Model Checking in an Industrial Environment

Ulrich Hensel[1], Eva Fordran[2], Matthias Fruth[1/2], Hu Shaoyu[1], Madabhushi Srinivasan[1]

[1] AMD Saxony LLC&Co.KG, Dresden Design Center, M/S I21-DC, PF 110 110,
D-01330 Dresden, Mail: ulrich.hensel@amd.com

[2] Fraunhofer-Institut für Integrierte Schaltungen IIS, Außenstelle EAS Dresden
Zeunerstraße 38, 01069 Dresden
Mail: fordran@eas.iis.fhg.de

Abstract - This paper presents experiences in applying model checking to register transfer level (RTL) design verification tasks. The presentation focuses on the description of typical verification problems and their formal capture as well as the application of reduction techniques. Moreover the paper briefly reports on those spots in the verification flow where model checking may be applied successfully.
The paper discusses various rather typical RTL blocks that were subject to formal verification: FIFO control logics and arbiter blocks. These real-world examples provide a set of interesting verification challenges such as proper clock descriptions, synchronous constraints, multiple unaligned clock domains, usage of partial reference models, application of reduction techniques.

1 Introduction

This paper presents experiences gained during the application of model checking to design verification at AMD’s Dresden Design Center (DDC).

The standard RTL (register transfer level) design verification at the DDC is currently based on simulation on both block and system level. The simulation environment exploits different high-level approaches and tools for an efficient test vector generation, check and coverage analysis. In particular, test benches and tests are written using a high level verification language providing random generation based on constraint solving, temporal expressions, definition and collection of functional coverage, means for reuse of verification components, etc.

Based on this language the DDC follows the method of coverage driven verification where coverage analysis is fed back to test case constraints to improve the coverage of a (randomized) test case. Coverage driven verification ensures a high level of confidence once the coverage goal is reached while it keeps at the same time the number of required test cases to a minimum.

Formal verification based on model checking complements this simulative approach. Model checking seems to fit well into the following verification stages:

- **Early in the design cycle**: Model checking may produce sanity and basic functional results on block level very quickly. Reset properties, simple data flow, and request-acknowledge schemes seem to be simple properties and reveal missing assumptions or errors quickly.
- **Concurrently with system level simulation**: Especially crucial blocks like (inhouse) IP or central control structures deserve a full-fledge formal verification. This on the other hand requires a sufficient behavioral interface description in the architecture specification. In

fact, the applications in this paper belong mainly to this scenario.

— **Late in the design cycle, corner case search and debug aid**: Double-check and debug error conditions that appeared on system level simulation or even emulation. In this case model checking can be applied on a block that turned out to produce rather unstable verification results. Model checking can help in identifying the error cause quickly once the error condition has been put into a safety property.

The paper starts with a brief introduction into model checking methodology in Section 2. The description of the application domain follows in Section 3. In particular, Section 3 presents the designs under verification FIFOs and arbiters and the properties (informally) and, moreover, touches briefly on the verification results. Section 4 finally discusses the lessons we have learned from our model checking adventure including clocking schemes, modelling techniques, and reduction techniques.

2 A Brief Model Checking Primer

Today verification based on formal model checking achieves more and more acceptance. In contrast to simulation, formal methods can exhaustively prove the correctness of a design with respect to a set of properties. But they suffer from the so-called ‘state space explosion’ and therefore run successfully on small to medium-sized modules or blocks only [2, 6, 8]. Model checking is also more suitable for control logic blocks rather than arithmetic blocks.
Instead of developing test cases or test vectors, model checking starts with the formalization of a given specification within properties. Properties describe the behavior of a design using a temporal logic. For instance a property could require that each request is answered by a grant eventually if no reset is asserted. The model checker attempts to verify such properties for all input pattern sequences, that is, there is no test vector needed. However, the input behaviors need to be legal: therefore constraints can be formulated and applied. The constraint is akin to a property however it is assumed to be true for the verification run. For instance a constraint could ensure that a certain input is stable with respect to to an enable signal.

Model checking hunts the whole reachable state space. The result is either that ‘the property holds’ (at all possible circumstances) or ‘the property fails’. If the property fails, the model checker gives an error trace for this property that can be debugged using a waveform viewer. The principle of model checking is shown in Fig. 1.

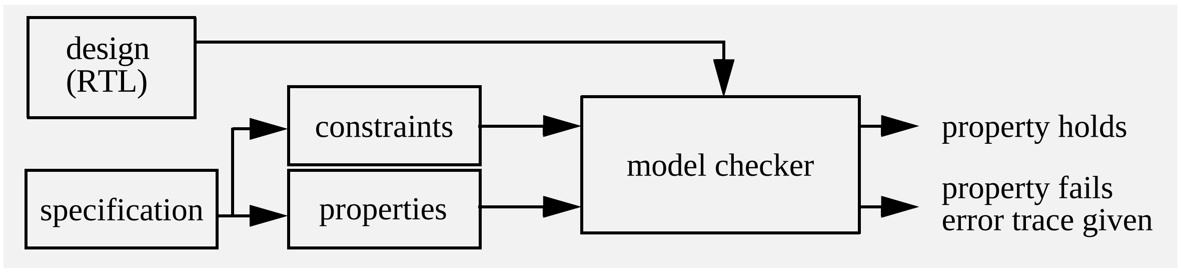

Figure 1:principle of model checking

Using model checking for real industrial designs needs preliminary arrangements of modeling, abstraction techniques, clock handling, reduction techniques, etc. The publications [1, 3, 4, 5, 9] discuss various possibilities of handling industrial designs for verification. A combination of different formal verification methods (Symbolic Trajectory Evaluation and model checking in [4], theorem proving and model checking in [5], pseudo-random testing and model checking in [1]) allows to shrink the state space for model checking. Abstraction [9] and reduction techniques chase the same goal. H.Choi and others [3] show one possibility to model unsynchronized clocks. This paper focuses mainly on clock modeling and reduction methods.

3 Designs-Under-Verification (DUVs) - Industrial Examples

The designs under verification considered in this paper are subblocks of PC infrastructure chips also called chipsets. These chipsets inhabit a variety of peripheral controllers. From the verification point of view the application domain has the following characteristics:

- — There are almost no data path processing blocks containing, for instance, large multipliers.
- — Most of the structures are control oriented such as address decoders, DMA controllers, arbitration schemes, timer blocks, etc.
- — There are many different clock domains.

3.1 FIFOs

FIFOs are normally implemented using registers or with a RAM. The paper discusses FIFOs based on dual-port-RAMs only. Such FIFOs can write and read at the same time. They have different pointers for write and read. The paper discusses the verification of distinct FIFOs. The enumeration below presents a set of general properties applicable to both FIFOs.

- **General FIFO properties**
  - — After an active reset, the input-, output- and internal registers or signals transition to their defined reset values.
  - — If there is no write or read operation then the pointers hold their values.
  - — If the FIFO is writing/reading then the write/read pointer is incrementing.
  - — If the two pointers are equal then the FIFO is full or the FIFO is empty.
  - — The read pointer always follows the write pointer
  - — If the FIFO is full then writing does not change the content
  - — If the FIFO is empty then reading does not produce valid results
  - — Data written successfully into the FIFO can be read out if there is no reset in between (Data content check)
  - — Data ordering is preserved.
- **FIFO_1: a dual clock FIFO IP containing the RAM**

The first example is a parameterized FIFO block intended as reusable component. The desired reuse justifies the application of formal methods to achieve a very high confidence in the block.

FIFO_1 (for an interface see Fig. 2) is based on a dual port RAM. The FIFO comes equipped with a parameterizable depth and width. The target instantiation of the block carries 256 32bit words.

The clock signal *in_clk* drives the input domain and the clock signal *out_clk* the output domain. The clocks are asynchronous and there is no known frequency relationship. Also the clock edges have no relationship. Thus the FIFO can be used as a multi-purpose synchronization block. The FIFO block should be verified in a black box approach basically as given i.e. including the RAM.

The main goal for this verification is to prove data consistency and ordering.

- **FIFO_1: Results**

FIFO_1s original target size together with its free ranging clock domains make the model checker fail because it runs out of memory. Reduction in the parameter values however results in a success-

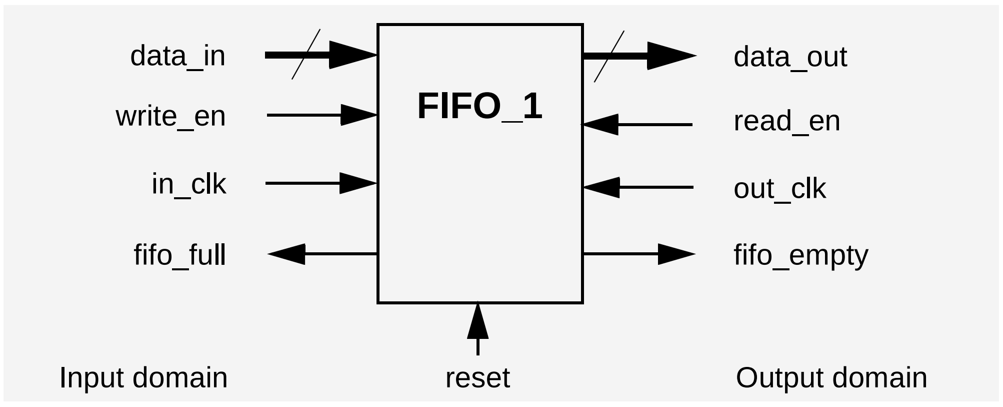

Figure 2:interface of FIFO_1

ful verification: model checking of the downscaled 8x2 FIFO proves all properties including the demanding data consistency and ordering properties. Runtimes are less than an hour.

- **FIFO_2: a dual clock FIFO controller**

FIFO_2 is a control block within a DMA controller (Fig. 3). The block serves as a coordinator between a package handler and a host protocol which belong to two clock domains. The package handler receives data from downstream and writes them into a 32-deep 4-byte-wide buffer. The 4-bit signal *write_en* indicates the progress of the bytes written into one element of the buffer. Every time four bytes writes, the controller increases the pointer *write_ptr* holding the position of the writer. As a counterpart, the host protocol reads out the data from the buffer and sends them upstream. The signal *read_en* indicates the read action and four bytes are read out from the buffer within one read cycle. The pointer *read_ptr* holding the position of the reader is increased whenever the read action is done. All the pointers are controlled in a wrapped around manner. In contrast to FIFO_1 the two clocks stand in fixed relationship however they are not phase locked that is their edges are not aligned. They run at a speed relationship of 3 to 5. The signal *diff_ptr* indicates how many elements reside in the FIFOs buffer.

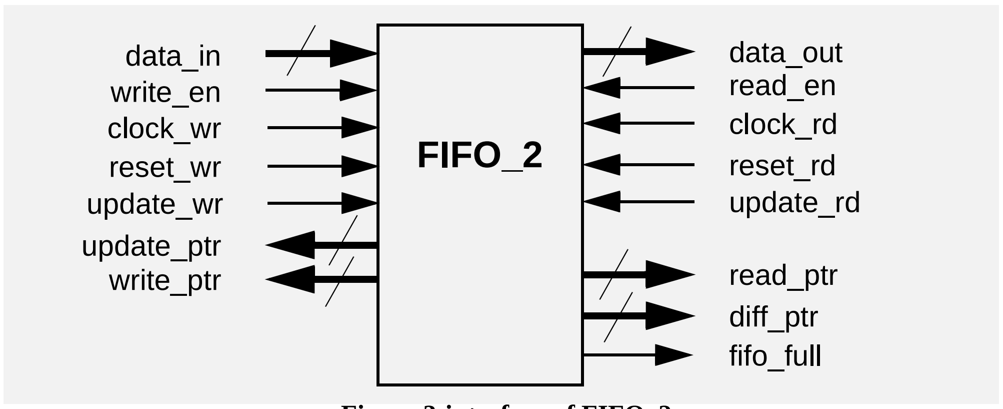

Figure 3:interface of FIFO_2

- **FIFO_2: Results**

FIFO_2 is also a rather large block for model checking. Down-scaling as in FIFO_1 cannot be applied easily because this design is not parameterized. The model checker runs only successfully after using its reduction techniques. Such techniques, offered by the used model checker, are, for instance, one step or iterative cone of influence reductions, deleting inactive variables, etc. All reduction techniques pursue the target of shrinking the state space.

Constraints model the two alive clocks using counters to realize their fixed relationship 3:5.

The model checker shows that one corner case property does not hold: The signal *fifo_full* is not asserted even when FIFO_2 contains 32 elements, because the signal *diff_ptr* never reaches the value 32. The examination of the counter example reveals the cause of failure: fifo_full is only asserted when the 33rd element is written. The specification did not cover this specific behavior of fifo_full.

3.2 Arbiter

Arbiter are control devices that manage exclusive access of a set of requestors to a common resource. The paper touches on the verification of two different arbiters; general properties applying to both are:

- **General arbiter properties**

  — After active reset each signal or register gets its reset value.

  — Always there is at most one grant.

  — No grant is given, while resource is busy.

  — There is no grant without a request.

  — A request leads to a grant.

  — Each request gets its grant fairly.

- **DUV3: Arbiter_1**

The informal specification of the small arbiter _1 reads as follows:
The arbiter arbitrates the requests of four different queues following a round-robin arbitration scheme (Fig. 4) with the option to prioritize posted requests. The priority selection is controlled by a single bit. If posted requests have priority, then non-posted requests will be granted after three arbitration cycle for posted requests at the latest.

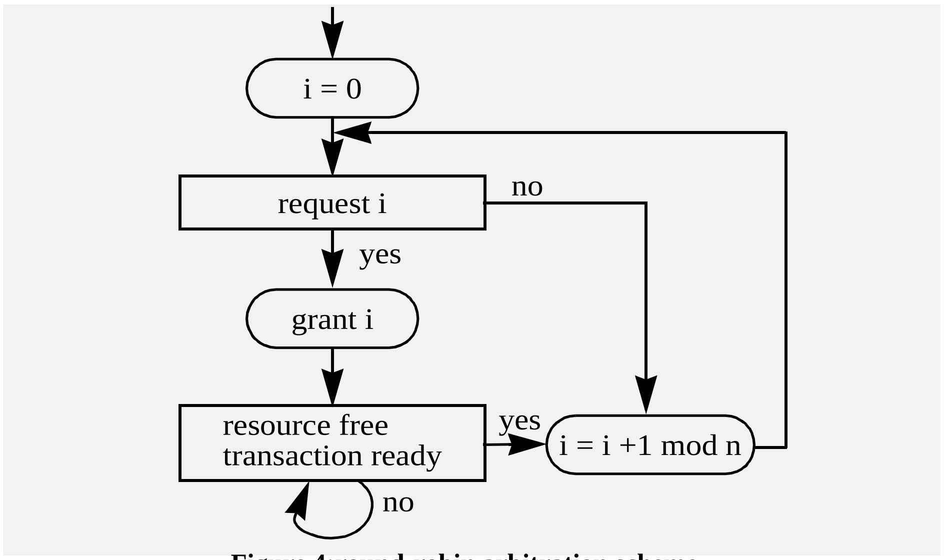

Figure 4:round-robin arbitration scheme

- **Arbiter_1: Results**

Arbiter_1 is a very small block. The model checker needs only a few seconds to verify all properties of the arbiter.

At first the model checker disproved the fairness property. Inspection of the error trace (counter example) given by the model checker revealed the cause: some input signals have to fulfill stability constraints. The specification did not describe these constraints and was updated accordingly. Using the additional constraints the model checker proves the fairness property ‘Each request gets its grant within 10 clocks’.

This demonstrates that model checking may not only find design bugs but may also lead to more precise specifications. In fact, model checking often uncovers “understood assumptions” that need to be documented explicitly to ensure correct block integration.

- **DUV4: Arbiter_2**

The informal specification of the arbiter_2 reads as follows:
The arbiter arbitrates the requests of 15 bus masters. The arbiter supports two arbitration levels with different priorities (fig. 5). Within each level a fair arbitration after a round-robin scheme takes place between all possible bus masters. All bus master requests can be programmed to which level they belong and a master can only request on this specified level.

Fig. 5 shows the two arbitration levels and the requests for bus masters 3, 5, 6, 12 and 13, were the masters 5 and 12 are prioritized over the other masters. The high request L in Level 1 shows that there are low priority requests. If the last high priority grant was 5 and the last low priority grant was 3, than the next grants after round robin at each level will be 12, 6, 5, 12, 13, 5, 12...and so on, assumed the requests will stay all the time.

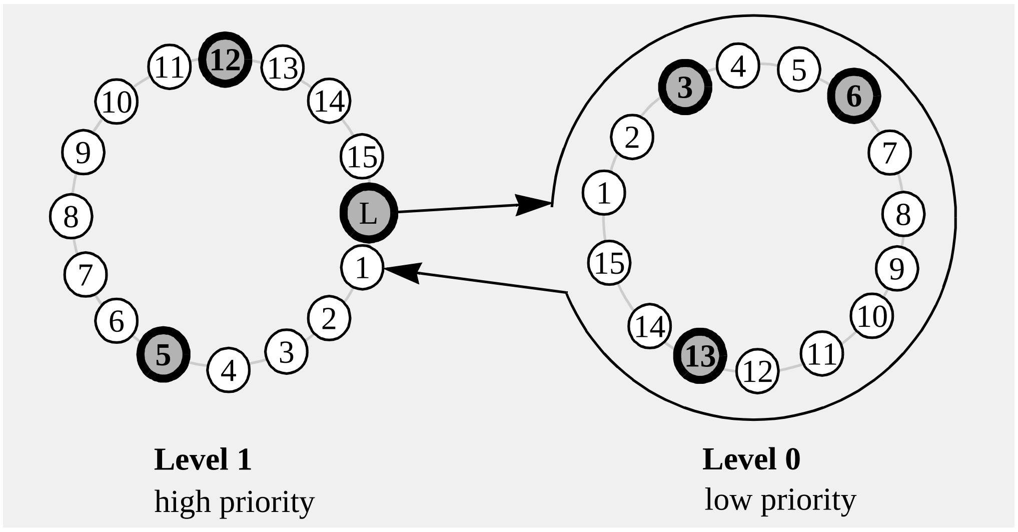

Figure 5:two level arbitration

- **Arbiter_2: Results**

Arbiter_2 is larger and more complicated than arbiter_1. It is possible, that a request gets his grant after 65 other grants were served.

Another approach to reduce the verification complexity is the decomposition of properties. Instead of attempting the verification of the entire fairness property (each request gets its grant after at most 65 cycles) we can prove the fairness property for each request/grant pair separately while all other requests run unconstrained. This property slicing leads to a reduction of the state space and thus the model checker can succeed. The verification of the fairness property for one request/grant pair consumes approx. 55 hours without using additional reduction techniques. Of course this verification had to be repeated for each request/grant pair. Although this method consumed a lot of computation time it finally produced a complete fairness proof.

4 Model Checking Lessons Learnt

This section discusses lessons we have learned in the process of applying model checking during product development.

4.1 Solving the Ground Problem: stepping back and forth between formal and informal

The application of model checking during product development poses a couple “soft” challenges besides the various “hard” ones such as capacity and expressiveness. Nevertheless these soft application obstacles still may hamper the introduction of model checking.

The specification of blocks and subblocks has to contain enough behavioral interface description to enable a black box verification at a granularity level suitable for model checking. Very often block and subblock specifications are rather structural than behavioral. Structural descriptions lend themselves well for implementation but the derivation of properties requires great deals of interpretation. Good examples for behavioral descriptions use wave forms, explicit (informal) descriptions of assumptions, message sequence diagrams, etc.

While the formalization of properties from a more or less complete informal specification has been commonly recognized as a problem, we experienced also a resistance when presenting the verification results to a non-specialist audience. It is a non-trivial task to demonstrate that a set of properties verified under certain constraints provides the desired complete coverage on all input situations and all parts of the design. The 100 per cent coverage that is often stated as the virtue of formal proof obviously depends on the completeness of the set of properties as well as the set of applied constraints. We desired to have a theorem prover like proof management where the proof tool keeps track of all unproven assumptions and holes in the verification.

As the model checker did not support such proof management we used a simple HTML presentation of constraints and properties to keep track of the verification status(e.g. see Table 1).

General design operating modes	**Query/ Property Name**	**Description**	**General Assumptions**	**Assump tions**	**Goal**	**Status/ Results**	**Remarks**
Reset behavior	reset_behavior	Assert resets and check if pointers go back to the original values.	clock_constraints	reset is active	radr=0 wadr = 0 diff_ptr=0 fifo_full=0	verified	There is no global reset signal, resets must be asserted respectively in their own domain.
Eventually	aft_rd_radr_inc	After a valid read the read ptr will update.	clock_constraints	a valid read	radr ++	verified	The counter should increase in a wrap-around way.

Table 1: sample HTML verification result presentation

4.2 Clocks, Clock Relationships, (A)Synchronous Inputs

From the model checker perspective, clocks without any additional constraints are just ordinary inputs. Our designs however are synchronous with respect to a clock or at least have synchronous clock domains. Neglecting the synchronicity of inputs with respect to clock leads almost necessarily to false negatives. Therefore we need to constrain clocks as well as the respective synchronous inputs.

With our tool the simplest way is to define a clock in terms of cranks where a crank is an atomic state transition, for instance we could define the simplest clock by (using pseudo code)

```
clock clk == 1 crank up; 1 crank down
```

The synchronicity of an input with respect to a clock could then be defined by a constraint

```
always(stable(in) or negedge(clk))
```

meaning that the input can only change on negative edges of clk and therefore is synchronous with respect to the positive edge of clk.

For a fully synchronous design these assumptions suffice. However as presented in Section 3.1, blocks subjected to model checking may have two clock domains and the clocks may be unrelated at all. In this case a clock definition in cranks is not expressive enough because these clocks are not phase locked. Instead we just assume that clocks are alive by requiring the constraints

```
after (clk == 0) eventually (clk == 1)
```

and

```
after (clk == 1) eventually (clk == 0)
```

Note that using this constraint we do not restrict the clocks to any duty cycle.

These constraints were sufficient for FIFO_1. However FIFO_2 required to specify unlocked clocks of a certain relationship (namely 3:5). We achieve that by implementing counters that count each on each positive edge of the respective clock (modulo 3 and 5 respectively) and requiring additional constraints like

```
after (clk_1_cnt == 3) always (clk_2_cnt == 5)
```

and

```
after (clk_2_cnt == 5) always (clk_1_cnt == 3)
```

These constraints fix the relationship between the two clocks while leaving them unconstrained enough to wind their way freely.

4.3 Properties vs. Partial Reference Models

In the course of describing FIFO as well as Arbiter properties, the model checker language ability to describe partial reference model turned out to be crucial.

For the FIFO we employed counters and storages to express ordering.

For the Arbiter we created a partial reference model for the round robin algorithm allowing for a stronger form of a liveness property.

4.4 Verification windows technique

A specification describes an associated class of models. The stronger the property is specified, the fewer models are contained in the class. If either the property is too tight or too loose against the desired requirements for the product then it will make the verification results less reliable. Some incorrect models may satisfy properties that are too weak. An effective way to prevent such situations is to write properties with verification windows such as

```
  after ... eventually ... within -delay 0 -duration 3 clock=rising
```

which strengthens the former eventuality property with a 3 clock-cycle window. The infinite testing scope is restricted to a finite one, thus, putting an upper bound on the eventually condition and ruling out all models that fulfil the eventually conditions later in time.

4.5 Reduction techniques

Facing the limitation of model checking to large designs we have to employ reduction techniques for a successful verification. During this case study we basically applied three techniques:

— **Explicit Model Downscaling**: the FIFO IP block comes equipped with parameters for the width and the depth. Reducing the parameter to small but still sufficiently expressive values allows model checking to succeed even without removing the memory.

— **Explicit Property Decomposition**: For the larger arbitration block we decomposed the properties into sets of properties regarding one request/grant pair respectively.

— **Reduction Algorithms**: The cone-of-influence reduction is the universal way of pruning away unrelated variables that increase the state space in vain. The more sophisticated type of reduction is the iterated reduction. A group of variables is pushed over in the model and non-deterministic many other variables will fall invalid like in the dominoes effect. If the model with the standing-erect variables is sufficient to make the property succeed, so will the initial model. Selection of the group of variables can be stochastic, but this will lead to pessimistic results in most cases. An educated selection needs a deep insight into the model, while as a compensation of the efforts, it will make the verification more efficient and avoid meaningless trials. Using educated reduction seeds we were able to achieve a considerable speed-up for quite large models.

References

[1] Mike G. Bartley, Darren Galpin, Tim Blackmore:
A Comparison of Three Verification Techniques: Directed Testing, Pseudo-Random Testing and Property Checking. DAC 2002, Louisiana, USA, p. 819-824

[2] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang:
Symbolic Model Checking: 10^{20} States and Beyond.
Proc. Fifth Ann. Symp. Logic in Computer Science, June 1990

[3] Hoon Choi, Byeong-Whee Yun, Yun-Tae Lee, Hyung-Lae Roh:
Model Checking of an Industrial Interface Logic for Two Unsynchronized Clocks.
DATE 2001, Designer's Forum, Munich, Germany, p. 93-97

[4] Scott Hazelhurst, Gila Kamhi, Osnat Weissberg, Limor Fix:
A Hybrid Verification Approach: Getting Deep into the Design.
DAC 2002, Louisiana, USA, p.111-116

[5] Roope Kaivola, Naren Narasimhan:
Formal Verification of the Pentium4 Floating-Point Multiplier.
DATE 2002, Paris, France, p.1- 8

[6] Thomas Kropf: *Introduction to Formal Hardware Verification.*
ISBN: 3-540-65445-3, Springer Verlag, 1998

[7] Robert P. Kurshan: *Formal Verification in a Commercial Setting.*
Design Automation Conference, June 1996

[8] Kenneth L. McMillan:
Symbolic Model Checking: An Approach to State Explosion Problem.
Kluwer Academic Publishers, 1993

[9] Marco A. Pena, Jordi Cortella, Enric Pastor, Alexander Smirnov:
A case study for the verification of complex timed circuits: IPCMOS.'
DATE 2002, Paris, France, p. 44-51