Using Formal Verification to Exhaustively Verify SoC Assemblies

Kenny Ranerup
ST-Ericsson
Lund, Sweden
Kenny.Ranerup@stericsson.com

Mark Handover
Mentor Graphics Corp.
Newbury, UK
Mark_Handover@mentor.com

Abstract— **State-of-the-art system on chip (SoC) design teams are being continually challenged to work at higher levels of productivity. The reasons why include increased design complexity and shrinking time-to-market windows. Functional verification continues to be the bottleneck in the development schedule, even as dynamic simulation methodologies have been much advanced.**

Faced with these challenges on a recent project, a group within ST-Ericsson decided to use formal verification to check an SoC assembly (that is, to do connectivity checking). This paper will detail the approach, the success of which depended on two key factors: 1) Automated creation of assertions, which in turn hinged on the regularity of many design features and the availability of specifications in machine readable form, and 2) Low sequential depth of the connectivity checks, which made it possible to achieve formal proofs at the chip level when combined with formal tools that can handle complete SoC designs.

The group demonstrated that creating and verifying properties with formal tools required less effort than using simulation alone, thus allowing for expanded verification scope.

One way the effectiveness of this approach was illustrated: during verification, a large number of bugs were found ranging from simple connectivity to interface bugs.

Keywords—formal verification; connectivity; SoC; assembly

I. INTRODUCTION

SoC assembly verification, often referred to as connectivity checking, is among the most time-consuming aspects of SoC verification. Connectivity checking asks the question: “Are all my design elements assembled correctly?” More precisely, it verifies that connections between the blocks of logic in a design are correct; for example, that an output port on one block is correctly connected to its target. But this task is not limited to port checking. It may also encompass pad checking, where complex multiplexing is likely to be employed, and clock checking, where it’s critical to ensure the correct clocks propagate to their intended destination under all modes of operation (mission as well as DFT modes).

For today’s SoC designs, connectivity checking is a daunting assignment given the large numbers of design blocks, complex clock structures, configurable bus-fabrics and point-to-point connections that can number in the thousands. This complexity presents an enormous challenge for the verification engineer, who will typically develop directed tests to toggle all the design signals of interest and then debug to determine why signal values did not transition or propagate as expected.

SoC debug can be particularly challenging. Often, when running dynamically at the SoC level, the result of incorrect IP integration may not show up for many cycles and will manifest itself as an incorrect value at the SoC outputs. Tracing this back to the source of the erroneous connectivity may require a huge amount of effort, which is only partially lessened by placing assertions in the IPs [1].

SoC integration issues impact simulation bring-up time. Given that a high percentage of integration problems are pure connectivity errors [2], verification of the SoC assembly is a crucial step.

Until recently, the modem group at ST-Ericsson approached connectivity verification via simulation. The group tried to write a series fairly complex testcases to toggle all signals between blocks in the design and then tried to track down reasons why certain signals didn’t toggle correctly. Developing the test cases and debugging them were incredibly time consuming processes, a fact that pushed the team to look for an alternative approach.

The group began a project to perform connectivity checking using formal verification techniques [3]. The result was exhaustive verification of the SoC interconnect using formal verification, thus eliminating the requirement for dynamic testing and dramatically reducing time to coverage closure.

Key elements of the success outlined in this paper included automatic property creation and an ability to analyze the complete design using formal verification tools.

II. VERIFICATION APPROACHES

There are several approaches to verifying connectivity using simulation. One is writing direct testcases trying to toggle the signals to be checked. However, setting up scenarios that can exercise a certain path can be time consuming, requiring complex and lengthy device configuration. Another issue with this approach is that it is difficult in simulation to determine when two signals are connected. Toggling source and destination signal at the same point in time is not sufficient to verify the connectivity since other signals can be involved. An alternative approach, one that unfortunately shares many problems with direct testing, is the use of constrained random

verification. At a subsystem level, this approach is often inefficient. For example, toggling all signals on a bus to see that all bits are connected and not swapped is very difficult when the communicating blocks cannot be directly controlled and observed.

At ST-Ericsson software driven verification addresses a large part of the subsystem verification. That is, software executing in the on-chip CPUs are used to stimulate and check the design. A major drawback is that the software normally can't observe behavior at signal level, which makes it unsuitable for connectivity verification.

There are also tools that can verify the structural connectivity of signals by traversing the design to determine if there are paths from source to destination points. This approach can be used for simple cases when there's no logic or sequential elements in the path but are otherwise unsuitable for this type of verification.

Formal verification compares favorably to these approaches. A big advantage is that there's no need to develop testcases for toggling signals. The formal analysis will implicitly analyze all possible signal combinations. Formal proofs also remove uncertainty about whether all cases have been covered. Setting up the device in the correct mode is often straightforward since the use of constraints can bypass lengthy setup sequences. Furthermore, the debug process is much more efficient. Only the signals involved in a failure are shown and the sequence that leads up to the failure is often minimal. And there is usually no need to develop any sort of testbench.

III. LTE Modem Design Description

ST-Ericsson develops cutting-edge mobile platforms and semiconductors across the broad spectrum of wireless technologies. Digital baseband SoCs such as the NOVATHOR™ DB8540 consist of a multi-mode wireless modem combined with application processors. The SoCs are used in mobile handsets such as Android phones, in which the modem handles the digital processing for the mobile communication and the application processor handles the user interface and Android applications. The multi-mode modems support a wide set of mobile communication standards: GSM, LTE FDD/TDD, HSPA+, TD-SCDMA, EDGE. The focus for this paper is the modem subsystem of these SoCs, though the approach is valid for a broad spectrum of SoC designs.

The modem design consists of a large number (~50) of IP blocks mostly performing computational functions that communicate through a special purpose communication network. A number of processors control modem functionality. Blocks are connected to a multi-level bus interconnect through which the processors can control the blocks. The bus interconnect is also used for high bandwidth mobile data (e.g. 150 Mb/s LTE). The implementation language is a mix of VHDL and Verilog.

The design is divided into several power domains and advanced dynamic clock gating is used throughout the design to reduce power consumption.

The design is memory intensive and contains 400 memory instances.

IV. Verification Flow

The first RTL verification is performed at the block level using coverage-driven constrained random simulation. This verification covers all block functionality and is tied to system design through reference models used in the block-level testbenches.

Subsystem verification (on several levels, all the way up to SoC level) then focuses on these questions:

- Are the blocks correctly connected?
- Does the subsystem meet performance and power requirements?
- Does the communication within the subsystem work?
- Does the communication outside of the subsystem work?
- Do the parts of the subsystem work together in the different major modes of operation (different power modes, mission/DFT-mode)?

Connectivity checking is separated from the other verification tasks for efficiency reasons. First, it requires a different dedicated verification solution. Second, the connectivity verification should be run very early in the subsystem verification process. Checking such things as clocks and resets early is crucial since problems in these areas can block large parts of the overall verification task.

Formal tools have many advantages in addressing connectivity verification. Such tools generally have little need for testbenches and are simple to set up and run checks, thus making formal verification fast compared to other techniques. Formal connectivity checking is therefore the first step in the subsystem verification and often run in parallel with other verification tasks. It is further part of the regression flow for each new subsystem release and typically completes long before other subsystem verification tasks.

V. Connectivity Property Set

A prerequisite for using formal property checking is having a complete property set, which conveniently is also required for verifying an SoC assembly.

In any SoC design, various types of connectivity may exist that require checking. However, simple point-to-point connectivity is most common.

A complex part of the interconnect is the reset and clock distribution, which contains logic such as muxes and clock gating. The implementation of this logic is distributed over the design and therefore not possible to verify at block level.

A similar system-wide interconnect is implemented for the DFT functions such as operation of memory BIST controllers and the control of clock and reset distribution.

A. *Point-to-point connectivity*

A large part of the connectivity checks in general include the following types of point to point connectivity:

- Unconditional point-to-point connectivity
- Point-to-point with delay
- Point-to-point with condition

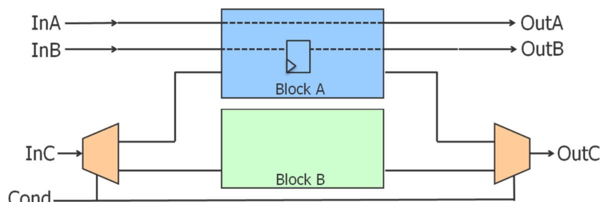

Figure 1. Common types of connectivity checks

These common types of connectivity are illustrated in figure 1 by the example paths: InA→OutA for unconditional point-to-point, InB→OutB for point-to-point with delay and InC→OutC for point-to-point with condition.

There could of course be many more unique types of combinations of the above. Most often the connectivity is between a point on the top level of the chip and a port on a block some levels of hierarchy below. This allows the path of the check to propagate through multiple blocks, which inherently tests connectivity at many nodes in the design. Typically the form of the property is simple and so is a good candidate for automation.

B. *Reset connectivity*

After integration it's crucial to verify that resets are working. Setting up simulations for this can be very time consuming.

Formal verification of the reset function can verify that reset signals are correctly connected (i.e., that the reset source controls the correct block/subsystem reset). It can also verify the correct reset values on the outputs of a block/subsystem. With a single reset this is of course not a difficult simulation problem. But in today's complex SoCs with many power and clock domains and complex IPs with separate resets, this becomes a non-trivial task.

Assuming there is no logic involved, resets can be treated as any other point-to-point connectivity checking. However, there can be multiple reset sources controlling a block's reset. For example, the reset source can be a combination of input pins, power on resets, software controlled resets and other functional resets. The reset sources are then combined through some logic to control a block's reset. Formal verification of this logic is no more difficult than connectivity checking. What is required is identifying all reset sources, the points that should be controlled by each reset source and finally the function of the reset logic, such as muxing for separate control of reset during DFT modes. After that, it is a simple matter of writing assertions that check that when a reset source is activated, the reset on a block is also activated.

The method assumes that resets are asynchronously activated, which is the normal case. If resets are clocked—for example, software-activated resets crossing a clock domain—it becomes more complex. Either the sequential behavior is not verified or clocks must be set up and the sequential behavior must be described in assertions or constraints.

In addition to checking the connectivity of resets, the verification is easily extended to check the reset value of all outputs. For example, when a subsystem reset is activated, the output signals of all blocks should take the values according to the specification. This can become more complex and also more important when the design has multiple power regions and multiple resets. However, as long as the expected values are specified, it is fairly simple to write the properties.

C. *Clock connectivity*

Clocks are the next logical step in verification. The function that should be verified is that the correct clock waveform is applied to the destination block when the clock is enabled. This also verifies that the correct clock enabler and correct clock source is used, that any clock gates are opened and that the clock is propagated to the block.

Consider an example. A clock source that is always on is distributed to a number of blocks, each having a clock gate controlling whether that block should have a clock. The clock gate is controlled individually for each block by registers. In this case, an assertion could be written that checks that the block clock is identical to the common clock source when the clock control register has enabled the block clock. This case is similar to conditional point-to-point connectivity checking, but the clock gate is often a sequential element, implemented with a latch on the enable signal.

The checking could be done without considering the sequential behavior by checking the paths separately. That is, by moving from clock source to clock gate, from clock enable register to clock gate enable port, from clock gate output to block clock. These checks would not involve any sequential behavior. However, it is fairly simple to check the combined behavior with assertions, thereby catching bugs such as polarity inversion of the clock gating signal.

The clocking scheme could of course be more complex, perhaps involving more signals controlling the clock gating, in which case these conditions must also be incorporated into the assertion.

In the ST-Ericsson modem design, the clocking architecture is quite complex due to the aggressive power saving requirements. It involves every block in the design. There are a large number of clocks, each with a number of sources that dynamically determine if a clock should be running or not.

The integration aspects of the clock design have been formally verified on subsystem level. At block-level the functionality of the clock control blocks have been fully proven formally. Formal techniques have become the preferred solution for verification of the clocking functionality, in part because of the critical nature of the function and also because the alternatives, direct test cases or constrained random test, are needlessly complex.

At subsystem level, the clocking verification consists of checking that the different sources are connected through the hierarchy and combined correctly to turn on the individual clocks. Although this verification is similar to the previously described connectivity verification, it involves both sequential behavior and logic functions. The formal verification also requires a number of constraints, partly to bypass a complex startup sequence such as powering up PLLs and partly to separate DFT mode from mission mode.

Some of the clock properties can be reused from the block level verification since that verification is also done formally. The reuse, however, does require some changes to the clock properties. It will of course simplify the reuse if the block level properties are developed with hierarchical reuse in mind.

D. DFT connectivity

A large part of the DFT functionality is contained in the memory BIST logic. Control of the memory BIST functionality is achieved through the use of global control signals and DFT control registers. This control is suitable for formal connectivity checking.

DFT scan and BIST modes require clock gates and resets to be controlled differently than mission mode. This control is via top-level signals and DFT control registers. The verification of this control can be included in the clock and reset verification described earlier, though in some cases it requires constraints for analyzing the design separately in DFT mode.

One example of DFT functionality that can be easily included is where DFT modes override the mission mode clock to force clock gates open. Both the connectivity of the DFT control signals and the correct function of the clock gating can be verified in this way.

Building upon the formal connectivity checking, it is possible to check other simple logic functions in a similar way. One example is the BIST *end* status signal. Each memory BIST controller raises this signal when it has completed the memory test. The *end* signals from all memory BISTs are then combined into a single global *end* signal that is simply the *AND* function of all the separate *end* signals. Verifying this function using subsystem-level simulation is very time consuming since it requires sequencing all the individual BIST end signals. With formal verification, it is simply a matter of capturing the *AND* logic in the form of a property.

VI. SPECIFICATIONS

The efficiency of creating the connectivity properties depends to a large degree on the specifications, especially on how easy it is to use the specification as input to the scripts that generate the connectivity properties. This reuse can be achieved either through using file formats that are machine readable, such as spreadsheets or XML files, or by having clearly specified rules that can be applied to the whole design, such as “all clock gates must be opened when the dft_clk_force signal is activated”.

The specifications used at ST-Ericsson for connectivity verification consist of a number of different sources. Most are architectural specifications of the design and are listed below. The remainder of this section describes these specifications and how they are used in connectivity checking.

- Integration spreadsheets
- Port lists
- Memory lists
- Hierarchy description
- Architecture specification
- Clocking architecture
- Power domains
- DFT architecture
- DFT configuration register description

These specifications are considered golden for connectivity verification.

Integration spreadsheets describe subsystem connectivity. The subsystem can consist of a large number of blocks and the spreadsheet describes where each block-port is connected. One verified subsystem has 3,500 point-to-point connections described in this way.

During integration the spreadsheet is used as input for generating the subsystem RTL. Here the connectivity check verifies that the integration process has been performed correctly. This connectivity check might seem superfluous since both RTL and properties are generated from the same source. Yet there are several reasons why this makes sense. The generated interconnect can have bugs due to the generator scripts. The interconnect generation is also done at several levels; a generated subsystem is instantiated in another subsystem, which is also generated. Thus there is a danger of cross-hierarchy bugs. Furthermore, design changes might be done without regenerating the design from the specification, for example when implementing ECOs.

Another specification is the port list which describes all subsystem- and block-level ports and includes data types, clock region and the reset values on all outputs. The architecture specification combined with the port lists are used to generate properties that check that block and subsystem outputs have correct reset values when the corresponding reset input is activated. This verifies the integration as well as the block design.

Memory lists in spreadsheet form and hierarchy descriptions in XML form are used to create properties— for example, power and retention control of the memories—that verify system-wide connectivity to all memory IPs.

An alternative or complementary approach for specification is to embed the information in the design. This might seem contradictory to the goal of verifying the design, but there is information needed for verification that is not the subject to be verified. One example is finding the hierarchical path to all memory instances, which is required for checking that power control signals are connected to all memory instances. These paths are not subject to verification, so taking this information

from the RTL design does not decrease the verification value of the connectivity.

The hierarchical paths could be extracted from an XML hierarchy file or from the RTL design. A simple method is to ensure that all memories in the design use a consistent naming scheme that allows identification of memories by their instance or entity names. Memory list and path information can then be automatically extracted from the RTL design database.

It's not only the quality and form of the specification that affect the effort needed to create the connectivity checking properties. The regularity/consistency of the design also has a big influence.

In large SoC designs, the different subsystems and blocks are often developed by different design teams that might have different design styles and solutions. This can make connectivity verification very time consuming. Enforcing common design styles and design solutions is therefore very important. One example of this is how the memory BIST is connected. There are eight large subsystems in the modem design that use memory BISTs. If each subsystem used a different way of connecting the BIST through the hierarchy (even though they comply to the same BIST design rules) the connectivity effort would be eight times larger.

In the ST-Ericsson modem design, this is solved by generating the BIST structure so that it is identical for all blocks in the design and also by enforcing how the control signals are connected through the hierarchy.

VII. Automatic Creation of Properties

Connectivity checks are usually very regular and repetitive in nature and lend themselves easily to script generation of the properties. There are advantages to automatically generating the properties. One is that there is less editing when design changes happen, such as alterations in the design hierarchy. It is also possible to make the input files to the generator in a compact readable format, making it much easier to review.

Example 1. Connectivity generator input file

```
{check} CDR connection N0-N4, Main PLL
{src}       $syscon_main_pll_cdr_entity
{srcports}  tst_pll_pf_n0,
            tst_pll_pf_n1,
            tst_pll_pf_n2,
            tst_pll_pf_n3,
            tst_pll_pf_n4,
            tst_pll_pf_enable
{dst}       $syscon_main_pll_entity
{dstports}  N0,N1,N2,N3,N4,ENABLE
{tag}       main_pll_n_ctrl
```

Here's one example of the input to a generator script describing a number of connections between two blocks in a compact format. This input file is manually written based on a specification of the corresponding functionality. The file describes the path to two blocks (*dst, src*), which are variables defined elsewhere. Next comes a list of the source port names (*srcports*) and a list of the corresponding port names on the destination block (*dstports*). The *tag* field describes a precondition, defined elsewhere, so the connectivity is only checked when this condition is set. From this file, six conditional point-to-point connectivity assertions are generated. The input file is much more compact and easier to review than the generated SystemVerilog assertions.

In the previous example, the input file was created just to provide data for checking connectivity. An alternative example is shown below. Here, the input file is used both for generating the RTL interconnect code and the data required for connectivity checking. The example is an extract from a spreadsheet file used to describe block-to-block connectivity, with each line describing two block-ports that should be connected. It also describes when ports are to be left unconnected or tied-off to a constant.

Example 2. Connectivity and integration spreadsheet file

```
cpu.paddrdbg[11..2];db.paddr[11..2]
cpu.paddrdbg31      ;db.paddr[31]
ac.prdata[31..0]    ;pb0.ac_prd[31..0]
ac.psel_vec_a       ;pb0.ac_evec_psel
cpu.rstreq          ;_to_open
cpu.nopwrdwn        ;_to_open
cpu.addr[31..12]    ;_to_constant ;(OTHERS => '0')
cpu.addrv           ;_to_constant ;'0'
```

The compactness of these formats is important. It is in principle possible to directly write the assertions without using any generator, but when the complete property set consists of 10,000 properties the result could be 100,000 lines of code and 10 MB of text. Without automation, the maintenance of such a property set is of course costly.

VIII. Considerations for Formal

Usually one would not consider running formal verification at the SoC level to check functional behavior; formal is usually a method that is targeted at block level verification [4]. The main concern is the size of the state space the formal tool needs to consider, which is both a function of the RTL design and the properties. With connectivity checking, the properties are usually trivial, often with little or no temporal element. Also by their nature, they typically involve only narrow slices of the design under test, allowing the formal tools to consider just a fraction of the total SoC state when attempting to prove the properties. As a result, formal verification can be readily deployed at the SoC level.

While connectivity properties are usually simple, there are a few things to consider when writing them. The examples below will be using SystemVerilog assertions (SVA) [5] but for formal connectivity checking, both PSL and SVA can be used. The syntax and semantics are very similar and the concepts exemplified here are useful in both languages.

A. Point-to-point properties

Assertions are clocked and as SoC designs contain many clocks, it must be determined which clock to use for each assertion. This can, however, be simplified. For combinational connectivity checks, the relationship between clocks is not relevant and therefore all clocks can be considered to be identical. This allows all properties to be setup to use the same clock, and thus to drive all clocks in the design.

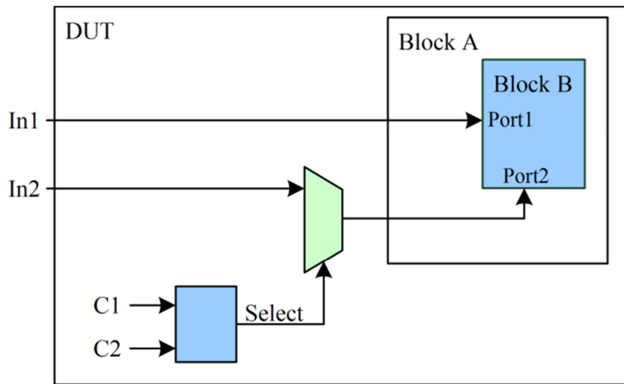

Figure 2. Design example used in the assertions below

The following example shows how a simple point-to-point connectivity check can be written in SVA using the design example in figure 2. It is simply an equivalence check of the two ports referenced using hierarchical dot notation. The example assumes that a default assertion clock has been defined (see example 7).

Example 3. Point to point connectivity check through hierarchy

```
assert_in1__blockb_port1:
  assert property (
    dut.in1 == dut.a.b.port1
  );
```

Conditional point-to-point connectivity is another common property. Conditional connectivity is simply a precondition for the equivalence check. This assumes that only combinational logic is involved. Often the condition may be used in many properties, so separating the condition from the property is generally a good idea. In the following example, also based on the design example in figure 2, the condition is described separately so it can be easily reused.

Example 4. Conditional point to point connectivity check

```
assign select2 = (
 (dut.c1 == 1) &&
 (dut.c2 == 1)
);
assert_select2_in2__b_port2: assert property (
  select2 |-> dut.in2 == dut.a.b.port2
);
```

An alternative method to implement this property would be to consider the condition to be static during the whole analysis. This can be done by using a constraint to set the design in the correct condition for the analysis of the interconnect.

When just some of the property set requires this constraint and other properties don't—for example, to separate DFT and mission mode—these must be analyzed in separate runs. The advantage of analyzing the modes separately is that the properties might be simpler to describe since they may not have to describe both modes of behavior. A disadvantage is that other modes must be analyzed in separate runs. There is also a risk of over-constraining the design, thereby missing verification of valid behavior.

An unconditional point-to-point property is simply an equivalence check and at first sight may be considered to be sufficient to check the connectivity of two nodes. However, this property would miss one common type of error: that a signal is mistakenly tied off to 0 or 1. In such a case, the equivalence still holds, so another type of property is needed. One simple solution is to add cover properties verifying that the involved signals can toggle.

Checking that the signals are not tied off in a conditional property must be done differently. A simple cover statement might not work since potentially the condition can toggle in the same cycle and cause the checked signal to toggle. A typical example of this would be a signal that has different behavior in DFT mode vs. mission mode, effectively having a mux on the signal controlled by the DFT mode condition. Just changing the mux select signal will toggle the output while the input to the mux might still be tied off.

The cover property for a conditional property should therefore be described differently. The following example shows how this can be accomplished by requiring that the condition is stable when the signal to be checked toggles.

Example 5. Conditional tie-off check

```
cover_select2_b_port2_rose: cover property (
  select2 && $stable(select2) &&
  $rose(dut.a.b.port2)
);
cover_select2_b_port2_fell: cover property (
  select2 && $stable(select2) &&
  $fell(dut.a.b.port2)
);
```

It would be possible to write such a property as an assertion rather than a cover property, but this may require use of liveness [5]. Liveness properties can be checked using formal techniques but may be more challenging to prove and debug. So the decision was made to express these conditional checks as cover properties.

B. *Connectivity and combinational logic*

Reset logic can also be checked. But in a case where there are several reset sources which can cause a blocks reset to be activated, a simple equivalence property cannot be used. The following example illustrates how this can be written using implication instead of equivalence.

Example 6. Reset connectivity check with multiple sources

```
assert_reset1_blocka_rst: assert property (
  dut.reset1 == 0 |-> dut.a.b.c.blocka.rstn == 0
);
assert_reset2_blocka_rst: assert property (
  dut.reset2 == 0 |-> dut.a.b.c.blocka.rstn == 0
);
```

Reset output values of blocks can also be checked. The following example is a complete SVA property file that defines property clocks, sets input resets through constraints, checks output values and also shows how the checker module is bound to the DUT using the SVA bind-construct.

Example 7. Reset value check

```
module reset_checker ( input ref_clk );
  default clocking ref_clock @(posedge ref_clk);
  endclocking

  // constraints: reset inputs are active
  assume property (dut.core_rst_n == 0);
  assume property (dut.dbg_rst_n  == 0);

  // assertions: check output values during reset
  assert_rst_mem_m0_awsize:
    assert property ( dut.mem_m0_awsize == 0);

  assert_rst_mem_m0_awvalid:
    assert property ( dut.mem_m0_awvalid == 0);
endmodule

// Bind tester module to the DUT
bind dut reset_checker i_reset_checker (.*);
```

The reset checking is an example of checking simple logic functions. Another example that was previously mentioned is to verify the global BIST end signal. Again, this combines the verification of connectivity with logic functions.

Example 8. Verifying a global logic function

```
assert global_bend: assert property (
  dut.global_bend == (
    dut.a.b.c.bend  &&
    dut.a.x.i1.bend &&
    dut.b.bend
  )
);
```

C. *Sequential properties*

The examples discussed so far have not included any sequential behavior. One example involving sequential logic is checking reset connectivity through reset synchronizers (i.e., two flip-flops in series synchronizing a reset signal). The reset source and reset destination are therefore not combinationally connected, so checking the connectivity involves sequential behavior. This can be verified formally, but there are also some simplifications that can be made.

Even when the connectivity checks include sequential behavior, the check is normally not verifying the exact sequential behavior. The connectivity might check that the release of reset reaches the destination, but usually it is not relevant how long time this takes in relationship to other clocks. So a reasonable simplification is to assume that all clocks are identical thereby avoiding the work to define the exact relationship between all clocks. In ST-Ericsson's modem design, approximately 200 clocks would otherwise have to be set up individually.

Clock connectivity checking can in principle be performed in the same way as other connectivity checks, but there is one additional aspect to consider. Normally it must be specified which clock each assertion is clocked on. To be able to check each part of a clock waveform, the assertion clock must be faster than the clock to be checked. If the assertion clock frequency is double the frequency of the checked clock, then each clock phase can be checked. It is therefore not sufficient to use the highest speed clock in the design to clock connectivity assertions that are checking the clock waveform. One solution is to create a "virtual" clock for the assertions. This is a clock that is not present in the design under test, but is created for the formal tool and for which the appropriate frequency can be specified.

Another aspect of clocking is that there are usually clocks that are generated by on-chip PLLs. The behavior of PLL models cannot be analyzed by the formal tool as they are not modeled as synthesizable RTL, which is a prerequisite for the formal analysis. However, the tool automatically identifies the clock outputs of the models and treats them as clocks in the analysis.

Likewise, derived clocks—gated clocks or clock dividers—are identified automatically. However, in this case the tool can analyze the behavior, so no simplifications are needed here.

D. *Reviewing properties*

Formal verification quality is only as good as the specification—that is, the properties. The properties can of course also have bugs, such as using the wrong signals or conditions. These bugs are usually easy to find since the formal tool will find counter examples. However, when the checks are sequential, it is much easier to describe the wrong behavior in the properties and the formal tool might not find any counter examples. The process of writing sequential assertions should therefore involve some way of reviewing the behavior of the assertions.

The method used at ST-Ericsson is to let the formal tool create waveform examples of the assertions (sanity waveforms) and review these. This is usually enough for connectivity checks, since the sequential behavior is usually very simple. If the sequential behavior is more complex, then a single example waveform might not be enough to show the full behavior of the assertion. Other techniques must then be used. These may include writing cover properties based on the assertion code to show that the intended sequence is possible and using the formal tool to create waveforms to review the sequences. Another technique often used to debug properties is to modify the properties so that they're known to fail and then review the counter example waveform created by the tool.

E. *Testbenches and binding properties*

Formal verification of connectivity requires very little in the form of testbenches. There is usually no need to define scenarios or testcases and go through a lengthy configuration phase of the DUT to reach the state where the connectivity can be verified. This means that the time from when the design is ready to begin verification to the first results can be much shorter than for other verification approaches.

What is required for formal verification is to develop the properties and a method to attach the properties to the DUT. The examples shown above are all based on two SystemVerilog features to attach the properties to the DUT; the bind-construct and hierarchical paths. The bind-construct allows a module to be instantiated at any point in the design without modifying the RTL of the design. In example 7, the bind-construct has been used to attach a module containing the

properties, to the top level of the DUT. Hierarchical paths allow access to any ports in the design through hierarchical dot notation from within the properties as illustrated in the examples.

Connectivity checking usually involves listing all endpoints to be checked and creating assertions for each port-pair. In some circumstances a substantially simpler method can be used. The SystemVerilog bind construct can be used to automatically attach assertions to the endpoint.

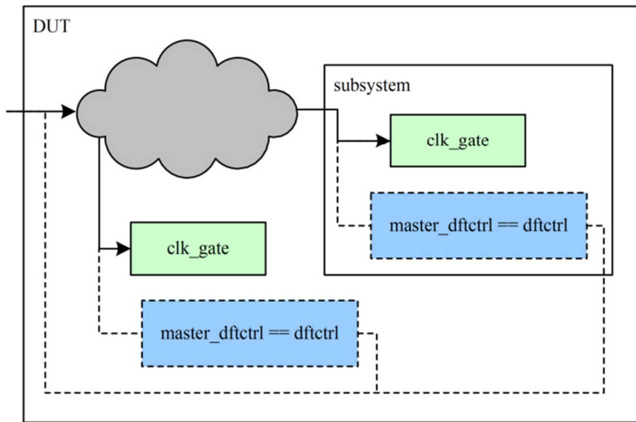

Figure 3. Automatic binding of checkers to all clk_gate instances

For example, consider a goal of verifying that in one mode, controlled by a single top-level control signal, all clock gates in the design are opened (see figure 3). One method to verify this is to use an assertion as previously described, but that would require listing all clock gates in the design. However, if you use a design style where all clock gates are implemented by instantiating the same module, you can use the bind construct to attach a checker module to all instances of that clock gate module without listing them.

In the bind statement, one port is then connected to the top-level control signal and another port to the local port on the clock gate module. In the attached checker module, a connectivity assertion is used to check that the top-level control signals always have the same value as the local port, thereby verifying the connectivity.

F. Black boxing

The RTL code used in formal connectivity checking is the same that is used for simulation based verification. However, there are usually parts of the code that can't be analyzed by the formal tool such as behavioral memory models, PLL models etc. This has little impact on the formal connectivity verification since the functions of these blocks are not involved. The solution for the formal tool is to exclude these behavioral models from the analysis, also referred to as black boxing. This means that the tool considers the content of the modules to be empty and the output of the modules can take any value at any time. The tool identifies behavioral constructs and will automatically black-box the corresponding modules.

This technique can also be used to reduce runtime of the formal tools. If there are no connectivity endpoints or through points inside a subsystem, it is possible to tell the tool to black box that complete subsystem. This can in some cases improve tool performance and also be used to carve up the property set into tranches, enabling the verification to be performed in multiple passes or phases if required. However, this also introduces the risk that assertions might fail on behaviors that are not possible since the design is under-constrained due to outputs of the black-boxed parts. (There is no risk of getting false proofs.)

IX. RESULTS

Since the introduction of formal connectivity verification, several ST-Ericsson projects have succeeded in verifying the connectivity exhaustively using formal methods. That is, all connectivity has been verified and all properties proven formally.

During the verification a large number of bugs were caught. In a project with a new design, approximately 100 bugs were found ranging from simple connectivity to interface bugs. In a subsequent project with the same structural design but upgraded functionality the number of bugs found was down to 40.

Among the key elements of the success of the approach:

- Due to the large number of connections to be verified, it is crucial to be able to automate the creation of assertions.
- The regularity of many design features such as interconnect, clocking, reset, DFT control, memories, memory BISTs, and the enforcement of consistent design solutions makes compact specifications and automation possible.
- The specifications are created in machine readable form such as spreadsheets and XML files.
- It is possible to verify the connectivity checks using formal tools on the full design. This depends to a large degree on the low sequential depth of the connectivity checks.
- Formal tools are available that could handle complete designs, both in terms of size and design style.

These factors should not be difficult to achieve in other SoC projects. Many of them are likely to be fulfilled already since they contribute to overall efficiency, pursuit of which is not limited to connectivity checking.

The creation of properties was achieved to a large degree through automatic methods using scripts to parse and translate the specifications. Previously, the properties were manually created requiring several man months of effort. The automatic approach takes just a few seconds to generate hundreds of properties, saving time in both the development and maintenance of properties. The scripts are also re-usable and extendable for future projects.

Table 1 summarizes the properties used for two different subsystems. The table differentiates between assertions created automatically and semi-automatically (created automatically

but edited manually). The need to edit manually was due to lack of information in the machine readable specifications or too many special cases.

TABLE I. PROPERTIES SUMMARY

Property Set		Number Of Properties		
		Automatic created Assertions	***Semi-automatic created Assertions***	***Manually created Constraints***
System I	Mission	3500	800	130
	DFT	1600	300	0
System II	Mission	6500	0	75
	DFT	0	0	0

The success of the method made it possible to widen the scope of verification during the project, thereby verifying substantially more properties than was initially planned. Due to the widened scope, it is not possible to directly compare the effort to implement formal verification versus that to just proceed with simulation-based verification. However, the time to develop and verify these properties is certainly much shorter than that generally expended when verifying using simulation.

The clock connectivity required most time per assertion to write as the checkers are more challenging. But it is also in this area that simulation has most problems. The clocking is a system-wide behavior that has a large number of cases. Covering this using functional coverage and constrained random simulation would be very time consuming. Using an assertion language for checking would be a good choice even if using a simulation-based verification approach for clock connectivity checking. However, with the formal alternative, you avoid the large task of finding scenarios that complete the coverage goal.

Before analysis the property set is sorted into groups, where the members of each group share common constraints, and separate formal runs are then applied to each group. For the complete modem design, it takes a few hours to run the proofs of one property group while the complete set might be run overnight.

It is also important to stress that expertise in formal verification methods is not so important for this type of verification. The team that implemented these methods at ST-Ericsson were by no means experts in formal verification. The successful approach employed at ST-Ericsson should therefore not be difficult to reproduce.

Most of the formal verification has been performed using Mentor Graphics' Questa® Formal product but the techniques described in this paper are fairly tool-independent and have been used with other formal tools.

X. CONCLUSIONS

The verification of SoC assemblies is a challenging process due to the number of blocks, point-to-point connections and complexity of clock structures. ST-Ericsson has successfully approached this using formal verification methods, achieving an efficient verification flow that is to a large degree automated and has proven to catch bugs efficiently. Indeed, thousands of properties have been proven and hundreds of bugs caught using this method.

Creating and proving connectivity properties is not difficult and doesn't require extensive experience of formal verification. However, given the amount of properties, automation of property creation is important. Achieving this is facilitated by machine-readable specifications. When ST-Ericsson started to deploy formal connectivity verification, a part of the specifications were machine-readable while others were not. Over time this has been improved substantially as the awareness of the upside of formal techniques increased the need for machine-readable specifications and consistent design solutions. The slowest improvement to achieve has been consistent design solutions. This is natural and due to the high cost of changing design solutions and the time it takes to unify solutions over multiple sites and design teams.

The awareness of formal verification has increased since deployment of formal connectivity verification and it has become a part of the verification toolbox even outside the area of connectivity verification. In the future, a natural development might be to extend connectivity to also verify the communication protocols used at the system level.

ACKNOWLEDGMENT

We thank the reviewers at ST-Ericsson and Mentor Graphics for their insightful comments and suggestions.

REFERENCES

[1] N. Bamford, Rekha K Bangalore, E. Chapman, H. Chavez, R. Dasari and Y. Lin, "Challenges in System on Chip Verification", 7th International Workshop on Microprocessor Test and Verification

[2] S.K. Roy, "Top Level SOC Interconnectivity Verification using Formal Techniques", 8th International Workshop on Microprocessor Test and Verification

[3] P.Yeung, H.Foster, "Planning Formal Verification Closure", Mentor Graphics World-Wide Web Page [Online]. Available http://www.mentor.com/products/fv/0-In_fv

[4] H. Foster, L. Loh, R Baham, V. Singhal, "Guidelines for creating a formal verification testplan", In Proc. DVCon 2006

[5] IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language, IEEE Std.1800-2009