Coverage Directed Generation of System-Level Test Cases for the Validation of a DSP System

Laurent Arditi[1], Hédi Boufaïed[1], Arnaud Cavanié[2], Vincent Stehlé[2]

Texas Instruments[1] and Simulog[2]

Abstract. We propose a complete methodology for the automatic generation of test cases in the context of digital circuit validation. Our approach is based on a software model of the system to verify in which some modules are written in the Esterel language. An initial test suite is simulated and the state coverage is computed. New test sequences are automatically generated to reach the missing states. We then convert those sequences into system-level test cases (i.e. instruction sequences) by a technique called "pipeline inversion". The method has been applied for the functional validation of an industrial DSP system giving promising results.

1 Introduction

Although formal verification is now well accepted in the micro-electronics industry, it is only a complementary technique and the primary validation and verification methods are still based on simulation. However, it has been shown that formal methods can also raise the quality of simulation-based verification while decreasing its cost[9]: a formal analysis of a circuit can generate tests which are then simulated. The tests are efficient because they are targeted at some coverage criteria[12][15][2]. That technique is usually called "white-box" verification since it uses the internal structure of the circuit to verify in order to generate the tests.

Our contribution extends this methodology to the system level. It is based on a software model of the system to verify which is partly written in the Esterel language[3]. The functional validation relies on an initial test suite constituted of system-level test cases (i.e. assembly language programs) which is run on the software model. The modules written in Esterel are formally analyzed so that it is possible to get an accurate measure of the state coverage provided by the initial test suite. Test sequences (also called "test vectors") are then automatically generated at the module level[1]. The next step is to translate those sequences into system-level test cases. We propose a technique called "pipeline inversion"

1. Texas Instruments. MS 21. BP 5. 06270 Villeneuve Loubet. France. *Tel:* +33 4 9322 2856. *Fax: +33 4 9322 2275. E-mail:* larditi@ti.com, h-boufaied@ti.com

2. Simulog. Les Taissounnières, Route des Dolines. 06560 Valbonne. France. *E-mail:* cavanie@simulog.fr, stehle@simulog.fr

to achieve that goal. Those test cases can then be reused for the validation of the hardware model as well as the physical chip.

We have applied our methodology for the validation of a commercial Digital Signal Processor (DSP) system built around Texas Instruments' TMS320C55x™[22]. For all system modules, we could significantly increase the state coverage (up to 100% for some modules).

Section 2 presents the context of our work. We describe in Section 3 our technique to automatically generate tests and in Section 4 their extension to the system-level by pipeline inversion. Practical results are given in the following Section. Finally, we discuss the advantages and weaknesses of our approach, while comparing with related works.

2 Modeling and validation flows

This Section describes the existing modeling and validation flows which have been applied before introducing formal methods in the project. We will show in the following Sections how formal methods could be integrated into those flows to increase the validation efficiency.

2.1 The TMS320C55x™ DSP system

Texas Instruments' TMS320C55x™[22] is a new ultra-low power DSP targeted at the third generation of wireless hand-sets. It is the successor of the popular TMS320C54x™[21]. This is not only a DSP-core but a whole system-on-chip because it includes a CPU (DSP-core) and various system modules such as internal memory controllers, external memory interface, Direct Memory Access (DMA) controller, instruction cache, peripheral bus controller and host-processor interface.

2.2 The software model

The final chip which has been designed was synthesized from a hardware model developed in VHDL language[13] at the Register Transfer (RT) level. We call it the "VHDL model". In parallel, a software model of the system was designed. We call it "C model", because it is written mainly in C/C++ language.

Both models share the same architecture and the same interfaces, down to a single level of sub-modules. From a temporal point of view, the C model is cycle-accurate, which means that signal commutations done within a cycle happen in the mean time. For each cycle, all output signals are equivalent on both C and VHDL models, given that the input signals are equivalent.

There are advantages in having a software model of a hardware system:

- Because software is easier to correct and to modify than hardware, software's development cycle time is shorter than hardware's. A software

model is thus functionally ready earlier than the hardware system it represents.

- A software simulator is an environment which provides elegant ways of solving synchronization and probing problems inherent to hardware instrumentation. This facilitates instrumentation, which eases development and performance analysis at the target level.

Those advantages allow to give the customers a usable model of the chip very early, letting them begin development on their side. It also makes possible to write test cases early, ensuring those tests are functionally correct, without waiting for the VHDL code to be ready.

Current hardware description languages development environments give the designer the possibility to simulate his code. Nevertheless, a separate software model (a simulator) has advantages that a Hardware Description Language (HDL) one lacks:

- A dedicated simulator is faster than HDL simulation by an order of magnitude at least.
- It is possible to decline the software model with various interfaces: command line, GUI, co-design frameworks,...
- It is even possible to encapsulate it with an HDL interface, providing the best of both worlds.
- The resulting HDL module behaves like the real HDL code would, but it is hardly possible to reverse-engineer this component into HDL code. This elegantly solves the problems of intellectual property and confidentiality which arise when distributing a model of the component to customers.

2.3 Hardware vs. Software validation

Our C model provides all the advantages listed above. But its main purpose is to allow validating the real component. This is a multi-step process where:

1. The software model is validated by running the test cases and checking some functional signatures.
2. The hardware model (at the different levels: RT, gate, transistor) is validated by simulating the same test cases. During the run on the software model, the inputs are extracted at each cycle and injected for the simulation of the hardware model. The outputs of both models are then compared and any mismatch is signalled.
3. The physical chip is validated using the same methodology.

That process clearly shows the importance of the set of test cases: if it is too weak, many bugs can reside in the software model and in the final chip.

2.4 Test cases

Instead of developing test benches and a huge set of test patterns, we have developed system-level test cases: they are assembly language programs. System-level test cases are easier to write, debug and maintain than test patterns dedicated to unitary testing. They exercise the whole system, producing inter-module communications, and thus taking into account the behavior of all the system modules. The resulting patterns can not be unrealistic (whereas isolated test patterns could).

In many cases, knowing a given test-case has finished without hanging is not enough to qualify it as “passed”. In addition, functional signatures are inserted by the test case developers or even automatically by a dedicated tool. The signatures check values of registers, memory locations, etc. and compare them with some expected values.

We mainly deal with three sorts of test-cases: completely hand-written test-cases, completely generated ones, and a hybridation of both.

- Hand-written test cases are a good starting point when starting the project afresh. Moreover, one often needs to hand-write dedicated test-cases to reproduce “bugs”.
- A different approach is to completely generate assembly test-cases with only a small description of the low-level functionnalities one wishes to test (control-flow instructions, data manipulations...). Dedicated generators have been developed for this purpose.
- Finally, hybrid test cases are obtained by declining a hand-written test-case in many “flavours”. For example, a given hand-written test case doing memory accesses can easily be declined to access different memory areas with different configurations.

The rest of this paper shows how test cases can be generated to increase the state coverage of the system modules.

3 Automatic generation of tests

3.1 Choosing a coverage metrics

When we started our work on automatic test generation we already had an initial database of test cases. Each one targets a given functionality of our system. Our goal was then to answer the following two questions:

- Do we cover enough functionalities and possible combinations?
- If not, how can we efficiently, and if possible automatically, increase the coverage?

To answer these questions, one must first precisely define a coverage criterion[10][2]. Coverages based on statements, expressions, conditions, etc. are too weak when dealing with concurrent systems. We have chosen to focus on *state*

coverage which appears to be a good compromise. Indeed, each system module is a Finite State Machine (FSM) which state coverage can be accurately measured and it is likely that an uncovered state reflects an untested scenario. However, directly coding FSM is not suitable for modeling and maintaining complex systems. The Esterel language will help us solving this problem as described below.

3.2 The Esterel language

Esterel[3] is an imperative language dedicated to the modeling of control-dominated reactive systems. It provides powerful primitives for expressing concurrency, communication and preemption. It has a formally defined semantics in terms of FSMs which makes Esterel programs behave deterministically. The Esterel compiler can generate C code, and, as it focuses on control aspects, data-handling is imported from the C language.

From a programming point of view, Esterel is more convenient than C for modeling many of our system's modules: its dedicated primitives make the code simpler, shorter, more readable and maintainable than C code. For short, it is closer to a specification while still remaining executable.

From a validation point of view, the benefit is that Esterel programs are compiled into well defined formalisms such as netlists or explicit automata[3]. This opens the door to the use of formal verification and analysis tools such as SIS[19], VIS[23], SMV[16] and Xeve[4].

Applying our methodology to the whole system at once is unrealistic: first, because of its size, second it would have needed to remodel all the modules in Esterel, including the CPU. Thus, we have chosen to apply it at the module level first and to extend it at the system level.

Seven of our system's modules which are control-dominated have been remodeled in Esterel. These modules are compiled into a netlist which is optimized using SIS[19][20]. The netlist is then converted into a C++ code so that it can be integrated in the rest of the C-model.

We discuss in Section 6.1 the advantages of using a high-level language such as Esterel compared to other formalisms.

3.3 Formally evaluating state-coverage

Having embedded the Esterel modules, we are now able to run a simulation and, for each module (noted M), get the *reached state set* (noted *Red*).

---

3. A netlist is a circuit at the gate level. It is represented as a set of Boolean equations assigning values to output signals and registers in function of input signals and previous values of the registers. The netlist implements an FSM but its size is usually much smaller than the equivalent explicit automata (polynomial vs. exponential).

To get a coverage measure, we must now determine the number of theoretically reachable states. The formal verification tool Xeve[4] is used for this purpose: taking as input the same netlist we used to generate the C++ code, it builds the BDD[5] of the *reachable state set* (noted *Rable*). The BDD is then traversed and explicitly enumerated. Notice that we do not need to abstract the model in order to extract its control-flow as in [10]. This is because we prohibit the modeling of data in the Esterel models. Therefore, the states we consider are only control states.

The *state coverage* is then defined as the ratio between the number of reached states and the number of reachable states. The difference between these two sets is called the *missing state set* (noted *Miss*).

3.4 Generation of test sequences

We have shown how to measure the state coverage. We now detail our method to automatically generate test sequences in order to increase that coverage. It is based on well-known techniques of reachability analysis.

We call a *test sequence* for M a sequence of input vectors applicable to M. To increase M's state coverage, the first step is to find test sequences leading to the missing states:

1. We first exploit *Miss*. For each of its elements *S*, we add one new output O_S to the module's netlist so that O_S is active iff the FSM of M is in the state *S*.
2. We then use a formal model checker (we have used Xeve[4] and SMV[16]), asking for each O_S to prove the property "O_S is never active".

Since we know these properties are false, we obtain counter-examples which are test sequences leading to the missing states.

A dedicated algorithm has also been implemented into Xeve so that we only need to input the netlist and *Red* to Xeve[1]. It implicitly computes *Miss* and automatically generates the corresponding test sequences. The whole validation flow starting from the Esterel description of a module to the generation of test sequences reaching missing states has been fully automated.

3.5 Taking environment constraints into account

The computation of *Rable* as presented above does not take into account potential constraints imposed on the module by its environment (i.e. CPU and other modules). The existence of such constraints often makes many theoretically reachable states actually unreachable. Taking these constraints into account is mandatory to contain the state explosion and to get a realistic evaluation of the state coverage. The other reason for considering these constraints relates to test sequence generation: a generated test sequence that does not respect the module's constraints is unusable since it is impossible to produce in practice. The

set of constraints applied to a given module is determined by the informal documents describing the module and the ones connected to it. In some cases that set is not maximal. Fortunately that does not corrupt the validity of our method but may only reduce its efficiency. A more important point one must take care of is the correctness of the constraints:

- They must not be contradictory, otherwise "false-positive" results will be returned because $false \rightarrow anything$. Thus, we first prove the constraints are not always false.
- They must not be too restrictive: they must at least allow to reach all the states we know they are reachable in the context of the real system. To do so, we benefit from the fact that our methodology is based on simulation. Indeed, we first verify that *Red* is included in *Rable* even after *Rable* has been reduced by the constraints. The efficiency of that cross-checking depends on the quality of the initial test suite: in the worst case, the initial test suite and *Red* are empty and thus *Red* is always included in *Rable* whatever the constraints are.

It is therefore an important task to regularly manually review the constraints. In practice, we apply the following iterative process:

1. Get *Red* by simulation, compute *Rable* and *Miss*.
2. Write a constraint set *Constr*.
3. If *Constr* is always *false*, it includes contradictory conjunctions, goto 2.
4. If *Constr* does not hold for all elements of *Red*, it is too strong, goto 2.
5. Generate test sequences to reach all elements of *Miss* under the constraint *Constr*.
6. If some test sequences are unrealistic, *Constr* is too weak, goto 2.

Both Xeve and SMV allow taking constraints into account. One solution is to express the constraints as an observer written in Esterel (i.e. a module which monitors inputs/outputs and signals constraint violations). But it has some limitations because liveness constraints are not expressible. In SMV, constraints, including liveness ones, are expressed in LTL or CTL[4].

3.6 From test sequences to test cases

A *test case* is a program. It may be written in C, assembly or any other language, but in the end it is a binary code executed by the system. Unlike a test sequence, a test case is not relative to any module in particular but to the whole system. We say that a *test case realizes a test sequence* if the execution of the test case eventually generates the test sequence.

---

4. Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) [7] are logics with additional timing operators: *G* (always), *F* (eventually), *U* (until), *X* (next step), *A* (all execution paths), *E* (some execution paths).

As we have already mentioned, it is mandatory that our methodology is able to produce test cases and not only module-level test sequences. The task of generating test cases from test sequences may be automated in some cases but it strongly depends on the targeted module M.

The problem has already been addressed in [15] and [2] but those works are in the most convenient case where the module to verify is the decoder of a processor. Therefore, generating input sequences for the module is nearly equivalent to generating instructions: the gap between test sequences and test cases is narrow and easy to cross. We are in a more general situation because the modules being verified are not part of the CPU but are connected to it, directly or not. They are also connected to other modules which are sometimes not formally defined.

The connection protocol between the CPU and the module is a key factor for the automation of the test case generation. Indeed, suppose test sequences have been generated for a module M which is directly connected to the CPU and only to it. Suppose there is a defined set of instructions which can send control signals to M in a deterministic way. Then it is possible to automate the generation of the test cases. But if M is not directly connected to the CPU, if several different modules can drive signals to M, it is difficult to deterministically and automatically convert test sequences into test cases. These two situations are the best and worst cases. We consider the later below and the former in Section 4.

3.7 Manual generation of test cases

We address here the critical case where there is no automatic way to translate test sequences into test cases. Test cases must be manually written but the sequences are of great interest in order to target the effort. Indeed, the sequences accurately show the functionalities and their combinations which have not been exercised. It is the task of the verification engineers to first ensure the asked sequences are realistic, and, if they are, to write programs which realize them.

Let us assume the module under test is M, an automatically generated sequence *T* is targeted at a missing state *S*. If *T* is realistic, an engineer writes a program *P* which should reach *S*. We provide different tools in order to ensure that assumption. The first and obvious one is to run *P* and check that *S* is now included in the reached state set. Another way to do that is to dump the sequence *T'* of inputs at M's boundary while running *P*. It then remains to verify that *T* is a subsequence of *T'*.

The manual generation of test cases is only usable and valuable when the number of missing states is manageable. But we have experienced an hybrid approach which consists in manually writing a few test cases and then automatically deriving them so that they exercise similar functionalities on different modules and with different configurations. We can also introduce some random parameters. The number of resulting test cases may be large but the effort to

produce them is reduced. We show in Section 5.3 how we used that approach, and how it helped increasing the state coverage.

4 Generation of test cases by pipeline inversion

We have presented our methodology to automatically generate test sequences targeted at state coverage. These sequences may be manually converted into test cases but it is of course preferable to provide a fully automatic flow from missing states to test cases as we propose below.

4.1 Pipeline inversion

Let M be a module connected to the CPU. Starting with a test sequence for M, the problem is to go backward across the CPU back to a program. Doing so needs a model of the CPU. Then an automated tool can show the input sequences at the CPU level which generate the targeted input sequences at M's level. Input sequences at the CPU level are assembly instruction sequences which are converted into real test cases in a straightforward way. We call that mechanism "pipeline inversion" and we formalize it here below.

We note I_M (respectively O_M) a sequence of inputs (resp. outputs) at the boundary of M. M(I_M) = O_M means that M eventually outputs O_M when its inputs are I_M but the timing between I_M and O_M is not specified. Similarly, CPU(I_{CPU}) = O_{CPU} means that the CPU eventually outputs O_{CPU} when its inputs are I_{CPU} (i.e. the execution of the program I_{CPU} generates the outputs O_{CPU}). As we are in a synchronous framework driven by a single clock, there is a one-to-one mapping between each sequence couple.

The pipeline inversion of O_{CPU} consists in finding a sequence I_{CPU} so that CPU(I_{CPU}) = O_{CPU}. The methodology presented in Section 3 allows to generate I_M given O_M. A pipeline inversion of I_M would then convert the test sequence at M's level into a test case. A first alternative to do so is:

1. Generate I_M,
2. Make a pipeline inversion of I_M so that I_{CPU} is generated.

But this approach is too weak, it may produce false-negative results: it may fail to generate I_{CPU} because the pipeline inversion of I_M is impossible. Suppose we want to generate a test case to produce O_M and that all the following conditions are met:

$$\exists I_M, M(I_M) = O_M \quad (1)$$
$$\exists J_M, (M(J_M) = O_M) \wedge (J_M \neq I_M) \quad (2)$$
$$\forall I_{CPU}, CPU(I_{CPU}) \neq I_M \quad (3)$$
$$\exists J_{CPU}, CPUJ_{CPU} = J_M \quad (4)$$

The test sequence generation algorithm can show only one input sequence satisfying a given output, if any. Therefore, the algorithm may show only I_M at step 1 (by Equation 1) and thus fails to generate any test case at step 2 (because of Equation 3). But if step 1 would have shown J_M (by Equation 2), the pipeline inversion would have succeeded and generated J_{CPU} (by Equation 4).

To overcome this drawback, we have extended the technique in order to connect the model of the CPU to the model of M, and perform the generation of M input sequences under the constraint of the pipeline inversion. That means that instead of running the two-step process presented above, we run a single-step process: Find I_{CPU} so that $M(CPU(I_{CPU})) = O_M$. Doing so, the formal analysis will succeed in generating all the O_M which can be produced by M and the CPU together.

That methodology rises performance issues since it requires a computation of the state space for the M-CPU product FSM instead of computing the state spaces for M and for CPU alone. The difference is not negligible since it represents one to two orders of magnitude in computation time. To reduce that weakness, we propose to first use the two-steps approach which is quite fast. We use the single-step approach only for the states which could not have been reached by the two-steps approach.

4.2 Our pipeline model

We have applied pipeline inversion to internal memory controllers DARAM, SARAM and APIRAM. Therefore the model of the pipeline we have described in Esterel is an abstraction of the real one which accurately models the communication with the memory controllers but not more. That abstraction is mandatory in order to avoid a state explosion. Here is the list of the abstractions we did:

- The pipeline model describes the execution of 50 instructions whereas the real set includes more than 400 instructions.
- The effects of the instructions are only described for the data control signals. That means the model describes for each instruction at which pipeline stages the requests are sent to the memory controllers and at which stages the ready are awaited.
- The model is pipelined but has only 3 stages whereas the real one has 7. We modeled only the stages at which data control signals are sent. For example the instruction fetch and decode stages are not relevant. Instead we express some simple constraints which are true if the running program is mapped into a fast access memory, and if interrupts which could perturb the regular instruction fetch mechanism do not happen.

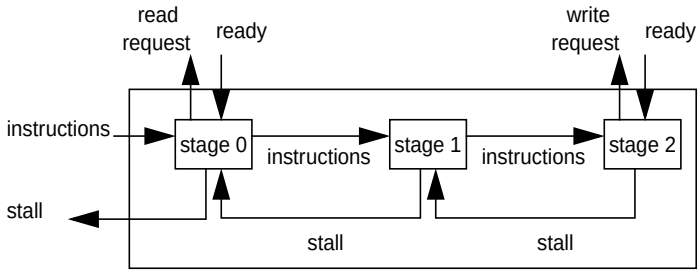

Figure 1 Overview of the pipeline architecture.

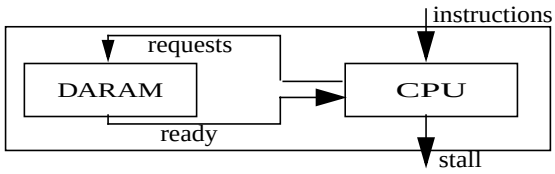

Figure 2 Connection of DARAM to the pipeline model.

- The model is not super-scalar whereas the real CPU can execute two instructions in parallel. When needed, the parallel instruction pairs are modeled as pseudo-instructions with equivalent behaviors.

Our Esterel model of the pipeline resembles the one proposed in[18]. It is built as follows (see also Figure 1). At the top-level, inputs are the instructions (one signal for each of the 50 possible instructions), and the acknowledgement signals (ready from memory controllers). The outputs are the request signals (to the memory controllers) and a general stall signal. The top-level consists of three sub-modules: stage 0, 1 and 2. Instructions signals are connected to stage 0 and are propagated to stage 1 and stage 2 when the pipeline is not stalled.

When an instruction arrives to stage 0, if it needs to perform read operations, requests are sent. On the next cycle, if all the awaited ready signals are active, the instruction may either “die” or it is propagated to stage 1 if it also needs to perform write operations. If the ready signals are not present, the pipeline is stalled. That means the instructions are not propagated and that no new instruction is accepted. Stage 2 is similar to stage 0 except that requests concern write operations and that instructions are not propagated any more.

4.3 Pipeline inversion in practice

We illustrate here the pipeline inversion on the validation of the DARAM module. DARAM is an internal memory controller which is connected to the CPU. Figure 2 shows how they are connected.

The DARAM and CPU models are compiled as netlists from the Esterel codes and optimized. DARAM’s netlist is instrumented to add outputs corre-

sponding to the missing states (see Section 3.4). The two netlists are then connected and converted into a code suitable for SMV.

Additional constraints are written to express CPU features not specified in the Esterel model. For each instruction, there is an input signal $instr_i$ which is active when the instruction has been fetched and decoded. Examples of constraints expressed in LTL are given below:

- No more than one instruction is decoded at each cycle:
  $G(\forall i, instr_i \rightarrow \forall j \neq i, \neg instr_j)$.
- No instruction is decoded when the CPU is stalled:
  $G(stall \rightarrow \forall i, \neg instr_i)$.
- Some instructions need a relocation of the stack pointer before being executed. In the same program, if one instruction relocates the stack pointer, then all following instructions must use the same location:
  $G(\forall i, instr_i \rightarrow XG(\forall j, instr_j \rightarrow (stack_i = stack_j)))$

All the constraints form an assertion called *rel* which is assumed. Finally we express temporal properties allowing to generate traces reaching the missing states. Those properties have the form $P_k{:}rel \rightarrow G\neg O_k$ where O_k is an output activated iff DARAM is in the missing state S_k.

We then ask SMV[5] to prove all P_k properties. If P_k is *true,* that means S_k is actually not reachable because of the pipeline behavior. The set of reachable states is thus reduced. When P_k is *false,* SMV produces a counter-example. It is a trace of the signals (including the $instr_i$ inputs) which lead to the emission of the output O_k, thus which reaches S_k.

We have developed an automatic tool in order to process SMV counter-examples and generate assembly language test cases. That tool selects only the signals named $instr_i$ in the counter-example. It translates them into assembly-language instructions which are included into a template program. Functional signatures are automatically inserted, the program is directly run on the software model and the new number of reachable states can be computed.

5 Practical results

5.1 Module description

We have applied the methodology presented above on different kinds of modules building a TMS320C55x™[22] system. The modules we focused on are shown on Figure 3: DARAM, SARAM and APIRAM (internal memory controllers), SAM (bus controller which handles communications with external peripherals), MMIP and MMID (external memory interfaces), DMA (direct memory access controller). They were modeled in Esterel. The development time

5. We used SMV for that purpose. But any other model checker is usable.

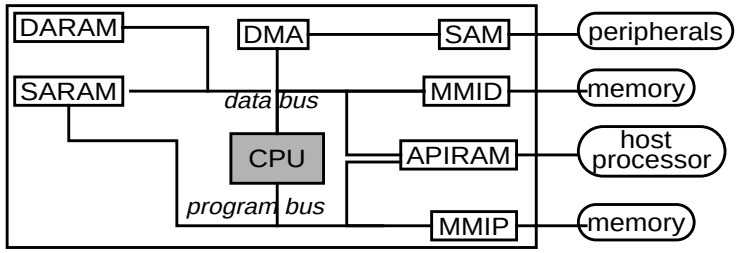

Figure 3 The DSP system.

was significantly shorter than the one required to model the same modules in C or in VHDL languages.

Module	DARAM	SARAM	APIRAM	SAM	MMIP	MMID	DMA
# inputs/outputs	18 / 101	32 / 164	35 / 159	33 / 122	22 / 59	49 / 157	24 / 46
#regs init/optim	63 / 30	91 / 27	91 / 26	86 / 80	78 / 30	95 / 67	98 / 73
time optim	41 sec	302 sec	83 sec	82 sec	286 sec	43 sec	428 sec
#reachable init	556	827	422	400	46,513	>1.3M	309
#reachable final	481	368	237	400	46,513	>1.3M	278
time reachable	2.1 sec	3.8 sec	2.5 sec	7.09 sec	40.7 sec	24 hours	23.2 sec
time enum	0.1 sec	0.1 sec	0.1 sec	0.7 sec	3.6 sec	-	0.7 sec

Table 1 System modules on which we applied the generation of test cases.

Table 1 shows the characteristics of the modules. The first row indicates the number of primary inputs and outputs, the second row is the number of registers before and after optimization using SIS[19]. We were faced to a state explosion for MMID: we could not use SIS because it requires the computation of the reachable state space. Instead, we reduced the number of registers using an ad-hoc approach which is less efficient. “time optim” is the CPU time[6] required to run the sequential optimization. “#reachable init” (resp. “#reachable final”) gives the number of reachable states obtained without (resp. with) environment constraints. That last number will be our basis for state coverage. The two last rows report the CPU time needed to implicitly build the reachable state sets with Xeve[4] and to explicitly enumerate them. One can see that, except for MMID, the computation and the enumeration of the reachable state spaces is fast. This is mainly due to the sequential optimization.

5.2 Generation of the initial reached states

We have compiled all the modules as C code and integrated them into the software model. We have then run the initial test suite which serves the designers

6. All CPU times are measured on a 360Mhz UltraSparc-II with 1Gb of physical memory and 2Gb of virtual memory.

for non-regression checking[7]. That test suite has been developed along the project during more than two years by several verification engineers. It consists of 48,268 test cases, running for more than 385 Millions of cycles. The simulation time is 120 hours. The initial state coverages obtained after the simulation of that test suite are given in Table 2. The rows show the number of states

Module	DARAM	SARAM	APIRAM	SAM	MMIP	MMID	DMA
#reached init	166	106	35	217	650	2289	218
coverage init	35%	29%	15%	54%	1%	?	78%
#missing	315	262	202	183	45863	?	60

Table 2 Resulting state coverage after simulation of initial test suite.

reached by the test suite, the state coverage it provides and the number of missing states. Notice that the traditional code coverage metrics[10] (statement, condition,...) obtained after the simulation of the same test suite on the VHDL model are in the 90-100% range, even for the modules where state coverage is very low.

5.3 Generation of new test cases

We have applied our test sequence and test case generation methodology. Table 3 shows the results. The two first rows reports the number of test sequences

Module	DARAM	SARAM	APIRAM	SAM	MMIP	MMID	DMA
#test sequences	315	261	201	80	-	-	60
time sequences	5,246 sec	18,836 sec	611 sec	140 sec	-	-	1,139 sec
#test cases final	315	248	201	32	13,752	10,560	10
#reached final	481	254	162	321	2,328	6,382	234
coverage final	100%	69%	68%	80%	5%	?	84%
cov final/cov init	2.9	2.4	4.5	1.48	5	2.8	1.1

Table 3 Generated test sequences, test cases and final state coverage.

which have been automatically generated and the CPU time required for that task. “#test cases final” is the number of system-level test cases which have been generated from the test sequences, automatically or manually. The three last rows show the coverage obtained after having run the new test cases and how it has increased compared to the coverage provided by the initial test suite.

---

7. We have also run real-world applications, representing several Billions of cycles. Those applications are Digital Signal Processing algorithms which exercise the CPU but do not increase the coverage of the system modules.

5.4 Remarks

For DARAM, SARAM and APIRAM, the generation of the test sequences is slow because it uses the pipeline inversion technique as described in Section 4.3. But the advantage is that the test cases are almost directly derived. This is the case for DARAM for which we could directly get a 100% coverage. The new tests are very efficient if we consider the number of new states reached over the simulation time: they allow to reach 315 states in 37 seconds whereas the initial test suite covers only 166 states in 120 hours. If we run only the 127 test cases of the suite which where developed to validate DARAM specifically, only 132 states are reached in 325 seconds.

Our pipeline models only the CPU data requests but SARAM and APIRAM are also connected to the instruction fetch unit, the DMA and the host processor. Therefore, the pipeline inversion generated incomplete test cases. We then automatically derived them so that they are executed with different combinations of requests from the instruction fetch unit, the DMA and the host processor. It allowed to reach a coverage a little bit less than 70%. It is not clear whether the remaining missing states are really reachable. We think the environment constraints we modeled are too tolerant (but not to restrictive).

For the SAM module, we did not use pipeline inversion because most of the requests are not originated by the CPU. The generation of test sequences is thus fast. From the 80 proposed sequences, we manually wrote 32 test cases we considered as being realistic and the most important ones. That allowed to significantly increase the coverage.

MMIP and MMID have too many states to try to generate test sequences reaching all the missing states. For MMIP, we generated just a few sequences, manually converted them into test cases and automatically derived them into several versions: different memory latencies, protocol configurations,... That approach is not rigorous enough to provide an acceptable coverage. However, we could multiple the coverage by a factor of 5. We did not investigate that methodology more deeply but we believe it is promising, especially if random parameters are introduced. We did not generate any test sequence for MMID but we reused the ones of DARAM that we derived into several test cases so that MMID is addressed instead of DARAM. The coverage of MMID could thus almost be tripled.

For DMA, we generated the missing test sequences and manually converted some into test cases. The other ones seemed to be unrealistic.

5.5 Bugs found

Our methodology and tools arrived late in the project life so that they have been applied after many bugs were already found and fixed. However, the generated test cases have shown unknown bugs in various modules, in the software model as well as in the hardware model.

The most interesting conclusion of our experiments is that very often, bugs were not located in the modules we were testing. For example, one of the test cases generated for MMID highlit a possible dead-lock in the DMA. We also found bugs in the CPU while running test cases of DARAM and APIRAM. This remark is important: it shows that a system-level validation approach has several advantages, one of them being that it can show bugs in parts of the system which were not directly being validated.

6 Conclusions

We have shown in this paper a methodology to automatically generate new test sequences targeted at state coverage. They may be automatically converted into system-level test cases using a technique called “pipeline inversion”. This validation flow has been integrated into a real world design project and allowed to significantly increase the coverage of system modules.

6.1 Comparison with related works

Early works (e.g. [6] and [14]) have investigated the use of formal methods to automatically generate tests. But they took place in an ATPG (Automatic Test Pattern Generation) context, thus at a low-level where the tests are not intended to verify the functionalities of the systems but to detect possible fabrication problems.

Our work is more related to [12][15] and [2]: methodologies to automatically generate tests targeted at state and transition coverages are proposed. They show applications for the verification of parts of real processors like an instruction decoder, allowing to easily turn test sequences into test cases. We are in a more general situation where the test sequences generated do not apply to a CPU, thus the need of the pipeline inversion technique to convert the test sequences into system-level test cases. This has been discussed in Section 3.6.

We also think a major advantage of our approach is that the models are written in the Esterel language. The models of [2] and [17] are explicit FSMs handwritten in the Murφ[8] and SMV[16] languages and are not reusable for any other purposes like simulation. With Esterel, the FSMs are not explicitly specified but automatically synthesized from a high-level form. The Esterel models are also executable allowing to build software simulators we release to our customers. The extraction of the control parts of the module is automatic: we do not need to work on data abstractions as in [12] or [17].

6.2 Perspectives

Our methodology suffers from some weakness we need to address. The main one is the fact that the state explosion stops the test case generation flow. Indirect approaches are still efficient (see results for MMIP and MMID) but not sat-

isfactory. We have already used reduction techniques but they only help keeping the BDD representation of the states compact. The sequential optimization does not help for modules having a too large state space (e.g. MMID). We are currently working on modular optimization which may allow to progress.

Our current approach needs at least one step where a state set is explicit (*Miss* or *Red*). This is a limitation which could be overridden: the idea is to build *Red* as a BDD during the simulation. Therefore the operation $Miss = Rable - Red$ could be done on BDDs. The generation of test sequences or test cases using the pipeline inversion would then need to use heuristics to produce tests so that each one pass through the maximum number of missing states. This would allow to generate a manageable number of test cases even if the number of missing states is large.

Other coverages may be considered[10][2][11]. The next step is to apply the methodology on transition coverage. We have conducted a first experiment on DARAM. Fully automatically generated test cases could raise the transition coverage from 10% up to 90%.

We are currently applying our methodology on a new project, at a larger scale, earlier in the project life, with a very limited initial test suite. Our future work also involves the generation of test cases for data-dominated modules.

References

[1] L. Arditi, A. Bouali, *et al*. "Using Esterel and Formal Methods to Increase the Confidence in the Functional Validation of a Commercial DSP". In *Workshop on Formal Methods for Industrial Critical Systems*, Trento, Italy, 1999.

[2] M. Benjamin, D. Geist, *et al*. "A Study in Coverage-Driven Test Generation". In *36th Design Automation Conference*, 1999.

[3] G. Berry, G. Gonthier. "The Esterel synchronous programming language: Design, semantics, implementation". In *Science of Computer Programming*, 19(2), 1992.

[4] A. Bouali. "XEVE, an Esterel Verification Environment". In *Computer Aided Verification*, LNCS 1427, 1998.

[5] R. Bryant. "Graph-based algorithms for boolean manipulation". In *IEEE Transactions on Computers*, C-35(8), 1986.

[6] H, Cho, G. Hachtel, F. Somenzi. "Fast Sequential ATPG Based on Implicit State Enumeration". In International Test Conference, 1991.

[7] E.M. Clarke, E.A. Emerson, A.P. Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". In *ACM Transactions on Programming Languages and Systems*, 8(2), 1986.

[8] D.L. Dill. "The Murφ verification system". In *Computer Aided Verification*, 1996

[9] D.L Dill. "What's Between Simulation and Formal Verification?". In *35th Design Automation Conference*, 1998.

[10] D. Drake, P. Cohen. “HDL Verification Coverage”. In *Integrated Systems Design Magazine*, June 1998.

[11] F. Fallah, P. Ashar, S. Devadas. “Simulation Vector Generation from HDL Descriptions for Observability-Enhanced Statement Coverage”. In *36th Design Automation Conference*, 1999.

[12] R.C. Ho, C. Han Yang, M.A. Horowitz, D.L. Dill. “Architecture Validation for Processors”. In *International Symposium of Computer Architecture*, 1995.

[13] *VHDL language reference manual*. IEEE standard 1076-1993. IEEE Press, 1994.

[14] T. Kropf, H.-J. Wunderlich. “A Common Approach to Test Generation and Hardware Verification Based on Temporal Logic”. In International Test Conference, 1991.

[15] D. Lewin, D. Lorenz, S. Ur. “A Methodology for Processor Implementation Verification”. In *Formal Methods in Computer Aided Design*, LNCS 1166,1996.

[16] K. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic, 1993.

[17] D. Moundanos, J. Abraham, Y. Hoskote.”Abstraction Techniques for Validation Coverage Analysis and Test Generation”. In *IEEE Transactions on Computers*, 47(1), 1998.

[18] S. Ramesh, P. Bhaduri. “Validation of Pipelined Processor Designs using Esterel Tools: A Case Study”. In *Computer Aided Verification*, LNCS 1633, 1999.

[19] E.M. Sentovitch, K.J. Singh, *et al*. “Sequential Circuit Design Using Synthesis and Optimization”. In *International Conference on Computer Design*, 1992.

[20] E.M. Sentovitch, H. Toma, G. Berry. “Latch Optimization in Circuits Generated from High-Level Descriptions”. In *International Conference on Computer-Aided Design*, 1996.

[21] *TMS320C54x DSP CPU Reference Set*. Texas Instruments, Literature Number SPRU131F, Apr. 1999.

[22] *TMS320C55x DSP CPU Reference Guide*. Texas Instruments, Literature Number SPRU371A, Feb. 2000. See also `http://www.ti.com/sc/c5510`

[23] The VIS Group. “VIS: A system for Verification and Synthesis”, In *Computer Aided Verification*, LNCS 1102, 1996.