Cost-efficient Block Verification for a UMTS Up-link Chip-rate Coprocessor

Klaus Winkelmann, Infineon Technologies AG
Hans-Joachim Trylus, Siemens AG
Dominik Stoffel, Kaiserslautern University
Görschwin Fey, Bremen University

Abstract

ASIC designs for future communication applications cannot be simulated exhaustively. Formal Property Checking is a powerful technology to overcome the limitations of current functional verification approaches. The paper reports on a large-scale experiment employing the CVE property checker for verifying the block-level functional correctness of a large ASIC.

This new verification methodology achieves substantial quality and productivity gains. The two biggest advantages are:

- *Coding and Verification can be done in parallel.*
- *The whole state space of a test case will be verified in a single run.*

Formal Property Checking simplifies and shortens the functional verification of large-scale ASICs at least in the same order of magnitude as Static Timing Analysis did for timing verification.

1 Challenges in Designing Complex Wireless Communication ASICs

With the aim of making it possible for people to communicate at any time, anywhere, and in any language, Siemens Mobile is developing a whole range of devices, appropriate network infrastructures and innovative applications for now and the future.

In the last decade the useable chip area for semicustom ASIC design increased rapidly. In the early nineties the maximum available gate size was about several 100k, today more than 20 million in a 0.11µm technology are possible. The attempt of using the available gate size results in the two main problems:

- How to meet the functional verification coverage.
- How to meet the correct timing of the ASIC.

With physical driven synthesis and static timing analysis the latter can be achieved more or less. The first one is the big challenge.

1.1 Context: UMTS

The third generation mobile communication systems are also called UMTS (Universal Mobile Telecommunication System). In Europe and most parts of Asia UMTS is based on the WCDMA (Wideband Code Division Multiple Access) Standard and uses a chip rate of 3,84MBits/s. With these chips, quasi-random bits generated from pseudo-noise (PN) sequences, the user data/bits are multiplied and thus are spread over a wider bandwidth. The PN sequences are orthogonal for different users and applications. This means multiplexing in the power plane. The spreading process consists of two separate multiplications. First, the multiplication with a real-valued channelisation code. Second, the multiplication with a complex-valued scrambling code

In CDMA systems all users share the same frequency spectrum. Thus every user gives a contribution to the background noise level of each other user. To minimize the noise level, and for other reasons, it is essential to have a good and fast power control in CDMA systems. One of the big advantages of CDMA systems is that these systems are relatively robust against multi-path fading. This is achieved by adding up the multi-path components after the correlation with a path specific delay and phase adjustment in the so-called Rake Receivers.

The needed overall performance for a cost efficient solution could only be reached by the use of a combination of ASICs and DSPs.

1.2 Design Characteristics

The uplink chip-rate coprocessor considered in this paper serves as a 'number-cruncher' for a DSP. The data path design of the ASIC is based around a generic multiply-and-accumulate engine, which performs the necessary correlation operations and multi-path combinations. The DSP controls these correlation operations, called tasks, via the control path of the ASIC, using parameter tables. The DSP is also responsible for starting and stopping the tasks by setting execution conditions, stored in execution tables.

The coprocessor consists of six major sub-chips. In total there are more than 200 logic blocks with approx. 85k FlipFlops and nearly 300 RAM instances. The area equivalent is more than 2 million used gates, clocked above 100MHz. The average data throughput lies in the range of several Gbit/s.

1.3 Verification Challenge

The main challenges for the verification of the ASIC are the complex task scheduling and the complex data path.

The task scheduling depends on many parameters like: spreading factor (7 different ones are possible),

1530-1591/04 $20.00 (c) 2004 IEEE

priority flags, mask flags and others. Most of the parameters are allowed to change every 10 ms (the duration of one frame). This results in a case space of more than 10^{18} cases. Obviously this case space could not be verified with classical approaches like simulation.

The data path involves the correlation and de-correlation of the signals. This is a non-trivial task that also depends on several parameters for a task.

2 Infineon's CVE Circuit Verification Environment

Infineon Technologies AG offers a broad range of semiconductor products for important target markets such as mobile communications and networks, access control and network security, car electronics, and more. In order to meet its highly demanding cost and quality targets, Infineon is using an advanced design flow incorporating state-of-the-art commercial tools, as well as innovative in-house tools.

2.1 Tool environment

The Infineon design flow includes formal verification by the in-house system CVE (Circuit Verification Environment, [2]), which has been developed by Siemens and Infineon for more than a decade. No commercial alternative is considered equivalent. Beyond in-house use, the tool is also available to co-operation partners, such as Siemens AG.

CVE consists of language front-ends including VHDL and Verilog, the *gatecomp* equivalence checker and a property checker called *gateprop*.

2.2 gateprop Property Checker

Functional verification, using *gateprop*, is based on

- compiling (automatically) the design into an internal finite state machine representation, and
- formalising (manually) its specification using a simple temporal property language, called ITL.

An ITL property is essentially a constraint on the design's signals over a finite time interval. To be valid, a property needs to hold for every observation window of the appropriate length, in every run. Using a proprietary combination of algorithms such as bounded model-checking [1], the tool checks each property and, if it is found to be not valid, produces a counter-example.

The basic concepts of the language are:

HDL flavour: The user chooses to write in either a VHDL or Verilog syntax, using familiar language constructs to quickly get up to speed with property writing.

Time steps: A property is written over a number of time steps, from time `t` (i.e. `t+0`) to a future time (e.g. time `t+4` after 4 clock cycles). Consequently, there are a few time constructs in the language (e.g. `at t`, `during[t,t+2]`, etc.).

Each property consists of a "prove" and an "assume" part.

Assume part: This part allows the designer to specify the working mode of the design under inspection. Assumptions such as 'no reset occurs at time t' are typically necessary to investigate if the design exhibits a particular behavior. Further typical assumptions are 'at time t the input connection_request is high' or 'there will be no write_request during time interval t+1 to t+5'.

Prove part: This part of the property specifies expected behavior. Typical assertions in the prove-part are 'the grant output is set at time t+5' or 'the write_acknowledge output will somewhere be issued within time interval t+1 to t+3'.

There are a number of language extensions that are designed to result in *concise but intuitive* properties, including data quantifiers, a powerful macro mechanism and time variables.

3 Block Verification

The verification approach in our ASIC project combines block-level formal verification with system-level simulation. Working in a bottom-up way, each block was verified formally before the system level is even coded.

The verification team was headed by one experienced CVE verification engineer. All verification engineers had a scientific background in formal methods and have been trained to use CVE. The major verification load was handled by this team, separating the concerns of designing and verifying. In addition, each designer underwent the same CVE training. Over the duration of the project the designers themselves increasingly started using the formal tool, e.g., adapting properties and running regressions.

In the verification process we can discern several phases. For each of them a close interaction of designer and verifier was practised:

Preparatory formalisation: Using the informal specification, and interaction with the designer team, first properties were written even before block level architecture was available. This required at least the entity description and saved some effort for the later phases – however in most cases the actual verification did also require some knowledge of implementation details, such as the names of state variables.

Initial block verification: Parallel to the coding of a block, properties were written, and as soon as the VHDL code could be successfully compiled, formal verification was started. Alternatively, the designer often ran a first simple simulation for a few standard cases before he handed the code over for verification. The latter option filters out some trivial bugs, but makes little difference from a global perspective. In this phase both trivial and

complex bugs were found (and fixed), and as a result a formal block-level specification existed which covered the block's function completely, and truthfully with respect to the implementation.

Block-level regression: When a new HDL version was checked in, the existing property suite was re-run, in some cases catching errors that were introduced by the change. This was often but not always possible without adapting the property suite.

Specification adaptation: As is common in a large innovative project as this, the system specification changed during the development duration. E.g., certain additional modes and flags were proposed by system engineering, and had to be incorporated into the already coded blocks. In this case properties had to be adapted in parallel with the code change, and re-checked. The complete regression suite was very helpful in these cases as it is by no means trivial to maintain the existing function while adding new ones.

Interface verification: Once two or more communicating blocks had been completely verified at the block level, the formal specification provided an excellent means to check their mutual interfaces. The ITL properties unequivocally describe the timing, handshake protocols and dependencies for each partner of an interface.

An example of such a situation is this: block A produces a result x, together with a "valid" flag x_valid. Block B, which consumes x, assumes that if x is not valid, it will not take the special value X'010', and acts on this assumption. None of the blocks is faulty by itself, but B's assumption is just not guaranteed by A.

Such dependencies were captured in ITL properties, and by carefully reviewing these, several bugs were discovered, which result from two designers' deviating understanding of the informal specification.

3.1 Verifying the Control Path

Verifying control logic is an ideal application of property checking. Instead of generating sophisticated stimuli to check the normal operation and all of the corner cases, properties cover all cases of an expected functionnality at once.

This section explains the verification of one module that contains mainly control logic.

The uplink chip-rate coprocessor consists of several modules that fulfil different tasks. Interfacing the connection to the antenna and to the digital signal processor is done by dedicated modules. The decoding of the signals is executed by four similar units that actually implement the necessary arithmetic functions. Of these four sub-units, the despreader is the one carrying the main work load in terms of operations per second.

The task controller of the despreader (TCD) holds the state of several hundreds of tasks and evaluates changes on their finite state machines. For each such task a change may be caused by a request of the DSP or by an execution condition becoming true. To meet the specification the TCD has to evaluate the execution condition of each task once during a calculation period (= 2048 clock cycles) and has to be able to consider all requests that occur during a calculation period.

A request of the DSP may lead to a transition in the task's FSM and may also cause the execution condition of the task to change. An execution condition becoming valid may switch a task from active (i.e., arithmetic operations for this task are executed) to inactive or vice versa. For each active task a job has to be issued to the arithmetic units following the task controller. Furthermore each task has assigned a task descriptor that is sent by the DSP. It defines several parameters for the execution of the task as for example the task time frame, the antenna where the data originates or the handset a task is related to. A change of the task descriptor is introduced by a request as well.

All tasks were defined by the same parameters. This symmetry was exploited to scale down the number of tasks for verification. Thus, model generation and verification were speeded up.

The property suite of the task controller can be grouped in three major categories:

- OPERATION/RESET-properties
  show that the task controller itself operates correctly, e.g., pipelines operate correctly, all tasks are evaluated during an iteration period, and the inter-process communication is fine.
- DATA-properties
  prove that data is correctly modified. For this, the correctness of accesses to RAMs is modelled. The properties prove that upon a read access the retrieved data is modified consistently with the specification and after that written back. In addition it is proven by properties that no intermediate access to the RAM occurs.
- ENVIRONMENT-properties
  verify that the needs arising from the connection to other modules are fulfilled, e.g. all requests that occur can be processed in time, and the number of jobs for succeeding modules is correct with respect to the amount of time needed to evaluate them.

On the first synthesizable version of the code a large number of bugs were found. This is due to the fact that no simulation run was preceding the formal verification, so a lot of simple bugs occurred. The advantage was that no effort was spent to set up a test bench at module level for the task controller.

The most important contribution of the formal verification was the detection of some difficult bugs that would not have been caught by a simulation run. In the following one such bug is considered in more detail.

Upon an update request the task descriptor of a task is changed. This may contain a correction of the delay between task time and system time. When this delay crosses a critical value a corner case has to be handled.

This was done correctly for so-called data tasks but not for control tasks. To find this bug in a simulation run the old task descriptor would have to provide a delay that is near the corner case. An update request upon the task would have to place a new task descriptor that corrected the delay by a critical value into the necessary direction. Also this would have to be done for a control, not a data task. Indeed only a few of the possible time delays are within reachability of the corner case, besides the time delay several other parameters are contained in the task descriptor, an update request is only one of several possible requests on a running task and half of the tasks are data tasks.

It is virtually impossible to cover all these combinations of conditions in a test bench even at block level, far less at system level.

3.2 Verifying the Data Path

Since the mission of the ASIC is to serve as a "number-crunching" coprocessor for a DSP it consists to a large extent of data path circuitry involving a lot of arithmetic. Initially, there was concern whether formal verification could at all be applied to the larger data path blocks since it is well-known that the proof engines in verification tools have problems dealing with arithmetic. Our experiences, however, showed that the data path blocks could indeed be formally verified with only a few and only minor restrictions. A great number of design errors were detected, some of which had a great probability of surviving a simulation-based verification step without being detected.

In this section we discuss data path verification with CVE *gateprop* using an example from the design.

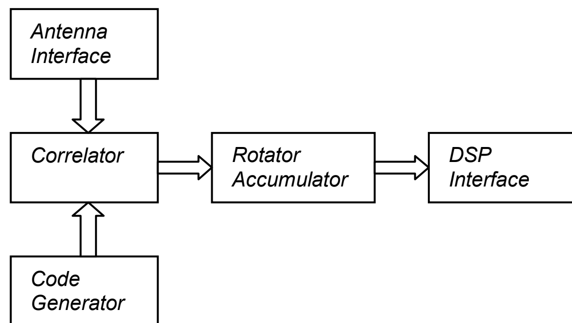

Fig. 1 Despreader

Fig. 1 shows a simplified view of the despreader, a typical signal processing component as it is found in several places in this ASIC. In the despreader, the raw signal coming from an antenna interface is correlated with de-spreading and de-scrambling codes. After correlation computation, the signal is rotated in order to compensate the rotation caused by the transmission channel. Finally, the reconstructed transmitted symbols are stored in buffers in an interface block where they can be picked up by the DSP. All computations are complex-valued. Some of these blocks are quite large. For example, the block labelled "code generator" has a size equivalent to 80000 NAND gates.

3.2.1 Code generator example

In our example, we take a closer look at a generator for a scrambling code which is part of the block labelled "code generator" in Fig. 1. The code generator is not only responsible for synthesizing various types of scrambling codes but also the de-spreading codes needed in the CDMA-typical correlation operations.

The code generator delivers a code word upon request by the correlator. The code word is a segment of a scrambling code sequence. The values of the sequence are typically binary or quaternary, and they are generated by generator polynomials starting from a given start state. For example, if the polynomial is $x^{25}+x^3+1$, then the first 25 bits $y(0), ..., y(24)$ of the sequence constitute the start state (which may be an arbitrarily chosen state other than 0), and the succeeding bits $y(i)$ are defined by $y(i) = y(i-22) + y(i-25) \; mod \; 2$, for $i > 24$. However, not individual bits of the scrambling code sequence are needed by the correlator but segments of 64 consecutive bits. Note that the sequences can be very long (e.g., ~50,000 bits). In order to allow the parallelism needed in this ASIC, the code segments must be produced and delivered to the correlator within only a few clock cycles, prohibiting a naive implementation based on (sequential) linear feedback shift registers (LFSRs). The typical solution for this problem is to use a direct code generation method involving matrix multiplications which allows to produce an arbitrary vector in the state space starting from any given start state. It is schematically shown in Fig. 2

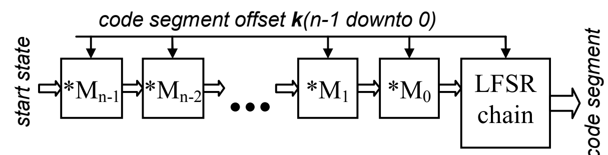

Fig. 2 - Code Segment Computation Scheme

The basic idea is the following. An arbitrary n-bit state vector S' (where n is the degree of the generator polynomial) can be obtained from a start state vector S by multiplying S with an appropriate $n \times n$-matrix M: $S' = M \cdot S$. If M_0 corresponds to a single application of the generator polynomial, then powers M of M_0, i.e., $M=(M_0)^p$, are matrices that can be used to "jump" to a state at an arbitrary offset p from the start state in the sequence. In a typical hardware implementation, certain selected matrices $M_i = (M_0)^k$ are stored as constants such that k is a power of 2: $k = 2^i$. This way, a code segment at

an offset p from the start state with $2^k \le p < 2^{k+1}$ can be computed using only $\lceil log_2 k \rceil$ instead of p matrix multiplications. Still, the depth of this data path and the hardware required is quite large. Since the value of p may range up to around 50,000, there are still 17 stages of matrix multiplications needed for computing an n-bit segment of the scrambling code sequence. Finally, the block labelled 'LFSR chain' in Fig. 2 computes the remaining (64-n) bits of the code segment.

How does one verify this large piece of data path, i.e., what exactly should the theorems check? The property language ITL with its HDL constructs makes it very easy to write properties that correspond closely to the structure of the RTL code of the design, since the very same constructs that are used in the RTL code can also be used for writing the theorems. However, simply importing RTL statements from the design into the theorems does not increase confidence in their correctness, since design errors may be imported as well. It is therefore crucial to find alternative formulations of the design functionality in order to obtain a second viewpoint on the implementation.

```
1:   theorem scrambling_code is
2:   for: i = 0..21;
3:   freeze:
4:       y0 = y_sequence @ t_done;
5:       y1 = y_sequence @ t_done+1;
6:   assume:
7:       during [t, t_done+1]: reset /= 1;
8:       at t+1: p=prev(p)+64;
9:   prove:
10:      at t_done+1:
11:          y1(i)=y0(64+i-22)+y0(64+i-25);
12:  end theorem;
```

Fig. 3 – Correctness Property for Code generator

A solution is sketched in Fig. 3. This theorem checks directly whether the generated code segments conform to the definition of the generator polynomial of the scrambling code. The structure of the implementation is not replicated in the theorem. For the theorem, it is assumed that a code segment at an arbitrary offset is requested from the code generator module at time **t**. At time **t+1**, another code segment is requested (line 8) with its offset **p** being larger by exactly 64, the bit-width of a generated code segment. In other words, the requested segments are assumed to be arbitrary but *consecutive*. The code generator produces both segments after a certain fixed delay. The first code segment is produced at time **t_done**, the second at time **t_done+1**. The **freeze** part of the theorem (lines 3-6) makes both code segments referable as **y0** and **y1**. In the **prove** part (lines 9-11), it is now checked whether the produced segments are indeed consecutive, and whether the generator polynomial condition holds. This theorem formally verifies the complete functionality of the scrambling code generator, including an implicit check of the correctness of the stored matrices and matrix multiplications. (Note that the theorem shown in Fig. 3 considers only the first 22 bits of the generated code segments (line 2). Additional similar theorems checking the remaining 42 bits as well as a theorem for the case p=0 are needed but omitted here for reasons of space and to keep things simple.)

A number of design errors were found by checking this theorem, including wrong addressing of the stored matrices and errors which were induced by introducing pipelining. Some of these errors had a very low probability to be detected in a test bench simulation. Note that, especially in this case, a simulation-based verification approach is greatly inferior to property checking. A scrambling code generator usually produces a pseudo-random output sequence which must be compared bit by bit to a golden sequence, a tedious task. Moreover, in each simulation run, only one out of a vast number of possible sequences can be checked, resulting in very low coverage.

While a theorem like the one in Fig. 3 concisely describes the required design functionality in a few lines of ITL code, the problems to be solved by the property checking engines of CVE can become very complex. However, after all design errors had been found and removed, the proofs of these theorems could be completed within short CPU time, with the exception of the largest of the scrambling code generators. In this case, an additional constraint had to be introduced into the theorem in order to reduce the problem size for the property checker: the start state of the code generator had to be restricted to an arbitrary but fixed value, making the theorem less general. Since, however, this verification problem is symmetric with respect to the variable 'start state', and generality in the much more important variable `code offset' p was still maintained, this restriction is not severe.

Except for this restriction which was applied to only one of the scrambling code generators, the complete code generator module was formally (i.e., exhaustively) verified with CVE, leading to an outstanding verification quality within short verification time. This would have been impossible in a traditional simulation-based verification approach.

3.2.2 Data path results

Similar results were obtained for the remaining blocks of the data path of Fig. 1. These blocks contain more of the standard arithmetic such as complex addition and multiplication. Although arithmetic is typically problematic for formal verification tools, most blocks could be verified by CVE. In some cases we did encounter complexity problems, which, however, could always be alleviated by manual interaction, for example, by reducing the bit widths of the arithmetic operators. Note that current research is directed at automating such approaches, e.g., [3,4,5,6], or at improving the backend proof engines in order to better cope with arithmetic circuits,

e.g., [7]. Typical design errors found in these blocks include wrong operand signs, wrong comparison operators (‘<’ instead of ‘≤’) and typical entity interface problems.

Note that property checking cannot completely replace simulation. For example, as stated before, designers usually maintain a test bench during the design process to be able to quickly simulate the basic functionality concurrently with coding and to immediately remove the most obvious design errors. Interestingly, after running these simulations, the designers very often felt sure that their designs were free of error, especially because their test benches operated the design at maximum load. A number of errors stayed undetected just because the designed system was simulated at extreme operating conditions, thereby simply disregarding the “ordinary” stimuli. These errors were, however quickly discovered by formal verification.

4 Overall Result

Formal verification has never before been applied to a large design project with the degree of coverage that we achieved. The block level verification was completely done by formal property checking. Comparing this to a traditional simulation-based approach, we have to consider several factors, as follows.

4.1 Verification Cost

It would be naive to believe that the benefits of formal verification come for free. The total *human effort* for writing properties in our project was in the order of 1.5 person years.

This has to be compared to the total human effort for HDL coding on the one hand, and writing test benches on the other. The total coding effort was close to 2 person years. As test benches at the module level were not used in this project, we can only relate this to previous projects and conclude that the formal approach requires less total effort than thorough block-level verification, but not drastically so. As formal verification started in parallel to coding, the whole block verification time was reduced by approx. 40%, which saves nearly two months design time.

The *computation time* for a complete regression run has to be compared to the total simulator run-time. On a basis of 30 blocks verified, we find that the sum of all verification run times is in the order of 50 CPU hours on standard Unix workstations. However, fewer than 2 % of all theorems (10 out of a total of about 700) account for more than 90% of this computation time. In other words 98% functional coverage is possible within only 5 hours. Given that simulator time is today one of the severely limiting factors in the ASIC quality assurance process, these figures show that formal property checking provides a very valuable progress here.

4.2 Quality Improvements

The *quality achieved* by block-level verification is probably the most important factor. It can be measured by the number of bugs which make it to the later stages of system simulation and emulation, or even to silicon. An analysis showed that most of the problems arising during system simulation were related to aspects not addressed on the formal approach, such as RAM interfaces and system-level wiring. On the other hand, property checking provably discovered a substantial number of “difficult” bugs, i.e. which are related to certain corner cases likely to be missed by block-level as well as system-level simulation. This was one of the main reasons why there was no redesign necessary for that ASIC.

4.3 Outlook

Time to market and first-time-right silicon are the most important targets in today’s ASIC development.

The promising results in this challenging ASIC project show that formal property checking has the ability to give ASIC verification a substantial boost.

At Siemens Mobile Networks’ UMTS ASIC Development, formal property checking is now one of the supporting pillars of future design flows.

5 References

[1] Biere, A. and Cimatti, A. and Clarke, E.M. and Zhu, Y. *Symbolic Model Checking Without BDDs*. TACAS’99, number 1579 in LNCS, pp 193–207, 1999.

[2] J. Bormann, C. Spalinger, *Formale Verifikation für Nicht-Formalisten*, IT+TI 2/2001.

[3] R. Brinkmann, *Using Symmetry for Problem Reduction in Bounded Model Checking on the Register-Transfer Level*, SymCon 01, Paphos, Cyprus, 2001.

[4] P. Johannsen, *Booster: Speeding Up RTL Property Checking of Digital Designs by Word-Level Abstraction*, Proceedings CAV'01, pp. 373-377, 2001.

[5] P. Johannsen, *Reducing Bitvector Satisfiability Problems to Scale Down Design Sizes for RTL Property Checking, Proceedings HLDVT'01, pp. 123-128, 2001.*

[6] P. Johannsen, R. Drechsler, *Formal Verification on the RT Level - Computing One-To-One Design Abstractions by Signal Width Reduction*, Proceedings VLSI-SOC'01, 2001.

[7] D. Stoffel, W. Kunz, *Verification of Integer Multipliers on the Arithmetic Bit Level*, Proceedings IEEE/ACM Intl. Conference on Computer-Aided Design (ICCAD-01), pp. 183-189, 2001.