Contributed Article

Automated Formal Method Verifies Highly-Configurable HW/SW Interface

By Joachim Knäblein and Hans Sahm, Alcatel-Lucent

This article describes how formal verification techniques significantly improved the verification productivity and quality of a complex HW/SW interface - traditionally verified using simulation-based methods - in a large SDH/SONET communications chip. The interface consisted of 31 distributed register map blocks, presenting a challenging verification task.

The formal verification approach reduced our verification effort and time by 70 percent compared to simulation, improved verification quality, and eliminated tedious and error-prone human involvement. The effort saving is even greater when regression tests are taken into account. As a result, we now routinely use both formal verification and conventional simulation-based verification, exploiting the complementary strengths of both approaches. The choice of approach is determined by criteria such as logic complexity, gate count, criticality and the potential for faster results.

The Alcatel-Lucent facility in Nürnberg designs large, state-of-the-art ASICs and FPGAs for next-generation telecommunication equipment. Not only are such designs growing more complex, but so is their verification. Alcatel-Lucent continuously invests in improvements for its functional RTL verification flow to meet these continuously growing verification challenges. Consequently, we joined with other companies and research institutes to initiate an industry collaboration – named HERKULES [1] – to achieve zero-defect development of hardware systems. We leveraged our HERKULES experience and technology in this project to verify the complex HW/SW interface.

The Approach

Telecommunication ASICs often incorporate a large number of registers which are used for communication with the system processor. Typically, the processor uses the aforementioned HW/SW interface to initiate a particular system function or to monitor system behavior. The register count may exceed 20,000 registers per device, with each register possessing tens of attributes such as address and register type.

Alcatel-Lucent uses a dedicated register development tool to manage and automate the development of programmable registers. This tool was enhanced to automatically generate SystemVerilog assertions (properties) from the register specification for subsequent formal verification. This feature can be used to efficiently verify the integration of automatically generated register blocks in the design. Although register block generation itself is considered reliable, errors can still be introduced during register block integration. The potential problems are listed in a later section.

We use the distributed register map approach in which registers are distributed over a number of register blocks, rather than concentrated in one large block. The advantages of this approach are:

- Connections between register signals and functional blocks are implemented locally by the block designers. This avoids the potential wiring errors associated with a single, large block.
- Block designers can verify the distributed model with their own register map at an early design stage.
- Distributed registers avoid the synthesis and verification problems associated with a single, large register block.

Traditionally, simulation-based register verification has been performed to check the correct functioning of the HW/SW interface. Such verification checked that every specified register was accessible by applying all available access modes, i.e. read/write with byte, word and long-word access. It also checked for address mirroring, and that accesses to one register had no impact on other registers.

In this simulation-based approach, every register instance had a built-in VHDL assertion that logged the accesses to a file. The simulation control file was created automatically by the register development tool. The simulation log file typically became very large and, depending on the register count, analysis of this log file was a tedious, error-prone job. The simulation ran from days to a week, and had to be repeated after each modification. It is this high complexity and low productivity situation that the introduction of formal verification significantly improved.

Automatic Register Map Test Flow

In the design under verification (DUV), there are 31 distributed register blocks with 3,217 registers. RAM addresses are not considered in this calculation. Each register carries a specific number of bits and fields (multi-bit values). The number of such fields in this design is 12,600. Using the traditional verification approach, and assuming that half of the registers are read/write, this would have required approximately 15,000 accesses to be simulated - three access modes and read/write combinations. Because every register carries 4 fields on average, the output listing would have had 60,000 entries to be analyzed manually, possibly supported by scripting.

By contrast, the formal verification approach works as follows:

1. The register development tool generates 12,600 x n properties (n = 3 to 5 depending on the type of field) required for the complete verification of the register blocks in the design. The properties connect to internal register block signals, so a configuration file maps the hierarchical paths to the referenced register signals.
2. The entire design is loaded into the formal verification tool, in this case, OneSpin Solutions' 360 MV tool.
3. For every register block in the design, we:
   - Black-box everything not required for verification of the register block. Thus, by default, everything is black-boxed except the modules which are below and above in the hierarchy of the register block under verification. This reduces design complexity dramatically.
   - Compile the remaining non-black-boxed parts.
   - Verify all the properties for this block.
   - Report the results.
4. Check reports for failing properties.

All of these steps are combined into one TCL script executed in the formal tool, which then automatically generates a list of those properties that do not pass on the design. This list is typically very short and thus enables a much more rapid review of verification results than the traditional simulation log. Both the properties generated by the register development tool and the script are re-useable in future designs without any change. The only design-specific contribution is the configuration file, which is read by the property generator.

Note that the configuration file also identifies blocks that may not be black boxed, for example, the processor interface module and address mapper blocks. The latter type of block is used to map addresses for multi-instantiated register blocks. Details of these kinds of blocks are given later.

Special Role of RAMs

The register development tool supports the assignment of a group of registers to a RAM block. Typically, this is done to optimize hardware resources. Since RAM implementations vary from vendor to vendor, it is currently not possible to automatically generate tests for RAM regions inside a register block. However, the property generator must know where RAMs are located in the address space in order to avoid the creation of properties for these RAM address ranges. Therefore the configuration file includes this information, too.

Automatic Property Generation for a Register Block

Basically, a register block includes a particular set of addresses. Thus, each address carries, for example, 32 bits allocated to fields. A typical address configuration is shown in figure 1.

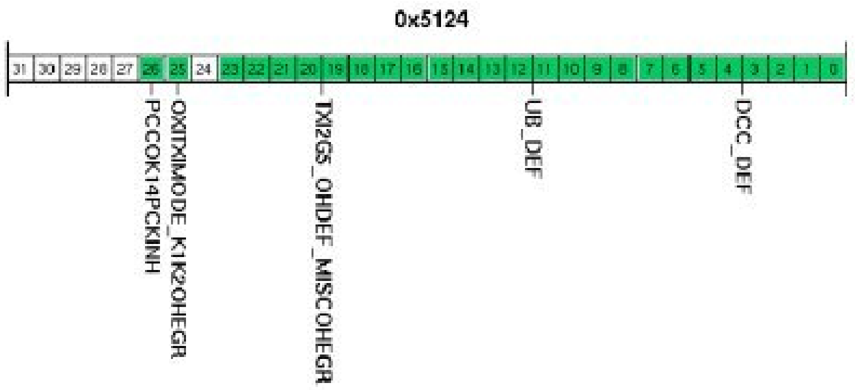


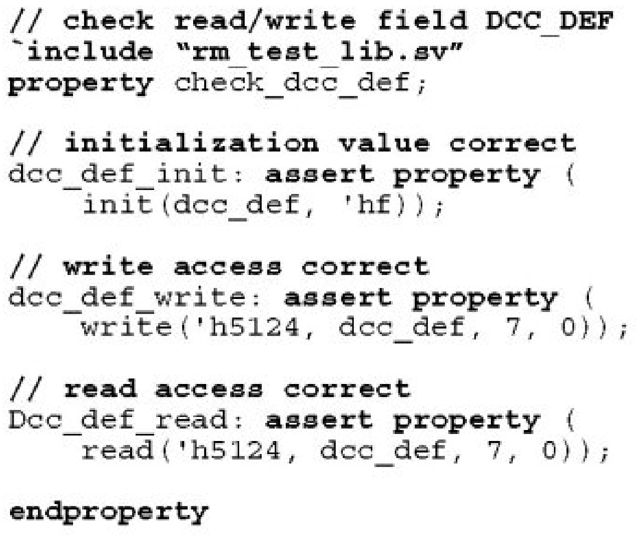
Figure 1: A typical address configuration (Source: Alcatel-Lucent)

The property generation script creates a set of properties for each field. Depending on the field type, the property set for a particular field comprises the following properties:

- A “read” property checks whether the value of the register block input which belongs to the particular field definition can be retrieved at the internal CPU-controller interface. Additionally, it checks that input values do not interfere.
- A “write” property checks whether a value from the CPU-interface is exactly written to the corresponding output port of the register block. Additionally, it checks that the output ports do not change when there is no relevant access.
- In the case of “write” fields a property checks the correct initialization value after reset.

- In the case of interrupt fields, it checks whether the field is correctly set and whether it contributes properly to interrupt generation.
- It creates one property which checks that there is no reaction on the register busses if an invalid address is applied. This property finds, for example, redundant register blocks.

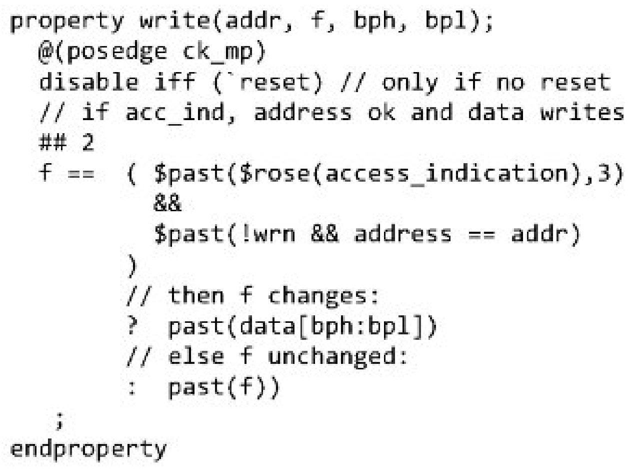
The properties – generated as SystemVerilog Assertions – for the Read/Write field DCC_DEF are shown in Figure 2. To improve readability, the instance path to the signal names has been removed:



```
// check read/write field DCC_DEF
`include “rm_test_lib.sv”
property check_dcc_def;

// initialization value correct
dcc_def_init: assert property (
    init(dcc_def, 'hf));

// write access correct
dcc_def_write: assert property (
    write('h5124, dcc_def, 7, 0));

// read access correct
Dcc_def_read: assert property (
    read('h5124, dcc_def, 7, 0));

endproperty
```

Figure 2: SystemVerilog Assertions for Read/Write field DCC_DEF (Source: Alcatel-Lucent)

A library called ‘rm_test_lib.sv’ provides the basic SVA property definitions. An example write property is shown in figure 3.



```
property write(addr, f, bph, bpl);
  @(posedge ck_mp)
  disable iff (`reset) // only if no reset
  // if acc_ind, address ok and data writes
  ## 2
  f ==  ( $past($rose(access_indication),3)
          &&
          $past(!wrn && address == addr)
        )
        // then f changes:
        ?  past(data[bph:bpl])
        // else f unchanged:
        :  past(f))
   ;
endproperty
```

Figure 3: Example write property (Source: Alcatel-Lucent)

Typical Errors in the HW/SW Interface

Although the automatically-generated register blocks are considered reliable, there is still the potential for errors in the HW/SW interface design. To detect such errors, the verification compares the HW/SW interface specification against its implementation. Some of the errors that typically occur are listed below. Any of these errors might be detected by manual or semi-automatic inspection of the simulation log file. However, all of these errors are guaranteed to be detected by formal verification of the generated properties. The integration of an entire register map has been forgotten.

- Inconsistencies between the register database and the design implementation that arise when the register database is changed without regeneration of register blocks.
- Signals of a register block are not connected or wrongly connected.
- Address mirroring of registers because internal address buses are defined too small.
- Some designs have a top level register map select block that provides select signals to the distributed register blocks. Erroneous generation of these select signals is a typical flaw.
- Errors in RAM implementations. Although not directly verified, specific error groups can still be detected.
- Address mapper blocks are implemented incorrectly.

The last two items require more explanation.

Errors in RAM Blocks

Although RAM-based realization of register groups is currently not supported directly by the verification, it is possible to detect specific error groups in RAM implementations. As described in the section 'Automatic Property Generation for a Register Block', there are several properties that are checked on every single field, and one property that is checked for the entire register implementation.

The field-related properties can detect errors in RAM regions inside the register block wherever such RAM implementations affect the timing of the register block. For instance, when a RAM block signals an erroneous wait cycle.

The register block-related property also fails when the address region specified in the configuration file is incorrect or absent. The formal tool finds a counter-example that uses the integrated RAM block to make the property fail. For example, a RAM region is implemented in the address range 0x0-0x1000 in a given register block. In the configuration file, the range is erroneously defined to be 0x1000-0x2000. The related property checks that there is no response for addresses between 0x0-0xfff, since the property is derived from the definition in the configuration file. The property fails because the RAM block responds only to addresses in the implemented address region.

Errors in Address Mapper Blocks

In certain design situations, it is desirable to instantiate the same register block more than once. However, the automatically-generated register block includes an address decoder that is tailored for a specific address region. Typically there are two ways to cope with this situation.

- The address width of a register block is reduced and an additional select signal is provided.
- A register block decodes the entire address and an address mapper is inserted into the address lines. The mapper re-calculates the address for a particular instantiation.

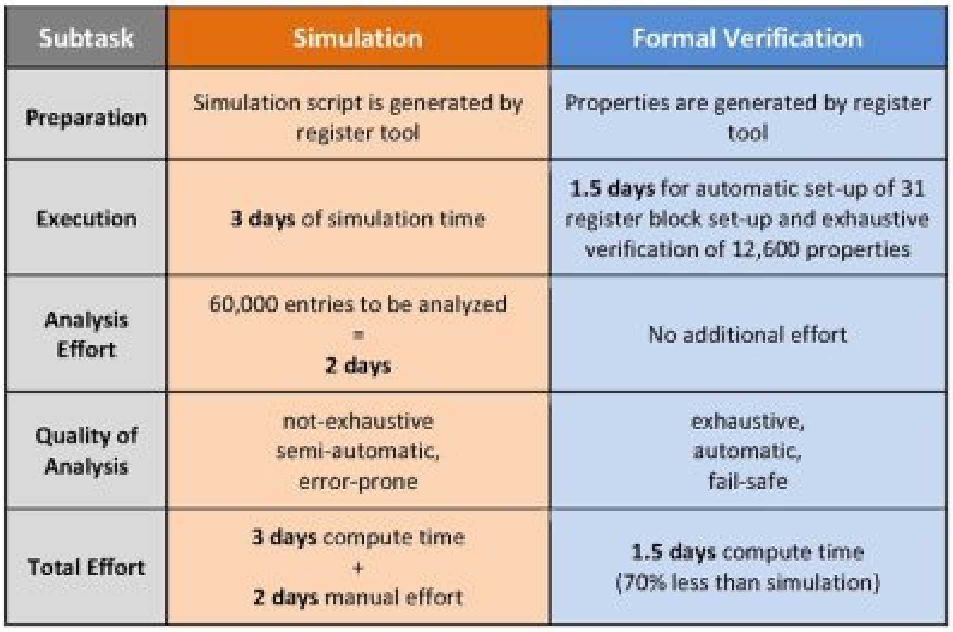
The first solution seems simpler, but is more prone to errors and less flexible than the second one. That's the reason why Alcatel-Lucent's Nürnberg site prefers the address mapper approach. Errors introduced during the manual development of the address mapper blocks are detected by the register map check, which compares the register map specification directly with its implementation.

Improved Verification and Design Quality

The formal verification approach was applied to a complete design. Three erroneous register block instantiations were detected, with incorrectly connected and dangling signals. Moreover, the approach detected inconsistencies between the register map specification and the implementation, caused by changes to the register database that had not been reflected in the implementation. These problems might have been detected during simulation-based verification, but the analysis would have been error-prone and much more time-consuming. We discuss these issues in the next section.

Comparing Verification Productivity

A verification productivity comparison shows the superiority of the new formal verification approach over the former simulation-based verification for this class of HW/SW interface designs. The central result of this verification project can be seen in the following table. Simulation results are interpolated based on previous designs. The formal verification results are expressed as average values; the run-time to prove individual properties varies from register block to register block and from field to field.



Subtask	Simulation	Formal Verification
Preparation	Simulation script is generated by register tool	Properties are generated by register tool
Execution	**3 days** of simulation time	**1.5 days** for automatic set-up of 31 register block set-up and exhaustive verification of 12,600 properties
Analysis Effort	60,000 entries to be analyzed = **2 days**	No additional effort
Quality of Analysis	not-exhaustive semi-automatic, error-prone	exhaustive, automatic, fail-safe
Total Effort	**3 days** compute time + **2 days** manual effort	**1.5 days** compute time (70% less than simulation)

Table 1: Verification effort: simulation and formal verification compared (Source: Alcatel-Lucent)

We achieved considerably better verification quality with 70 percent less time and effort using the formal verification approach. In particular, tedious and error-prone human involvement was significantly reduced. In practice, this productivity gain is even more significant when taking regressions into account. Typically, there are errors in the design with respect to the register map implementation, and its verification must be repeated several times. In each of these regressions the formal approach saves 3.5 days of run-time and analysis as shown in the table above. A fully-automated analysis of the simulation log file would save manual effort but would not change the fact that errors can be missed because the simulation itself is not and can never be exhaustive.

Joachim Knäblein works at Alcatel-Lucent in Nürnberg, Germany. He is a 17 year veteran of chip design, with hands-on experience in EDA tool development. Knäblein has dedicated most of the last two years to formal verification methodologies, namely model checking.

Hans Sahm is Technical Manager Hardware, R&D at Alcatel-Lucent

This article is based upon a paper funded by the German government, BMBF, project HERKULES, registration number 01M3082.

References

[1] HERKULES Project, Description of Work, August 2006, Alcatel-Lucent, Infineon, Bosch, OneSpin Solutions, Melexis, Concept Engineering.

Further Reading

Complete Formal Verification of TriCore2 and Other Processors. Bormann, Beyer, Maggiore, Siegel, Skalberg, Blackmore, Bruno; DVCon 2007

Achieving Completeness in IP Functional Verification. W. Büttner, M. Siegel. EE Times.

Using Formal Techniques to Debug the AMBA System-On-Chip Bus Protocol. A. Roychoudhury, T. Mitra, and S. Karri. DATE 2003.

Automatic Formal Verification of Fused-Multiply-Add FPUs. C. Jacobi, K. Weber, V. Paruthi, and J. Baumgartner. DATE 2005.

Can We Really Do Without the Support of Formal Methods in the Verification of Large Designs? Umberto Rossi. DAC 2005.