

# Transaction-Level Specification of VHDL Design Models

David A. Fura  
Levetate Design Systems  
P.O. Box 168, 4756 U. Village Pl. N.E.  
Seattle, Washington 98105 USA  
levetate@levetate.seanet.com

Arun K. Somani  
University of Washington  
Electrical Engineering and Comp. Sci. & Engineering  
Seattle, Washington 98195 USA  
somani@ee.washington.edu

**Abstract.** This paper addresses the modeling of hardware systems at high levels of abstraction. To support VHDL-based abstract specifications, a new modeling language is introduced and demonstrated that supports defining the abstraction linking specification models to designs. This language, called VIL (for VHDL Interface Language), provides a capability missing from existing hardware simulation environments. The manner in which this language supports greater verification rigor and automation is also described.

## 1 Introduction

Considerable recent attention has focused on the ‘top-down’ hardware design paradigm. By encouraging designers to construct and analyze system models at high levels of abstraction, this approach offers the potential for reduced hardware development costs and, at the same time, higher-quality designs. However, it is also understood that improvements in commercial design tools are needed to better support top-down design methods.

In this paper we address a fundamental weakness in existing methods for top-down design, that of inadequate support for developing abstract specifications. The specific purpose of this paper is threefold:

- (a) To explain how the modeling requirements of abstract hardware specifications are not being entirely met by commercially-available languages and tools.
- (b) To introduce a new hardware specification methodology and language support to address weaknesses in existing methods.
- (c) To point out how our language support can be exploited to increase the rigor and automation in hardware design environments.

The principal contribution of this paper is the demonstration of a new abstraction specification language called VIL (for VHDL Interface Language). To our

knowledge, VIL is the first formal language developed to represent system *abstraction* in simulation environments. We explain how VIL-like languages can support new classes of commercial design tools offering enhanced rigor and automation compared to existing methods.

In Section 2 of this paper we describe what the terms ‘abstraction’ and ‘abstract specification’ mean in the context of this paper. We describe a number of different forms of hardware abstraction as we distinguish our work from related prior work. We introduce a *transaction* level of abstraction as our top-most specification level.

In Section 3 we introduce a new approach to specify abstraction in simulation environments, using the abstraction language VIL. We demonstrate VIL support for modeling data abstraction and temporal abstraction. VIL support for specification re-use is also demonstrated.

In Section 4 we explain how VIL provides the foundation for new hardware design tools offering greater rigor and automation over existing methods. We overview a novel VHDL test bench generation system under development at Levetate Design Systems, and we point out future applications in formal verification and synthesis as well.

In Section 5 we close with a concluding discussion.

All related prior work is described in the applicable section below.

## 2 Abstract Hardware Modeling Overview

One of the widely-cited strengths of VHDL is precisely its support for modeling systems at high levels of abstraction. For example, the arrays, records, and enumerated types of this language provide an excellent way to encapsulate complex data within a small number of abstract signals.

Unfortunately, existing methods to incorporate VHDL specifications into simulation environments are generally deficient in one important area. For abstract specifications, these methods completely neglect the mappings that link the signals of the specification to those of the underlying design. This omission is important, as it can be

shown to limit the rigor and automation achievable in existing simulation environments.

The extent of this problem is dependent upon the nature of the relationships between the signals of the specification and those of the design. In other words, it depends on what types of ‘abstraction’ exist between these signals. To be clear about what we mean by this, we next describe four different forms of abstraction that are used to construct simple models to serve as specifications for more-complex design models. Following this, we introduce a new approach for constructing abstract specification models for designs that are expressed at the register transfer level (or RTL).

## 2.1 Abstraction

Of the four different types of abstraction described next, we borrow the terminology for the first three from [10]. The fourth form recognizes the recent movement towards graphical specification methods.

**Structural abstraction.** All approaches to abstraction serve to reduce the visible ‘complexity’ in some way. Structural complexity can be measured by the number of components or signals contained in a representation. Structural abstraction is a process of replacing a multi-component model by one with fewer components. Often, the new model contains only a single component, whose behavior is represented by a sequence of behavioral expressions.

Structural abstraction is one of the most common forms of abstraction used by hardware design engineers. In fact, it is structural abstraction that distinguishes the register transfer level from the underlying gate level.

**Data abstraction.** Data complexity is somewhat harder to characterize than structural complexity. One way employs the number of data values contained in the type used for the data. For example, the infinite number of values available for real-valued voltages makes these values more complex than those of the 9-valued VHDL type `std_logic`. Likewise, both of these types are more complex than 2-valued types such as the VHDL types `bit` or `boolean`. This characterization is consistent with the general view that booleans are indeed an abstraction of analog voltages.

**Temporal abstraction.** A good way to measure temporal complexity relies on the number of time steps used for a given segment of real time. A representation requiring more time steps is considered to be more complex than one requiring fewer. Many examples of temporal abstraction exist. For example, synchronous models, where time increments on the rising edge of a system clock, are temporal abstractions of models

where time is measured in fine-grained units of real time such as picoseconds.

**Visual abstraction.** Visual complexity is a measure of the mismatch between the representation of hardware systems and our natural ability to understand such models by looking at them. A good example of visual abstraction is the use of graphical representations in lieu of textual descriptions.

Examples of visual abstraction abound in the recent trend towards graphical specification methods. A number of commercially-available tools are available today that synthesize VHDL or Verilog state machine models directly from such descriptions.

**Discussion.** Today, when a general-purpose commercial design tool is claimed to operate on ‘specifications’ for RTL design models, this means that it works on graphical versions of the underlying simulation models. The type of abstraction associated with such specifications is clearly visual abstraction, but can include structural abstraction as well.

As pointed out above, a number of commercially-available tools exist today that can synthesize an RTL state machine model from a specification exhibiting visual abstraction. However, except for certain specialized applications, existing synthesis tools cannot handle specifications with data and temporal abstraction. For these problems, the design engineer must generally build the design and verify it through simulation.

The state-of-the-art today for general-purpose applications has the designer generate graphical versions of an RTL design model and an RTL test bench model to exercise the design. Significantly, the user of existing graphics-based synthesis tools is still required to define the system behavior for every RTL signal at every step of the RTL system clock. This is for both the design model and the test bench model. The use of graphics alone does not eliminate a difficult design task for either of these models, which still exhibit high data and temporal complexity.

## 2.2 Transaction-Level Modeling

In order to address the complexity of large hardware systems, it is clear that hardware specification methods must support levels of abstraction well above RTL. Furthermore, specifications that are only visual abstractions of RTL models are not sufficient, by themselves, for large systems. The large state spaces of such models would make them too incomprehensible and slow to execute for use as specifications.

A major problem with existing simulation environments is that even if an abstract specification model is built, there is no ready way to integrate it with the RTL design model.

To accomplish this today, the designer must construct a sophisticated test bench model at the data and temporal levels of the RTL design. Note that the design of such test benches relies on an informal notion of the required mappings between the signals of the specification and those of the design. Constructing simulation test benches is extremely hard today (e.g., see [4]).

To support efficient and effective simulation-based hardware design, we have developed a multi-level modeling approach with a number of important characteristics:

- (a) The top-most *specification level* is state machine-based, suitable for executable specifications.
- (b) The specification level is formally linked to the underlying design level through a new intermediate *abstraction level*. This level can be viewed as the specification for sophisticated test bench models, an observation that suggests new ideas for design tool development (see Section 4).
- (c) The *abstraction language VIL* that is used for the abstraction level has expressive power sufficient to handle complex subsystem interface protocols.

To our knowledge, this combination of properties is unique to the modeling paradigm that we describe next.

Figure 1 shows our modeling approach applied to an RTL design for an embedded hardware subsystem. The particular example under consideration here focuses on the abstract behavior associated with bus transactions, a class of behavior notoriously hard to verify by simulation. The figure depicts a 3-level hierarchy consisting of an RTL design model at the bottom, a *transaction-level*

specification model at the top, and an interfacing abstraction model in the middle.

For RTL models, a unit of time represents the duration of a single cycle of the system clock, and data are represented using bits and bit-vectors primarily. Because this level is already handled by the current generation of commercial logic synthesis tools, our interest naturally lies at the interface between RTL models and those levels residing above it.

In contrast to the RTL, the transaction-level signals can be viewed as abstract packets, and transaction-level time is seen to progress only as new transactions are initiated. Conceptually, the packet data type resembles the record type of VHDL. Typical fields for bus transactions include many familiar ones, including target addresses, data blocks, and so on.

The opcode field shown in this figure is less familiar. Its purpose is to abbreviate the wide range of (RTL) control-signal behavior that normally gets represented by a collection of informal timing diagrams. Transaction opcodes can include such information as transaction target information, directionality (read or write), as well as various safety and liveness properties expected of the transmitting system. An example safety property might be the non-interference with externally-sourced bus traffic, while an example liveness property might be an eventual reply to a received request.

While bus-based transactions represent the most familiar examples, transactions can actually be viewed in a more general way than this. Specifically, any sequence of low-level actions that results from the occurrence of a single RTL initiating event can be considered a transaction. We call the initiating event a *boundary event* to denote its

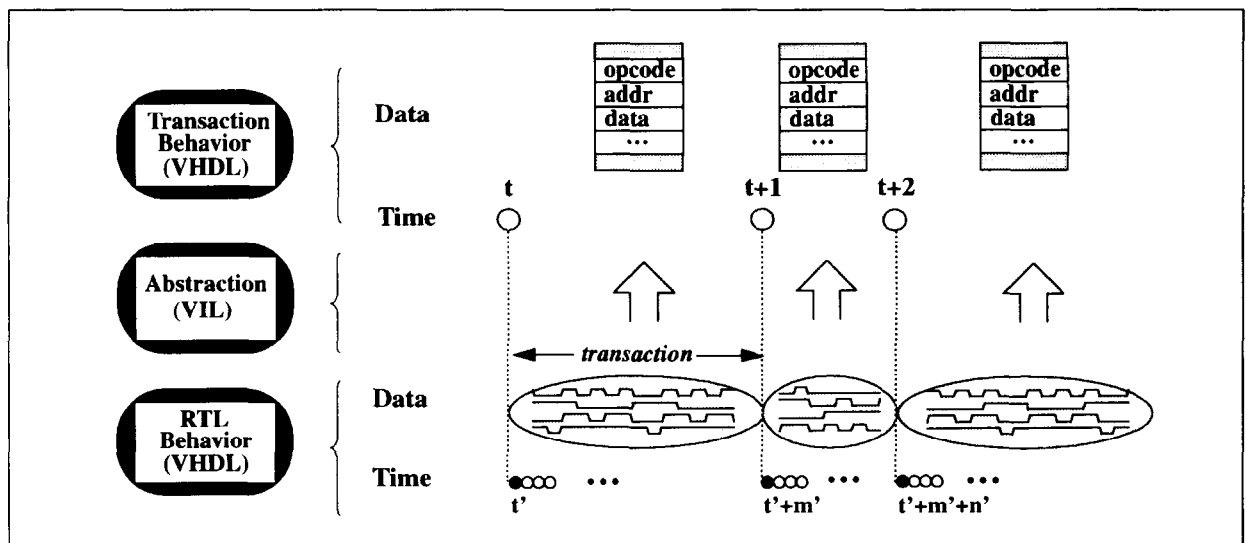


Figure 1: Transaction-Level Data and Time With Respect to the Implementing Register Transfer Level.

marking the temporal boundary of the abstract operations implemented by the low-level actions. In Figure 1, the boundary events might represent the arrival of bus mastership requests to initiate new bus transactions. The difference in time scales between the two levels is quite pronounced, as each transaction can represent many clock cycles worth of behavior.

As indicated by the arrows in Figure 1, the abstraction model shown in the middle of the figure associates the signals of the transaction level with those of the register transfer level. In fact, VIL definitions *define* the transaction-level signals in terms of the RTL signals. The ability to implement definitions such as this is a characteristic of formal approaches to hardware verification. However, we believe that this is the first time such a capability has been made available for simulation environments.

### 3 Transaction Modeling With VHDL and VIL

To provide a readily-understood description of some interesting aspects of our modeling approach, we will apply the approach to a very simple example in this section. The example is a single-bit full adder cell familiar from textbooks on computer architecture and arithmetic. We assume that the RTL design model for this adder contains the following two VHDL expressions for the sum and carry outputs, respectively:

```
sum <= a xor b xor cin;
cout <= (a and b) or (a and cin) or (b and cin);
```

The input signals **a** and **b** are the data inputs to the adder and **cin** is the carry in.

In the rest of this section we will focus on models for the specification of this adder and for the abstraction linking it to the design. We begin with a simple example where the specification is a data abstraction of the RTL design. We then update this example with a nontrivial interface protocol to show the handling of temporal abstraction. Abstraction model re-use is demonstrated as well. More details of the language VIL can be found in [8].

### 3.1 Data Abstraction

**VHDL specification.** Figure 2 shows a VHDL specification of the full adder using integer subtypes for the signals in lieu of the bits and bit vectors of the design. The subtypes **int1** and **int2** contain the ranges **0–1** and **0–3**, respectively, and are assumed to be defined in the package **types** in this example.

The input signals of line 8 are integer counterparts to the RTL inputs **a**, **b**, and **c**, while line 9 declares a new signal corresponding to both **cout** and **sum** of the design. In view of the behavior shown on line 15, we are tempted to interpret this new signal as some type of concatenation of the previous two signals. However, without yet knowing the abstraction mappings defining **cout\_sum**, this interpretation (the correct one) is only a plausible guess. All we can say for sure is that this new specification defines the output as the numerical sum of the three inputs.

**VIL abstraction.** The VIL model of Figure 3 formally links our new specification to the original RTL design. Aside from the use of different keywords (**spec\_entity** and **design\_entity**), and the location of the type declarations, the two entity blocks shown in Figure 3 are VHDL entity blocks. Capital letters are used to distinguish the signals of the two different levels of abstraction.

The signal relationships among the two levels are defined in the abstraction block that begins on line 17. As an example, line 19 says that specification signal **A** carries an integer version of the RTL signal **a**, for all times **t**. The time **t** is a reserved word in VIL representing the abstract time. The fact that it is also used for the concrete (RTL) signals here indicates that no temporal abstraction exists.

The VHDL function **to\_integer** maps the bits and bit vectors of the design to the integers of the specification. The function works in an expected way, with a bit value of '1' or 'H' mapped to integer 1 and everything else mapped to 0. This overloaded function also maps bit vectors to integers, with the rightmost position of the vector representing the least significant bit.

```

1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  library WORK;
4  use WORK.types.all;
5
6  entity add1s is
7  port (
8    A, B, CIN :in int1;
9    COUT_SUM :out int2
10 );
11 end add1s;
12
13 architecture d_abs of add1s is
14 begin
15   COUT_SUM <= A + B + CIN;
16 end rtl;
```

Figure 2: VHDL Specification for a Single-Bit Full Adder Reflecting Data Abstraction.

```

1  spec_entity add1s is
2  subtype int1 is integer range 0 to 1;
3  subtype int2 is integer range 0 to 3;
4  port (
5  A, B, CIN :in int1;
6  COUT_SUM :out int2
7  );
8  end add1s;
9
10 design_entity add1 is
11 port (
12 a, b, cin :in std_logic;
13 sum, cout :out std_logic
14 );
15 end add1;
16
17 abstraction d_abs of rtl is
18 begin
19 (A At t == to_integer (a) At t) ^
20 (B At t == to_integer (b) At t) ^
21 (CIN At t == to_integer (cin) At t) ^
22 (COUT_SUM At t == to_integer (cout & sum) At t)
23 end;

```

Figure 3: VIL Definition for Adder Data Abstraction.

In addition to these functional mappings, line 22 defines the concatenation of the two output signals that we speculated about earlier. The specification output **COUT\_SUM** is thus defined as the integer version of **cout** and **sum**, concatenated left-to-right using the VHDL operator for concatenation, **&**. VIL borrows VHDL syntax wherever possible.

Regarding again the function **to\_integer**, we note that this function operates on signals (e.g., **a**) rather than on the individual values carried by the signals (e.g., **a At t**). This is consistent with standard practice; for example, the functions of the standard logic package **std\_logic\_1164** operate over signals. This is the same thing as saying that they operate over the values carried by signals *at every time*, rather than for a specified single time. The function **to\_integer** is one of a number of standard functions recognized by VIL.

### 3.2 Four-Phase Protocol Modeling

In order to demonstrate non-trivial temporal abstraction for our adder example, we need a more complex interface protocol than the combinational flow-through we have been using.

**Informal description.** One simple protocol that meets our needs is the standard four-phase protocol described in Figure 4. Part (a) of this figure shows the directionality of two new signals implementing the protocol control. The timing diagram of part (b) provides an informal description of the protocol itself.

The control signals **rqtin** and **ackout** carry the directionalities implied by their names only to support their application to the full adder example below. The environment will source the request signal **rqtin**, while the adder will drive the acknowledgement signal **ackout**.

In part (b), we assume that signals **rqtin** and **ackout** are both low prior to the beginning of a transaction. The signal **rqtin** is brought high by the environment to initiate a new transaction, an action constituting the boundary event for this protocol.

In response to the arrival of a high **rqtin**, the adder cell is required to raise **ackout**; in turn, the environment is required to then lower **rqtin**, an act which leads to the subsequent lowering of **ackout** as well. Although this is not evident from the diagram, we assume that the transitions are unbounded in time; they must occur 'eventually' at

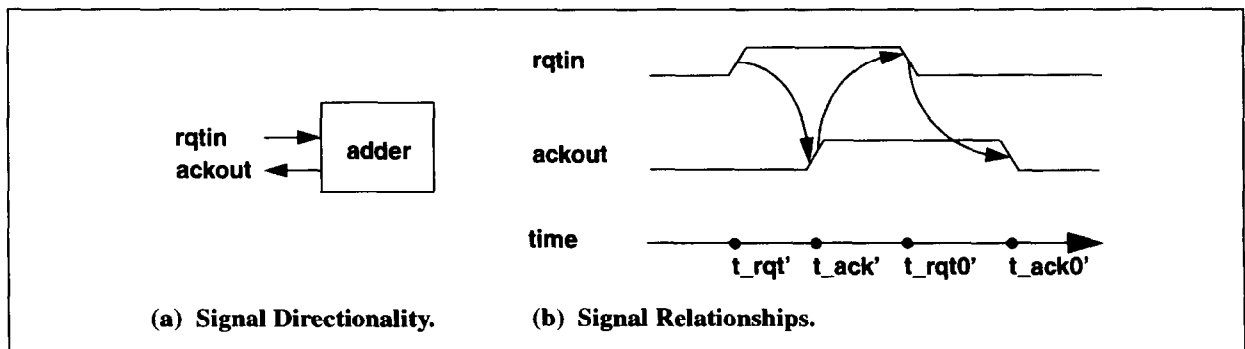


Figure 4: Informal Description of a Four-Phase Interface Protocol.

some future points in time. The times shown at the bottom of the figure mark the signal transitions just described.

**VIL formal model.** Timing diagrams can provide highly intuitive descriptions of protocol behavior; however, with few exceptions (e.g., [7]), they lack the formal semantics necessary to support formal reasoning. For example, many timing diagrams (as above) fail to distinguish between events that must occur on a specific cycle, within a bounded period of time, or just eventually.

In order to support the reasoning necessary to accommodate a range of tools encompassing simulation and a number of formal approaches, VIL provides a semantics that is rigorously defined by a formal embedding in higher-order logic (e.g., [3]). VIL borrows constructs from existing temporal logics [9] and prior work described in [10]. Despite its formality VIL provides a concise and intuitive syntax.

Figure 5 describes how VIL can be applied to our example protocol. Part (a) shows a VIL formula (*rqt\_sat*) specifying the ‘request’ behavior implemented by the *rqtin* signal, while *ack\_sat* of part (b) specifies the ‘acknowledge’ behavior implemented by *ackout*.

VIL uses VHDL-style aliases to define the four points in time shown in Figure 4, along with a number of new VIL constructs. For example, the expression for *t\_rqt'*

on line 2 says that *t\_rqt'* is the time when *rqtin* changes to ‘1’ for the *arb*’th time since time 1, where *arb* is some unspecified, or arbitrary, time. The use of the VIL keyword *arb* here conveys the idea that the formula covers *all* instances of *rqtin* changing to ‘1’.

The other three times are offsets from *t\_rqt'*. For example, line 3 says that *t\_ack'* is the time when *ackout* changes to ‘1’ for the 0’th count (first time) since *t\_rqt'*. VIL counts are implemented as natural numbers starting at zero.

The body of the VIL formula in part (a) defines the required behavior for *rqtin*. This formula requires three things of *rqtin*:

- (a) If *rqtin* changes to ‘1’ then it must remain ‘1’ while *ackout* remains ‘0’ (the period of time between *t\_rqt'* and *t\_ack'*). This is specified by lines 7 and 8 of the VIL formula.
- (b) If *rqtin* changes to ‘1’, and if *ackout* subsequently changes to ‘1’, then *rqtin* must eventually change to ‘0’. This property is specified by formula lines 7, 9, and 10 and covers the time interval between *t\_ack'* and *t\_rqt0'*.

Since the definition of *t\_ack'* (line 3) provides no information as to when *ackout* changes to ‘1’, line 9 describes an *eventual* occurrence of this event. Other VIL constructs are available to specify time-bounded

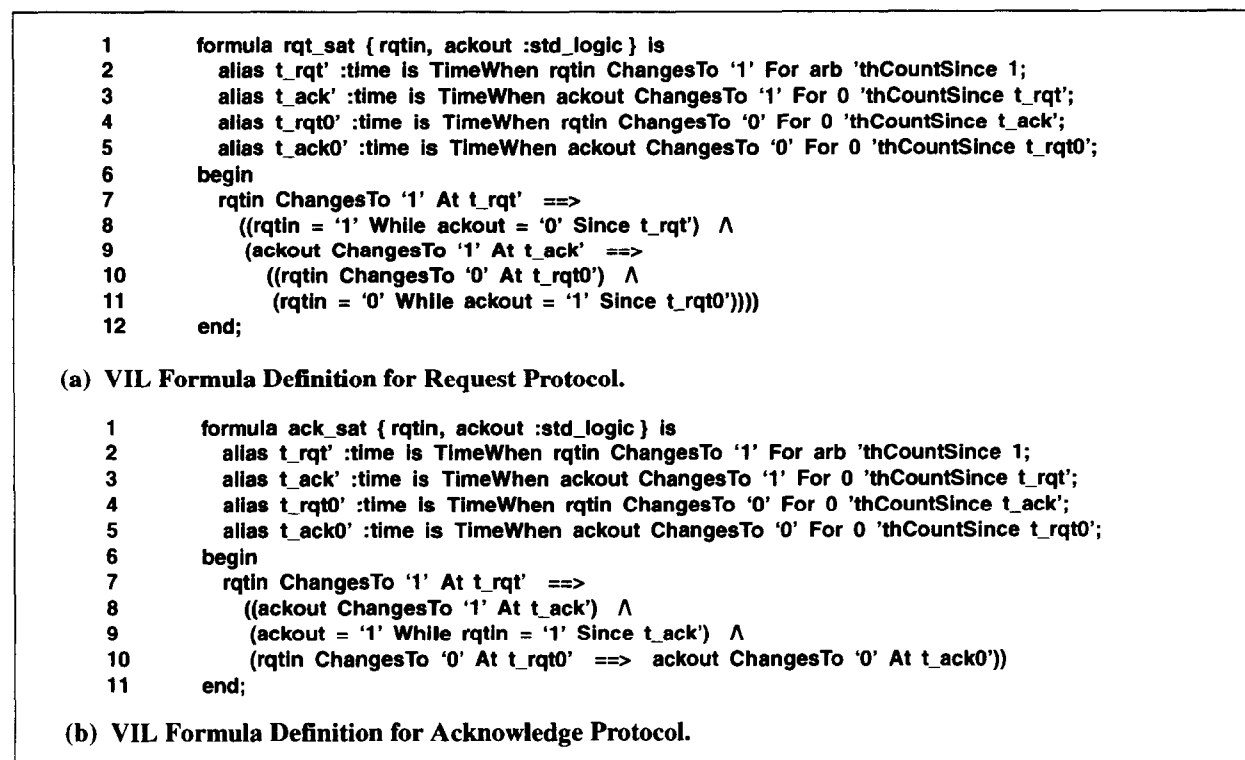


Figure 5: VIL Formal Specification of Four-Phase Protocol.

event occurrences and future events that must occur at specific times.

- (c) Finally, if **rqtin** changes to '1', and if **ackout** subsequently changes to '1', then once **rqtin** changes to '0' it will remain '0' while **ackout** remains '1'. Lines 7, 9, and 11 specify this property, which covers the time interval between **t\_rqt0'** and **t\_ack0'**.

Most of the VIL constructs used in part (a) have meanings consistent with their English syntax. The main exceptions are the symbols **∧** and **==>**, representing conjunction (logical and) and implication (if-then), respectively. These two operators are somewhat special in that they operate over sampled signal *values*, rather than the signals themselves. By way of contrast, the VHDL operator **and** operates over signals.

The formula of part (a), taken as a whole, produces a value that can be interpreted as either logical true or false. By specifying that the value of the formula **rqt\_sat** is true, we would make the statement that the requirements laid out in (a)–(c) above are satisfied.

The formula in part (b) of Figure 5 specifies the behavior for **ackout** over the same three intervals of time covered in part (a). This formula says that if **rqtin** changes to '1' then:

- (a) **ackout** eventually changes to '1';
- (b) after changing to '1', **ackout** remains '1' while **rqtin** remains '1'; and
- (c) after changing to '1', if **rqtin** subsequently changes to '0' then **ackout** eventually changes to '0'.

**Discussion.** In moving away from the graphical timing diagram of Figure 4 to the text of Figure 5, we appear to be bucking the recent trend *towards* graphical formalisms. Although VIL provides a high degree of user-friendliness, there is probably nothing more friendly than the visual abstraction offered by timing diagrams.

To provide a protocol specification environment integrating the benefits of graphics with the formality of VIL, the tools described in Section 4 include one that supports protocol animation. The tool produces VHDL models to display VIL formulas as timing diagrams using the graphical output of existing simulators. The tool provides fast and (nearly) automatic graphical feedback to designers as they construct VIL models.

### 3.3 Temporal Abstraction

**VHDL specification.** The new interface protocol introduces an interesting modeling issue into our adder specification. This specification must capture the adder's new protocol obligations, as well as the protocol assumptions it can levy on its environment. At the same time, it is desirable to keep the specification as simple as possible.

An effective approach to model interface signal control borrows the notion of *opcodes* used in computer processor modeling. Specifically, we can define unique opcodes for the individual types of abstract behavior that we want to distinguish. We can also define a separate opcode to represent 'illegal' behavior, for systems failing to implement their portion of the protocol under consideration.

Figure 6 shows the transaction-level specification for our adder updated with the protocol interface for its inputs and outputs. Four new opcodes differentiate the new entity block from the previous entity block of Figure 2. The input opcode **RST\_OP** abstracts the behavior of the RTL **reset** signal, while **LIVE\_OP** and **RQT\_OP** do the same for **rqtin**. The output **ACK\_OP** is an abstract version of the RTL **ackout** signal. These are described in more detail below.

All four opcodes share the same enumerated data type, **OpTy**, with values **SAT** and **XXX**. **SAT** represents protocol satisfaction while **XXX** indicates illegal behavior. We assume that this type is declared in the package **types**.

The transaction-level behavior beginning on line 15 contains two main extensions from that of Figure 2. Lines 19–

```

1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  library WORK;
4  use WORK.types.all;
5
6  entity add2s is
7  port (
8    A, B, CIN :in int1;
9    RST_OP, LIVE_OP, RQT_OP :in OpTy;
10   COUT_SUM :out int2;
11   ACK_OP :out OpTy
12 );
13 end add2s;
14
15 architecture trans of add2s is
16 begin
17   process (A, B, CIN, RST_OP, LIVE_OP, RQT_OP)
18   begin
19     assert ( RST_OP = SAT and
20             LIVE_OP = SAT and
21             RQT_OP = SAT );
22     COUT_SUM <= A + B + CIN;
23     ACK_OP <= SAT;
24   end process;
25 end trans;

```

Figure 6: VHDL Specification for a Single-Bit Full Adder Reflecting Data and Temporal Abstraction.

21 precondition the behavior of the adder on a well-behaved environment. In other words, we levy no requirements on our adder if it receives illegal opcode values for **RST\_OP**, **LIVE\_OP**, or **RQT\_OP**.

In addition to the specified behavior for **COUT\_SUM** repeated from before, the adder is required to satisfy its own portion of the four-phase interface protocol. This is reflected on line 23 with the transmission of **SAT** on the output **ACK\_OP**.

**VIL abstraction.** As before, it is the signal mappings contained in the abstraction model that actually give meaning to the abstract signals of the transaction level. Figure 7 shows the abstraction block of a VIL definition providing these mappings.

Lines 27 and 28 of the model contain aliases for two significant concrete times of this protocol. The concrete time **t'** is a reserved word denoting the beginning of the **t'**th transaction. Based on the alias defined on line 27, we can conclude that the boundary event for this protocol is: **rqtin ChangesTo '1'**. This alias establishes a formal link between the temporal streams at the two levels of abstraction. As in Figure 1, abstract time **t** occurs at the same real time as concrete time **t'**. With respect to Figure 4, **t'** corresponds to the time **t\_rqt'** shown there.

The second alias defines **t\_ack'** as the time when the adder raises **ackout** for the first time since the beginning of the transaction. This time corresponds to its name-sake in Figure 4.

The signals **A**, **B**, **CIN**, and **COUT\_SUM** are all defined using the sampling-based abstraction used previously. The difference here is that the sampling points are the aliased times just described, rather than time **t**. The three inputs are seen to be sampled at the beginning of the transaction, while the outputs are sampled at **t\_ack'**.

The concise mappings used for the protocol opcodes **RQT\_OP** and **ACK\_OP** benefit from the pre-defined

formulas **rqt\_sat** and **ack\_sat** shown in Figure 5. If we assume that these formulas are contained in the file **four\_phase.vil**, then the C-style **#include** on line 24 is sufficient to bring them into our model.

This facility for specification re-use is an important aspect of the VIL language. It will permit us, for example, to establish libraries of pre-defined protocol formulas that can be accessed by all VIL users. We expect that this sharing of models will bring great practical benefits to the future construction of abstraction specifications.

Returning now to the definition of **RQT\_OP** on line 35, we can observe a new 'selection-based' form of abstraction at work, in contrast to the sampling-based methods used earlier. Line 35 says that **RQT\_OP At t** is equal to **SAT** precisely when the formula **rqt\_sat { rqtin, ackout }** is true; otherwise it equals **XXX**. Again, a true value for the formula means that the environment is satisfying its part of the four-phase protocol with respect to the **rqtin** signal. The definition of **ACK\_OP** on line 37 provides a corresponding interpretation with respect to **ackout**.

The definition of **LIVE\_OP** on line 34 specifies a type of 'liveness' that would be required in a subsequent verification of the adder. The reader will note that the protocol definition given by the two formulas of Figure 5 relies on the existence of a rising **rqtin** to define the beginning of a transaction. However, nowhere does it state that such an event must actually occur. In fact, this property is not one that we would expect of a protocol definition, which is only concerned with what happens after a transaction is initiated. The occurrence of the **t'**th transaction is indicated by a value of **SAT** for **LIVE\_OP**.

Finally, the definition of **RST\_OP** on line 33 specifies the required behavior of the RTL **reset** signal. This signal satisfies its 'protocol' if it provides a high value at time **0** and then a low value for all times afterwards.

```

24  #include "four_phase.vil"
25
26  abstraction trans of rtl is
27    alias t' :time is TimeWhen rqtin ChangesTo '1' For t 'thCountSince 1;
28    alias t_ack' :time is TimeWhen ackout ChangesTo '1' For 0 'thCountSince t';
29  begin
30    ( A At t == to_integer(a) At t' ) ^
31    ( B At t == to_integer(b) At t' ) ^
32    ( CIN At t == to_integer(cin) At t' ) ^
33    ( RST_OP At t == SAT If ( reset = '1' At 0 ^ Henceforth reset = '0' Since 1 ) Else XXX ) ^
34    ( LIVE_OP At t == SAT If ( rqtin ChangesTo '1' For t 'thCountSince 1 At t' ) Else XXX ) ^
35    ( RQT_OP At t == SAT If ( rqt_sat { rqtin, ackout } ) Else XXX ) ^
36    ( COUT_SUM At t == to_integer(cout & sum) At t_ack' ) ^
37    ( ACK_OP At t == SAT If ( ack_sat { rqtin, ackout } ) Else XXX )
38  end;

```

Figure 7: Partial VIL Definition for Adder Data and Temporal Abstraction.

#### 4 VIL-Based Design Tool Support

One could make the argument that formalizing the interfaces for RTL designs is a cost-effective way to avoid design mistakes early in the design process. However, much greater benefit is possible from design tools that exploit the information contained in VIL definitions.

**Test bench generation.** Work at Levetate Design Systems and the University of Washington has led to a novel approach to test bench synthesis supporting advanced simulation environments. Figure 8 shows one such environment that includes test benches for both the design and the specification. The significant aspect of this environment is the reduced user interaction required to verify an RTL design against a transaction-level specification.

In fact, other modes supported by our tool exhibit even greater autonomy, at both the input selection side and the output certification side. This tool also supports the protocol animation pointed out in Section 3.2.

Figure 9 shows the role of this Simulation Self-test Tool<sup>1</sup> (or SST) in simulation model construction. The motivation for this tool is the observation that the design test bench shown in Figure 8 *implements* the abstraction linking the signals of the design to those of the specification. It is conceptually straightforward to generate this model from an abstraction specification written in

<sup>1</sup> Patent pending.

VIL. Other approaches for test bench construction rely on the structure of the design (e.g., [6]) or on visual abstractions (e.g., [2]) to serve as specification languages. SST is expected to enter beta testing in the 2nd half of this year.

**Formal verification tools.** All formal approaches to verification require explicit models for both the design and the specification. The 3-level modeling paradigm shown in Figure 9 supports the modeling needs of these approaches, and suggests ideas for new commercial design tools based on, for example, model checking [1] and theorem proving [5]. Hardware design engineers trained on VIL in simulation environments would have a straightforward transition to the newer formal verification tools that use the same input languages.

**Behavioral synthesis tools.** Finally, a new generation of behavioral synthesis tools can be envisioned that has the designer produce the top two levels of the hierarchy in Figure 9. The new information provided by the VIL abstraction model should enable these tools to better serve general-purpose design applications.

#### 5 Conclusions

Current RTL simulation practices are plagued by inadequate specification languages and the associated tool support necessary to automate labor-intensive and time-consuming activities. As a result, simulation-based verification today is inefficient with regard to the time and money needed to achieve high-quality designs.

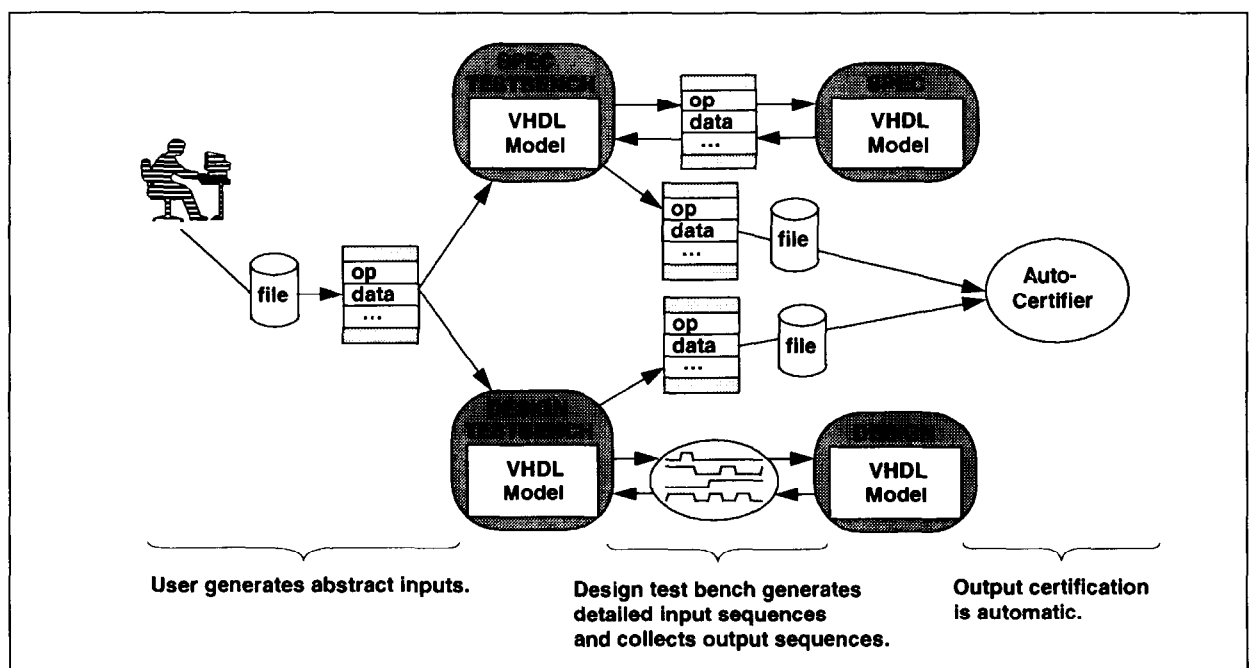
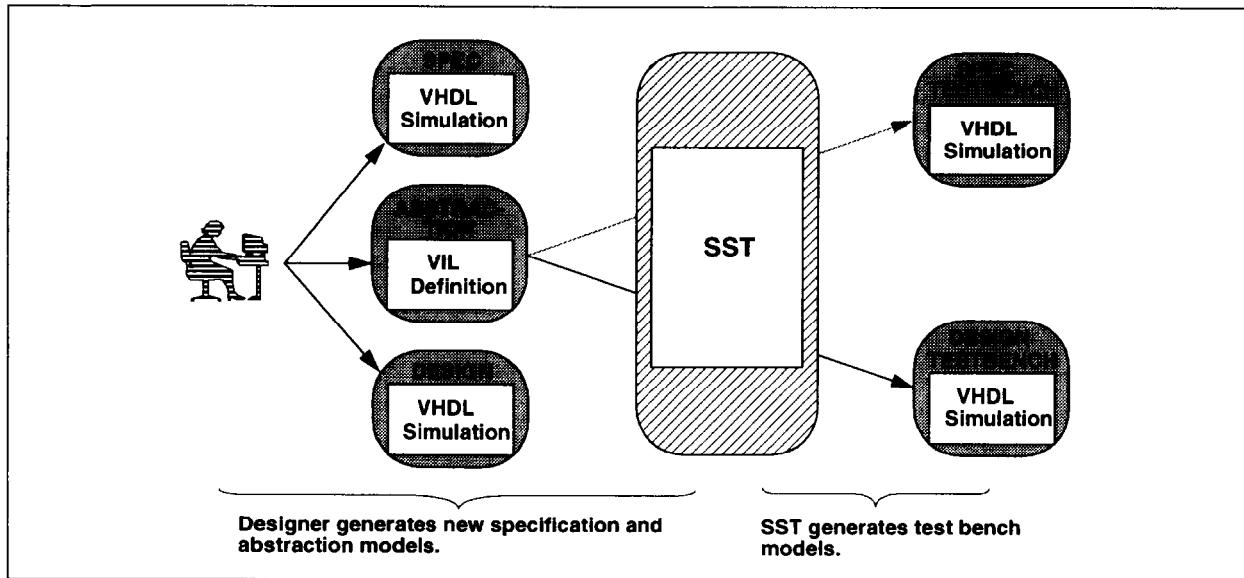


Figure 8: Macro-Automated Simulation Environment.



**Figure 9: Tool Support for Automated Test Bench Generation.**

A key missing element in existing simulation environments is an explicit model for the abstraction linking the signals of the design to those of the specification. To our knowledge, the language VIL that we have demonstrated in this paper offers the first general-purpose and formal means to construct abstraction definitions for simulation environments. VIL provides first-of-a-kind support for the data and temporal abstraction required for high-level specification models.

Although VIL is a new language, it employs intuitive, English-like syntax to model concepts familiar to design engineers. We believe that VIL-based modeling can be easily mastered by hardware design engineers. The mathematical foundations of VIL are considerably easier to understand than calculus, for example.

The primary reason that VIL-like languages are not already widespread today is the lack of motivation for design engineers to learn them. This situation is changing however. A number of advanced tools, including SST, model checkers, and other formal methods, require them. The advantages that these tools offer in rigor and automation could eventually lead to a role for interface languages in abstraction modeling resembling the role of state machines in behavioral modeling.

## 6 References

[1] J.R. Burch, et. al., "Symbolic Model Checking for Sequential Circuit Verification," *IEEE Transactions on Computer-Aided Design*, April 1994.

- [2] "ESDA Software Adds Enhancements and Links to Verilog Testbench Tool," *Electronic Design*, June 12, 1995.
- [3] D.A. Fura, *Abstract Interpreter Modeling and Verification Methods for Dependable Embedded Hardware Systems*, Ph.D. Thesis, Electrical Engineering Department, University of Washington, March 1995.
- [4] R. Goering, "Synthesis Falls Short When SOS Calls," *Electronic Engineering Times*, June 19, 1995.
- [5] M.J.C. Gordon and T.F. Melham, *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, Cambridge University Press, 1993.
- [6] S. Kapur, J.R. Armstrong, and S.R. Rao, "An Automatic Test Bench Generation System," in *VHDL International Users Forum*, May 1994.
- [7] G. Kutty, et.al., "A Graphical Interval Logic Toolset for Verifying Concurrent Systems," in C. Courcoubetis (ed.), *Computer Aided Verification*, 1993.
- [8] Levetate Design Systems, "VIL Prototype Language Definition," Technical Report, 1995.
- [9] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer-Verlag, 1992.
- [10] T.F. Melham, "Abstraction Mechanisms for Hardware Verification," in G. Birtwistle and P.A. Subrahmanyam (eds.), *VLSI Specification, Verification, and Synthesis*, 1988.