A Methodology for SoC Top-Level Validation using Esterel Studio

Lionel Blanc
Esterel Technologies
Twins 1
679 Av Julien Lefèbvre
+33 (0)4 92 02 40 40
lionel.blanc@esterel-technologies.com

Amar Bouali
Esterel Technologies
Twins 1
679 Av Julien Lefèbvre
+33 (0)4 92 02 40 40
amar.bouali@esterel-technologies.com

Jérôme Dormoy
Esterel Technologies
Twins 1
679 Av Julien Lefèbvre
+33 (0)4 92 02 40 40
jerome.dormoy@esterel-technologies.com

Olivier Meunier
Esterel Technologies
Twins 1
679 Av Julien Lefèbvre
+33 (0)4 92 02 40 40
olivier.meunier@esterel-technologies.com

ABSTRACT

In this paper, we suggest a methodological framework addressing the System on Chip top-level validation. Our systematic approach works on a powerful abstraction of IP blocks called transactional model. We state the transactional modeling "philosophy", its benefits, and its limitations. The methodology is illustrated step by step on a simplified example. Its effective implementation is realized within Esterel Studio tool synchronous approach, formal verification and test generation capabilities. This methodology is an opportunity to reduce SoC top-level validation time, increasing confidence through larger and relevant functional coverage.

Categories and Subject Descriptors

M.1.6 [**Design Methods Track**]: Testing, test generation and debugging.

General Terms

Design, Verification.

Keywords

System on Chip, Integration, Validation, Transactional Model, ATPG, Synchronous Approach, ESTEREL STUDIO, SYNCCHARTS.

1. INTRODUCTION

When it comes to validating the System on Chip (SoC), there are two issues to consider. First, we need to verify the IP cores thoroughly and separately. Secondly, we need to verify their integration in the system; arduous challenge in design of very large chips. Indeed, the number of test vectors to exhaustively test SoC grows exponentially with the number of components and inputs of the system. Now, compound behaviors of multiple IP blocks generate huge state spaces, such that it is humanly impossible to consider all the test vectors that could cause the design to fail. In this context, Automatic Test Pattern Generation (ATPG) constitutes a rationalization of the SoC testing process. The most widely used approach to generate verification tests automatically is pseudo-random generation. However, coverage is often doubtful. Formal methods may be efficiently applied to generate deterministic test vectors covering all the possible behaviors rather than a limited and uncertain range of behaviors [7]. However, behavior models need to be complete and accurate. Unfortunately, in the present SoC framework, most of the IP cores design knowledge remains with the IP provider or designer. Even though one would have the necessary knowledge, time-to-market pressure is incompatible with IP core formalization.

From the previous paragraph, it clearly appears that:

- Apply formal methods to automatically generate test vectors provides some consistence to the coverage.
- Due to time-to-market pressure, insufficient IP core knowledge (and potential combinational explosion), it is not reasonable to formally prove complex chips.

In this context, the paper suggests a methodological framework for SoC top-level validation, test-oriented and based on formal synchronous approach using Esterel Studio. Esterel Studio development environment provides behavioral specification, simulation, formal verification, automatic test generation, coverage analysis and code generation.

The paper is organized as follows: Section 2 introduces the simplistic EASY application for methodology illustration purpose. Section 3 familiarizes the reader with the synchronous approach. Section 4 presents the methodology's different steps and benefits. The

Conference '00, Month 1-2, 2000, City, State.

methodology steps are first exercised on the EASY example. Then we give some guidelines to implement this methodology on real industrial cases. Finally, Section 5 summarizes the methodology characteristics and presents a few perspectives.

2. THE "EASY" APPLICATION

In this section, we present a simple system called "EASY" to illustrate the methodology. The "EASY" example is composed of:

- a CPU (embedding software),
- three IP blocks performing functions. These IP blocks could represent I/O peripheral, CODEC components, etc. In a generic way, IP blocks can be viewed via their registers, i.e. to access a function one needs to access registers (set i^{th} bit to 1, etc.). IP1, IP2 and IP3 blocks export their functions via one or two command registers (Figure 1). Basically, the CPU or a client IP block can read or write registers and wait for a "done" event such as a completion interrupt.

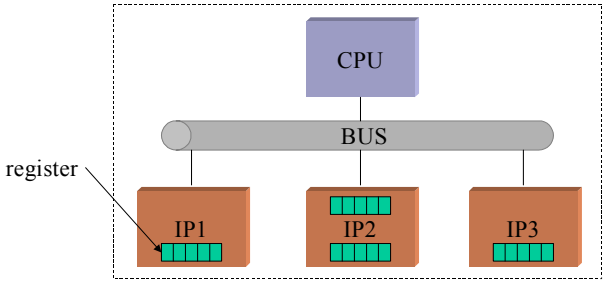

Figure 1: "EASY" example

3. THE SYNCHRONOUS PARADIGMS

ESTEREL [3] and SYNCCHARTS [2] (E&S) synchronous languages are dedicated to control-dominated reactive system modeling. These systems exhibit fundamental differences in comparison with "classical" algorithmic systems:

- they have to handle a large set of sporadic and/or periodic events potentially simultaneous,
- they have to cope with concurrent behaviors,
- they have to perform complex synchronizations.

Therefore, E&S introduce preemptions, suspension, concurrency and hierarchy as native concepts.

These formalisms are founded on a mathematically well-defined semantic: a relevant abstraction conferring to design a behavioral determinism and reducing notably combinational complexity to consider. Obviously, this abstraction makes a physical sense since a physical representation compliant with the "abstract" (logical) model can be derived (hardware circuit translation).

The ESTEREL STUDIO compiler compiles a program into a Finite State Mealy Machine (FSM). The FSM internal representation can be either explicit or implicit by means of Boolean equation systems. This model is the common base for simulation, code generation, formal verification, and test generation. Model-checking and test generation are performed using efficient BDD techniques [4].

ESTEREL STUDIO [1] unifies E&S formalisms (see Figure 2). It provides a design chain without discontinuities from specification to validated hardware or software code. Therefore, it is a sound candidate in a top-down co-design flow [6]. ESTEREL STUDIO provides an automatic test pattern generator, handling reachable state space (RSS) via BDDs. The aim is to generate deterministic sequences ensuring the state coverage completeness, i.e. produce test vectors in order to cover 100% of the RSS.

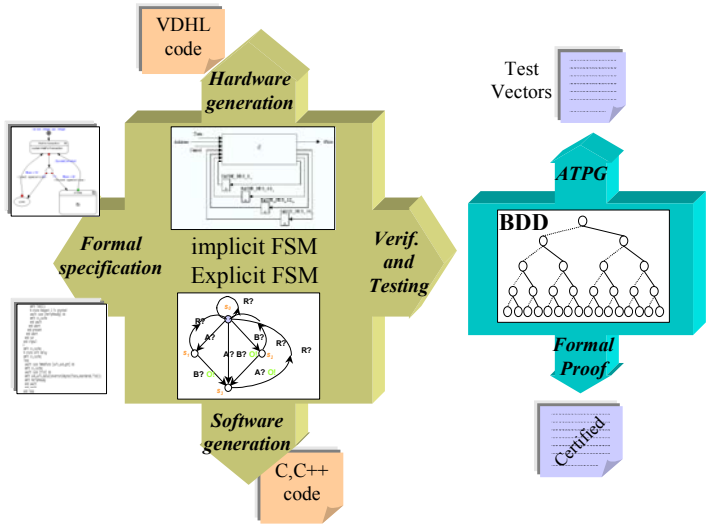

Figure 2: Esterel Studio overview

4. THE SUGGESTED METHODOLOGY

Our approach assumes that IP and components to integrate are locally validated by the providers. So, the integration phase consists in validating a multiple block composition, i.e. validate the proper interoperability and interactions among SoC blocks, e.g. that there are deadlock-free. In the current SoC framework (re-use and IP approach), we have to take into consideration the following statement: the accurate knowledge of IP blocks remains with the IP providers (both for trusted party IP or for internally developed IP). Therefore, a "client" of an IP block generally only knows the services (functions) that it renders: it is the user's view. The *service* notion is a

software-oriented concept; the *transaction* is a more suitable term in a hardware context characterizing ordered service accesses. A transactional description (TD) considers IP blocks via the service access points or interface ports: it is a "black-box" view (high-level description). TD gets rid of complete and heavy cycle accurate behavior modeling. TD exhibits the following major characteristics:

- it does not require a high-level expertise (easy to model and to understand); transactional models are easy to express and intuitive! Transactions essentially express asynchronous requests – responses,
- it allows a flexibility on block boundary (interfaces), i.e. transactional models do not need to be boundary accurate,
- it is not implementation-dependant. Indeed, transactions request temporally ordered services; the way the services are provided does not impact on them! Therefore, common transactional models do not need to be cycle accurate. In certain cases, especially to show evidence of deadlocks, "gray-boxes" can be modeled. Gray-boxes are hybrid description coupling transactional with local cycle-accurate modeling.

So, the privileged approach to validate SoC integration is the following:

- design transactional models (TM) using a formal approach. TM constitutes formalized TD;
- from these deterministic models, automatically generate covering test vectors to exercise SoC.

ESTEREL STUDIO (ES) is the appropriate tool to achieve this methodology given that:

- E&S formalisms are particularly suited to TM (indeed, transactions denote high-level protocols[1])
- it offers a powerful symbolic ATPG.

Note that TM is a deterministic "sampling" of the SoC behaviors, i.e. only a subset of significant behaviors is considered. ES makes it possible to consider all the possible behaviors of this abstraction.

4.1 A Step-by-Step Description

In this section, we precisely describe the methodology using a step-by-step representation:

Step 1: ***Describe each IP block as black-boxes at a transactional level***

At this stage, IP blocks only export the offered and rendered services. In a generic way, a service may be arbitrarily acknowledged or not (Figure 3).

[1] contrary to bus level protocol

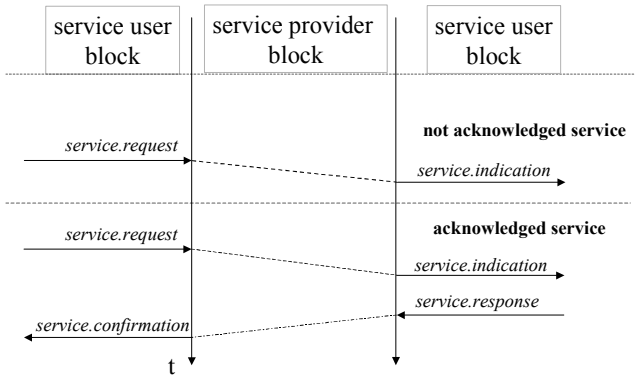

Figure 3: Service access transaction

Now, let us consider the "EASY" application. On Figure 4, we characterize IP1 block at transactional level via its register. This block provides two unacknowledged services: reading or writing (requests) inside a command register and the associated "done" indications. These synchronization events allowing the next reading or writing access (a kind of READY signal).

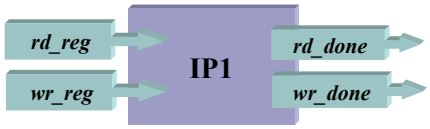

Figure 4: I/O block black-box view

IP block services can often be viewed through (sequentially ordered) *Read / Write* and *ReadDone / WriteDone* transactions. Indeed, realize a service often consists in updating a register content.

Step 2: ***Model the CPU as a transaction initiator***.

In this step, the CPU is viewed as a way to stimulate IP block services:

- Design transactional model for each IP block in order to trigger and synchronize its services (named Service Access model). So, there are as many Service Access (SA) models as there are IP blocks.
- Put in parallel (concurrent composition) all the SA models.

Note that, the CPU core interface should aggregate the conjugated interfaces of each SA model (see example hereafter).

A CPU core may generate ordered deterministically service accesses, but generally, requests are sporadic. Therefore, non-deterministic interleaving among services may be needed to "cover" realistic transactions (requests are sporadic or asynchronous by essence). E&S formalisms consider non-determinism as an environment property. So,

we can cope with non-determinism by introducing extra-inputs.

On Figure 5, each IP block involved in the "EASY" platform is described as a black-box (resulting from step 1).

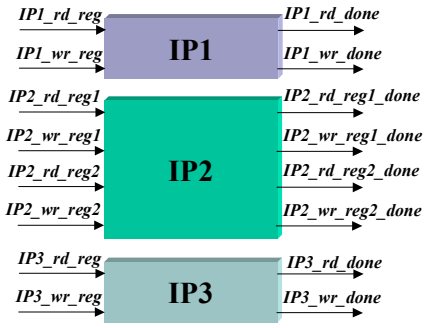

Figure 5: "black boxes" IP for "EASY" platform

Then, the CPU core interface (Figure 6) is the aggregation of all IP blocks conjugated interfaces, i.e. each element produced by an IP block is consumed by the CPU (for synchronization reason) while each element consumed by an IP block is produced by CPU. So, the CPU core is client of the IP blocks.

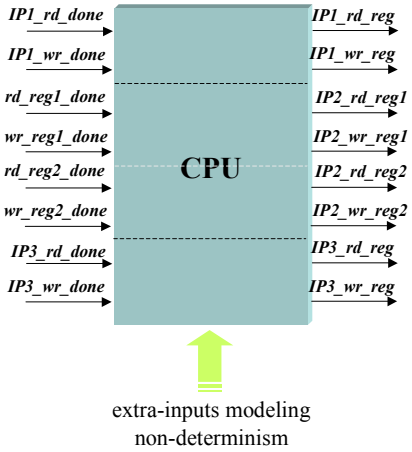

Figure 6: CPU core interface

Concerning the SA model (behavior at transactional level) of IP blocks, we opt for a generic model abstracting a register (Figure 7). The "left path" models the reading access to the register while the "right path" models the writing access to the register. Note the asynchronous waiting of "done" events as well as "*CPU_read_req*" and "*CPU_write_req*" extra-inputs to introduce a non-deterministic interleaving among accesses to the register. Next, all the SA models are concurrently composed (see Figure 8) to constitute a non-constraint CPU, i.e. all the IP blocks are fully independent. Note the multiple instantiation of "register" module to build IP2 block behavior.

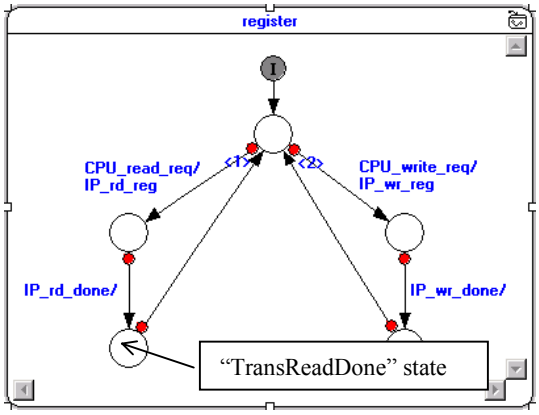

Figure 7: transactional model of a register

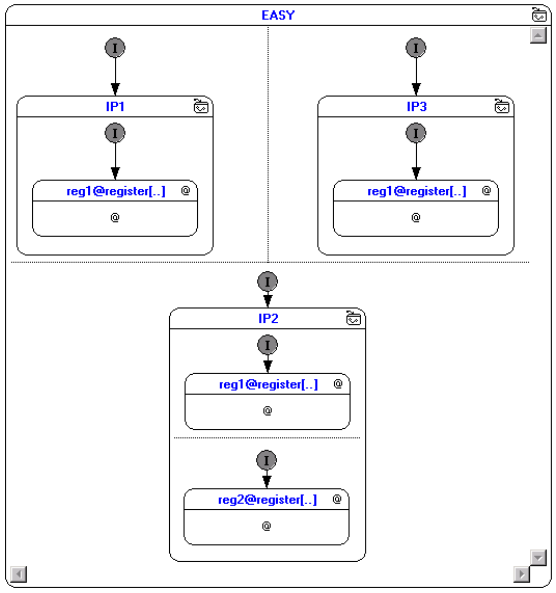

Figure 8: CPU model

Step 3: ***ESTEREL STUDIO ATPG generates top-level transactional test patterns.***

ESTEREL STUDIO ATPG supports two generation modes:

Short mode: unitary testing

> for each state *s* of the RSS, generate a sequence from initial state to state *s*.

Long mode is based on the prolongation of short sequences in order to reach as many new states as possible. This mode reduces sequence number and generates deep sequences (exciting several functionalities) closer to handwritten functional sequences. We recommend to use long mode.

ESTEREL STUDIO ATPG applied on "EASY" example gives the following results:

EASY platform in a few number	
Reachable State Space	626
Number of generated sequences	246
Number of transactions (rd/wr – rd_done/wr_done)	≈ 600
Number of cumulated reaction	1576

Hereafter, a partial generated sequence is extracted from the generated test cases (using Esterel Studio Simulator Input format). The considered sequence simultaneously and alternatively excites registers of the three IP blocks. ";" token causes a program reaction with the specified input vector.

```
%-------------------------------------------------------------
----
%----- Sequence:      1
%----- Covering:      205 new states
%----- Total covered: 205 new states
%----- Remaining:     421 uncovered
%-------------------------------------------------------------
;
% Outputs:
CPU_2_IP1_write_req CPU_2_IP3_write_req CPU_2_IP2_write_re
CPU_2_IP2_read_req_1 ;
% Outputs: IP1_wr_reg IP3_wr_reg IP2_wr_reg_2 IP2_rd_reg_1
IP3_wr_done ; % Outputs:
IP1_wr_done ; % Outputs:
CPU_2_IP3_write_req IP2_rd_done_1 ; % Outputs: IP3_wr_reg
CPU_2_IP1_write_req IP3_wr_done ; % Outputs: IP1_wr_reg
IP1_wr_done CPU_2_IP2_write_req_1 ; % Outputs: IP2_wr_reg_
CPU_2_IP3_write_req IP2_wr_done_1 ; % Outputs: IP3_wr_reg
CPU_2_IP1_read_req IP3_wr_done ; % Outputs: IP1_rd_reg
IP1_rd_done CPU_2_IP2_read_req_1 ; % Outputs: IP2_rd_reg_1
```

Figure 9: a generated pattern (partially viewed)

Exercising this sequence onto the CPU model would effectively produce service (read or write) access requests to IP block. On Figure 9, at each simulation step, the CPU produced requests are shown using comments (following the "% Outputs" token).

Step 4: Esterel Studio code generator produces CPU code

From CPU model issued from steps 1 and 2, Esterel Studio automatically generates consistent code (in the privileged target language[2]).

Step 5: ***Generated transactional test cases are mapped on test code***

Step 3 produces a set of transactional level test cases. From these test vectors a testbench should be derived. A testbench is a piece of code which object is to exercise the CPU generated code at step 4. Remember that CPU model characterizes a transaction initiator, which goal is to request sequentially or concurrently IP block services. The CPU model may be viewed like an embedded software generator. However, this component is not autonomous. To be self-sufficient, CPU requires test cases which setup the extra-inputs modeling non-determinism.

In this context, the testbench implements a simple execution machine [3], i.e. a software dedicated to a (co-simulation) platform, which role is to excite IP block codes (RTL, C, etc.) by requesting services (see Figure 10).

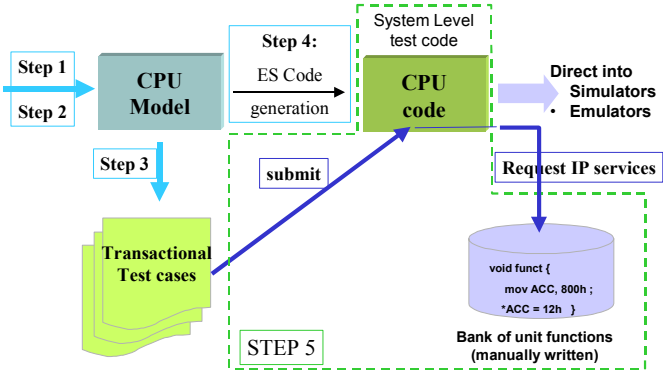

Figure 10: test code

It is a trouble-free task to derive a testbench from generated transactional test cases and generated CPU code. Indeed, the Esterel Studio compiler provides an API exporting:

- the reaction function consistent with specification,
- the interface functions:
  - Input functions to set the environment input events,
  - Output functions to affect the environment.

A naming convention correlates function calls and logical signals issued from model.

[2] C,C++,VHDL,Verilog

Let us now consider the EASY application to illustrate this step.

1) *Derive CPU testbench from transactional test cases* (let us assume that C language is the privileged language), i.e. write the "main" function which exercises the generated code with the generated test cases:

```
void main(){
// first generated sequence
EASY_reset(); // behavior resetting
EASY(); // a reaction => tick
EASY_CPU_2_IP1_write_req();
EASY_CPU_2_IP2_read_req_1();
EASY_CPU_2_IP3_write_req();
// R/W events simultaneity
EASY();
…
// new sequence
EASY_reset();
…
}
```

Note that generated "model_I_signal()" input functions[3] introduce a (first-level of) bufferization ensuring the input vector consistency during the reaction.

Synchronization signals ("done" signals) call for a specific processing. These signals enable to produce functionally and temporally consistent sequences (wait for an acknowledge, etc.). Contrary to extra-inputs which may directly request input interface functions, synchronization event presence depends on the IP block reactivity. Synchronization events should be expanded using the suggested skeleton:

```
void EASY_SYNC_IP1_rd_done() {
  /* Your code {
  e.g. wait for a completion interrupt
  wait_IT(IP1);
  } */
  //Set IP1_rd_done for next reaction
  EASY_I_IP1_rd_done();
}
void main(){
// first generated sequence
EASY_reset(); // behavior resetting
EASY(); // a reaction => tick
EASY_CPU_2_IP1_write_req();
EASY_SYNC_IP1_rd_done();//instead of EASY_I_IP1_rd_done
// Done event simultaneity
EASY();
…
// new sequence
EASY_reset();
…
}
```

Manually written "model_SYNC_sig()" synchronization functions make possible complex command protocols to wait for a service completion.

According to the behavior, a given input vector causes outputs to be emitted during the reaction. These actions are directed via an action table and consist in function calls. For instance, to request write service for IP1, the CPU outputs the "IP1_wr_req" signal. During a reaction, the "IP1_wr_req" signal presence will result in the "EASY_O_IP1_wr_req()" function call.

2) Write "model_O_signal()" output functions or link with existing symbols (library)

The output function content has in charge the effective implementation of the corresponding service using low-level primitives.

```
void EASY_O_IP1_rd_req() {
// for instance
  asm {
     mov AL, 06h
     mov AH, 10h
     …
  }
}
```

The methodology recommends to embed CPU core in the integration platform. Although that is not mandatory, this approach seems to be more homogeneous and allows to dynamically detect timing miss-matches, for instance.

4.2 Scaling-up

Previously, we have focused on a "ideal" transactional method. Unfortunately, the SoC framework is undoubtedly trickier. However, ES remains especially efficient for the reasons described hereafter.

Transactional modeling follows an iterative process: successive refinements are required to produce TM. First refinement delivers a non-constraint CPU model; future refinements introduce integration constraints, environment constraints, etc. In this framework, a modular approach is necessary. Moreover, so that the model development process converges, a k^{th} model refinement should only be a specialization of the $(k-1)^{th}$ model refinement; this characteristic identifies *incremental modeling*. Incremental modeling capability is a "language" property. E&S formalisms offer **modularity and compositionality** properties allowing to apply an incremental modeling approach.

For instance, let us assume that the EASY application now performs a read-burst service only applicable for IP1 and IP3. The read-burst mode realizes several sequential read accesses (statically defined) atomically, i.e. during a read burst mode, "write" actions are forbidden. To model this behavior, we instantiate the "normal" register previously designed, append a textual block modeling the burst mode and introduce a few synchronizations[4] (Figure 11). The

[3] These functions are automatically generated by Esterel compiler to enable a "safe" access to logical signals

[4] in() trigger tests the activity of a state. Register module exports "transReadDone" (Figure 7).

burst size is given by an explicitly defined constant ("cst_size_burst"). Note the ability to develop hybrid graphical/textual description via SYNCCHARTS meta-language.

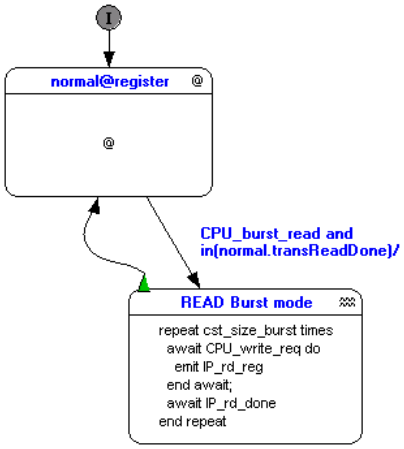

Figure 11: register with a read burst mode

ESTEREL STUDIO ATPG applied on a modified version of the "EASY" model, that is with the read-burst mode, gives the following results, with "cst_burst_size"=5:

EASY platform in a few number	
Reachable State Space	10 001
Number of generated sequences	3 595
Number of transactions (rd/wr – rd_done/wr_done)	≈ 22 000
Number of cumulated reaction	46 091

As the CPU model is under-constrained, i.e. the IP blocks are fully independents, the RSS exponentially grows (therefore, the number of generated sequences and transactions exponentially increases). As a result, it appears fundamental to *control the testbench expansion* in order to keep simulation time compatible with the time-to-market pressure. Testbench expansion may be precisely addressed by directing its tests using environmental constraints. Indeed, constraints often correlate several IP block behaviors, i.e. reduce the RSS.

For instance, let us assume that the EASY application has to take into account the following integration constraint: "*IP1 and IP3 read burst mode are mutually exclusive*", i.e. during a IP1 (respectively IP3) read-burst, IP3 (respectively IP1) cannot request a transfer in read-burst mode.

The E&S modular approach makes the constraint composition effortless. To model the previously defined constraint, we suggest to create a concurrent component (Figure 12) inhibiting, when it is relevant, the register normal mode behavior using communication signals. Inhibiting or freezing a behavior is modeled using the suspension mechanism (Figure 13).

ESTEREL STUDIO ATPG applied on a constrained version of the "EASY" model, that is with the exclusive read-burst mode constraint, gives the following results, with "cst_burst_size"=5:

EASY platform in a few number	
Reachable State Space	2 626
Number of generated sequences	756
Number of transactions (rd/wr – rd_done/wr_done)	≈ 3 900
Number of cumulated reaction	7 960

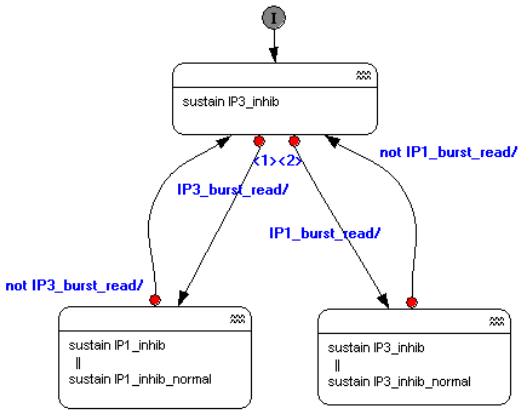

Figure 12: integration constraint modeling

Hardware architecture commonly embeds software. Embedded software debugging should be addressed as soon as possible in the SoC development process. Transactional models may serve as hardware reference model to validate, by advance, the embedded software before the availability of precise hardware description (RTL for instance). As a result, this approach can save simulation times compared to costly cycle accurate model.

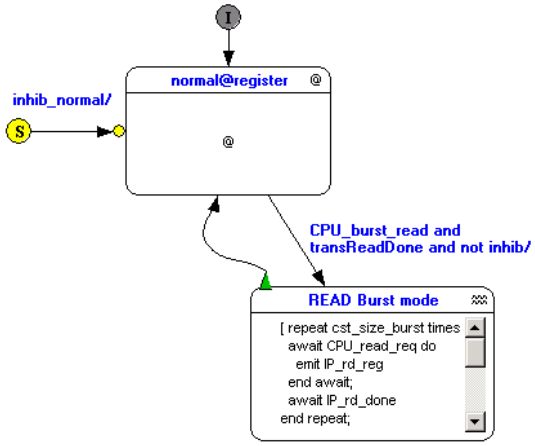

Figure 13: Integration constraint modeling

5. CONCLUSION

We have presented a methodology for the validation of IP blocks interoperability in SoC designs. SoC top-level validation is a touchy and decisive stage in the current IP approach and reuse framework. In traditional testing, the IP blocks are seen as black boxes. We have presented a methodology that adds flexibility, increases the quality of tests, while reducing the validation time for IP blocks interoperability validation. Our methodology suggests the use of a synchronous formalism such as E&S within which the SoC design is modeled: IP blocks are easily described as concurrent transactional models (TM) with different levels of abstraction. TMs appear to be a sound abstraction and provide enhanced scaling-up capabilities.

6. REFERENCES

[1] Esterel Studio in a nutshell
http://www.esterel-technologies.com/esterel/nutshell.htm

[2] André C. "Representation and Analysis of Reactive Behaviors: A Synchronous Approach", CESA'96, IEEE-SMC, Lille, France

[3] Berry G. "The Foundations of Esterel, Proof, Language and Interaction": Essays in Honour of Robin Milner, G. Plotkin, C. Stirling and M. Tofte, editors, MIT Press, Foundations of Computing Series, 2000.

[4] Bryant, R.E, "Graph-Based Algorithms for Boolean Functions Manipulation", IEEE Transactions on Computers, C-35(8):677-691, 1986

[5] Boufaied H. "Machines d'exécution pour langages synchrones". PhD thesis (in french), Doctorat de l'Université de Nice-Sophia Antipolis, 1998

[6] Pelissier G. et al. "Using Esterel Approach to Design Complex Systems", EDP 2001, Monterey, CA

[7] L. Arditi et al., "Using formal methods to increase the confidence in the Validation of a Commercial DSP", in proc. of the Int'l ERCIM workshop on Formal Methods for Industrial Critical Systems, FMICS 1999.