Improving the Quality of Bounded Model Checking by Means of Coverage Estimation

Ulrich Kühne Daniel Große Rolf Drechsler

Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
{ulrichk, grosse, drechsle}@informatik.uni-bremen.de

Abstract

Formal verification has become an important step in circuit and system design. A prominent technique is Bounded Model Checking (BMC) which is widely used in industry. In BMC it is checked if certain properties hold for the design. But even if all properties could be successfully verified, it is difficult to determine if the properties cover the entire functional behavior of the circuit. Recently, a new approach for estimating coverage in BMC has been presented that can easily be integrated in existing BMC tools. In this paper we give experimental results on the application of the technique to the block-level verification of a RISC CPU. The experiments show that the costs for coverage estimation are comparable to the verification costs. Furthermore it is demonstrated how the technique can be applied to achieve full coverage on a higher level. As an example, we investigate the instruction set verification of a RISC CPU.

1. Introduction

These days the design of circuits and systems is a very challenging task. Besides the design of such systems – that are part of very different kinds of devices – showing the correct functional behavior has become the most important issue. For this purpose simulation based verification techniques are not sufficient since they cannot guarantee design correctness. Therefore, formal methods have gained large attention because they allow to prove the correctness of the design. A very successful formal technique to verify that a finite state system satisfies a temporal property is *Bounded Model Checking* (BMC) [3]. In BMC the system is unfolded for k time frames and together with the property converted into a *Boolean Satisfiability* (SAT) problem [4]. If the corresponding SAT instance is satisfiable a counter-example of length k has been found. Due to significant improvements in the tools for SAT solving [10, 11, 5] BMC can be applied to large designs and is widely used in industry [13, 2].

But so far the completeness of the property set was only ensured manually by a careful analysis of the verification team. Recently, an automatic approach to estimate the achieved coverage for BMC has been proposed [8]. The approach generates a coverage property for each important signal. If the considered properties do not describe the signal's entire behavior, the coverage property fails and a counter-example is generated. From the counter-example an uncovered scenario can be derived. Analyzing these counter-examples and adding corresponding properties allows the verification engineer to stepwise close the coverage gap.

The contribution of this paper is twofold: First, we want to quantify the computational costs for the application of the coverage approach in comparison to pure verification. This very important aspect has not been studied in [8]. Therefore we apply the coverage approach to the block-level verification of a RISC CPU as a non trivial example. The verification and the coverage tests are described in detail by means of examples. Then, the run-times for verification and coverage estimation are analyzed.

Second, we investigate the estimation of coverage on a higher level. Based on the results of the complete block-level verification we consider the RISC CPU at the top-level. Typically, at this level properties for each CPU instruction are formulated. In such a property the exact behavior of all involved hardware blocks with respect to the considered CPU instruction is specified. In other words the effect that results from the execution of an instruction including the communication of hardware blocks is checked. We show that the coverage approach can be used to guarantee coverage at this level. By the means of a detailed example the suggested notion of higher level coverage based on proven correct instructions is discussed and the costs for the coverage check are analyzed. Following a certain property style is helpful for achieving full functional coverage by reducing the number of uncovered scenarios.

The rest of the paper is structured as follows. In Section 2 BMC and the coverage approach are briefly reviewed. The basic data and modeling aspects of the RISC CPU are provided in Section 3. In Section 4 the verification and coverage estimation at different levels as well as the cost comparison are presented. Finally, in Section 5 the paper is summarized.

2. Preliminaries

In this section first an overview on BMC is given. Afterwards the coverage approach from [8] is briefly reviewed.

2.1. Bounded Model Checking

For the verification and the coverage tests, we use BMC as described in [13]. Thus, a property only argues over a finite time interval. For a design with its transition relation T_δ, a BMC instance for a property p over the finite interval $[0, c]$ is given by:

$$\bigwedge_{i=0}^{c-1} T_\delta(s_i, s_{i+1}) \ \wedge \neg\, p$$

This verification problem can be formulated as a SAT problem by unrolling the circuit for c time frames and generating logic for the property. In contrast to [3] there is no restriction for the state s_0 in the first time frame during the proof. This may lead to false negatives, i.e. counter-examples that start from an unreachable state. In such a case these states are excluded by adding additional assumptions to the property. But, for BMC as used here, it is not necessary to determine the diameter of the underlying sequential circuit, i.e. if the SAT instance is unsatisfiable the property holds.

As input language to the verification tool CheckSyC [6] we use a subset of PSL (*Property Specification Language* [1]). In the following we assume that each property is an implication, i.e. the property has the form $always(A \rightarrow C)$. A is the antecedent and C is the consequent of the property and both consist of a timed expression. A timed expression is formulated on top of variables that are evaluated at different points in time within the time interval $[0, c]$ of the property. The operators in a timed expression are the typical HDL operators, e.g. logic and, logic or, arithmetic operators and relational operators. The timing is expressed using the PSL operators $next$ and $prev$.

2.2. Coverage Estimation

The basic idea of the coverage approach presented in [8] is the following: First, for each output o of the circuit all proven properties are identified which argue over o. Then it is checked whether there exists a scenario where o is *not* determined by the set of properties. Here, not determined means that an input and state assignment has been found where no consequent of the set of properties specifies the value of o unambiguously.

This approach is implemented by introducing a multiplexor for each bit that is driven by the output o and the inverted value of o. Then the coverage check can be performed by generating a coverage property for each considered output o. This coverage property is used to show that the multiplexor is forced to select the original value of o, assuming all involved properties. Now if the coverage property for the output o holds then o is covered by the properties. Otherwise from the resulting counter-example an uncovered scenario can be derived.

For the details on the construction of the coverage property we refer the reader to [8].

Complete coverage in terms of the approach is achieved by considering all outputs of a circuit. If all outputs are successfully proven to be covered by the properties then the functional behavior of the circuit is fully specified.

3. RISC CPU

3.1. Basic Data

In Figure 1 the main components of the RISC CPU are shown. The CPU has been designed as a Harvard architecture. The data width of the program memory and the data memory is 16 bit. The size of the program memory is 4 KByte and the size of the data memory is 128 KByte. The length of an instruction is 16 bit. Due to page limitation we only briefly describe the five different classes of instructions in the following:

- 6 load/store instructions (movement of data between register bank and data memory or I/O device, loading of a constant into high- or low-byte of register)
- 8 arithmetic instructions (addition/subtraction with and without carry, left/right rotation and shift)
- 8 logic instructions (bit by bit negation, bit by bit exor, conjunction/disjunction of two operands, masking, inverting, clearing and setting of single bits of an operand)
- 5 jump instructions (unconditional jump, conditional jump, jump on set/cleared carry or zero flag)
- 5 other instructions (stack instructions push and pop, program halt, subroutine call, return from subroutine)

3.2. Modeling

The RISC CPU has been modeled in SystemC [12, 9]. SystemC is a freely available C++ class library, which provides the ability to model hardware at different levels of abstraction. The hierarchical structure of the RISC CPU is reflected in the SystemC implementation in such a way that each hardware block corresponds to a SystemC module. A module in terms of SystemC is a special C++ class. The behavior of a module is defined by processes.

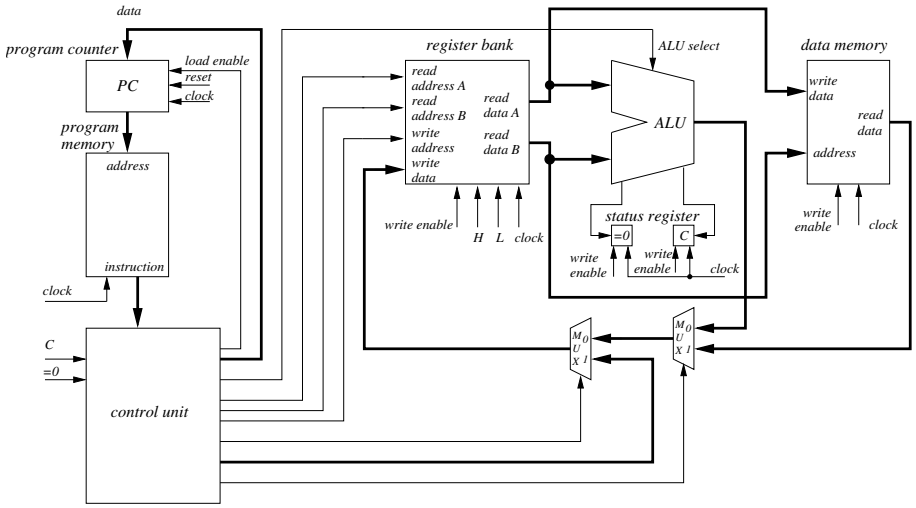

Figure 1. Structure of the RISC CPU including data and instruction memory

4. Verification and Coverage

First in this section the block-level verification of the RISC CPU is described. Then the coverage approach is applied at this level. We discuss in detail how revealed coverage gaps are closed. Then we suggest a notion of coverage on the top-level of the RISC CPU and present the verification and coverage estimation at this level. During the application of the coverage approach we always compare the verification and coverage estimation costs.

All experiments that are reported have been carried out on an Intel Pentium M with 1.7 GHz and 1 GB main memory under the Linux operating system.

4.1. Block-level Verification

In order to guarantee the correct behavior of the RISC CPU, it has been verified using BMC. In a first step, for each of the hardware blocks it is checked, whether the input/output behavior of the implemented circuit matches the specification. Therefore a number of properties have been formulated in PSL.

As the program counter (PC) will serve as an example, it is described first. The PC has an internal 11 bit register `pc` which holds the current program address. The address is shown at the output `pcout`, while the output `pcinc` shows the current address increased by 1. The PC is reset to address 0 by setting the input `reset` to 1. If the load enable input `le` is set to 1, the PC is loaded with the address from input `din`. Otherwise it is increased by 1 in every cycle if the PC is enabled., i.e. if the enable input `en` is set to 1.

In Figure 2 some of the properties for the PC can be seen. The first property `RESET` checks the correct behavior after a reset. The second property `INC` checks that the PC is increased if it is enabled, there is no reset, no load and if the

```
1  property RESET =
2    always(
3      reset == 1
4    ) -> (
5      next(
6        pcout == 0 && pcinc == 1
7      )
8    );
9
10 property INC =
11   always(
12     reset == 0 && le == 0 &&
13     pc < 2047
14   ) -> (
15     next(
16       (prev(en) == 1) ?
17       (pcout == prev(pc) + 1) :
18       (pcout == prev(pc))
19     )
20   );
21
22 property LOAD =
23   always(
24     reset == 0 &&
25     le == 1
26   ) -> (
27     next(
28       (prev(en) == 1) ?
29       (pcout == prev(din)) :
30       (pcout == prev(pc))
31     )
32   );
```

Figure 2. Properties for the program counter.

Table 1. Results for the verification.

Block	#p	CPU time	max. mem
register bank	5	2.95 s	15 MB
program counter	4	0.12 s	9 MB
control unit	19	0.26 s	8 MB
data memory	2	4.09 s	44 MB
program memory	2	1.35 s	23 MB
ALU	18	5.27 s	16 MB

end of the address space has not been reached yet. The third property `LOAD` checks the load functionality of the PC.

For all hardware blocks of the CPU properties have been specified in a similar way. In Table 1 the results of the block-level verification are shown. The first column gives the name of the hardware block. The second column provides the number of properties that have been written for the respective block. In the last two columns the total CPU time for the verification and the maximal used memory during the verification are given. As can be seen, the verification can be carried out very fast using BMC. Note that the sizes of the memories have been reduced in the verification model.

4.2. Block-Level Coverage

If all properties hold the coverage check can be performed in a next step. Following the approach described in Section 2.2, for each single output of each hardware module it is checked, if its behavior is specified unambiguously by the properties. Therefore a coverage property is generated for each output.

Figure 3 shows the coverage property for the output `pcout` of the PC. In line 2 the multiplexor needed for the coverage check is inserted using a special command enclosed in a comment. The command instructs our BMC tool to replace the original output `pcout` by a multiplexor construct – consisting of a multiplexor for each output bit, as described in [8] – and to rename it to `pcout_orig`. Now the original value `pcout_orig` is routed to the output `pcout` iff the signal `select` is set to 1. In lines 4 to 25 the original properties are assumed[1]. Furthermore it is assumed that at time point 0 the select signal is set to 1 (line 27), i.e. we are dealing with the original value of the circuit on output `pcout` at time point 0.

Under these assumptions we want to prove that the select signal has to be 1 at time point 1 as well (line 29), meaning that the output is determined in any case. The coverage property can be automatically checked using BMC. As a result, the property fails and a counter-example is generated, see Figure 4. As can be derived from the trace, the value of `pcout` is not specified in the case that the PC has already

[1]This is expressed in PSL using the c-like ?-operator for an if-then-else construct, i.e. the property $A \rightarrow C$ is transformed to $A\ ?\ C\ :\ 1$ due to syntactical restrictions of our PSL parser.

```
1  property PCOUT_COV =
2  // @insertMuxForSignal: pcout select
3  always(
4    // RESET
5    ((reset == 1) ?
6     (next(
7        pcout == 0 && pcinc == 1
8      )) : 1) &&
9
10   // INC
11   (((reset == 0) && (le == 0) &&
12     (pc < 2047)) ?
13    (next(
14       (prev(en) == 1) ?
15       (pcout == prev(pc) + 1) :
16       (pcout == prev(pc))
17     )) : 1) &&
18
19   // LOAD
20   (((reset == 0) && (le == 1)) ?
21    (next(
22       (prev(en) == 1) ?
23       (pcout == prev(din)) :
24       (pcout == prev(pc))
25     )) : 1) &&
26
27   select == 1
28 ) -> (
29   next(select == 1) // covered?
30 );
```

Figure 3. Coverage property for the program counter.

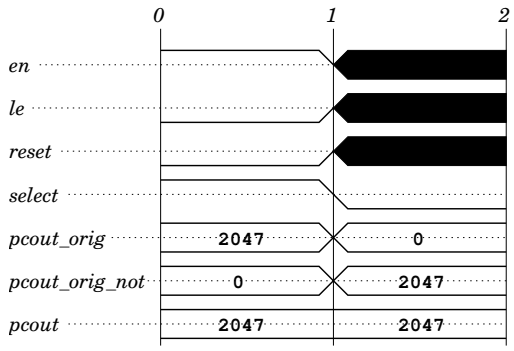

Figure 4. Counter-example for program counter coverage.

reached the end of the address space (address 2047) at time point 0. Looking again at the properties in Figure 2, it can be seen that the gap is due to the assumption in line 13 of property `INC`.

Note that the trace of the counter-example gives us information on the actual behavior of the circuit in the scenario that has not been properly specified. The original value that

Table 2. Results for the block-level coverage.

Block	#o	CPU time	mem
register bank	2	1.50 s	18 MB
program counter	2	0.08 s	9 MB
control unit	24	0.08 s	9 MB
data memory	1	2.88 s	42 MB
program memory	1	1.15 s	25 MB
ALU	3	8.58 s	32 MB

is generated by the circuit is shown as signal `pcout_orig` in the trace. Obviously the PC performs a wrap-around and starts over at address 0 when it reaches the end of the address space.

There are different possibilities how to close the coverage gap that has been revealed by our approach. One could add a property which describes the behavior of the PC in the forgotten scenario and then check again for coverage. As an alternative, the scenario can be directly excluded in the coverage property. For the example at hand we choose to exclude the scenario because the specification itself does not specify the behavior of the PC at the end of the address space. It is left to the programmer to avoid this situation. The scenario can be excluded from the coverage check by the additional assumption $(pc < 2047)$ in the coverage property. With this additional assumption the coverage property holds, i.e. the output `pcout` is fully covered by the properties with respect to all important scenarios.

In the same way coverage properties have been generated for the outputs of all hardware blocks of the RISC CPU. The results of the final full coverage proof are shown in Table 2. The first column gives the name of the hardware block. In the second column the number of outputs are given that have been checked for coverage in the respective block. The last two columns show the run-time and the needed memory for the coverage check. Note that the run-times and memory requirements are in the same order of magnitude as for the verification described in Section 4.1.

In total, if all properties hold that have been specified during the verification and if they cover the entire behavior of a design in terms of the coverage approach, then the properties form a complete and non-ambiguous specification of the design. This complete verification can be achieved stepwise by closing all coverage gaps that are being revealed by our approach.

Whenever a coverage check fails and a counter-example is generated for the uncovered scenario, the verification engineer has to decide whether the gap has to be considered harmful. The counter-example provides information on the actual behavior of the circuit in the uncovered scenario. If the behavior does not meet the specification then there is a bug in the design. If the behavior conforms to the specification then there is a coverage gap and the verification has to be completed by adding a property. Afterwards the coverage check has to be performed again.

Table 3. Results for the instruction set verification.

Category	#p	CPU time	max. mem
load/store	6	95.45 s	122 MB
arithmetic	8	487.25 s	153 MB
logical	8	48.65 s	90 MB
jump	5	16.53 s	80 MB
other, reset	6	27.32 s	89 MB

But as for the example above it is also possible that the specification itself is incomplete and the verification engineer has left out certain scenarios intentionally. In this case the unspecified scenario can be excluded from the coverage check by an additional assumption in the coverage property.

In any case our approach provides a feedback for the verification engineer and enables him to reason about the uncovered scenarios. In this way it supports design understanding and it helps to improve the quality of the verification.

4.3. Top-Level Verification

Based on the successful verification of all involved hardware blocks, the instruction set of the RISC CPU has been formally verified. A property has been formulated for each of the 32 instructions that checks if the effects of the instruction meet the specification. These properties affect all of the hardware blocks. Due to page limitation we only give the results of the verification, see Table 3. For details we refer the reader to [7]. The first column gives the category of the verified instructions. The number of properties for the respective category can be found in the second column. The last two columns give the total run-time and the maximum memory needed during the verification.

4.4. Top-Level Coverage

In contrast to the block-level verification the properties for the instructions of the RISC CPU do not consider single outputs or signals. In fact the instruction set verification involves different hardware blocks and their communication. Therefore the notion of coverage at this level is not as clear as for single hardware blocks. Obviously it is not sufficient to prove the coverage of the outputs of the CPU because the input/output interface is only affected by few instructions. To be sure that the properties form a complete specification of the circuit's behavior, the state bits have to be considered as well. If the state of the circuit is uniquely determined at any point in time, its behavior is fully covered in terms of our approach.

As an example the status bits of the RISC CPU are considered – the zero flag and the carry flag that indicate the

```
1  assign OPCODE = instr.range(15,11);
2  assign DEST   = instr.range(10,0);
3
4  property JMP =
5  always(
6    reset == 0 &&
7    OPCODE == "11110"
8  ) -> (
9
10   // jump to target address
11   next( pc.pc == DEST ) &&
12
13   // no sideeffects
14   next(
15     (stat.C == prev(stat.C)) &&
16     (stat.Z == prev(stat.Z)) &&
17     (reg.reg[0] == prev(reg.reg[0])) &&
18     (reg.reg[1] == prev(reg.reg[1])) &&
19     [...]
20     (reg.reg[7] == prev(reg.reg[7]))
21   )
22 );
```

Figure 5. Property for the jump instruction.

result of the last logical or arithmetic operation (see also Figure 1). Among the properties of top-level verification there are 32 properties for the instructions and a reset property. In order to achieve full coverage with this partitioning of the properties, *every* property has to define the value of the status bits, regardless whether the respective instruction changes the flags or not.

As an example consider the property for the jump instruction in Figure 5. It states that whenever there is no reset (line 6) and the current instruction is a `JMP` (line 7) then the program counter is set to the target address in the next cycle (line 11). This obviously describes the correct behavior of the jump instruction. However, in order to achieve full coverage the property must also specify what the jump instruction does *not* do. This is expressed in lines 14 to 21 which state that in the next cycle the status bits and the content of the registers remain unchanged.

As a consequence, all properties have to be assumed in the coverage property for the status bits. In this way, a large monolithic property is generated including *all* instructions. However, the final coverage proof could be carried out faster than some of the original properties that have been used for verification. The coverage proof took 72.68 CPU seconds and used 119 MB of main memory. This clearly demonstrates the feasibility of our approach. It could be proved that under all circumstances the value of the status bits is unambiguously defined by the instruction set properties.

5. Conclusions

In this paper we have demonstrated how coverage estimation can be applied in Bounded Model Checking. In a case study it has been proven that all the outputs of the hardware components of a RISC CPU are fully covered by a set of properties. Results on run-time and memory usage of the verification and the coverage check have been provided. It could be shown that the costs for the coverage check are comparable to the verification costs.

Furthermore we have presented an approach how the technique from [8] can be applied on a higher level. As an example we have shown that the status bits of a RISC CPU are fully described by a set of properties used for the instruction set verification. In order to achieve full coverage, the properties have to be written in an appropriate style.

Overall, the techniques presented here offer improvements in the quality of verification and in design understanding while being easy to integrate.

References

[1] *Accellera Property Specification Language Reference Manual, version 1.1*. http://www.pslsugar.org, 2005.

[2] N. Amla, X. Du, A. Kuehlmann, R. P. Kurshan, and K. L. McMillan. An analysis of SAT-based model checking techniques in an industrial environment. In *Correct Hardware Design and Verification Methods (CHARME)*, pages 254–268, 2005.

[3] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *LNCS*, pages 193–207. Springer Verlag, 1999.

[4] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:506–521, 1960.

[5] N. Eén and N. Sörensson. An extensible SAT solver. In *SAT 2003*, volume 2919 of *LNCS*, pages 502–518, 2004.

[6] D. Große and R. Drechsler. *CheckSyC*: An efficient property checker for RTL SystemC designs. In *IEEE International Symposium on Circuits and Systems*, pages 4167–4170, 2005.

[7] D. Große, U. Kühne, and R. Drechsler. Hw/sw coverification of embedded systems using bounded model checking. In *Great Lakes Symp. VLSI*, pages 43–48, 2006.

[8] D. Große, U. Kühne, and R. Drechsler. Estimating coverage in bounded model checking. In *Design, Automation and Test in Europe*, 2007.

[9] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[10] J. Marques-Silva and K. Sakallah. GRASP – a new search algorithm for satisfiability. In *Int'l Conf. on CAD*, pages 220–227, 1996.

[11] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conf.*, pages 530–535, 2001.

[12] Synopsys Inc., CoWare Inc., and Frontier Design Inc., http://www.systemc.org. *Functional Specification for SystemC 2.0*.

[13] K. Winkelmann, H.-J. Trylus, D. Stoffel, and G. Fey. Cost-efficient block verification for a UMTS up-link chip-rate coprocessor. In *Design, Automation and Test in Europe*, volume 1, pages 162–167, 2004.