A Practical Approach to the Formal Verification of SoCs with Symbolic Model-Checking

Emil Dumitrescu

A Practical Approach to the Formal Verification of SoCs with Symbolic Model-Checking

Emil DUMITRESCU
TIMA Laboratory, Grenoble, FRANCE

Abstract: The successful application of model-checking to industrial designs calls for a minimal set of efficiency criteria. This work addresses these issues, based on the linear-time model-checking verification of an instruction cache controller designed by ST Microelectronics.

1. INTRODUCTION

We present a formal verification approach concerning the main modules of a SoC. This approach has been experimented on a system designed by ST Microelectronics: the digital core of a cellular phone, which implements two main features: one real-time component performs digital signal processing, and implements the communication protocol; the second component is in charge of all user-interface functionalities.

Given its high degree of complexity, the elaboration of this system involves a large design team. The verification task is separated from the actual design and performed by verification engineers who are not involved in the design process. This paper presents, from the point of view of a verification engineer, a set of model checking verification strategies applying to the different parts of a SoC. The presentation is illustrated using, as a running-example, one SoC module: an instruction cache controller.

Model Checking state of the art

Temporal logic and symbolic model checking present considerable advantages for the specification, analysis and verification of real-life circuits. It is now common practice to use this technique and traditional simulation jointly, as two complementary tools. Unfortunately, the use of model checking on an industrial design often requires non-negligible efforts, despite the "push-button" advantage advertised for this technique. The fully automated approach of symbolic model checking is limited by combinational explosion of the BDD model representation [4], [1]. To solve the exponential space complexity problem, two main directions are being explored.

Various alternative representations have been successfully tried. Word-level operations have shown efficient symbolic representations [2]. As for the

general representation of a controller state space, ZBDDs [6] seem to bring a good spatial improvement compared to traditional BDDs.

The second important research direction is the enhancement of symbolic model-checking algorithms. Several algorithms are very efficient for a particular class of circuits, but can give bad results if applied to another class [9]. In [7], a heuristic analysis of the circuit representation, based on variable dependency matrices, is presented, and can be used to choose the appropriate verification algorithm, if it exists.

Despite these improvements, the asymptotic exponential complexity has not been eliminated. Besides, some of them are not implemented inside industrial verification tools accepting a standard HDL entry. Hence, care must be taken when using this technique for verifying an arbitrary circuit. Some essential guidelines for model checking a design already exist. For instance, the proof of arithmetic operations is very inefficient. As for control parts, they cannot have an arbitrary size. Deciding whether a circuit is reasonably sized depends on several aspects: number of lines, number of state variables, complexity of the transition and output functions, the importance and the size of the data part, as well as the possible depth of the corresponding finite state machine. All these aspects require a good knowledge of the circuit.

On the other hand, classical efficiency guidelines concerning model checking include design decomposition according to assume-guarantee techniques [3], data-path abstraction, case splitting, and data type reduction [5]. These techniques successfully apply to most common RT-level designs but they require a good knowledge of the existing temporal logics, and also of the symbolic model checking mechanism.

In this paper we show the practical application of several of the above techniques on a SoC design. Section II presents the sub-module taken as running-illustration together with the characteristics of the VHDL model. Section III is the core of this paper, in which we formalize the design specification and explain the various simplifications that we applied to the model to check the requested properties. Section IV gives our results: the figures showing that a simplification strategy cannot be avoided. The last section concludes on the task of the verification engineer.

2. THE ST-MICROELECTRONICS CACHE CONTROLLER

2.1 Overview of the architecture

The cache controller is actually a memory interface connecting a DSP on the one side, and internal (on-chip) and external memory banks on the other side, as shown in Figure˚1. At each clock cycle, the DSP can issue a fetch request to the cache controller. If the requested word is present in the on-chip memory, it is delivered with a three clock cycles constant latency. Requests are pipelined towards the internal memory. Each word delivery corresponds to a request received three clocks earlier.

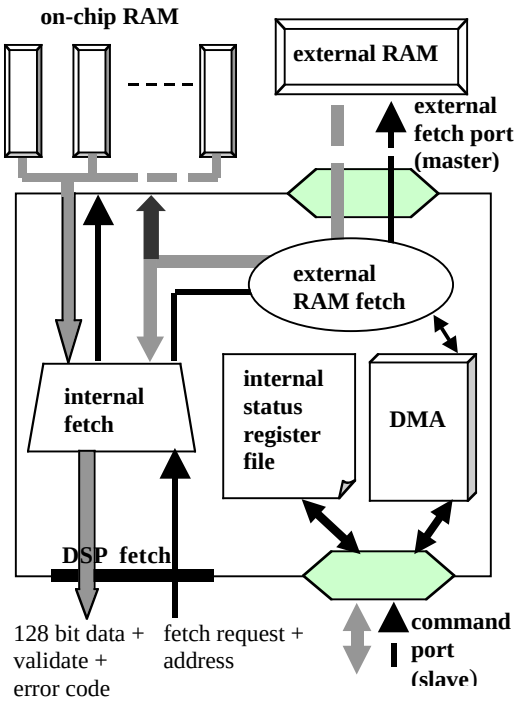

Figure 1 - Architecture of the cache controller

In case the requested word is not in the on-chip memory, the fetch request is transmitted to the *external RAM fetch* engine. A read request is issued through the **external fetch port**. This request may be answered within an arbitrary amount of time, which is usually very long.

Unlike classical mechanisms, cache misses are not followed by an automatic download of a new memory page. It is the responsibility of the DSP to order the refreshment of one particular memory bank. To do so, it transmits a download command to the *DMA* engine, via the **command port,** specifying which memory bank is to be loaded, and which memory page is needed. In parallel, it updates the *internal status register file* to indicate that the bank being loaded is no longer valid. Fetch requests to an invalid bank are answered with an error. The DMA engine controls the external RAM fetch engine, which will perform as many word transfers as needed to fill the bank. When the download completes, the bank is validated.

During the DMA transfer of a bank, using the external fetch port, other valid banks must remain accessible for DSP fetch requests. Moreover, if a fetch request corresponds to an external address, it must be issued. Hence, a DMA transfer and an external fetch must share the port in a fair manner.

However, if a transaction arrives on the command port, internal fetches are interrupted. This happens because the transaction may invalidate a bank, and hence allowing further fetches may be unsafe.

The **internal status register file** records the current status of the cache controller: which banks are valid or invalid, the status of current and pending DMA transfers, and the base addresses of each page recorded inside the internal banks.

2.2 The VHDL model

The whole implementation of the cache controller is described using RT-level VHDL. A simple code inspection has revealed that this design does not present any explicit hierarchy indications. The separation between the different engines

represented above (internal fetch, external fetch, DMA, etc.) is not clearly done by the designer, who has preferred to keep all these behaviors inside the same VHDL architecture. Thus, these engines are implemented as collections of interconnected clock synchronized and combinational processes. Hence, it is difficult to reason about the different parts of the design in an independent manner, by applying hierarchical design decomposition.

The control part of this design is predominant. However, transition functions often use chained comparisons between bit vector objects, for instance each time an address is analyzed. This complicates their symbolic representation.

A small arithmetic computation is implemented inside the DMA engine: a counter indicates the index of the last word transferred from the external memory to an internal bank. The upper limit of the counter reflects the size of a bank, which currently equals 1024 words. The reachability analysis step may become extremely inefficient, because the state space exploration of this counter requires 1024 iterations, which are added to the exploration of the global system.

Besides, reasoning about aggregate objects can generally not benefit from symmetry considerations, because the different parts of a bit vector most often follow different paths.

The most representative quantitative figures concerning this design are the following: it is about 1500 lines long. The VHDL elaboration gives a total number of 1000 flip-flops. Address busses are 30 bits wide, and data buses are 32 bits wide. The width of the fetch port is 128 bits.

Given the features of this design, it is impossible to use the ÒpushbuttonÓ approach of model checking. A number of preliminary manipulations are necessary, which are presented in the following section.

3. THE VERIFICATION TASK

3.1 The formal specification

The specification document of a circuit implicitly defines the properties that need be verified. Ideally, a verification engineer should not even have to look inside the HDL code, as it should be enough to formalize and verify all the assertions given by the specification. If all these assertions are proved correct, then the circuit may be considered ÒcorrectÓ However, a specification is rarely clear and explicit. Many aspects are often left unspecified; hence, the first important step to achieve is to set up a verification plan addressing the key behaviors of the design. The correctness of the design is assessed with respect to this plan. Obviously, it is important to define a large number of unrelated assertions.

The set of assertions written define the formal specification of the design. They are split into two classes: environment assumptions and design properties.

3.1.1 Environment assumptions

Constraints on the environment are most often required, as designs usually work under the assumption that it never exhibits an unexpected behavior. An environment constraint is a collection of invariants declared either as Boolean formulae, or as simple state machines. For example, a Boolean formula could say: "it is never true that the DSP sends a request while the cache controller has its DSP fetch stalled". A simple state machine environment would say: "after a request signal is risen, it must remain high, until an acknowledge is received".

The environment assumptions are composed with the finite state machine description of the design. Unfortunately, the size of the result is usually much bigger than the design alone, except for very simple assumptions that assign constants to primary inputs, allowing obvious simplifications on the design.

3.1.2 Design properties

The design properties are temporal formulae that the design should satisfy. Safety and liveness can usually be expressed. In our experience, response times are always constant, or at least bounded. Hence, instead of expressing liveness requirements, which are usually very hard to prove, similar safety properties can be written.

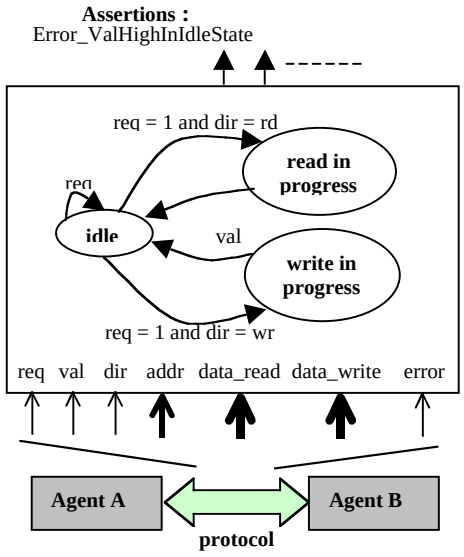

Figure 2 - A monitor specification for the command and external fetch ports

The concept of monitor is a very interesting alternative to temporal logic for writing properties [8]. Figure 2 displays the principle that we have applied. Let A and B be two components communicating with a protocol on their interconnection signals. The monitor takes as inputs the signals of interest that must be observed, and provides as outputs one bit for each Boolean formula that must evaluate to true. It is written as a component, which is instantiated together with A and B in the enclosing architecture. The body of the monitor itself can be written as one or more finite state machines whose state transitions are triggered by events on the observed signals.

For instance, in Figure 2, one output displayed says that an error occurs if a "validate" is asserted without a previous request. It is verified like an invariant. Thus, reasoning about the past is done in a natural way using this specification method, unlike most classical temporal logics available.

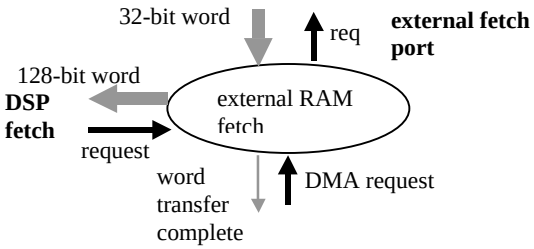

Figure 3 - Isolating one functional block

Writing a monitor as a component has no side effects on the description of the design itself. It allows easy plug-and-play of the added VHDL text for verification purposes, which can easily be removed for synthesis.

3.2 The model simplification

3.2.1 Isolating functional blocks

The most natural verification approach consists in writing a set of properties relating the inputs and the outputs of the cache controller. However, given the size of this design, only a small part of them can be expected to give a result when run by a model checker, as their symbolic representation probably uses an important part of the circuit description. In order to be able to go further, decomposition is vital.

After a detailed code inspection, the different functionalities of the cache controller have been isolated as shown in Figure˚1. It is now possible to reason about each part of the design independently, and write a set of local properties for each one of them. Consider for instance the external RAM fetch engine (Figure 3).

A DMA or a fetch request can trigger the transfer of an external word. Thus, the *req* signal of the external fetch port is asserted together with the address that is required. The following properties should be satisfied:

P1. the *req* signal cannot be asserted spontaneously. At least one incoming request should be present, either on the DSP fetch side or on the DMA side;

P2. conversely, the *req* signal cannot be de-asserted unless a response has been received on the external fetch port. However, it is also true that this request holds as long as the incoming request that triggered it.

The first property states that a *req* rising event can only be triggered by one of the two incoming DSP fetch or DMA requests. The Boolean function defining the *req* signal depends on the values of these two incoming requests. But as the external RAM fetch engine remains connected to the rest of the circuit, these two signals have their own transition functions. They depend on other intermediate signals, and so on, until a set of primary inputs is reached. Thus, the verification of this property is inefficient, due to complicated transition functions that involve both the DSP fetch port and the command port.

3.2.2 Eliminating useless parts

At the local level of the external fetch engine, the behaviors associated to the incoming requests can be abstracted away. Indeed, a request is issued on *req* **when** an incoming request arrives. Hence, we may consider both incoming requests as *primary inputs*. Thus, for verifying these two particular properties, both the DSP fetch and the DMA functional blocks may be eliminated. The transition function of *req* becomes much more simple and the verification of property P1 is done at the local level of the external fetch engine. This simplification is done manually. The verification engineer directs the model-checker to override the transition functions of the incoming requests.

According to these simplifications, the proof of properties P1 and P2 runs on a new model consisting of the external RAM fetch separated from its environment. The interpretation of the verification results is the following. If the property is true at the local level, this result has been obtained under the assumption that **any** behavior can occur on the incoming request inputs. The original implementations of these inputs define more restrictive behaviors, which are necessarily included in the set of all possible behaviors. Hence, the property is also true in the original implementation.

However, in case the property is proven false, it is not safe anymore to draw a conclusion at the global level of the circuit. Erroneous behaviors of the incoming request signals may cause the property failure, while they are not allowed by the initial description. This case is illustrated by property P2, which says that *req* may fall if the incoming requests fall. The same simplified model remains interesting, but it is obvious that the property will fail: the scenario displaying an incoming request rising and then falling can occur, because any behavior is now allowed for the incoming requests. The model must be complemented with the assumption that incoming requests remain high once they are asserted, until a transfer is completed. Thus, the erroneous behavior is eliminated and the property passes.

3.2.3 Using non-determinism

Eliminating a part of a design for simplification purposes means deleting all the behaviors it implements. However, some of the behaviors deleted may still be needed so that the model obtained after simplification remains realistic. In other terms, a simplified model focusing on the desired behaviors must replace the eliminated part of the initial description, while still allowing all other. When an appropriate scenario occurs, the behaviors needed are exhibited. All other scenarios correspond to an unspecified behavior. Thus, the simplified model obtained is both *abstract* and *non-deterministic*.

In our example, a model must implement the assumption required for proving property P2. It must ensure that when a request rises it must remain high until a transfer is completed. However, the request may rise at an arbitrary moment.

The abstract non-deterministic model replacing the initial description is usually written by the verification engineer. Hence, it must be proved correct before it can be used. The correctness criterion states that, while focusing on desired behaviors, the model allows all the behaviors defined by the original transition functions it replaces.

This is synonymous with *refinement checking*[5]. Replacing a part of the design (in this example, the definitions of the two incoming requests) by a simpler, non-deterministic, abstract definition, together with performing the subsequent checks is known as *assume-guarantee* reasoning[3].

3.2.4 Behavioral partitioning

The different functional blocks that are running in parallel may execute independently, or they can influence the execution of each other. Hence, different execution scenarios are possible. Any property a design should satisfy must be valid for all scenarios that may occur.

The verification of a property can be split into a set of sub-goals, one for each interesting scenario. Each sub-goal asserts that the initial property is true under the assumption that only one scenario may occur. If all sub-goals are verified, then the initial property may also be true.

For instance, a verification goal asserts that transactions on the command port of the cache controller are correct. This goal can be split into two sub-goals, according to the following scenarios: (1) only read transactions are allowed and (2) only write transactions are allowed. If both read and write transactions are proved correct, we may conclude that all transaction on this port are correct. The two sub-goals constrain the signal indicating the direction (read or write) to constant values. Thus, the size of their underlying model becomes considerably lower due to BDD simplifications.

Behavioral partitioning acts both as a powerful simplification method and as an environment constraining method. However, its application needs care: if all sub-goals are true, the initial goal is not necessarily true. For instance, assume that a bug occurs only when a read transaction is followed by a write transaction. If behavioral partitioning is employed, this bug shall not be uncovered.

3.3 Classical simplification methods that are inapplicable

When a design manipulates bit-vector objects, a natural simplification consists in reducing their size to only a few bits wide. This is especially appropriate for data. However, control bit-vectors are not always symmetric, as individual bits or bit fields may correspond to different meanings. This is the case for bit-vector objects implemented inside the cache controller. In that case, most properties have to be split into as many sub-goals as there are bit fields in the vector (up to the number of bits in the worst case).

Concerning the refinement-checking technique as a support for the assume-guarantee reasoning, most industrial verification tools do not implement the mechanical check. Thus, before using an abstract model, a manual assessment with respect to the initial design must be performed. In practice, this limits the method to abstract models that are very simple.

4. RESULTS

For verification purposes, the initial model has been slightly modified. In order to solve the reachability problem stated at ⁄2.2 the verification has run under the assumption that memory banks contain only 4 words instead of 1024.

It is important to note that in order to set up a property a "design cycle" must be followed. As it is impossible to know in advance whether a verification task ends within a reasonable amount of time and resources, it is important to establish a limit on the memory size to be used, and kill the verification if this limit is exceeded. If this happens, a model simplification should be attempted before re-trying a new verification.

The cache controller has been verified using Cadence FormalCheck v2.3 on a Sparc Station Ultra 2-60, running at 400 MHz with 2 Gbytes of memory.

A total number of 50 properties have been written. We highlight a few interesting classes of properties that we have written, as well as the importance of the model simplification used in achieving good verification times (Table 1).

Behavioral partitioning is the only model simplification needed for verifying the *internal fetch (IF)* functional block: to make sure that fetches are not interrupted, the incoming requests on the command port are deactivated. Two properties are representative:

IF1. a fetch request is acknowledged within 3 clock cycles;

IF2. a rising acknowledge corresponds to a fetch received 3 cycles earlier.

The verification of the *external RAM fetch (ERF)* functional block has needed more important model simplification efforts, as it had to be cut from the rest of the design. We highlight two properties for this block:

ERF1. external fetch requests are never retracted;

ERF2. DMA and internal fetch incoming requests are treated fairly.

The *DMA* functional block must comply with a very important requirement (DMA): it must not write into the internal RAM if it is in use. The model is simplified by cutting the functional block from the remaining design, and by using behavioral partitioning.

Table 1 — Different properties and verification results

Functional block	Temporal logic (TL) or monitor (M)	State variables/ model depth	Model simplification technique	Time (sec)/ Memory (MB)
IF-1	TL	153/19	behav.part.	63/16,5
IF-2	M	163/19	behav.part.	52/16,5
ERF-1	M	210/ -	-	out of memory
	M	10/11	cut	49/16.5
ERF-2	TL	300/-	-	time limit exceeded
	TL	256/33	cut	65/31
DMA	TL	310/ -	-	out of memory
	TL	147/57	cut + behav.part.	62/16

In Table 1, the first two columns list the properties and the way they are written; the third column gives the number of state variables and the number of iterations necessary to check a property. The fourth column tells which simplification technique, if any, has been applied. The last column gives the performance in computation time and memory.

For properties ERF1,˚2 and DMA, we see that simplification by cutting away a functional block is essential. If this technique is not applied, either the proof runs out of memory or it lasts several days, which is impractical. One third of the 50 properties written have been verified using functional cut. For the remaining properties, behavioral partitioning and environment constraining assumptions were sufficient to obtain verification answers within useful time.

5. CONCLUSION

The set of model simplification guidelines presented here allowed a reasonable functional coverage for the verification of the cache controller design. All the functional blocks have been addressed, despite the lack of hierarchy in the RTL description of the design.

A few minor incompatibilities have been faced, concerning the synthesizable HDL code used by the designer team, which is not always recognized by the verification tool.

This work also raises an interesting question: would the verification have been quicker if the designers had partitioned the design? In theory, the answer should be ÒyesÓ in practice, it is probably ÒnoÓ In order to avoid looking inside a component, its documentation should be very detailed, clear, and maintained up to date as the design evolves, which is very hard to achieve. Using the existing documentation as an aid to understanding the behavior of a component seems more realistic.

Thus, we believe that in the current context, given the available tools, it is essential that verification engineers are aware of the implementation details of the design they verify. On the other hand, the ignorant application of automatic model-checking fails in most cases. Knowledge about the underlying theory of this technique is vital.

REFERENCES

[1]. R. E. Bryant, *Graph-based algorithms for Boolean function manipulation*, IEEE Transactions on Computing, vol. C-35, no. 8, Aug. 1986, pp. 677-691

[2]. R. E. Bryant, Y. Chen, *Verification of arithmetic circuits using binary moment diagrams*, International Journal on Software Tools for Technology Transfer, vol. 3, no. 2, Springer, pp. 137-155, Mar. 2001

[3]. T. A. Henzinger, S. Qadeer, and S. K. Rajamani, *You assume, we guarantee: methodology and case studies* Proceedings of the Tenth International Conference on Computer-aided Verification (CAV 1998), Lecture Notes in Computer Science 1427, pp. 440-451

[4]. K.L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, Boston, MA, 1994

[5]. K.L. McMillan, *Verification of infinite state systems by compositional model checking*, in L. Pierre and T. Kropf, editors, CHARME99, vol. 1703 of LNCS, pp. 219-233

[6]. S. Minato, *Zero-suppressed BDDs and their applications*, International Journal on Software Tools for Technology Transfer, vol. 3, no. 2, Springer, pp. 156-170, Mar. 2001

[7]. Moon, J. H. Kukula, K. Ravi and F. Somenzi, *To Split or to Conjoin: The question in Image Computation*, in Proceedings of DAC, Los Angeles, CA., June 2000, pp. 23-28

[8]. K. Shimizu, D. Dill and A. J. Hu, *Monitor Based Specification of PCI* - Proceedings of the Third International Conference of FMCAD, November 2000

[9]. R.K. Ranjan, A. Aziz, R.K. Brayton, B.F. Plessier and C. Pixley, *Efficient BDD algorithms for FSM synthesis and verification* Presented at IWLS95, Lake Tahoe, CA, May 1995