Shortening Verification Time Using the CoverAll™ Toolset to Automate Assertion Based Verification

White Paper

April 2010

SOLID OAK TECHNOLOGIES

www.solidoaktech.com

1. INTRODUCTION

Logical and functional bugs continue to be the leading cause of costly silicon re-spins.[1][2] These bugs generally fall into three categories. First, the function wasn't tested. Second, the function wasn't checked. Lastly, the function wasn't tested or checked because it wasn't defined.

Many people have spent countless hours trying to address the first case, testing everything. The problem is not the inability to write enough tests or to randomly stimulate everything. The problem is the ability to define what to test and to prove that it has all been tested. Until now, someone had to translate the high level specification into a verification plan, execute it, and hope there's enough coverage in the plan to ensure a working device.

The second case, checking for correct behavior, usually occurs when one engineer designs a function and another verifies it. The designer fails to convey enough information about how the logic is supposed to work to allow it to be accurately checked. The difficulty is to come up with a way to check the RTL without just rewriting it in a different language. The problem with this approach is obvious; the bug will be in both representations. Until now, this has been the major obstacle in checking functionality.

The last case, not tested and not checked occurs because the function is not defined, the inputs don't function as described and the specification did not define what happens in that case. The designer designs only what's defined and the verification engineer tests only what's defined. But what happens when the inputs do not work as specified? Does a packet get dropped when dropping packets is not allowed OR does it lock up… Oh no! (re-spin bug). Until now, it was up to the architect writing the specification to cover all cases, even the invalid ones.

Solid Oak Technologies has developed a methodology which solves these three issues, using tools designed to automate the solution and shorten the time to verification complete. Today a designer can completely define a function with flow diagrams and timing diagrams utilizing Solid Oak's easy to enter and modify graphics tool, CoverAll™, compile them in seconds, and have a complete set of assertions and coverage statements to verify the function to 100% coverage. How does this address all three issues?

In the first case, testing everything is easily solved by this methodology. Using flowcharts and timing diagrams completely defines every possible path/combination a function can have. CoverAll™ takes this graphical representation and converts it into path coverage statements in either the PSL[3] or SVA[4] language which can be included in simulations or formal verification. Once all of these cover statements have been exercised, the design has 100% functional coverage (i.e. everything tested).

For the second case, checking everything is also easily solved with CoverAll™. Using flowcharts and timing diagrams defines the behavior of every RTL register. The tool takes this graphical representation and converts it into assertions in either PSL or SVAs which can be included in simulations or formal verification. Once all analysis has been run and all assertions pass (not just don't fail), the function is 100% verified (i.e. everything checked).

For the last case, every possibility defined is also easily solved with CoverAll™. Using the graphical flowchart entry tool forces the developer to define all cases, even the elusive "else" case. This means the architect and designer must address how to handle all error cases so that catastrophic failure does not occur when the inevitable "it is not supposed to do that" occurs.

Solid Oak's approach is based on automating the generation of assertions and cover statements from the specification. This methodology puts the effort where it needs to be, on the specification. There are plenty of examples and texts on how to create assertions[5][6] and data showing that design teams aren't using them[7] and why[8]. Automating the generation of assertions and cover statements from the flow diagrams and timing diagrams solves the problems with using assertions. Until now, the process of developing assertions was relatively ad-hoc and time consuming. Even designers experienced with the use of assertions often struggle to determine if they have placed all the right assertions in all the right places.

How does Solid Oak's methodology and CoverAll™ toolset shorten verification time? A typical chip design flow starts with a specification and then design and verification begin in parallel. The design phase usually takes the least amount of time and the verification the most. The verification cycle consists of developing a testbench, predicting the behavior, and checking the behavior. Predicting and checking the behavior takes the majority of that time. With Solid Oak's approach, the testbench must still be developed but the predicting and checking is automated from the specification. This eliminates the time required to develop the predictors and checkers with the added benefit of capturing the exact design intent from the diagrams. When the designer is ready, let the debug begin!

This paper details how to create useful timing and flow diagrams so they may be utilized to document a design's functionality and identify the assertions and cover statements necessary for full functional coverage with the CoverAll™ toolset. In addition, the method will be illustrated by way of a real design example using as a test case the OpenCores I^{2}C controller project[9].

2. Creating Specifications with Useful Visual Diagrams

Wikipedia states **"A picture is worth a thousand words** is a familiar proverb that refers to the idea that complex stories can be told with just a single still image, or that an image may be more influential than a substantial amount of text. It also aptly characterizes the goals of visualization where large amounts of data must be absorbed quickly."

As cited in the Collett Research[2], incorrect or incomplete specifications lead to design flaws and that design complexity and incorrect interface assumptions are the major contributors.

A critical part of the functional specification for any design is the block-by-block detailed description. A well written block-level description should contain three main sections:

1) A Block Diagram and high-level description of the intended functionality

2) Interface Descriptions, with detailed timing diagrams

3) Detailed Functional Description with flow diagrams

The content of the interface descriptions and flow diagrams is vital since these are critical in the creation of a complete functional specification. These graphical representations are useful for many purposes, including the development of behavioral models and the RTL design; however, the main focus of this paper is how to generate a complete set of assertions for functional coverage points. The goal is to make these diagrams "more influential than a substantial amount of text" in an effort to reduce the ambiguity of the design intent and "to tell a complex story with a single image".

2.1 Creating Interface Diagrams

At the block level, Interface Diagrams serve the purpose of defining the protocol that allows two or more blocks to transfer data. For example, in the Opencores I^2C controller core project, a Wishbone compatible interface is utilized to control the core operations. In a traditional functional specification, the following timing diagrams would most likely be found (or simply referenced):

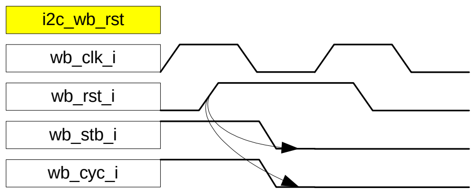


Figure 1 Wishbone Reset Timing Diagram

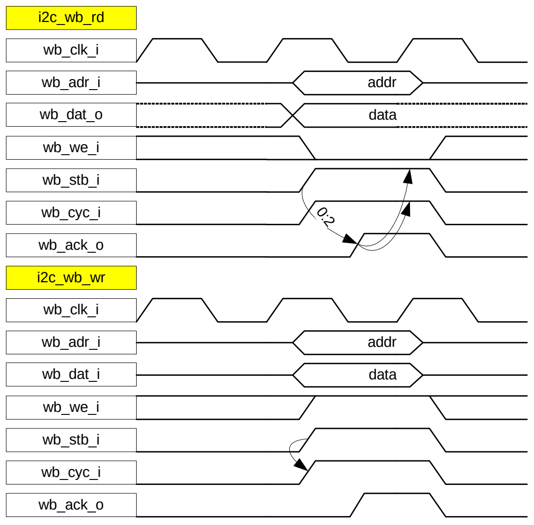


Figure 2 Wishbone Read and Write Cycle Timing

The assertions are automatically extracted with the CoverAll™ TD2Assert module to capture the protocol from the diagrams:

```
property td2a_i2c_top_wb_td_td_assert1;
@(posedge wb_clk_i) ((wb_rst_i) |-> ~(wb_stb_i));
endproperty
assert property(td2a_i2c_top_wb_td_td_assert1);

property td2a_i2c_top_wb_td_td_assert2;
@(posedge wb_clk_i)
((wb_rst_i) |-> ~(wb_cyc_i));
endproperty
assert property(td2a_i2c_top_wb_td_td_assert2);

property td2a_i2c_top_wb_td_td_assert3;
@(posedge wb_clk_i) ((wb_stb_i) |-> ##[0:2] (wb_ack_o));
endproperty
assert property(td2a_i2c_top_wb_td_td_assert3);

property td2a_i2c_top_wb_td_td_assert4;
@(posedge wb_clk_i) ((wb_ack_o) |=> ~(wb_stb_i));
endproperty
assert property(td2a_i2c_top_wb_td_td_assert4);
property td2a_i2c_top_wb_td_td_assert5;
@(posedge wb_clk_i) ((wb_ack_o) |=> ~(wb_cyc_i));
endproperty
assert property(td2a_i2c_top_wb_td_td_assert5);

property td2a_i2c_top_wb_td_td_assert6;
@(posedge wb_clk_i) ((wb_stb_i) |-> (wb_cyc_i));
endproperty
assert property(td2a_i2c_top_wb_td_td_assert6);
```

As seen from the diagrams, assertions flow naturally from the arrows that indicate action/reaction pairs.

However, there is functionality missing from these diagrams. Obviously, since the Wishbone interface is intended to control the operation of the I^2C controller, there are additional timing diagrams needed that portray what occurs when the wishbone interface instructs the core to perform some higher function than a simple read or write. One example of a required diagram is shown in Figure 3.

This figure represents the wishbone cycle required to initiate a start condition for a read or write of the I^2C bus. The sequences and assertions for this functionality are:

```
sequence td2a_i2c_sta_wr;
@(posedge wb_clk_i)
(
((wb_adr_i ==? 3'b100) && (wb_dat_i ==? 8'b1001xxxx) && wb_we_i
&& wb_stb_i && wb_cyc_i && core_en && ~(sta | sto) && ~(rd | wr)
&& ~tip && ~done && ~irq_flag) ##1
(core_en && (sta | sto) && (rd | wr) && ~tip && ~done &&
~irq_flag) ##1
(core_en && (sta | sto) && (rd | wr) && tip && ~done &&
~irq_flag) ##[1:10000]
(core_en && (sta | sto) && (rd | wr) && tip && done && ~irq_flag)
##1
(core_en && ~(sta | sto) && ~(rd | wr) && tip && ~done &&
irq_flag) ##1
(core_en && ~(sta | sto) && ~(rd | wr) && ~tip && ~done &&
irq_flag) ##[1:100]
((wb_adr_i ==? 3'b100) && (wb_dat_i ==? 8'bxxxxxxx1) && wb_we_i
&& wb_stb_i && wb_cyc_i && irq_flag) ##1
(~irq_flag)
);
Endsequence
TD2A_I2C_STA_WR : cover property (td2a_i2c_sta_wr);

property td2a_i2c_top_td1_td_assert2;
@(posedge wb_clk_i)
(((rd | wr) && (sta | sto)) |=> (tip));
endproperty
assert property(td2a_i2c_top_td1_td_assert2);

property td2a_i2c_top_td1_td_assert3;
@(posedge wb_clk_i)
((done) |=> ~(rd | wr));
endproperty
assert property(td2a_i2c_top_td1_td_assert3);

property td2a_i2c_top_td1_td_assert4;
@(posedge wb_clk_i)
((~(rd | wr) && ~(sta | sto)) |=> ~(tip));
endproperty
assert property(td2a_i2c_top_td1_td_assert4);

property td2a_i2c_top_td1_td_assert5;
@(posedge wb_clk_i)
((done) |=> ~(sta | sto));
endproperty
assert property(td2a_i2c_top_td1_td_assert5);
```

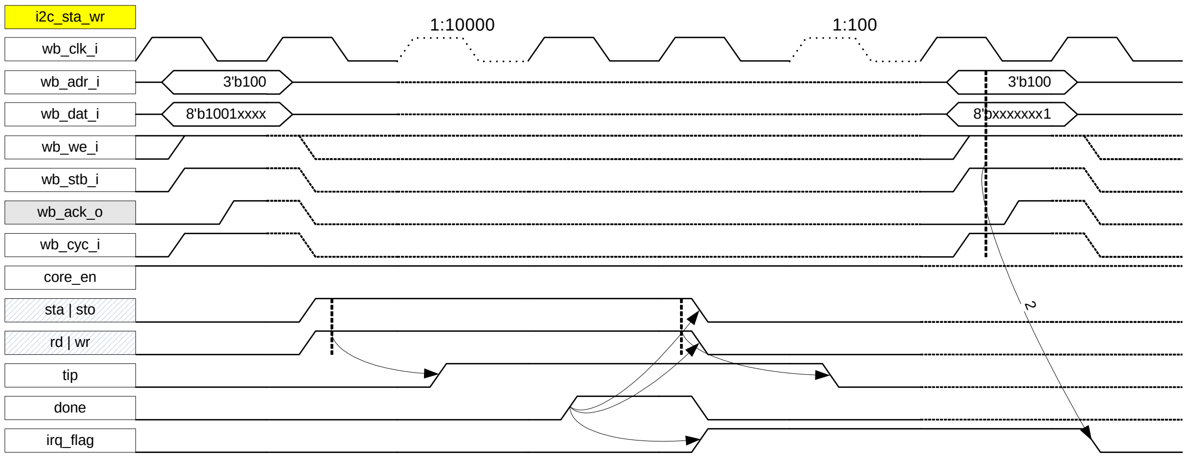


Figure 3 I^{2}C Controller Command Sequence

```
property td2a_i2c_top_td1_td_assert6;
@(posedge wb_clk_i)
((done) |=> (irq_flag));
endproperty
assert property(td2a_i2c_top_td1_td_assert6);

property td2a_i2c_top_td1_td_assert7;
@(posedge wb_clk_i)
(((wb_adr_i ==? 3'b100) && (wb_dat_i ==? 8'bxxxxxxx1) && wb_we_i
&& wb_stb_i && wb_cyc_i) |=> ##2 ~(irq_flag));
endproperty
assert property(td2a_i2c_top_td1_td_assert7);
```

These examples show how timing diagrams can be utilized to not only capture design intent, but also how the CoverAll™ TD2Assert module automatically generates the functional asserts and sequences to prove the intent is properly implemented.

2.2 Creating Flow Diagrams

Simply stated, a flow diagram represents the transfer function from the inputs to the outputs. Refer to the sample flow diagram in Figure 5. The rules and implications represented by the symbols in the flow diagrams are as follows:

- Start Block

  Start Blocks are represented by oval symbols. Each Diagram can have one and only one Start Block.

  The Start block contains the keywords: "clock" and a definition of the reset mechanism: "async_reset" if present.

  A single Arrow connects the start block to the first process block. An embedded signal expression in the arrow indicates a synchronous reset condition.

- Signals

  Signal relationships embedded in Arrows, Process Blocks and Decision blocks follow the Verilog HDL syntax rules.

- Arrows

  Arrows represent the direction of flow between blocks.

  Arrows with signal names embedded represent clock qualifiers.

- Process Blocks

  Process blocks are represented as rectangular shapes.

  Movement between Process blocks is synchronous, i.e. it occurs at the active edge of the clock.

- Decision Blocks

  Decision blocks are represented by diamond shapes.

  All Decision blocks must have "Y" and "N" outcomes.

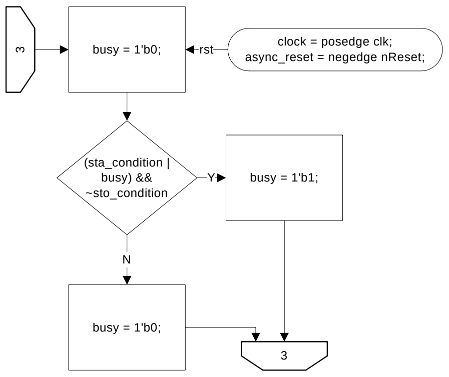
  Movement between Decision blocks is asynchronous, i.e. it does not occur at a clock edge.

  Movement between Decision blocks and Process blocks is synchronous, i.e. it occurs at the active edge of the clock.

The use of these symbols is demonstrated below in the flow diagram for the simple RTL circuit that describes the function of the "busy" signal taken from the i2c_master_bit_ctrl.v of the OpenCores I^{2}C controller source.

```
always @ (posedge clk or negedge nReset)
  if (!nReset)
    busy <= #1 1'b0;
  else if (rst)
    busy <= #1 1'b0;
  else
    busy <= #1 (sta_condition | busy) & ~sto_condition;
```

Figure 4 "busy" Signal RTL Code



3
busy = 1'b0;
rst
clock = posedge clk;
async_reset = negedge nReset;
(sta_condition | busy) && ~sto_condition
Y
busy = 1'b1;
N
busy = 1'b0;
3

Figure 5 "busy" Signal Flow Diagram

The next step is to extract the required functional coverage assertions and path covers from the flow diagram. Each transition between the start block, all process blocks and all decision blocks will map to an assertion. The path to each process block will generate a cover property.

For the example of the “busy” signal above, use Solid Oak’s CoverAll™ FC2Assert module to extract the following SVA[1] assertions based on the diagram of Figure 5:

```
// async reset
property fc2a_busy_arst;
@(posedge clk)
(!nReset |-> busy == 1'b0);
endproperty
assert property(fc2a_busy_arst);
// sync reset
property fc2a_busy_srst;
@(posedge clk)
disable iff(!nReset)
(rst |=> busy === 1'b0);
endproperty
assert property(fc2a_busy_srst);
// property # 1
property fc2a_busy_prop1;
@(posedge clk)
disable iff((!nReset) | (rst))
((((sta_condition |busy) &&~sto_condition)) |=> busy === 1'b1);
endproperty
assert property(fc2a_busy_prop1);
// property # 2
property fc2a_busy_prop2;
@(posedge clk)
disable iff((!nReset) | (rst))
((~((sta_condition |busy) &&~sto_condition)) |=> busy === 1'b0);
endproperty
assert property(fc2a_busy_prop2);
// path covers
FC2A_BUSY_PATH_1 : cover property (@(posedge clk)
(((sta_condition |busy) &&~sto_condition)));
FC2A_BUSY_PATH_2 : cover property (@(posedge clk)
(~((sta_condition |busy) &&~sto_condition)));
```

Note that the FC2Assert module generates:

- One asynchronous reset assertion.
- One synchronous reset assertion.
- One assertion for each process block (2 total).
- Two path properties for the logic paths to each of the process blocks.
- Automatically injects the disable conditions

Although this is a very simple circuit, it requires four functional coverage assertions and two path covers. As you review these assertions, note that the number and complexity of the assertions and path covers required for thorough coverage will expand with the complexity of the design block. The ability to generate these coverage assertions and path covers automatically for the diagrams is a huge time saver. That leaves time to work on the design specification, i.e. creating the flow and timing diagrams which good design practice dictates you should have be doing in the first place.

3. EXAMPLE: I^{2}C CONTROLLER

This section will illustrate the benefits of applying the CoverAll™ methodology, by way of an example, using as a test case the OpenCores I^{2}C interface controller project[9]:

- the process of creating a visual flow diagram and extracting the design assertions for a complete module, and
- the benefit of analyzing an imported design (14% of imported designs contain bugs[2])

[1]Equivalent assertions can be automatically generated in the Property Specification Language (PSL)

3.1 The I^{2}C Architecture

I^{2}C is a two-wire, bidirectional serial bus that provides a simple method of data exchange between devices[10]. The I^{2}C Controller module (i2c_master_top) drives and responds to the external two-wire I^{2}C interface to control data transfer according to the standard protocol. This design contains 3 modules:

- i2c_master_top – top level module that includes the Wishbone interface and core registers (status and control) and instantiates the i2c_master_byte module
- i2c_master_byte_ctrl – contains the byte level i2c state machine, i2c bus shift reqister and instantiates the i2c_master_bit_ctrl module
- i2c_master_bit_ctrl – contains the bit level i2c state machine as well as the clock generator, bus synchronizers, and arbitration logic

3.2 Diagrams

Covering the three modules requires the following number of diagrams to be created:

	Diagrams	
Module	**Flow**	**Timing**
i2c_master_top	3	9
i2c_master_byte_ctrl	5	-
i2c_master_bit_ctrl	7	1

The figures below show the state and logic flow diagrams for the i2c_master_byte_ctrl module. The outputs of the state machine diagram are not shown.

3.3 Assertions

The numbers of assertions automatically generated to cover the three modules is shown in the table below:

	Assertions				
	Flow Diagrams			Timing Diagrams	
Module	**Prop**	**Seq**	**Path**	**Prop**	**Seq**
i2c_master_top	55	-	25	22	20
i2c_master_byte_ctrl	63	9	60	-	-
i2c_master_bit_ctrl	103	5	40	-	5

Listed below are some of the assertions for the flow diagram of Figure 6.

```
// async Reset
property fc2a_i2c_byte_sm_c_state_arst;
@(posedge clk)
(!nReset |-> c_state == ST_IDLE);
endproperty
assert property(fc2a_i2c_byte_sm_c_state_arst);
// sync reset
property fc2a_i2c_byte_sm_c_state_srst;
@(posedge clk)
disable iff((!nReset))
(rst | i2c_al |=> c_state == ST_IDLE);
endproperty
assert property(fc2a_i2c_byte_sm_c_state_srst);
// state transition
property fc2a_i2c_byte_sm_fsm_trans2_ST_START_to_ST_READ;
@(posedge clk)
disable iff ((!nReset) | (rst | i2c_al))
((c_state==ST_START && (core_ack) & (read)) |=> (c_state ==
ST_READ));
```

```
endproperty
assert property(fc2a_i2c_byte_sm_fsm_trans2_ST_START_to_ST_READ);
// state sequence
sequence fc2a_i2c_byte_sm_fsm_seq0;
@(posedge clk)
(
((c_state==ST_IDLE) && (go) && (start) ##1 (c_state==ST_START))
##[1:$]
((c_state==ST_START) && (core_ack) && (read) ##1
(c_state==ST_READ)) ##[1:$]
((c_state==ST_READ) && (core_ack) && (~(|dcnt)) ##1
(c_state==ST_ACK)) ##[1:$]
((c_state==ST_ACK) && (core_ack) && (stop) ##1
(c_state==ST_STOP)) ##[1:$]
((c_state==ST_STOP) && (core_ack) ##1 (c_state==ST_IDLE))
);
endsequence
// state holding property
property fc2a_i2c_byte_sm_fsm_hold_0;
@(posedge clk)
disable iff((!nReset) | (rst | i2c_al))
((c_state==ST_STOP) && ~(core_ack) |=> (c_state==ST_STOP));
endproperty
assert property (fc2a_i2c_byte_sm_fsm_hold_0);
// Legal States property
property fc2a_i2c_byte_sm_c_state_fsm_legal_states;
@(posedge clk)
c_state inside {ST_ACK, ST_IDLE, ST_READ, ST_START, ST_STOP,
ST_WRITE};
endproperty
assert property (fc2a_i2c_byte_sm_c_state_fsm_legal_states);
// One Hot property
property fc2a_i2c_byte_sm_c_state_one_hot;
@(posedge clk)
$onehot0(c_state);
endproperty
assert property (fc2a_i2c_byte_sm_c_state_one_hot);
```

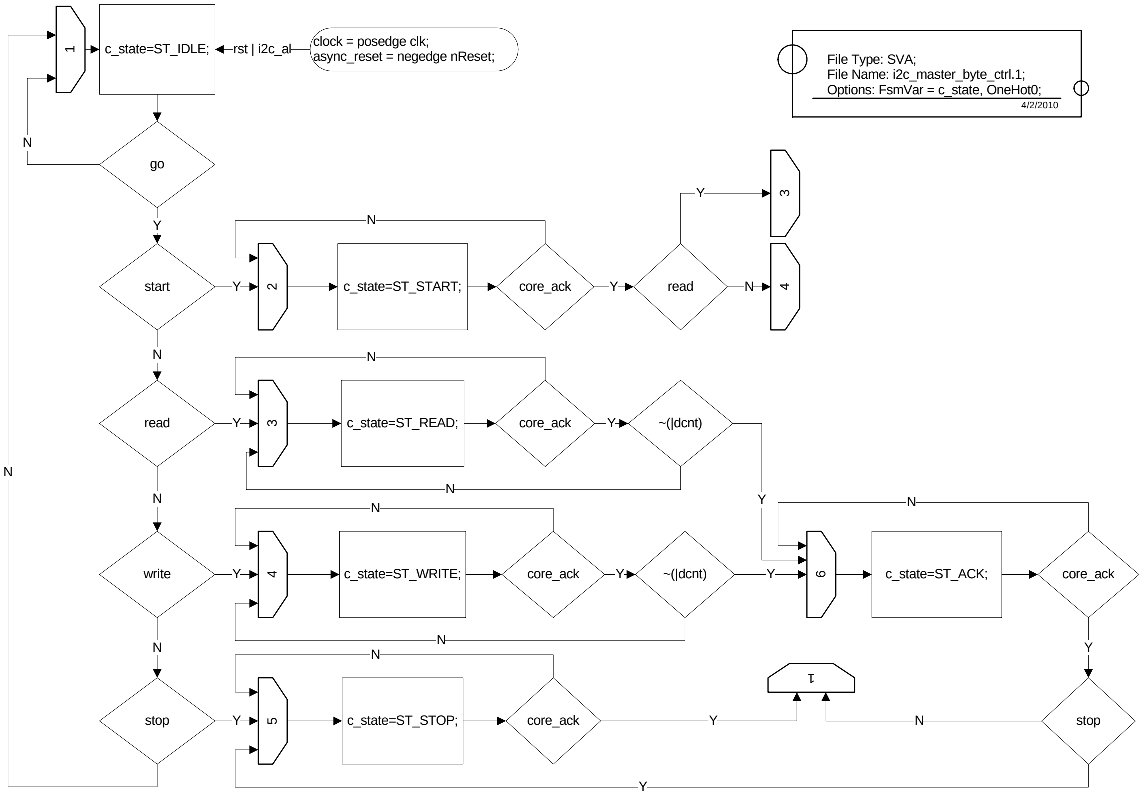


Figure 6 I^{2}C Byte Controller State Machine

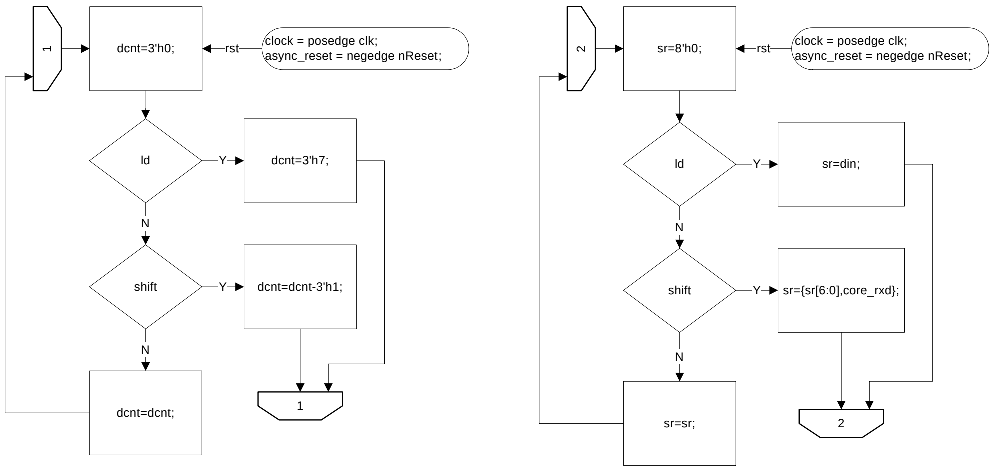


Figure 7 I^{2}C Byte Controller Logic

Listed below are the Assertions and Covers for the "dcnt" flow diagram in Figure 7.

```
// async reset property for signal dcnt
property fc2a_dcnt_arst;
@(posedge clk) (!nReset |-> dcnt == 3'h0);
endproperty
assert property(fc2a_dcnt_arst);
// sync reset property
property fc2a_dcnt_srst;
@(posedge clk) disable iff(!nReset)
(rst |=> dcnt === 3'h0);
endproperty
assert property(fc2a_dcnt_srst);
// property #1 for signal dcnt
property fc2a_dcnt_prop1;
@(posedge clk) disable iff((!nReset) | (rst))
(((shift) & ~(ld)) |=> dcnt === $past(dcnt)-3'h1);
endproperty
assert property(fc2a_dcnt_prop1);
// property #2 for signal dcnt
property fc2a_dcnt_prop2;
@(posedge clk) disable iff((!nReset) | (rst))
(((ld)) |=> dcnt === 3'h7);
endproperty
assert property(fc2a_dcnt_prop2);
// property #3 for signal dcnt
property fc2a_dcnt_prop3;
@(posedge clk) disable iff((!nReset) | (rst))
((~(shift) & ~(ld)) |=> dcnt === $past(dcnt));
endproperty
assert property(fc2a_dcnt_prop3);
// cover properties for signal dcnt
FC2A_DCNT_PATH_1 : cover property (@(posedge clk) ((shift) &
~(ld)));
FC2A_DCNT_PATH_2 : cover property (@(posedge clk) ((ld)));
FC2A_DCNT_PATH_3 : cover property (@(posedge clk) (~(shift) &
~(ld)));
```

3.4 Simulation Results

Simulating with the original test bench from the OpenCores site, the following coverage statistics can be obtained.

Table 1 - Initial Coverage Metrics

Coverage	i2c_master_top	i2c_master_byte_ctrl	i2c_master_bit_ctrl
Statement	35/47 (74.5%)	56/67 (83.8%)	124/140 (88.6%)
Branch	30/37 (81.1%)	48/52 (92.3%)	54/61 (88.5%)
UDP Expression	18/23 (78.3%)	9/9 (100%)	24/42 (57.1%)
UDP Condition	5/6 (83.3%)	1/3 (33.3%)	22/24 (91.7%)
FEC Expression	22/34 (64.7%)	14/14 (100%)	22/22 (100%)
FEC Condition	6/8 (75.0%)	2/4 (50%)	40/42 (95.2%)
FSM State	N/A	6/6 (100%)	18/18 (100%)
FSM Transition	N/A	12/13 (92.3%)	22/22 (100%)

As seen in the table, the code coverage is low. In addition, by examining the missing code coverage a number of items were noted:

1. The test bench tests the I^{2}C in the combined format, however, it always does a write (slave internal memory address) after a start or a repeated start. Thus the start followed by a read never occurs and is not covered in the code coverage.
2. The tip signal, which indicates transfer in progress, is not activated by the sto (STOP) bit in the CR (command) register. This appears to be a bug in the design.
3. No arbitration is attempted.
4. The I^{2}C slave device in the testbench issues a timing warning.
5. The startup time after the wishbone interface initiates an I^{2}C start is very long.

Could these issues have been avoided entirely or at least discovered quickly? Let's look at item #2 above. If the designer had written a flow diagram as shown in Figure 8, i.e. with no "sto" signal in the tip logic and

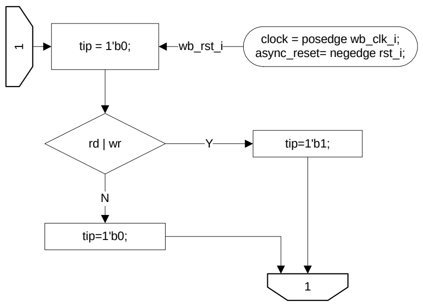


Figure 8 - "tip" Logic Diagram

in addition created the Timing Diagram shown in Figure 9 below, the oversight would have been caught by the assertions generated by the CoverAll™ TD2Assert module and circled in red (shown below).

```
property td2a_i2c_top_td2_td_assert1;
@(posedge wb_clk_i)
((~(rd | wr) && sto) |=> (tip));
endproperty
assert property(td2a_i2c_top_td2_td_assert1);
```

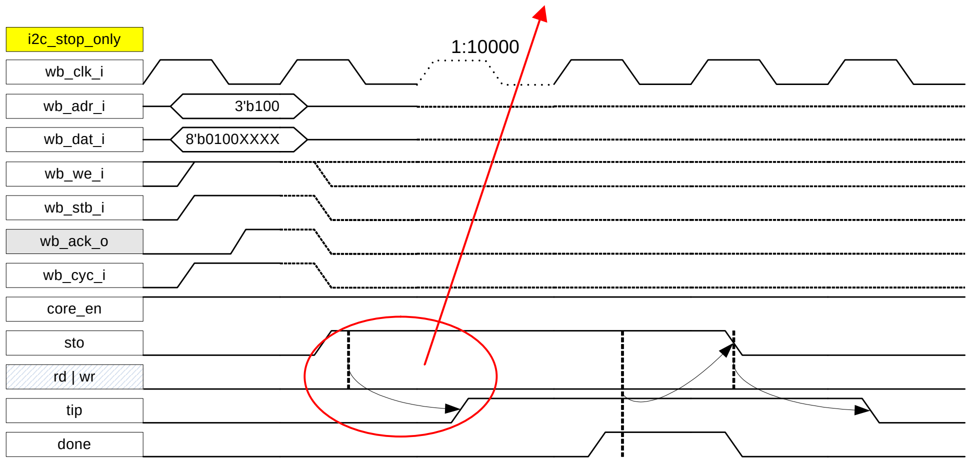


Figure 9 - Timing Diagram for a "Stop Only" Condition

This example points out the need to clearly define what the Design Intent is whether in Flow Diagrams or Timing Diagrams or both.

3.5 Improving the Test Bench

Can't we just fix the testbench to improve the Code Coverage and the Functional Coverage will follow? Let's see. The following modifications were made to the testbench to improve coverage:

- Reading and writing all registers.
- Enabling and servicing the interrupt logic.
- Asserting the "wb_rst_i" synchronous reset.
- Expanding for a two master mode and forcing an arbitration collision.

The following modifications were made to the RTL after creating flow and timing diagrams for each change:

- added the "sto" signal to the tip logic.
- fixed the bit controller state machine to properly generate a repeated start.
- fixed the slave wait logic to properly implement clock stretching.
- fixed the slow startup timing from first wishbone write to data on the I^2C bus.

The final coverage is shown below (functional coverage is shaded).

Coverage	i2c_master_top	i2c_master_byte_ctrl	i2c_master_bit_ctrl
Statement	47/47 (100%)	67/67 (100%)	162/162 (100%)
Branch	37/37 (100%)	52/52 (100%)	81/81 (100%)
UDP Expression	24/24 (100%)	9/9 (100%)	27/46 (58.7%)
UDP Condition	6/6 (100%)	3/3 (100%)	9/9 (100%)
FEC Expression	32/36 (88.9%)	14/14 (100%)	25/26 (1007%)
FEC Condition	7/8(87.5%)	4/4 (100%)	12/12 (80.9%)
FSM State	N/A	6/6 (100%)	20/20 (100%)
FSM Transition	N/A	13/13 (100%)	25/25 (100%)
Directive	41/45 (91.1%)	67/69(97.1%)	47/50 (94.0%)
Assertions	76/77 (98.7%)	62/63(98.4%)	103/103(100%)

Table 2 – Final Coverage Metrics

Code Coverage looks really good, right? All of the missing coverage is the result of:

- An RTL construct, "|c_state", in the I^2C arbitration logic. We will need to write more tests to get this to trigger in all states.
- The fact that the Wishbone interface signals 'wb_stb_i' and 'wb_cyc_i' are always active together. We can ignore this.
- Once signal 'core_en' is set, it stays set. We can ignore this.
- Redundant logic in the following statement (scl_oen is always high in states rd_c and wr_c):

  ```
  slave_wait_in = scl_oen && !sSCL && ((c_state==rd_c) |
  (c_state==wr_c));
  ```

  we can fix this by removing the scl_oen term.

Was the code coverage a good indicator of functional coverage?

Let's examine the functional coverage obtained by extracting the assertions, paths and sequences from the flow and timing diagrams generated for this design. The missing functional coverage shows:

1. A "NACK" response was never tested during an I^2C read or write command.
2. Arbitration loss was only tested after a start condition and never occurred during the subsequent or last write of a transfer.
3. The byte controller state machine allows the IDLE->START->READ and IDLE->READ transitions but these were never tested.
4. Arbitration can be detected when the bit controller state machine is in states "wr_c" or "wr_d" but was never tested in state "wr_d".

These items, with the exception of #3, would never have been discovered by code coverage nor would the fixed logic have been proven correct without the added assertions, path and functional coverage generated by the CoverAll™ TD2Assert and FC2Assert modules.

4. Formal Verification

One additional benefit of automatically generating design properties and coverage points with assertions is the ability to leverage formal verification. Formal verification can obtain greater coverage of design state space, and quicker, than simulation alone.

Assertions (assert statements) that pass and are covered in simulation provide some level of confidence that the design is working as intended. However, coverage of assertions in simulation is limited to that achieved by the finite range of stimulus that is applied to the design by the robustness of the test environment as is seen by the limited coverage of the arbitration functionality in the I^2C test case. In contrast, formal verification can exhaustively explore all possible legal input stimulus combinations and sequences. In many cases, this results in a valuable proof that the design will always work as intended in any environment. A proof is tantamount to 100% coverage with respect to a particular property of the design. Also, counterexamples can be formally generated, which demonstrate a sequence of input combinations that can cause the design to malfunction. Due to the exhaustive nature of formal analysis, the counterexamples which are generated often involve corner case events that are not likely to be discovered in simulation, and thus can highlight bugs that would otherwise go undetected.

4.1 Proofs

In this example, the three modules of the I^2C controller have a total of 365 target assertions. The formal tool[11] reported a solution for all but 14 of these properties after a 4 hour run of the proof engine and two(2) 24 hour runs of the deep counterexample engine.

```
---------------------------------------
Check Summary                     Total
---------------------------------------
Check Constraints                    19
Statically Proven                   238
  Vacuous                            55
  Maybe Vacuous                      32
Inconclusive Targets                 14
  Bounded Proofs                      0
  Timeout                            14
Fired Targets                       113
---------------------------------------
Total Checks                        384
---------------------------------------
```

In order to achieve these proofs, a few additional assertions were used to act as additional constraints defining the legal range of inputs to be explored by the formal engine.

```
// Make sure clock divisor is 0001
property CLK_DIV0;
  @(posedge wb_clk_i)
  ((wb_cyc_i && wb_stb_i && wb_we_i && (wb_adr_i == 0)) ->
(wb_dat_i == 8'h01));
endproperty
property CLK_DIV1;
  @(posedge wb_clk_i)
  ((wb_cyc_i && wb_stb_i && wb_we_i && (wb_adr_i == 1)) ->
(wb_dat_i == 8'h00));
endproperty
// once a command is issued, no new commands (wishbone writes)
until this one finishes
property CMD_HOLD;
  @(posedge wb_clk_i)
  ((tip) -> (!wb_we_i));
endproperty

// except for arbitration scl_pad_i should follow scl_padoen_o
property SCL_I_follows_SCL_PADOEN_O_1;
  @(posedge wb_clk_i)
  ((scl_padoen_o) -> (scl_pad_i));
endproperty
property SCL_I_follows_SCL_PADOEN_O_0;
  @(posedge wb_clk_i)
  ((!scl_padoen_o) -> (!scl_pad_i));
endproperty

// except for arbitration sda_pad_i should follow sda_padoen_o
property SDA_I_follows_SDA_PADOEN_O_1;
  @(posedge wb_clk_i)
  ((sda_padoen_o) -> (sda_pad_i));
endproperty
property SDA_I_follows_SDA_PADOEN_O_0;
  @(posedge wb_clk_i)
  ((!sda_padoen_o) -> (!sda_pad_i));
endproperty

assume property (SCL_I_follows_SCL_PADOEN_O_1);
assume property (SCL_I_follows_SCL_PADOEN_O_0);
assume property (SDA_I_follows_SDA_PADOEN_O_1);
assume property (SDA_I_follows_SDA_PADOEN_O_0);
assume property (CLK_DIV0);
assume property (CLK_DIV1);
assume property (CMD_HOLD);
```

These ensure that false counterexamples are not generated through unintended stimulus. For example, the CLK_DIV0 and CLK_DIV1 ensure the I^{2}C's clock divisor is set to 0x0001, not 0x0000 as the formal tool would like to shorten the proof radius. A clock divisor of 0x0000 creates unintended and incorrect behavior of the design.

To facilitate coverage on the arbitration based covers, an additional 4 hour run of the proof engine and 12 hour run of the deep counterexample engine was done with the following constraints removed (to allow collisions).

```
assume property (SCL_I_follows_SCL_PADOEN_O_1);
assume property (SCL_I_follows_SCL_PADOEN_O_0);
assume property (SDA_I_follows_SDA_PADOEN_O_1);
assume property (SDA_I_follows_SDA_PADOEN_O_0);
```

This resulted in all but six(6) of the original 365 targets having been proven or covered. These six are all mutli-clock sequence events.

4.2 Sanity Waveforms

When proofs are reported by formal tools, it is important to ensure that the design constraints are not too restrictive. Sanity waveforms help with that task.

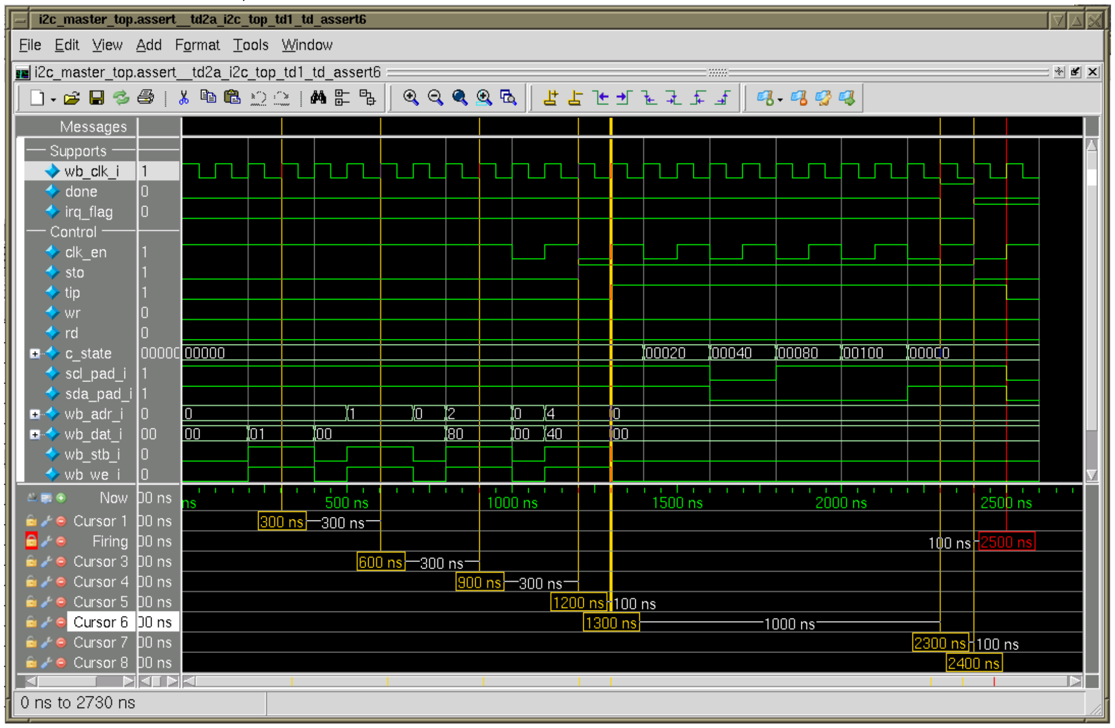


Figure 10 Sanity waveform for timing diagram property td2a_i2c_top_td1_td_assert6

Properties involving the implication operator, taking the form:

antecedent -> consequent

For any assertion of this type, the formal tools can generate a stimulus sequence that causes the antecedent event to occur, followed by the consequent within the expected response time.

Consider the following property (the bottom one, shown in red) from the waveform of Figure 3:

```
property td2a_i2c_top_td1_td_assert6;
@(posedge wb_clk_i)
((done) |=> (irq_flag));
endproperty
assert property(td2a_i2c_top_td1_td_assert6);
```

This property states that signal “irq_flag” will go active immediately after the “done” signal activates as is shown in Figure 3. The sanity waveform generated by the tools is shown in Figure 10. Here we see that the formal tools did the following to hit the coverage target:

- At 300ns and 600ns the clock divider is set to 0x0001 by writes to addresses 0x0 and 0x1.
- At 900ns the “core_en” signal is set by a write of 0x80 to address 0x2.
- At 1200ns a stop only cycle is initiated by a write of 0x40 at address 0x4 which activates the I^{2}C controller state machines.
- At 1300ns the “tip” signal goes active indicating a transfer is in progress.
- At 2300ns the “done” signal goes active and the antecedent condition is satisfied
- In the next cycle, at 2400ns, the consequent event occurs when the “irg_flag” signal goes active.

This behavior matches our original timing diagram. Reviewing the sanity waveforms for all the proven design properties is useful to gain confidence that the constraints are not too tight and the formal tools are achieving meaningful coverage of the state space. Any reported proof is suspect when sanity waveforms cannot be generated.

4.3 Coverage Points

Coverage points (cover statements) are also targets of formal verification. The formal tools can generate the required pattern of stimulus that exercises the design to hit a coverage point. In this example, there are 125 cover statements and each of them was covered by formal methods. However, in order to apply formal to some of these coverage points, some of the sequences defined for simulation had to be modified. The modification necessary was to change the temporal constructs such as [1:10000] to [1:$] and the comparison operator “==?” to “==”.

When the formal tools can hit the coverage points, it again provides valuable coverage feedback from the formal analysis. If all coverage points can be hit without finding counterexamples or with reporting proofs of all assertions, then it indicates that the formal tool has been configured to achieve maximum coverage of the state space – the design has been explored to its corner cases without resulting in a failure.

In the I^{2}C example, a handful of cover statement were not covered by the original testbench. The formal tool covered them and generated the waveforms similar to the sanity waveforms described earlier so that the coverage of the event can be visualized.

5. Conclusions

With designs and design teams becoming larger, starting RTL coding before completing a functional specification increases the program risk. It becomes much more efficient to spend the time early in the design cycle to develop a useful functional specification by utilizing the interface and flow diagram techniques described in this paper. The CoverAll™ toolset can then be employed to extract the necessary functional assertions, sequences and path covers so that the design team is assured that the design intent is properly captured and verified.

Verification teams will gain the advantage of having immediate feedback when issues occur not when they propagate to the output pins of the DUT.

A quality set of assertions provides a means for effective measurement of functional coverage in simulation and enhancement of coverage using formal methods to counter the growing complexity and improve the success rate of silicon design teams.

6. ACKNOWLEDGMENTS

Richard Herveille, designer and maintainer of the I^{2}C OpenCores project.

Roger Sabbagh, of Mentor Graphics, for his help in the explanation of the formal verification portion of this paper.

7. REFERENCES

[1] From the article “Design For Verification–Blueprint for Quality and Productivity”, in the June 2003 Synopsys Compiler by Rindert Schutten

[2] Collett International Research Inc., 2004 IC/ASIC Functional Verification Study, 1/2005

[3] Property Specification Language (PSL), Accellera/IEEE, www.eda-stds.org/ieee-1850/www.eda.org/vfv

[4] SystemVerilog Assertions (SVA), Accellera/IEEE, www.systemverilog.org

[5] Assertion Based Design 2nd Edition, H. Foster et.al., Kluwer Academic Publishers, 2004

[6] Using PSL/Sugar for Formal and Dynamic Verification, Cohen et al., VhdlCohen Publishing, 2004

[7] Deep Chip, DVcon 07 Item 13, 05/24/07

[8] From the article “SystemVerilog Gains A Foothold In Verification”, in the May 2006 Electronic Design by David Maliniak

[9] OpenCores I^{2}C Project [rev 57] http://www.opencores.org/projects.cgi/web/i2c/overview

[10] The I^{2}C-Bus Specification http://www.nxp.com/acrobat_download/literature/9398/39340011.pdf

[11] Mentor Graphics 0in Formal Tools, V2.6n