# Abstraction-Directed Synthesis of VHDL Test Benches

David A. Fura
Levetate Design Systems
4756 University Village Place N.E. #168
Seattle, Washington 98105 USA
dfura@levetate.com

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

**Abstract.** This paper describes a new automated approach to generate simulation test bench models. A distinguishing feature of this approach is the use of abstraction models to serve as inputs to the test bench generation process. The approach implementation within a new VHDL test bench synthesis tool is demonstrated on an illustrative example. This new tool is evaluated with respect to existing tools and shown to provide better support for multi-level simulation environments. Future work is discussed for implementing advantageous aspects of competing approaches.

## 1 Introduction

High-quality simulation test benches are essential elements of modern simulation environments for complex hardware systems. This is particularly true for the verification of designs expressed at the gate or register transfer level (RTL) against behavioral specifications expressed at higher levels of abstraction. The large number of test cases presented by these verifications increasingly demands the automation provided by self-testing test benches.

It is recognized today that the magnitude of the test bench generation problem is tracking the increasing complexity of today's system designs. Many would even argue that developing high-quality test benches has become more difficult than designing the systems themselves. Evidence of this is seen in numerous company reports of hardware projects where the number of verification engineers greatly exceeds the number of design engineers.

The increasing magnitude and importance of the test bench problem is also demonstrated by the recent proliferation of test bench synthesis methods and tools. Examples of recent research in this area have been reported in, for example, [11], [13], [1], and [14]. Commercial CAD tool vendors have also introduced a number of new systems within the past year, as reported in [6], [4], [7], [8], [5], and [9].

The major contribution of this paper is the demonstration of a new approach for test bench synthesis that is fundamentally different from existing methods and advantageous in a number of important respects. Our approach is the first (to our knowledge) to provide automatic synthesis of simulation models serving multi-level simulation environments, where the specification and design models reside at different levels of data and temporal abstraction. Our approach also promotes a 'specification-oriented' design paradigm supporting a coherent growth path to future hardware design methods of significantly greater rigor and automation.

A distinguishing element of our approach is the use of 'abstraction models' to serve as the input to our synthesis algorithm. Abstraction models define the mappings between the signals of abstract specifications and those of their underlying designs. The models are expressed in a new language called VIL (VHDL Interface Language) that is the first abstraction language (we believe) to ever be used in a simulation environment.

We have implemented our synthesis approach in a prototype tool called SST (Simulation Self-test Tool). SST generates test benches automatically from input models expressed in VIL. The current implementation of SST supports VHDL environments.

In Section 2 of this paper we present related work in the areas of test bench synthesis and modern simulation methodologies. We explain how these methodologies are not being supported by existing commercial synthesis tools.

In Section 3 we overview our approach to test bench synthesis and compare it to existing methods. Our approach's better support for multi-level simulation is explained.

In Section 4 we demonstrate our approach in action using the single-bit full adder example introduced in [10]. SST support for specification animation, interface protocol modeling, and design verification is described.

In Section 5 we finish with a concluding discussion.

## 2 Related Work

The utility of a test bench synthesis tool can be judged by the range of simulation environments it supports. One important question to ask of a tool is whether or not it can handle leading-edge environments capable of verifying large multi-board systems. It turns out that none of the existing tools support the multi-level simulation environments necessary for these systems.

### 2.1 Multi-level Simulation Environments

A good look into industry simulation practices for large hardware systems is contained within papers presented by Bell Northern Research [12] and IBM [3]. One of the most interesting aspects of both of these efforts is the use of multi-level simulation methods, where abstract behavioral simulation models serve as specifications for the RTL design models. In neither of these efforts were the specification-level signals of the same form as the RTL signals. The "high-level data types" of [12] and the "packets" of [3] are both data and temporal abstractions (see [10]) of the underlying RTL.

Multi-level design environments provide benefits in at least two ways. First, the creation and testing of explicit models for the specification helps to flush out the mistakes often present in natural language specifications. The abstract nature of the specification models makes them both easy to understand and quick to execute.

A second advantage of executable specifications is their support for improved rigor and automation in the design verification process, since an abstract specification model provides the basis for certifying design model outputs. In theory, a simulation test bench model can be designed that sends 'equivalent' inputs into the specification and the design and then tests for 'equivalence' over the specification and design outputs.

In practice, designing such self-checking test benches is quite difficult since it requires implementing mappings among the signals of two different levels of abstraction. For example, a specification-level input might be mapped into an implementing sequence of design-level inputs prior to its application to the specification model. At the output side, the design-level output sequences might be mapped up to the specification level in preparation for their certification against the specification model outputs.

Both of these types of mappings are especially tough to implement for the complex interface protocols used in hardware systems today. In fact, the work of [12] discovered that the size of the test bench code greatly exceeded the code size for the chip designs themselves. Even worse, the disparity was seen to grow as the complexity of the chips increased, suggesting that the test bench generation problem is one that threatens to only worsen in the future.

### 2.2 Test Bench Synthesis Tools

In both of the papers cited in the previous section, the test bench signal mappings were implemented as hand-coded simulation models. In reviewing the existing methods and tools for test bench synthesis, we will pay particular attention to their ability to automate the generation of such models. We will consider methods that are mature enough to be offered as commercial products.

**RTL state machine graphics.** In [1], researchers from Virginia Tech describe a system that converts graphical state machine-based specifications into VHDL test benches. A recently announced commercial tool from Summit Design, Inc. [4] also includes this capability, now in support of both Verilog and VHDL.

Due to this approach's use of "high level" test bench specifications, it might appear to support multi-level environments. However, an inspection of the graphical specifications used here reveals fundamental differences from those used in the target environments. The graphical specifications exhibit neither the data abstraction nor the temporal abstraction of the target environments – they are essentially pictorial versions of the RTL designs.

Because of these specification differences, test benches derived from RTL graphics do not meet the needs of multi-level simulation environments. They cannot establish the mappings linking the signals of RTL designs to the abstract specifications necessary to achieve automatic design output certification, for example.

**RTL timing diagram graphics.** To our knowledge, Borriello was the first to suggest the use of graphical timing diagrams as specification languages for simulation control (e.g., [2]). Currently, there are a number of commercial tools that synthesize test benches from interface protocol specifications expressed as timing diagrams. DS Diagonal Systems AG [6], Summit Design, Inc. [4], Chronology Corp. [5], and SynaptiCAD, Inc. [9] all have announced and/or described such a tool in the past year.

However, none of these tools, by themselves, support multi-level simulation environments. The reason for this, similar to that of the previous set, is that they all operate at the level of the RTL design. Because the tool inputs do not define the mappings linking the signals of the design to those of the abstract specification, it is impossible for the tools to implement these mappings within the synthesized test benches.

A second problem faced by timing diagram-based tools is the relative incompleteness of the information carried by

timing diagrams. That is, while timing diagrams are useful for describing interface protocols, they do not define the functionality (data processing) of the systems being interfaced. Therefore, if one wants a tool producing test benches that can certify design outputs automatically, then the required design functionality must be specified using a second approach. Because this required functionality would be expressed at the abstraction level of the design, it would be difficult to avoid constructing a specification significantly different from the design itself. At best, expressing the required results this way more than duplicates the effort put into an abstract specification. At worst, any mistakes present within the design could easily be repeated within the design's specified behavior, and thereby invalidate the simulation results.

**Regression testing.** A recently announced test bench tool from Synopsys Inc. [7] synthesizes test benches for individual blocks of a system. The tool works by capturing and storing the inputs and outputs of one or more blocks during system-level simulations that are themselves controlled by user-developed system test benches. The stored block-level inputs and outputs are used by the synthesized test benches for blocks that are to be subsequently tested in isolation from the rest of the system. Such block-level tests would occur after a block design was changed, for example. The testing of individual blocks in isolation from the rest of the system results in much shorter simulation times than repeated system-level tests.

It is clear that this approach does not, by itself, support multi-level simulation environments. For example, the problem remains of constructing the system-level test bench in the first place.

**Test bench programming.** Another recently announced test bench tool from Systems Science Inc. [8] generates Verilog test benches from specifications written in a new 'verification language' called Vera. The language constructs of Vera are especially well suited for specifying test bench activities, such as certifying that a signal value arrives within a specified window of time. Vera programs are more concise and easier to write than the equivalent Verilog test bench code.

For reasons familiar from above, this tool cannot by itself support the test bench requirements of our targeted multi-level simulation environments. Even as the programs written in Vera are clearly test bench 'specifications,' they also describe behavior at the level of the RTL design. Because they do not specify the mappings linking the RTL signals to their behavioral-level counterparts, the generated test benches cannot implement these mappings.

## 3 Abstraction-Directed Test Bench Synthesis

The inability of existing test bench tools to support multi-level simulation environments is understood simply by examining the quality of the information that they process. Test benches required by these environments must map the signals of the specification and the design to a common format. For a tool to create such test benches requires it to understand the specification for the mapping. None of the existing tools have access to such information.

This is not surprising, since (to our knowledge) no notation has even existed to represent the required mappings prior to the development of VIL (see [10]). VIL is an 'abstraction specification' language that permits designers to define mappings linking the signals of specifications to those of the underlying designs. VIL models actually define the specification-level signals in terms of the design-level signals. These models supply the information needed by test bench generation tools serving advanced multi-level simulation environments. Examples of VIL models are given in [10] and below.

Figure 1 demonstrates the role of VIL in a new 'specification-oriented' design paradigm that features greater designer attention to issues of specification. This paradigm makes available to CAD tools much of the information that has always been used by design engineers, but that has been in formats so informal as to be inaccessible to tools.

The test bench generation tools represented by Figure 1 promise a greatly reduced level of effort in the work currently being expended in detailed test bench coding for RTL designs. This is in exchange for some additional work put into system specification where it belongs.

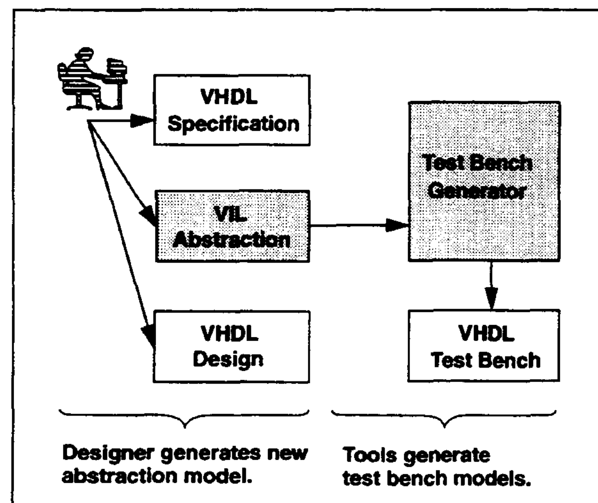


**Figure 1: Abstraction-Directed Test Bench Synthesis.**

An additional benefit of this specification-oriented design approach is its support for advanced methods in formal verification and behavioral synthesis. Formal methods to verify designs against abstract specifications *require* all three of the models shown in Figure 1. VIL serves the needs of these methods very well since it evolved out of a theorem proving environment and has a semantics formally defined by an embedding in higher-order logic. VIL can also serve as a constraint language to guide the behavioral synthesis of VHDL design models. SST provides design engineers a gradual introduction to the specification methods required by these future hardware design tools of greater rigor and automation.

## 4 Test Bench Synthesis Using SST

SST (Simulation Self-test Tool[1]) is a test bench generation system under development at Levetate Design Systems. The current implementation of SST translates VIL abstraction specifications into VHDL simulation models for a number of different simulation scenarios:

(a) *Specification test benches* support the standalone testing of specification models.

(b) *Design test benches* support the standalone testing of design models.

(c) *Protocol test benches* support the VIL modeling of interface (e.g., bus) protocols.

(d) *Self-testing test benches* support the autonomous verification of design models against their specifications.

In this section we illustrate how the models produced by SST support coherent system development methodologies. In particular, we describe a methodology comprising the following three steps:

Step I: Specification model development using specification test benches – (a) above.

Step II: Abstraction model development using protocol test benches – (c) above.

Step III: Design model development using self-checking test benches – (d) above.

The model generation for each of these steps is demonstrated in its own subsection below (Sections 4.3, 4.4, and 4.5, respectively). Section 4.1 starts things off by describing a simple circuit application that we use to illustrate these steps. Section 4.2 presents the VHDL specification model and VIL abstraction model for the example circuit. These are repeated from [10].

1 Patent pending.

### 4.1 Adder Example Overview

The example that we use is a continuation of the one used in [10]. The application is a single-bit full adder cell familiar from textbooks on computer architecture and arithmetic. The RTL design model for this adder is assumed to contain 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.

To illustrate modeling issues related to temporal abstraction, some complexity is added into the example in the form of the standard four-phase protocol described in Figure 2. Part (a) of this figure shows the directionality of two new signals implementing the protocol control. The control signals **rqtin** and **ackout** carry the directionalities implied by their names, with the environment sourcing the request signal **rqtin** and the adder driving the acknowledgement signal **ackout.**

The timing diagram of part (b) provides an informal description of the protocol itself. To begin, it is assumed that signals **rqtin** and **ackout** are both low prior to the beginning of a transaction. The signal **rqtin** is set high by the environment to initiate a new transaction.

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. The times shown in the figure mark the signal transitions just described. Although not shown in the figure, it is assumed that the adder inputs **a**, **b**, and **cin** are active at the beginning of

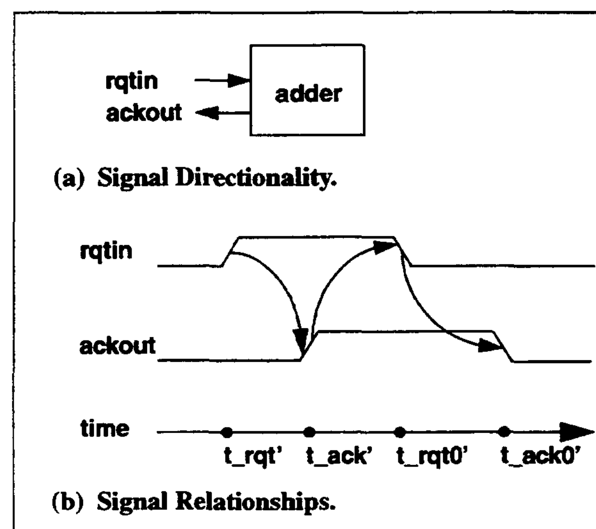


**Figure 2: Informal Description of a Four-Phase Interface Protocol.**

the transaction, while the outputs **sum** and **cout** are active on the rising edge of **ackout**.

### 4.2 The Specification and Abstraction Models

**VHDL specification.** Figure 3 shows the 'transaction-level' specification for the example adder circuit. At the transaction level of abstraction the duration of a single time step represents an entire transaction worth of behavior. For the protocol of Figure 2, this corresponds to the behavior occurring between the rising edges of **rqtin**.

The input signals **A**, **B**, and **CIN** of the specification are integer counterparts to the RTL inputs **a**, **b**, and **cin**, while the output **COUT_SUM** corresponds to the RTL pair **cout** and **sum**. The types **:int1** and **:int2** are integer subtypes with ranges **0–1** and **0–3**, respectively. The declarations for these types are assumed to be located in the package **types**. In fact, SST automatically produces this package from the information contained in the VIL abstraction.

The input **RST_OP** abstracts the behavior of the RTL **reset** signal, while **LIVE_OP** and **RQT_OP** do the same for the **rqtin** signal. The output **ACK_OP** is an abstract version of the RTL **ackout** signal. Each of these signals carries 'opcode' values from the set {**SAT**, **XXX**}, which represent protocol satisfaction and non-satisfaction, respectively. The type **:OpTy** is a VHDL enumerated type containing these two values, and is again declared in the SST-generated **types** package.

```
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 3: VHDL Specification for Adder with Data and Temporal Abstraction.**

A better understanding of these signals can be gained from their use in the architecture block beginning on line 15 of Figure 3. To begin, lines 19–21 precondition the behavior of the adder on a well-defined environment. In other words, no requirements are levied on the adder if it receives illegal opcodes for **RST_OP**, **LIVE_OP**, or **RQT_OP**. The adder design is therefore permitted to rely on certain assumptions regarding the behavior of the RTL **reset** and **rqtin** signals. The assumption specifics are formally represented in the VIL abstraction model below.

Finally, lines 22 and 23 of the specification define the requirements for the two transaction-level outputs. The use of integer subtypes at this level is seen to provide a highly-intuitive use of numerical addition for the **COUT_SUM** output. The specified value of **SAT** for **ACK_OP** indicates that the adder is required to satisfy its protocol obligations with respect to the RTL **ackout** signal. Again, the VIL abstraction model formally defines what these obligations mean at the design level.

**VIL abstraction.** Figure 4 shows the interesting parts of a VIL abstraction model linking the transaction-level adder specification to the RTL design. The parts not shown here are VHDL-like entity blocks for the specification and design. The specification entity contains the VHDL type declarations for **:int1**, **:int2**, and **:OpTy**.

Lines 27 and 28 of the model contain aliases for two significant concrete times of the four-phase protocol used in this example. The concrete time **t'** is a VIL reserved word denoting the beginning of the **t** 'th transaction. It is defined here as the time when **rqtin** changes to **'1'** for the **t** 'th count since time **1**. This time corresponds to the time **t_rqt'** of Figure 2. The second alias defines **t_ack'** as the time when the adder raises **ackout** for the first time since the beginning of the transaction. All 'counts' in VIL are natural numbers that begin at 0.

As might be inferred from above, the abstract time **t** is another reserved word in VIL denoting the **t** 'th, or 'current,' specification-level time (the transaction count). VIL models define abstraction mappings by focusing on an arbitrary single (the **t** 'th) transaction.

Lines 30–37 of the model define the eight transaction-level signals with respect to the underlying RTL signals. The adder data input mappings on lines 30–32 are the most straightforward to understand. For example, line 30 says that input **A** is an integer version of RTL signal **a** sampled at time **t'**, which is again the beginning of the transaction.

Line 36 defines the transaction-level data output **COUT_SUM** as an integer version of **cout** and **sum**, sampled at **t_ack'** and concatenated left to right.

```
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' )  Λ
31        ( B At t  ==  to_integer (b) At t' )  Λ
32        ( CIN At t  ==  to_integer (cin) At t' )  Λ
33        ( RST_OP At t  ==  SAT If ( reset = '1' At 0  Λ  Henceforth reset = '0' Since 1 ) Else XXX )  Λ
34        ( LIVE_OP At t  ==  SAT If ( rqtin ChangesTo '1' For t 'thCountSince 1 At t' ) Else XXX )  Λ
35        ( RQT_OP At t  ==  SAT If ( rqt_sat { rqtin, ackout } ) Else XXX )  Λ
36        ( COUT_SUM At t  ==  to_integer (cout & sum) At t_ack' )  Λ
37        ( ACK_OP At t  ==  SAT If ( ack_sat { rqtin, ackout } ) Else XXX )
38      end;
```

**Figure 4: Partial VIL Definition for Adder Data and Temporal Abstraction.**

The remaining four opcode mappings use a 'selection-based' form of abstraction in lieu of the 'sampling-based' methods of the previous four. On line 33, the **RST_OP** signal is defined to carry the value **SAT** precisely when the RTL **reset** is high at time **0**, and low for all times afterwards. Recall from above that a value of **SAT** is asserted for **RST_OP** in the transaction-level specification. The abstraction mapping presented here defines very precisely what this means at the level of the RTL design.

Lines 35 and 37 of the abstraction specify the mappings for the four-phase protocol opcodes **RQT_OP** and **ACK_OP**, respectively. Line 35, for example, says that the 'request' portion of the protocol is met precisely when the formula **rqt_sat {rqtin, ackout}** is true. Note that this formula is defined in the file **four_phase.vil**. The C-style **#include** statement on line 24 brings this formula into our model, as well as the definition for **ack_sat** used on line 37. The meanings of these two formulas will become evident from the descriptions of the following sections.

The final abstraction mapping is for the transaction signal **LIVE_OP**. Line 34 says that **LIVE_OP** is **SAT** precisely when **rqtin** actually changes to **'1'** for the **t** 'th count (at time **t'**). In other words, a value of **SAT** here indicates that the environment initiates the **t** 'th (or current) transaction.

### 4.3 Step I: Specification Model Development

In an ideal top-down system development the issue of specification correctness is one that should be addressed early on. A certified top-level specification, in conjunction with its companion abstraction model, provides the basis for determining design model correctness, and is useful in the earliest stages of detailed design.

Because the transaction-level model of Figure 3 is the top-most level in the modeling hierarchy for our adder, determining the correctness of this model can only be accomplished by a designer-implemented certification of the simulation results at this level.

Figure 5 shows the structure of SST-generated test benches that support specification-level testing. As shown in the figure, these test benches contain two major elements, aside from the instantiation of the specification model. Consistent with their names, the 'select input' and 'store output' subsystems select the specification inputs and store the specification outputs, respectively.

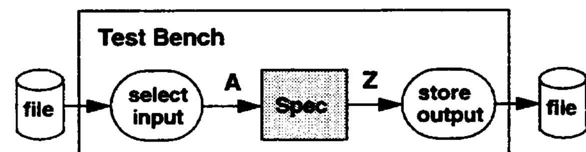


**Figure 5: SST Specification Test Bench Structure.**

**Variability in input data.** One of the important requirements of a test bench is that it not artificially restrict the choices of input data values provided to the model under test. The data types of the input signals should provide the only restrictions on these choices.

The 'select input' block of Figure 5 selects input values from the full data space defined by the input data types. Three different options for accomplishing this are presented to the designer at simulation time:

(a) *Vector-based* selection has the test bench read specification model inputs from the input file.

(b) *Random* selection has the test bench select specification inputs using pseudo-random number generation methods.

(c) *Exhaustive* selection has the test bench select specification inputs using a deterministic march through all possible values. Of course, the designer determines

how far the simulation actually gets through these values by controlling the length of the simulation run.

**Adder example.** Figure 6 demonstrates an SST-generated test bench applied to the example adder specification of Figure 3. This figure was obtained from a postscript screen dump of an actual simulator run.[2] The first eight waveforms show the declared signals of Figure 3, while the last two, for **y_clock** and **ytime**, show the timekeeping signals added by SST. The simulator displays all signal names in lower case.

As can be inferred from this figure, the input data were chosen using exhaustive selection, with the signals **A**, **B**, and **CIN** running through all eight combinations. Only the legal value **SAT** is used for the three opcode inputs (**RST_OP**, **LIVE_OP**, and **RQT_OP**) since the adder is not required to handle illegal input behavior.

From this figure, it is easy to see that the specification model satisfies our expectations for it. The output **COUT_SUM** is indeed the numerical sum of **A**, **B**, and **CIN**, while **ACK_OP** is **SAT** for each input scenario.

**File input/output.** In addition to supporting graphical feedback, the test bench also stores away the simulation results into an output file. The selected inputs are stored along with the outputs to assist any post-processing of the results. All of the type conversion functions necessary for file I/O are generated by SST and located with the type declarations in the **types** package.

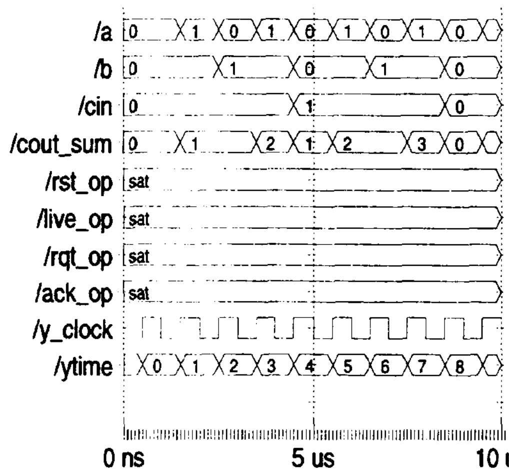


**Figure 6: Simulation Graphics for Adder Specification Animation.**

2 Using the V-System VHDL simulator from Model Technology Incorporated.

### 4.4 Step II: Abstraction Model Development

Within the design methodology being described here, the development of VIL abstraction models is clearly the step most unfamiliar to design engineers in industry. However, a number of techniques exist to help make abstraction modeling an everyday engineering activity:

(a) *Model reuse.* The abstraction model in Figure 4 provides a good illustration of model reuse in action, where the VIL model for the four-phase protocol was simply lifted from a 'protocol library.' There is no reason why standard interface protocols cannot be modeled in standard VIL-like notations.

(b) *Graphical input.* Most engineers would probably favor graphical entry over the textual VIL descriptions required by SST. While SST does not currently support graphical entry, incorporating this capability into a future version of SST is a reasonable step beyond the current state-of-the-art.

(c) *Protocol animation.* The graphical support that SST does currently offer is a protocol animation tool that provides pictorial representations of VIL models using the graphics already supplied by the simulator. This tool is the subject of the remainder of this section.

Figure 7 shows the structure of protocol animation test benches produced by SST. As shown in the figure, these test benches contain two separate regions, with the 'master' region implementing the protocol master functionality and the 'slave' region handling the slave functionality.

The regions implement a common signal flow. The 'select input' and 'store output' blocks are similar to those of the specification test benches described in the last section.

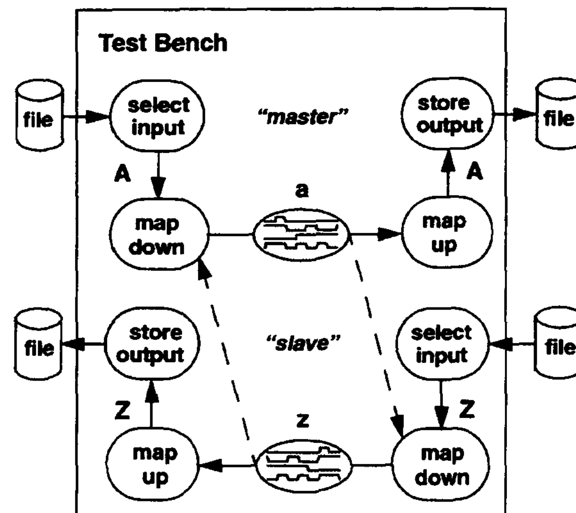


**Figure 7: SST Protocol Test Bench Structure.**

The two new blocks, 'map down' and 'map up,' implement abstraction mappings in the directions spec-to-design and design-to-spec, respectively.

The waveforms shown in the figure represent the design-level signal flow displayed using the simulator graphics. These waveforms provide a direct graphical implementation of the formulas contained in the VIL source model. When the formulas are those specifying the protocol control, then these waveforms are instances of the protocol implementation. The waveforms are obtained following a short sequence of three steps: abstraction model compile, test bench compile, and test bench simulation.

It is an important aspect of protocol test benches that they interface with neither specification models nor design models. Protocol development is therefore able to proceed in parallel with, and independent of, either specification or design development.

**Adder example.** Figure 8 shows actual waveforms for an SST protocol test bench implementing the adder abstraction model of Figure 4. This figure displays, in addition to the clock, only those signals participating in the protocol control: **rqtin** and **ackout**. Three different signal values are shown in this figure: **'0'** (low), **'1'** (high), and '–' (don't care). The don't care values are displayed as the intermediate dashed lines in the figure. They represent signals whose values are unconstrained by the protocol.

The required behavior of **rqtin** and **ackout** is defined by the formulas **rqt_sat** and **ack_sat** that were presented in [10].[3] The graphical formula 'samples' captured in Figure 9 provide an intuitive description of this behavior.

The formula **rqt_sat** says that, once **rqtin** rises it should remain high until **ackout** rises. Following this, **rqtin** is required to eventually fall, and then remain low until **ackout** falls. All of this behavior can be observed in the waveform for **rqtin**.

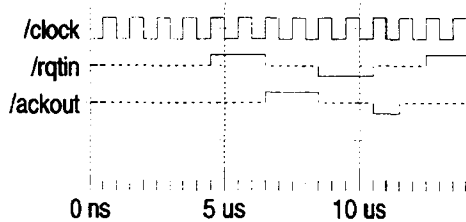


**Figure 8: Simulation Graphics for Adder Protocol Animation.**

3 A minor change has replaced the **While** operator of [10] with one called **Until** that offers virtually the same behavior.

The formula **ack_sat** says that, following the rise of **rqtin**, **ackout** must eventually rise and then remain high until **rqtin** falls. Following the fall of **rqtin**, **ackout** is required to eventually fall itself. This behavior is clearly displayed in the waveform for **ackout**.

The use of don't cares helps to make clear precisely what is and what isn't being specified by VIL formulas. In fact, their use in Figure 8 helps to highlight a specification mistake that existed in our original model of the four-phase protocol. This is the model displayed in the figure.

Recalling the discussion in Section 4.1, the signals **rqtin** and **ackout** are both assumed to be low prior to the rise of **rqtin** to begin a new transaction. Note, however, that **ackout** carries a value of don't care at this time. In other words, **ack_sat** has failed to sufficiently constrain **ackout** to be low. The current version of **ack_sat** includes the requirement that **ackout** remain low until **rqtin** rises.

### 4.5 Step III: Design Verification

The final step in the top-down design methodology described in this paper is to construct and verify an RTL design model against the behavioral specification model. This is done most efficiently using self-testing test benches.

Figure 9 shows the structure of self-testing test benches produced by SST. As shown in the figure, these test benches interface with both the specification model and the design model. Except for the 'certify output' block, all of the test bench internal blocks shown in this figure are similar to those described in earlier sections.

The 'certify output' block compares the results produced by the design model against those produced by the specification model. The specification outputs are taken in directly, while those from the design are first translated to the specification level in the 'map up' block. The results (pass/fail) are recorded into an output file (along with the corresponding specification-level inputs).

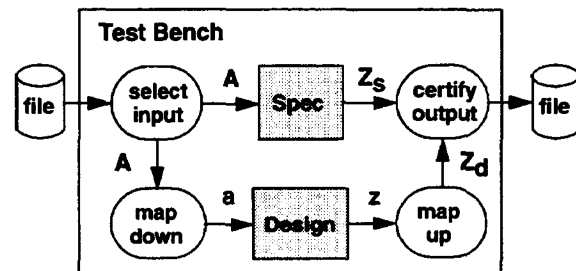


**Figure 9: SST Self-testing Test Bench Structure.**

Achieving meaningful output comparisons requires that the specification and design operate over common inputs. As seen in Figure 9, the 'select input' block indeed passes the same input values to both the specification and design. The values received by the design are first translated, as expected, in the 'map down' block.

**Variability in event times.** Interface protocols generally allow non-determinism in the response times of certain events occurring over the shared interface signals. For example, a protocol might say that an acknowledgement must occur within some bounded time after a request occurs, or perhaps just eventually. It is important that test benches avoid, to the extent practical, the introduction of timing constraints not present in the protocol specification. SST implements event time selection in two ways:

(a) *Random* selection has the test bench select event times using pseudo-random number generation methods.

(b) *Edge-case* selection has the test bench select input event times at the beginnings and endings of the time intervals specified in the interface protocols (in development).

**Adder example.** Figure 10 shows simulator waveforms for a self-testing test bench produced by SST for our adder example. The waveforms show the RTL design signals and cover somewhat more than a single transaction worth of behavior

It is interesting to note the correlation between the test bench-sourced signals and the VIL abstraction model of Figure 4. For example, the waveforms show the **reset** signal being high at time 0 and then low for all times afterwards. This is clearly the RTL implementation of a **SAT**-valued **RST_OP**, as defined by the abstraction in Figure 4. In the context of Figure 9, the 'select input' block selects the only legal value available for **RST_OP** (**SAT**) and then the 'map down' block converts this into the RTL **reset** behavior shown in Figure 10.

Recall that the adder abstraction model defines the specification-level data inputs as integer counterparts to the RTL data sampled at the beginning of the transaction. Looking at these relationships in the usual 'upward' direction, we would expect that, for example, a value of **'0'** for **cin** at transaction start would be equivalent to an integer **0** for **CIN**, and so on.

The waveforms of Figure 10 show the more interesting effects of the downward mapping on specification-level values **0**, **0**, and **0** for **A**, **B**, and **CIN**. Note that the RTL inputs displayed in these waveforms are low only at the start of the transaction and are otherwise don't care.

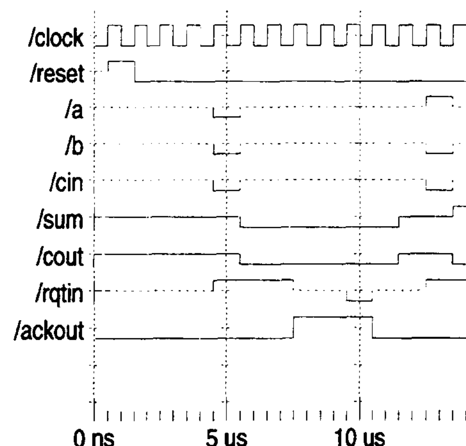


**Figure 10: Simulation Graphics for Adder Design Verification.**

Although this may seem surprising at first, it should be noted that the abstraction model says nothing about the value of the RTL inputs anywhere other than at transaction start. Where the RTL signals are unconstrained by the abstraction, they are driven as don't care by the test bench. Therefore, a faulty design that sampled these inputs on the wrong cycle (and processed don't cares) would very quickly expose itself as faulty.[4]

The design model drives values onto **sum**, **cout**, and **ackout**, thus there are no don't cares present on these signals. The design is seen to correctly implement the transaction in Figure 10 since **sum** and **cout** carry correct values on the rising edge of **ackout**, and **ackout** satisfies its own protocol obligations.

## 5 Discussion

In this paper we have presented a prototype hardware design tool called SST that is the first tool (to our knowledge) that provides automatic synthesis of simulation models serving multi-level simulation environments. This paper demonstrated specific SST support for environments employing RTL design models and behavioral specification models expressed at higher levels of data and temporal abstraction. SST-generated simulation models were described for a number of scenarios including: (i) specification animation, (ii) interface protocol animation, and (iii) autonomous verification of RTL designs against behavioral specifications.

4 SST currently supports the 9-valued **:std_logic** data type.

SST was compared to other test bench tools in this paper and shown to be unique in its use of abstraction models to serve as inputs to the test bench generation process. The information contained in these models specifies the mappings linking the signals of specifications to the signals of their underlying designs. As shown in Figure 9 of Section 4.5, it is precisely the test bench implementation of these mappings that accounts for SST's success in multi-level simulation environments. Existing methods cannot support these environments simply because the information that they process fails to adequately specify abstraction mappings.

Elements of the SST input language VIL (VHDL Interface Language) were presented in this paper. VIL is (to our knowledge) the first abstraction language ever used in a hardware simulation environment. It provides a concise and intuitive syntax that we believe is suitable for applications in industry today, and it has a formal semantics that offers a solid foundation for on-going work in formal verification and behavioral synthesis. VIL therefore provides a common 'bridging' notation to help ease the transition toward future design processes of greater rigor and automation.

Future work on SST will focus on protocol library development and improved user interfaces. Our work on protocol libraries will ensure that SST requires minimal abstraction modeling effort as an extensive set of VIL models will be made available with the tool. We also plan to investigate the integration of graphical timing diagram methods with VIL to better support those applications where it is necessary to design custom protocols. We are currently developing a graphical windows interface for SST to provide an integrated, easy-to-use environment for managing simulation model development.

## 6 Acknowledgements

Thanks to David Pellerin of Accolade Design Automation for suggestions on the presentation of material in this paper.

## 7 References

[1] J.R. Armstrong, et. al., "High Level Generation of VHDL Test Benches," in *VHDL International Users' Forum*, April 1995.

[2] G. Borriello, *A New Interface Specification Methodology and its Application to Transducer Synthesis*, Ph.D. Thesis and Report No. UCB/CSD 88/430, Computer Science Division (EECS), University of California, Berkeley, May 1988.

[3] M.C. Cogswell and C.C. Paynton, "A Mixed-Level Self-Verifying VHDL Simulation Environment with Selective Random Control of Data Transactions," in *VHDL International Users' Forum*, October 1995.

[4] "ESDA Software Adds Enhancements and Links to Verilog Testbench Tool," *Electronic Design*, June 12, 1995.

[5] "Automatically Generate VHDL and Verilog Testbench Models from Digital Timing Diagrams," *Electronic Design*, October 13, 1995.

[6] "Swiss Tool Maker Looks to Fill Top-Down-Design Void," *Electronic Engineering Times*, June 19, 1995.

[7] "Synopsys VSS Automates Regression Testing," *Electronic Engineering Times*, September 11, 1995.

[8] "'Vera' Talks Verification," *Electronic Engineering Times*, September 25, 1995.

[9] "Timing Diagram Editor Displays Waveforms," *Electronic Engineering Times*, October 9, 1995.

[10] D.A. Fura and A.K. Somani, "Transaction-Level Specification of VHDL Design Models," in *VHDL International Users' Forum*, October 1995.

[11] S. Kapor, J.R. Armstrong, and S.R. Rao, "An Automatic Test Bench Generation System," in *VHDL International Users' Forum*, May 1994.

[12] A. Silburt, et. al., "Accelerating Concurrent Hardware Design with Behavioural Modelling and System Simulation," in *32nd Design Automation Conference*, June 1995.

[13] M.F. Sullivan, et. al., "Integrating Hierarchical Test Benches into an Evolving VHDL Design Environment," in *VHDL International Users' Forum*, May 1994.

[14] C.R. Unkle, W.G. Swavely, and J. Nagy, "Development of a WAVES Compatible Testbench for Board-Level Test," in *VHDL International Users' Forum*, April 1995.