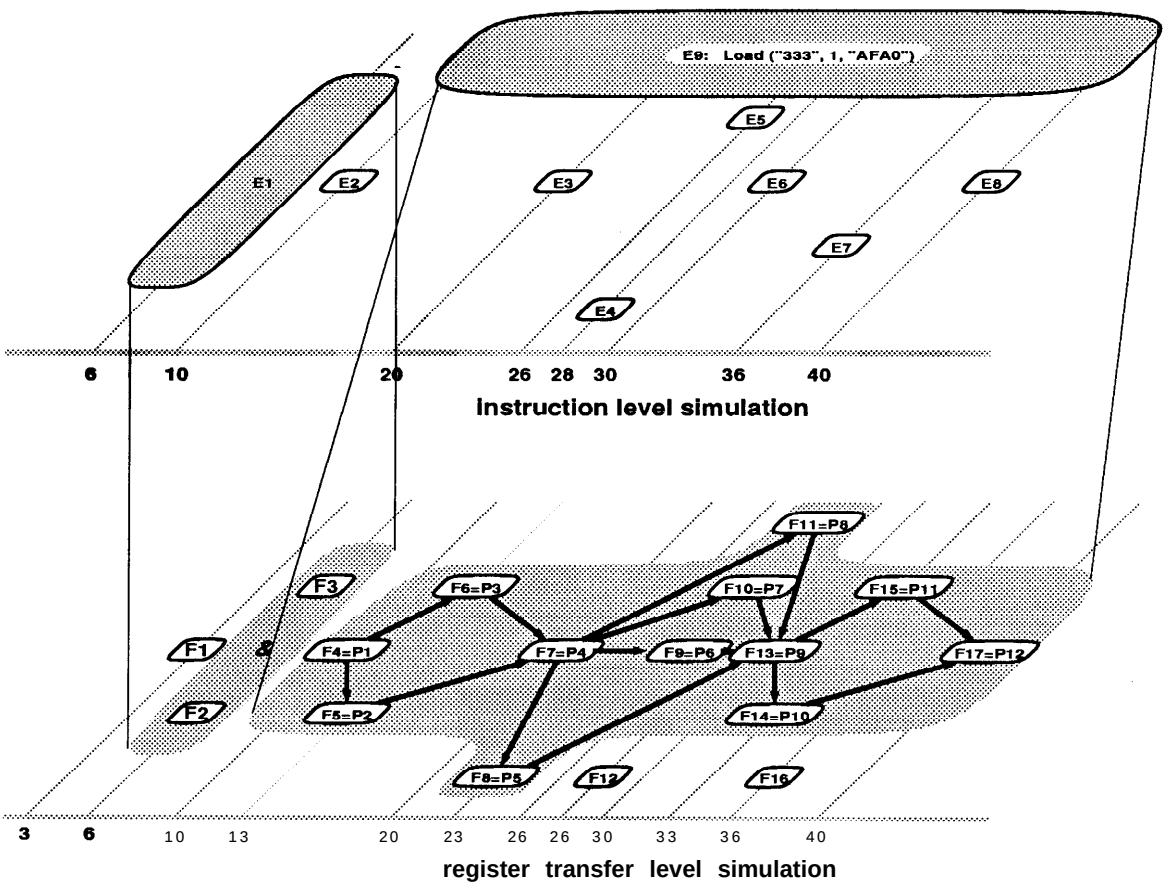

Figure 7: Correspondences between CPU simulations

the instruction level event E1.

The instruction level event E9 corresponds to the more complex pattern, described previously and consisting of events F4, F5, F6, F7, F8, F9, F10, F11, F13, F14, F15, and F17. In Figure 7, the events that make up the **Load** mapping instance are given both their name, and the name of the mapping event template they match. For example, F4=P1 means that event F4 is matched by the **Load** pattern template P1.

4 Comparative Validat ion

There are several ways to use event pattern mappings to implement comparative validation whereby simulations of different levels of a hierarchical design are compared for consistency. A very straightforward and easily implemented method is a post-mortem comparison. Assume we have two specifications and mappings between them that define the actions of one specification as patterns of actions of the other.

- execute the lower level specification and produce a simulation of low level events (*low level simulation)* ;
- use the mapping to translate the low level simulation into a partially ordered set of high level events *(mapped high level simulation)* ;

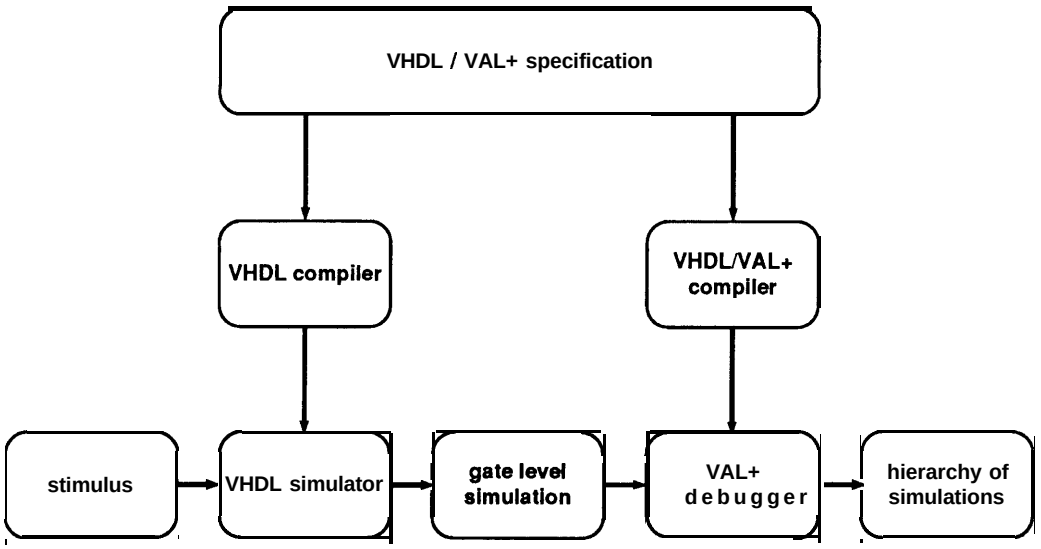

Figure 8: Debugger environment

- check that the mapped high level simulation satisfies the constraints of the high level specification.

More sophisticated uses of mappings are also possible, including on-the-fly translation of, and error detection in, very detailed large simulations.

To investigate the application of pattern mappings, a *debugger* has been implemented and used in debugging gate level simulations. Using event pattern mappings, the debugger extracts, from a VHDL gate level simulation, simulations at the register transfer level and the instruction level. The three simulations, at the gate, register transfer and instruction levels are called the *simulation hierarchy*. The designer specifies in VAL+ (an annotation language for VHDL) the actions each component can perform. Examples of such actions are for the register transfer level entity the action Oe (output enable), and for the instruction level entity CPU the action **Load.** The designer also specifies in VAL+ a mapping between each action and a pattern of events at a lower level of abstraction. For example, the register transfer level action Oe is specified as a pattern of gate level events, and the instruction level action **Load** is specified as a pattern of register transfer level events.

Figure 8 describes the debugger environment. The VHDL / VAL+ specification is compiled in two programs, a simulator and a debugger. The simulator, compiled from the VHDL gate level specification, takes as input a *stimulus* and produces a simulation at the gate level. The debugger, compiled from the VAL+ specification, takes as input the gate level simulation, and produces simulations at the register transfer and instruction levels.

Debugging the CPU design consists of (1) executing the VHDL gate level specification on a VHDL simulator, and (2) generating the register transfer and instruction level simulations using the simulation interpreter. The simulated CPU example had four levels of abstraction, gate level, elementary register transfer level (one–bit buffer, one–bit register, ...), register transfer level (multiple bit buffer, multiple bit register), and instruction level (the CPU itself). For a five instructions stimulus, the gate level simulation consists of 8073 events, the elementary register transfer simu-

lation consists of 334 events, the register transfer level simulation consists of 73 events, and the instruction level simulation consists of 5 events. These numbers show how well the event pattern mappings help structure the gate level simulation. To understand the problem of browsing through the gate level simulation, printing the gate level simulation, with as horizontal coordinate the VHDL signals and vertical coordinate the timepoints, would require a rather unmanageable page 4880 characters wide and 680 lines long.

The absence of one or more events, in the instruction level simulation, or the incorrect value of an event parameter is the indication of an error in the register transfer level simulation. The error can be traced to a missing or incorrect event in the register transfer simulation. The absence of an event in the register transfer level simulation can be traced either to wrong connections in the network of register transfer level components or errors in those components. This method allows for fast bug detection and pinpointing.

The debugger helped find five bugs in the CPU gate level specification (including errors in the logic of the clock enable circuitry of the registers, in the logic unit, and in connections between components). Four out of five bugs were found inspecting the top level (instruction level) simulation. The fifth bug was found inspecting the register transfer simulation, as a consequence of another bug at the instruction level. Those bugs could have been found, however very difficultly, by inspecting the gate level simulations, but the VAL+ simulation interpreter was of great help in pinpointing rapidly the bugs.

The debugger was also used in debugging two other designs : (1) a string matching chip, with specifications at the register transfer and chip levels ; and (2) a computer, consisting of two memory boards, a central processing unit, and a data transfer bus based on the VME protocol [7]. The register transfer level specification of the string matching chip is a systolic array that demonstrates the ability of the pattern language to handle pipelined behavior. The computer was specified at the board and system levels, and shows the benefit of using event pattern mappings in a asynchronous design (the VME bus protocol is asynchronous).

5 A language for event patterns

In this section a simple language for defining event patterns, and its semantics, are described informally. The main language features are (1) its simplicity (only four operators to express pattern of events), (2) its ability to define both mappings and constraints on the behavior, and (3) its ability to define context by means of guards.

5.1 The event pattern construct

Events occurring in a simulation are called *actual events* to distinguish them from event patterns.

Figure 9 shows an event pattern in which A, B, C, and D are event templates. The pattern will match any set of four actual events that are instances of A . . . D (as explained below) and whose ordering in the simulation matches the graph.

An event pattern has three parts : (1) a set of declarations of pattern variables and their types, (2) the set of event templates in the pattern (i.e., the nodes in the graph), together with guards to be checked when a possible matching event occurs in a simulation, and a name (or label) for each of the event templates, and (3) an ordering among the event templates of the pattern (i.e., the arcs of the graph).

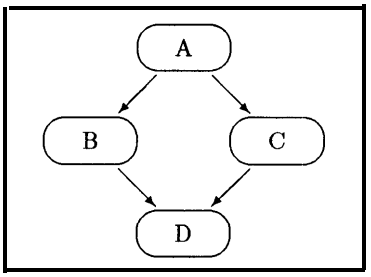

Figure 9: Event pattern

```
declare
  ?pattern_variable_1 : type_1 ;
  ?pattern_variable_2 : type_2 ;
  ...
nodes
  event_template_1 ;
  event_template_2 ;
  event_template_3 ;
  ...
pattern
  pattern-description ;
end ;
```

5.2 Pattern variables and Matching

Pattern variables are placeholders for values. Pattern variables have a name and a type, and are **initially** *unset* — i. e., have no value. Lexically, pattern variables identifiers always start with a ?, in order to make pattern descriptions more readable.

A pattern variable is replaced by a value during the process of *matching*. The pattern variable is then said to be *set*. A pattern variable cannot be assigned to, and cannot be subjected to the standard value–changing operations of its type. After it has been set, its value **can** be read or selected using the standard read–only operations of its type.

A pattern variable may occur in more than one event template of an event pattern. When it is replaced by a value during matching, all of its occurrences are replaced by the same value.

5.3 Pattern nodes

The generic form of a pattern node is :

```
name : event_template
    where guard ;
```

Each event template declaration (or node) in a pattern describes a set of possible events in a **simulation.** An event **template declaration has three parts, some of which are optional.**

1. **name** : each event template is given a unique name. Names are used to define where a template occurs in the graph (ordering) of a pattern. An event template may occur more than once in a pattern graph.

2. **event template** : an event template consists of the name of an action together with expressions for values of the action parameters. These parameter expressions may contain pattern variables as well as ordinary program variables. An expression may be evaluated only after all of its pattern variables have been set. Parameter expressions in event templates are optional.

3. **guard** : A guard is a boolean expression that restricts the number of actual events that can be matched by an event template by requiring that the boolean condition is true in order for a match to be successful. Guards are optional.

5.4 Matching event templates

An event template **matches** an actual event if (1) the action names in the template and the event are the same, (2) there is a replacement of pattern variables by values such that the template and actual event are identical when all expressions in the template are evaluated, and (3) any (optional) guard evaluates to true.

Examples of event templates and matches. Assuming ?data, **?ins** and **?addr** are pattern variables the following are examples of event templates :

1. **LD : Load.** This template matches any **Load** actual event in a simulation irrespective of its parameter values.

2. **LD : Load ("CCC", *1*, "AFA0").** This template does not contain any pattern variables. It matches a **Load** actual event if the address parameter is "CCC", the register parameter is 1, and the word parameter has the string value "AFA0".

3. **LD : Load ("CCC", 1, ?data).** This template contains the pattern variable ?data. If the ?data pattern variable is set to **V**, the template will only match simulation events Load("CCC", 1, **V**). If the ?data pattern variable is unset, the template matches all **Load** events in a simulation that have their first and second parameters set to "CCC" and 1 respectively, and sets the ?data pattern variable to the value of the **data** parameter of the actual **Load** event.

4. **LD : Load (?addr, 1, ?data) where ?addr = ?ins.address.** This template contains three pattern variables, **?addr**, ?data and **?ins.** If the **?ins** pattern variable is set to "3CCC", and the actual event **Load** (" **CCC**", *1* , "AFA0") occurs, the event template will match the actual event. However, if the **?ins** pattern variable is set to "3CCC" before the **Load** actual event occurs, and the event **Load** ("334", 1, "AFA0") occurs, the event template will not match the actual event.

5.5 Patterns

A pattern declaration consists of a sequence of expressions built up from the event templates and the **connectives** ->(followed by), &(both of), |(either of) and **when . . . then . . . before...**

1. **A -> B** matches an actual **A** and the first actual **B** that follows it in the ordering of a simulation.

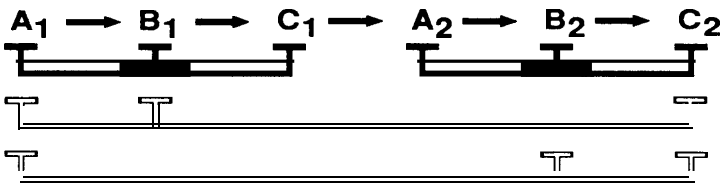

Figure 10: Matching semantics

2. A & B matches an actual A and an actual B provided they are not ordered in the simulation (i.e. occur in parallel).

3. A | B matches an actual A or an actual B.

4. **when A then B before** C matches A followed by a B, provided no C occurred before B. This construct **is** extremely useful to specify both constraints and mappings, as illustrated in section 5.7 and 5.8.

For example, the pattern corresponding to graph (9) appears in the following example. This pattern will match any part of a simulation where an **A** occurs first, followed by both **B** and C, which must themselves occur in parallel; these actual events B and C must then be followed by the same D.

```
...
nodes
  A : . . . ;
  B : . . . ;
  C : . . . ;
  D : . . . ;
pattern
  A -> (B & C) -> D;
end ;
```

5.6 Matching event patterns

An event pattern matches a partially ordered set of actual events if :

1. there is a replacement of pattern variables for values under which each of its template instances matches one of the actual events. Each template is thus associated with an actual event. If a pattern variable occurs several times in a pattern all its occurrences must be bound to the same value.
2. the ordering of templates is the same as the ordering of associated actual events.
3. the pattern matches minimal intervals in the simulation *(stingy matching).*

The semantics of pattern matching adopted here is called stingy matching. This means, informally, that a pattern, say **A -> B -> C**, matches a set of actual events A_1, B_1 and C_1 in a simulation, if (1) the actual events have the same ordering as the templates **A, B** and **C** in the pattern, and (2) the actual events occur in a time interval [T1,T2] (T1 being the greatest lower bound of the actual events timestamp's lowerbounds, and **T2** the least upper bound), and no other match of the pattern occurs in a lesser time interval ([U 1, U2], with **U 1 > T1** and **U2 < T2.** If several matches occur in the same time interval, they are all reported as independent occurrences of the same pattern.

Figure 10 shows all occurrences of the pattern **A -> B -> C** in a simulation that has two instances of event templates **A, B** and **C**. The pattern occurrences shown in solid lines satisfy the stingy semantics. In the outlined pattern occurrences, although the actual events have the same ordering as the pattern templates, the smaller interval requirement of the stingy semantics is not satisfied. This semantics is admittedly restrictive, and other pattern matching semantics could be adopted.

5.7 A CPU instruction level constraint

Constraints on simulations are also expressed using patterns. This provides a facility for specifying, in an entity interface, properties that must be satisfied by any architecture for that entity. The CPU constraint discussed in section 2.1, may be rewritten using template names as follows :

```
declare
    ?ins : InstrT ;
    ?din : WordT ;
nodes
    IL : Instr (?ins, ?din) where ?ins.opcode = ld ;
    C1, C2, C3, C4 : clk ;
    LD : Load (?ins.address, ?ins.register, ?din) ;
pattern
    when IL then
        LD
    before (C1 -> C2 -> C3 -> C4) ;
end ;
```

This pattern indicates that whenever an **Instr** action is executed with its **?ins** parameter corresponding to a load instruction **(where ?ins.opcode = ld)**, then a **Load** action must occur within the next three clock cycles. The **Load** action parameters must be respectively bound to (1) the address field of the **?ins** pattern variable ; (2) the register field of the **?ins** pattern variable ; and (3) the **?din** parameter. The formal and informal description of the constraint match closely.

5.8 The mapping for the action Load

The mapping for action **Load** has been informally and graphically described in section 3.1. Here is the text of this mapping in the pattern language.

```
mapping load ( ?addr : AddrT ;
```

```
                ?reg  : Int2 ;
                ?data : WordT ) is

  ?ins : WordT ;
nodes
  P1 : Clk ;
  P2 : C.State (if4) ;
  P3 : IR.Load (?ins) where instF (?ins) = ld ;
  P4 : Clk ;
  P5 : C.State (ld) ;
  P6 : AEB.Oe (?addr) where ?addr = ?ins (2..4) ;
  P7 : DIB.Oe (?data) ;
  P8 : C.Re ('1');
  P9 : Clk ;
  P10 : C.State (ifl) ;
  P11 : R(?reg) .Load (?data) ;
  P12 : Clk ;
pattern
  P1 -> (P2 & P3) -> P4
     -> (P5 & P6 & P7 & P8)
     -> P9 -> (P10 & P11) -> P12 ;
end load ;
```

The pattern variables **?addr, ?reg, ?data** are the parameters, of the mapping, and **?ins** is a local pattern variable. The templates and graph are shown in Figure 5. Templates P3 and P6 have guards, indicating that, for template P3, the opcode field of the instruction (**instF (?ins)**) must have the value **ld**, and that, for template P6, the address must correspond to the last three digits of the instruction word (**?addr = ?ins**(2..4)).

The pattern defines the ordering relationships between the templates, that are shown graphically in Figure 5, in a condensed form. For example, the part of the pattern P1 -> (P2 & P3) -> P4 is equivalent to the four ordering relationships P1 -> P2, P1 -> P3, P2 -> **P4**, and P3 -> **P4.**

6 Conclusions and Continuations

We conclude that event pattern mappings, with at least the expressive power of the constructs reported here, are essential to permit formal definition of the refinement relationships in hierarchical designs of concurrent systems. We believe they are a mandatory foundation for automating consistency checking of design levels of concurrent systems, design levels that may be several refinement steps apart.

We have not investigated the application of pattern mappings to formal proof of consistency of hierarchical designs, or to automated synthesis of a lower level specification from a higher level one.

At present we are developing and refining the pattern language and mappings outlined in this paper. More sophisticated pattern definition constructs are useful in the kinds of examples we are currently experimenting with, for example, constructs for defining conditional and repetitive

patterns. We are also implementing the application of mappings to post-mortem comparative validation of hierarchical designs in VAL/VHDL.

As stated previously, the full treatment of partial orderings induced by mappings is being undertaken on the TSL-1.5 project. The application of pattern mappings as an abstraction mechanism for computational modeling of distributed local-time systems is also being studied in the TSL–1.5 project.

References

[1] *Reference Manual for the Ada Programming Language.* U. S. Department of Defense, U. S. Government Printing Office, ANSI/MIL-STD-1815A edition, January 1983.

[2] C.J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Australian Computer Science Communications,* 10(1):56–66, February 1988.

[3] *HSL-FX Reference manual.* Nippon Telegraph Telephone, 1988.

[4] F. Mattern. *Determining the Partial Order of Distributed Events.* Technical Report SFB124–28/87, University of Kaiserslautern, Federal Republic of Germany, 1987.

[5] *VHDL Language Reference Manual.* October 1986. IEEE Preliminary Version 7.2.

[6] *IEEE Standard VHDL Language Reference Manual.* IEEE, Inc., 345 East 47th Street, New York, NY, 10017, March 1987. IEEE Standard 1076-1987.

[7] *VME bus specification manual Rev c.* Signetics, Feb. 1985.

A The example : the 16-bit CPU

The previous sections explained the principles and use of event patterns and mappings, referring to the CPU example. This section gives a more complete version of the example, the CPU utility package, the CPU entity interface, and the CPU RTL component interfaces.

A.1 Data types and operations

All types and operations relative to the CPU are gathered in a package.

The **bit** type is an enumeration type with two literals, '1' and '0'. The **Int2** type is an integer whose range is **0** to 3. Types **AddrT** and **WordT** are arrays of 3 and 4 hexadecimal characters respectively, representing 12-bit and 16-bit vectors. The instruction type OpCodeT is an enumeration with four literals, **ld** (load), **st** (store), **ex** (execute), and **id** (idle). The instruction type InstrT is a variant record : when the **opcode** is **ld** or **st,** the record has two fields, a **register** and an **address ;** when the **opcode** is **ex**, the record has three fields, two registers **(register1** and **register2**), and an **operation** ; when the **opcode** is **id,** the record is empty. The type RegistersT is an array of 4 16-bit vectors **(WordT).**

```
package cpu_p is
  type bit is ('0','1') ;
  type Int2 is integer range 0..3 ;
  type AddrT is string (1..3) ;
  type WordT is string (1..4) ;
  type OpcodeT is (Id, st, ex, id) ;
  type AluOpT is (andL, orL, notL, xorL) ;
  function AluF ( a, b : WordT ;
                  op : AluOpT ) return WordT ;
  type InstrT ( opcode : opcodeT ) is record
      case opcode is
      when ld I st => register : Int2 ;
                      address : AddrT ;
      when ex      => register1 : Int2 ;
                      register2 : Int2 ;
                      operation : AluOpT ;
      when id      => null ;
      end case ;
  end record ;
  type RegistersT is
    array (Int2) of WordT ;
end cpu_p ;
```

A.2 State model

The behavior of the CPU depends on its past behavior. One way to take into account the CPU past behavior is to store the relevant parts of that behavior in an *internal state.* In order to write constraints on each of the CPU internal actions, it is necessary to declare the CPU internal state.

A special construct is required for that purpose, the *state model declaration.* In the CPU, the state model is of type **Registersi,** that is, an array of four 16-bit vectors. Change in the state have the same syntax as events. For example, assuming the state model declaration :

```
state model is bit ;
```

a change of the state to value '1' is written `State ('1')`. In the CPU, a change of state whereby bit vector 2 takes the value **"AFAO"** will be written `State(2) ("AFA0")`.

A.3 Instruction level specification

At the top level, the CPU is represented by its interface and its instruction set. The CPU can be acted upon in two ways : the sequencing of its operation is controlled by a clock (**clk**), and it can be fed an instruction. It can perform 4 out-actions. **Re** (read-enable) ; **We** (write-enable) ; **Dout** (data output) ; and **Addr** (address). The CPU has also four internal actions, corresponding to each of the four instructions the CPU can perform : **Load, Store, Exec** and **Idle.**

```
with cpu_p ;

design CPU is
    -- ports
  in action Clk ;
  in action Instr ( instr : WordT ;
                    din : WordT ) ;
  out action Re ( b : bit ) ;
  out action We ( b : bit ) ;
  out action Dout ( b : WordT ) ;
  out action Addr ( a : AddrT ) ;
      action Load ( addr : AddrT ;
                    reg : Int2 ;
                    data : WordT ) ;
      action Store ( addr : AddrT ;
                     reg : Int2 ;
                     data : WordT ) ;
      action Exec ( regl : Int2 ;
                    reg2 : Int2 ;
                    op : AluOpT ;
                    opl, op2 : WordT ;
                    res : WordT ) ;
      action Idle ;

  state model is RegistersT ;

  declare
    ?ins : InstrT ;
    ?din : WordT ;
  pattern
```

```
  when Instr (?ins, ?din) where ?ins.opcode = ld then
      Load ( ?ins.address, ?ins.register, ?din) &
      State (?ins.register) (?din)
  before Clk -> Clk -> Clk -> Clk ;
end ;
declare
  ?ins :  InstrT ;
  ?din : WordT ;
pattern
  when Instr (?ins,  ?din) where ?ins.opcode = st then
      Store ( ?ins.address, ?ins.register,  State (?ins.register))
  before Clk -> Clk -> Clk -> Clk ;
end ;
declare
  ?ins :  InstrT ;
  ?din : WordT ;
pattern
  when Instr (?ins,  ?din) where ?ins.opcode = ex then
      Exec   ( ?ins.register1, ?ins.register2,
               ?ins.operation,
               State (?ins.register1), State (?ins.register2),
               AluF ( State (?ins.register1),
                      State (?ins.register2),
                      ?ins.operation
                    )
               ) &
      State (?ins.register2) ( AluF ( State (?ins.register1),
                                      State (?ins.register2),
                                      ?ins.operation


  before Clk -> Clk -> Clk -> Clk -> Clk ;
end ;
declare
  ?ins :  InstrT ;
  ?din : WordT ;
pattern
  when Instr (?ins,  ?din) where ?ins.opcode = id then
      Idle ;
  before Clk ;
end ;
declare
  ?ins :  InstrT ;
  ?din : WordT ;
pattern
  when    Instr (?ins,  ?din) where ?ins.opcode = ld
       -> Clk -> Clk then
      Re ('1') &
      Addr (?ins.addr) ;
  before Clk ;
```

```
  end ;
  declare
    ?ins : InstrT ;
    ?din : WordT ;
  pattern
    when    Instr (?ins, ?din) where ?ins.opcode = st
         -> Clk -> Clk then
        We ('1') &
        Dout (State (?ins.register1)) &
        Addr (?ins.addr) ;
    before Clk ;
  end ;

end CPU ;
```

Following the state model declarations, are constraints that define the behavior of the CPU. For example, whenever an event occurs on the CPU in-action **Instr,** with the **opcode** field of its **ins** parameter bound to the value **ld,** an internal action **Load** must occur within the next three clock cycles, with its three parameters bound to the address field of the **?ins** placeholder, the register field of the **?ins** placeholder, and the **?din** placeholder respectively. The last constraint in the CPU design states that, whenever an **event** on the in-action **Instr** occurs, with the **opcode** field of its ?ins parameter bound to the value **st,** followed by an event on the **Clk** in-action, followed again by a **Clk** in-action, then an event on three CPU out-actions must occur before the next event on the CPU **Clk** in-action : one on the action **We,** with its parameter bound to '1', one on the action **Dout,** with its parameter bound to the value of the ?ins.registerlth bit-vector of the State, and one on the action **Addr,** with its parameter bound to the **addr** field of the **?ins** pattern variable.

A.4 Register transfer level specification

Once the design specification is completed, the design is broken up in several components reflecting the architecture of the CPU. Each component is specified using the same technique as the high level design. The architecture also describes the flow of communication between the components. Textual and graphical descriptions of the CPU register transfer level specification appear in section 2.2.

A.4.1 Component interfaces

The code for the architecture assumes the following components. The informal specifications given in section 2.2, are completed by a state model and some constraints.

• **controller** : The controller is a state machine. It has two in-actions, **Input** and **Clk** ; twelve out-actions, **Re** (read enable), **We** (write enable), **Roe1** (first register output enable), **Roe2** (second register output enable), **Rce** (register clock enable), **Irce** (instruction register clock enable), **Irrst** (instruction register reset), **Acce** (accumulator clock enable), **Acoe** (Accumulator output enable), **Diboe** (data in buffer clock enable), **Doboe** (data out buffer output enable), and **Aeboe** (address buffer output enable). The controller state model is an enumeration type with eight literals : four instruction-fetch literals **(ifl, if2, if3,** and **if4**), two execute (**ex1** and **ex2**), one load literal **(ld),** and one store literal (**st**). The controller sequences and promotes in-actions in other CPU components. We describe on of the constraints on the CPU behavior. The constraint expresses that, whenever

there is an event on the **Clk** action, with the **State** equal to **if2**, then the **State** must change to **if3** and there must be an event on out-action **Irrst**, with its parameter bound to '1', before the next event on the action **Clk**.

```
design CONTROLLER is
  in action Input ( opcd : OpcodeT ;
                    r1 : Int2 ;
                    r2 : Int2 ) ;
  in action Clk ;
  out action Re ( b : bit ) ;
  out action We ( b : bit ) ;
  out action Roe1 ( r : Int2 ;
                    b : bit ) ;
  out action Roe2 ( r : Int2 ;
                    b : bit ) ;
  out action Rce ( r : Int2 ;
                   b : bit ) ;
  out action Irce ( b : bit ) ;
  out action Irrst ( b : bit ) ;
  out action Acce ( b : bit ) ;
  out action Acoe ( b : bit ) ;
  out action Diboe ( b : bit ) ;
  out action Doboe ( b : bit ) ;
  out action Aeboe ( b : bit ) ;
  state model is (ifl, if2, if3, if4, exl, ex2, ld, st) ;
  pattern
      when Clk where State = if2 then
          Irrst ('1') ;
          State (if3) ;
      before Clk ;
  end ;
end CONTROLLER ;
```

- **registers.** The register component has four in-actions, **Din** (input data), **Ce** (clock enable), **Oe** (output enable), and **Clk** (clock); one out-action, **Dout** (output data); and two internal actions, **Load** and **En**. The state model is of type **WordT**. If the clock is enabled (i.e. the value of the parameter of the last action **Ce** was true), the register loads the value of its data input on each occurrence of the **Clk** action. When the register output is enabled (i.e. the value of the parameter of the last action **Oe** was '1'), the register outputs its internal state (value of the parameter of the last occurrence of the internal action **Load)** through its out-action **Dout**, and issues an **En** action with its parameter bound to the value of register state.

```
design REGISTER is
  in action Oe ( b : bit ) ;
  in action Ce ( b : bit ) ;
  in action Clk ;
  in action Din ( value : WordT ) ;
```

```
  out action Dout ( value : WordT ) ;
      action En ( s : WordT ) ;
      action Load ( s : WordT ) ;
  state model is WordT ;
  declare
      ?data : WordT ;
  pattern
      when (Ce ('1')& Din (?data)) -> Clk then
          Load (?data) &
          State (?data) ;
      before Clk ;
  end ;
  pattern
      when Oe ('1') then
          En (State) &
          Dout (State)
      before Clk ;
  end ;
end REGISTER ;
```

- **registers with two outputs.** The two-output register component has five in-actions, **Din** (input data), Ce (clock enable), Oe1 (first output enable), Oe2 (second output enable), and **Clk** (clock) ; two out-actions, Dout1 (output data), and Dout2 (output data) ; and three internal actions, **Load**, En1 and **En2.** The behavior of the two-output register is similar to the behavior of the register, with input Oe1 and Oe2 controlling Dout1 and **Dout2** respectively.

```
design TWO-OUTPUT-REGISTER is
  in action Oel ( b : bit ) ;
  in action Oe2 ( b : bit ) ;
  in action Ce ( b : bit ) ;
  in action Clk ;
  in action Din ( value : WordT ) ;
  out action Doutl ( value : WordT ) ;
  out action Dout2 ( value : WordT ) ;
      action Load ( s : WordT ) ;
      action En1 ( w : WordT ) ;
      action En2 ( w : WordT ) ;
  state model is WordT ;
  declare
      ?data : WordT ;
  pattern
      when (Ce ('1')& Din (?data)) -> Clk then
          Load (?data) &
          State (?data) ;
      before Clk ;
  end ;
  pattern
      when Oel ('1') then
          En1 (State) &
```

```
        Outpl (State)
      before Clk ;
  pattern
      when Oe2 ('1') then
        En2 (State) &
        Outp2 (State)
      before Clk ;
  end ;
end TWO_OUTPUT_REGISTER ;
```

- **logic unit.** The logic unit component has three in-actions, the two operands **Opl** and Op2, and the operation selection **Op** ; and one out-action, **Dout.** The logic unit does not have a state model. The logic unit can perform one of four operations **(and, or, not, xor)** on the two operands, depending on the operation selection input.

```
design LOGIC-UNIT is
  in action Opl ( x : WordT ) ;
  in action Op2 ( x : WordT ) ;
  in action Op ( o : AluOpT ) ;
  out action Dout ( x : WordT ) ;
  declare
      ?data1, ?data2 : WordT ;
      ?operation : AluOpT ;
  pattern
      when Opl (?data1) & Op2 (?data) & Op (?operation) then
        Dout (AluF (?data1, ?data2, ?operation)
      end when ;
  end ;
end LOGIC UNIT ;
```

- **buffers.** The buffer component has two in-actions, Oe (output enable) and **Din** (data in) ; one out-action **Dout** (data out) ; and one internal action **En.** The buffer does not have a state model. When the buffer output is enabled (i. e. the value of the parameter of the last action [illegible] was true), the buffer outputs its input value (i. e. the value of the parameter of the last occurrence of the in-action **Din).**

```
design BUFFER16 is
  in action Oe ( b : bit ) ;
  in action Din ( w : WordT ) ;
  out action Dout ( w : WordT ) ;
      action En ( w : WordT ) ;
  declare
      ?data : WordT ;
  pattern
      when Din (?data) -> Oe ('1') then
        Dout (?data) &
        En (?data)
```

```
        end when ;
    end ;
end BUFFER16 ;
```

```
design BUFFER12 is
    in action Oe ( b : bit ) ;
    in action Inp ( a : AddrT ) ;
    out action Outp ( a : AddrT ) ;
        action En ( a : AddrT ) ;
    declare
        ?data : WordT ;
    pattern
        when Din (?data) -> Oe ('1') then
            Dout (?data) &
            En (?data)
        end when ;
    end ;
end BUFFER12 ;
```