Validation of functional processor descriptions by test generation

Fabrice Baray - ST-Microelectronics, Central R&D, France
Philippe Codognet - Paris 6 University, LIP6, France
Daniel Diaz - Paris 1 University, France
Henri Michel - ST-Microlectronics, Central R&D, France

Abstract. Microprocessor design deals with many types of specifications : from functional models (SystemC or proprietary languages) to hardware description languages such as VHDL or Verilog. Functional descriptions are key to the development of new processors or System On Chips at STMicroelectronics. In order to reduce validation effort and meet aggressive time to market requirements, it is essential to discover all potential defects as early as possible in the functional model, which is at the center of the design methodology.
In this paper we address the problem of automatic generation of high quality testsuites for microprocessor functional models. We introduce an overview of a software tool based on constraint solving techniques which analyzes the control flow of the initial description in order to generate tests for each path. We achieve better path coverage than typical hand-written assembler testsuites.
STM7, a STMicroelectronics micro-controller, has been used to check this novel test generation method. Composed of about 700 various instructions and 3000 lines of SystemC-like code, it is complex enough to assess the performances of our solution.

Keywords : code-based test generation, verification of functional processor descriptions, constraint solving techniques, finite domain constraints

Introduction

Current design methodology for microprocessors or complete *System On Chip* depends on the availability of a *simulation model*. Such a model is used for the following purposes :

1. Early development of a compiler or more generally a full *software tool chain*. The development of software can thus proceed in parallel with the development of hardware.
2. Early development of embedded software.
3. Using codesign, it is possible with such simulation models to benchmark and prototype complete platforms made of blocks which may not exist on silicon yet.
4. Hardware Verification : the simulation model serves as a *golden reference*, for a microprocessor or platform.

Being so crucial, the quality of a simulation model may affect drastically the timing of a design project.

Classification of simulation models

Simulation models are software programs which make the best effort to represent correctly the behavior of a hardware block. Ideally they should perform the right computations and produce accurate results, and optionally they should produce the right results at the right instants of time. Accordingly, we distinguish between :

– bit-true models : the computed results may be compared bit per bit with what produces the hardware
– cycle-accurate models : the results (changes of output signals) are produced at exactly the same pace as the hardware.

Note that a simulation model which is not bit true may still be of some use to conduct benchmarking and performance estimation. In this paper we will concentrate on bit-accurate software simulation models and refer to them as *functional models* or equivalently (in the context of microprocessors) to ISS *Instruction Set Simulators*.

They represent the view of the machine that the programmer has when writing software. A software developer does not have to know details dealing with pipeline, branch prediction, existence of multi execution units and other micro architectural aspects in order to write functionally correct software, even in assembly language. This view of the machine is in terms of architectural state (contents of registers, memories), the execution of an instruction causes a change from a stable architectural state to a new architectural state. In this view there is no notion of parallel executions or instructions whose executions may involve several cycles.

Use of a simulation model for Hardware Verification

Though formal verification methods are standard for microprocessors verification, the complete verification flow for a complex microprocessor still includes pseudo-random generation of very large testsuites (see for example [1] for the full picture of a complete verification flow).

The principle of verification using pseudo-random generation is to make sure that small sequences of assembler codes exhibit the same behavior on the HDL model (typically VHDL or Verilog) and on the functional simulator.

In this method small sequences of assembler codes are produced, with random (but directed) generators. A typical tool for this task is the Genesys tool ([2], [3]), another typical alternative is the Specman tool from Verisity Inc. The aim of these sequences is to test the VHDL model, and thus exercise all complex micro architectural (pipeline, branch prediction . . .) mentioned above. Obviously, this method just includes a comparison between the HDL model and the reference simulator. If both the HDL model and the ISS exhibit the same bug, nothing more can be said on the matter. What is important though is to be sure that the test vectors, completely encompass all the possible behaviors of the reference simulator.

Testsuites for ISS model validation

In order to reduce validation effort and meet aggressive time to market requirements, it is essential to discover all potential defects as early as possible in the functional model, which is at the center of the design methodology. This requirement can be met with testsuites of high quality for the functional models, quality depending on 2 factors :

1. completeness : a testsuite should capture all the possible behaviors of a model, in order to be able to observe differences of behavior between models. This qualitative completeness requirement being measured by a *coverage criterion*.

2. small or minimal : a testsuite should be small enough to be complete without being too huge and thus slow to run.

The automatic generated ISS testsuite is intended for the following uses :

- manual inspection of the results for compliance with the architecture manual
- comparison with an early version of the VHDL model (without all the complex mechanisms)
- comparison with an already existing ISS

The remainder of this article consists of three main parts. The first one describes our general method of validation for our functional model of processor. The second one addresses technical points on test vectors generation with some experience results. The third one shows experiment results on a case study.

Method of validation

Functional description language

Modeling a processor in a functional way requires an adequate language able to describe the desired behaviors. Many instruction set simulator (ISS) are written in languages which are either a variant of a classical language like C++ in the case of SystemC, or had-hoc languages (for example *flexsim/idl* an ST Microelectronics proprietary language). Both alternatives share the same semantic objectives :

- types with specific sizes : integer on n bits signed or not, with arithmetic operations dealing with casting rules ;
- bit manipulation : bit extraction, range of bit extraction, concatenation of two variables ;
- delayed assignment : specific hardware style constraint used to describe that an assignment is deferred to the end of a functional cycle ;
- assignment of undefined values in order to handle non deterministic behaviors ;
- syntax specific decoding statement, to ease development of complex instruction set processors.

An ISS description represents one functional cycle of the processor in terms of memory and register contents modifications. We consider that the memory and register values represent the whole architectural state (there is no hidden internal state). As a consequence, the input values of a test vector are only composed of a value for each element of the state. An execution of a test initializes values of registers and values of some memory cells before running the cycle. Furthermode, in our context, it is enough to test all possible execution paths of one cycle to validate completly the simulator. Generating tests for more than one cycle can be useful in the case of an error which is not visible while executing only one cycle, but this cannot be the case if the processor state is entirely visible.

The following example shows typical ISS coding style and constructs previously mentioned :

```
uint<8> mem[16777216];
uint<8> X;
uint<8> Y;
uint<24> PC;

uint<8> arith_inc_dec(uint<4> opcode1,
                      uint<8> reg) {
  switch (opcode1) {
    case 0b0011 : { // DEC X
      return(reg-1);
      break;
    }
    case 0b0101 : { // INC X
      return(reg+1);
      break;
    }
  }
}

void cycle() {
  uint<8> fetch;
  uint<4> opcode0;

  fetch=mem[PC];    // first fetch
  PC=(uint<24>)(PC + 1);
  opcode0=extract_range(fetch,7,4);
  switch (opcode0) {
    case 0b1111 : { // arithmetic
      uint<4> opcode1;
      opcode1=extract_range(fetch,3,0);
      if ((opcode1==0b0011) ||
          (opcode1==0b0101))
        X=arith_inc_dec(opcode1,X);
      break;
    }
    case 0b0110 : { // X=Y+op1
      uint<8> op1;

      op1=mem[PC];     // second fetch
      PC=(uint<24>)(PC+1);

      X=Y+op1;
      break;
    }
  }
}
```

Model based approach

In this paper we consider a model based approach for the test generation method. The entry point of the tool is directly the functional model of the processor rather than other solutions starting from less precisely instruction set descriptions. This approach allows a more sophisticated solution leading to more efficient test vectors. The figure 1 shows the architecture of our tool called STTVC (ST Test Vectors Constructor).

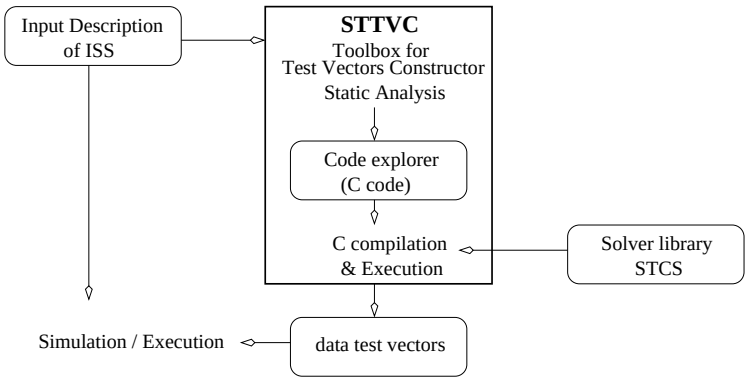

Figure 1. STTVC tool architecture

From the ISS description (functional description) we produce a C program (a *code explorer*) which is then compiled with an ISS independent part : the STCS solver library. This translation process is just a static analysis of initial model. *Code explorer* computes the test vectors which reveals the types of errors we want to detect in the ISS functional model. A test vector is composed of input values for all input elements in the model, and expected output values.

The efficiency of a test generation method is measured in terms of "*coverage*" [4]. A *code coverage* (or line coverage) percentage represents the size of the subset of all statements activated by the test vectors. Activating all paths in the code is a more restrictive coverage criterion than code coverage. It is even more demanding than the notion of *condition coverage* : considering the whole testsuite all conditions ("*if*" in particular) have been exercised as *true* or *false*.

Figure 2 illustrates the differences between coverage criteria on a little example with only two input values used in conditions c1 and c2 and four statements a,b,c and d. 100% code coverage can be reached with only one test vector which exercises the path (1). Unfortunately in this test vector all conditions are true. 100% condition coverage can be achieved by adding to the testsuite a second test vector for which both conditions are false (path (2)). For 100% path coverage all four paths need to be exercised.

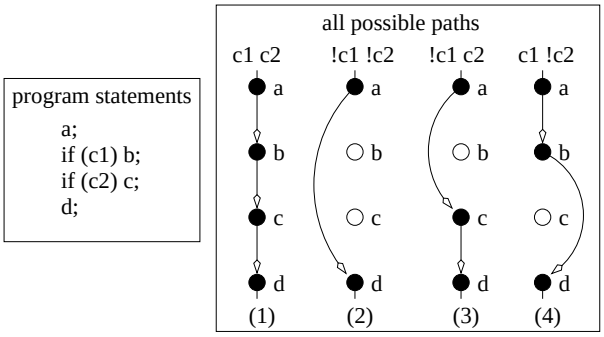

Figure 2. Code or path coverage

In order to find as many errors as possible, path coverage is more powerful than code coverage. In the context of microprocessor validation, we expected that doing a full path coverage for the functional description of one cycle would give better results, in terms of error detection, than the classical assembler testsuite simulation. Consequently, our translation in STTVC decomposes the description into paths and the produced *code explorer* program generates vectors independently for each path.

Constraint solving and tests generation

In this section, we consider the problem of finding test vectors for only one path of the functional description control flow. A path is composed of ordered statements of two types : assignments of variables and conditions on variables. Conditions on variables come from the test statements of the initial description and constraint directly the input values for testing the path. This problem can be view as a classical *constraint satisfaction problem* (CSP for short) [5,6,7].

A constraint is a logical relation among several unknowns (or variables), each one taking a value in a given domain of possible values. A constraint thus restricts the possible values that variables can take. A *constraint satisfaction problem* (CSP for short) is defined as a triple $< \mathcal{X}, \mathcal{D}, \mathcal{C} >$, where $\mathcal{X}$ is a set of variables, $\mathcal{D}$ is a set of domains, that is, a finite set of possible

values (one for each variable), and $\mathcal{C}$ a set of constraints restricting the values that the variables can simultaneously take. As in classical CSPs, we consider in this study finite domains for the variables (integers or naturals) and a solver based on arc-consistency techniques, in the spirit of [8]. Such a solver keeps internal representation of variable domains in order to handle all kind of constraints. For instance, the domain of an unsigned variable X, constrained by X *less than* a constant C can be exactly defined by an interval representation $[0 \dots C]$. However for much more constrained variables, the interval representation is only an approximation of their effective domain.

Solving constraints consists in first reducing the variable domains through local propagation techniques and then finding values for each constrained variable in a *labeling* phase, that is, iteratively *grounding* variables (fixing values for variables) and propagating its effect on other variable domains (by applying again the same arc-consistency techniques). The labeling phase can be improved by using heuristics for the order in which variables are considered and for the order in which values are considered in the variable domains.

The first idea to solve our test generation problem was to use the existing solver GNUProlog [9], like in [10,11]. But considering our input language, some points are difficult to solve with this solver :

- defining typed variables with fixed bit size can be solved by simple domain reduction. But the casting operation between them can result in some loss of information on domain values. Another problem of the GNU Prolog solver is that it handles only natural but not integer (signed) variables ;
- constraints on bit manipulation have to be handled specifically : bits extraction on variables such as $X = Y[5 : 3]$ (equating X to the slice of bits 3 to 5 of Y), and concatenation of variables such as $X = (Y : Z)$ (equating X to the concatenation of Y and Z). [10] proposes to consider variables as bit vectors and constraints on variable bits as several independent constraints on individual bits. This solution however cannot be efficiently combined with arithmetic constraints on the same variables and thus prevent some possible domain reduction ;
- array manipulations in the microprocessor description can simply be considered by adding to the solver a list of variables but the number of created variables would become too large if the array represents memory.

Due to these reasons, and to improve efficiency in the constraint solving process, one of the main contribution of this work is the development of a dedicated constraint solver, named STCS. STCS benefits from the experience of GNUProlog. New specific constraints based on our input language characteristics are introduced, for instance (X, Y, Z are variables, C is a constant) *logical and* (X_Eq_Y_and_Z), *logical shift* (X_Eq_Y_slr_C), *bit concatenation* (X_Eq_Y_concat_Z) and *bit extraction* (X_Eq_Y_extract_C). The solver implements the specific constraints with two internal representations of the variable domains : one for the arithmetic constraints and one to handle efficiently the *bit* based constraints.

On our little example, five paths are analyzed and the results give values for the registers and some memory cells. Notes that for the first three paths, the Y register is not used, and the input

and output values are free. Each path corresponds to one instruction, manually annotated into the following results :

```
// Arithmetic (false opcode1)
inputs  : [ Y={0 .. 255}  PC=0  mem[0]=240  X={0 .. 255}  ]
outputs : [ Y={0 .. 255}  PC=1  X={0 .. 255}   ]

// DEC X (with overflow)
inputs  : [ Y={0 .. 255}  PC=0  mem[0]=243  X=0 ]
outputs : [ Y={0 .. 255}  PC=1  X=255   ]

// INC X
inputs  : [ Y={0 .. 255}  PC=0  mem[0]=245  X=0 ]
outputs : [ Y={0 .. 255}  PC=1  X=1   ]

// X=Y+op1
inputs  : [ Y=0  PC=0  mem[1]=0  mem[0]=96  X={0 .. 255} ]
outputs : [ Y=0  PC=2  X=0   ]

// Implicit default for switch (opcode0)
inputs  : [ Y={0 .. 255}  PC=0  mem[0]=0  X={0 .. 255} ]
outputs : [ Y={0 .. 255}  PC=1  X={0 .. 255}   ]
```

For instance, the *dec* path induces a constraints store of length 12, with many temporary variables defined in order to calculate the expressions of the input description. These variables are named by T0, T1, ... and prefixed by the function name in which they are defined.

constraint name	operand 1	operand 2	operand 3
X_Lt_C	PC	16777216	
X_Eq_T_I	cycle::T4	[]mem	PC
X_Eq_Y_p_C	cycle::T5	PC	1
X_Eq_cast_Y	cycle::T6	cycle::T5	
X_Eq_Y_extract_C	cycle::T7	cycle::T4	240
X_Eq_C	cycle::T7	15	
X_Eq_Y_extract_C	cycle::T8	cycle::T4	15
X_Eq_C	cycle::T8	3	
X_Neq_C	cycle::T8	5	
X_Eq_C	cycle::T8	3	
X_Eq_Y_p_C	arith_inc_dec::T0	X	-1
X_Eq_cast_Y	arith_inc_dec::T1	arith_inc_dec::T0	

Case study : STM7 micro-controller

In order to evaluate the performances of our test generation method and the STTVC tool introduced in the previous sections, we used a case study on a commercial ST microprocessor. The STM7 is a micro-controller with 6 internal registers, 23 main addressing modes and 45 instructions, its high level description includes about 3000 lines of code. Combining the addressing modes with the instruction set gives about 700 various instructions.

The flow used to conduct the case study is given in figure 3.

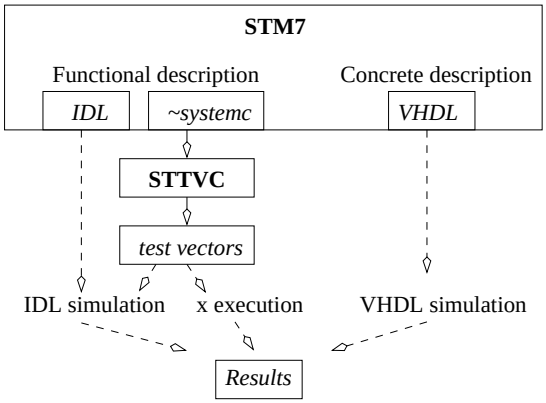

Figure 3. STM7 test flow

Starting from a functional description, test vectors are generated with the STTVC tool described earlier and are simulated with the three platforms shown in Figure 3. The VHDL simulator is not able to simulate only a single cycle of the description and to directly input and output values for registers. Then, for that platform, the test vectors must be encapsulated into an assembler program which initializes the registers, let the processor reach a stable state by appending NOP instructions to the test and saves the context after each test cycle. For each test vectors we compare execution on the two simulators (that is, we compare the outputs they produce). In addition, efficiency results in terms of percentage of coverage and simulation time can be computed.

The generated testsuite is composed of 1922 test vectors and can be executed for acquiring error detection and coverage percentages in order to compare with coverages obtained with the simulation of the classical manually written assembler testsuite. In terms of number of functional cycles, the STM7 assembler testsuite is ten times bigger than our test vectors simulation, with 18941 cycles, and takes more time to run compared with only 1922 cycles of our solution. This is a very encouraging result. This important difference in simulation time has to be taken into account for all the coverage criteria analyzed below.

Table 1 compares the assembler testsuite and the STTVC test vectors simulation in terms of statement coverage. The covered statements are decomposed into two classes : NBB for normal basic blocks (649 total) and IBB for implicit basic blocks (48 total) which correspond to the *else* part of an *if* statement without *else* or to the *default* part of a *switch* statement without *default*.

test type	covered	percentage
STTVC NBB	644	99.2%
ASM NBB	642	98.9%
STTVC IBB	35	72.9%
ASM IBB	0	0%

Table 1. Statement coverage results

With this criterion, the results are quite similar : for the assembler testsuite the IBB blocks are not covered because it is based on classical micro-controller utilization. We can suppose that the 22 percent of IBB not covered by STTVC tests generation correspond to effectively “non reachable” IBB.

For results on the path coverage criterion, the assembler testsuite is simulated on the model with an annotation in order to dump the path description. By construction, the STTVC tests must cover all paths of the code description. First of all, we can observe that the ASM simulation doesn’t give any path not generated by STTVC. Without any external tool to verify our implementation (to our knowledge there is no path coverage measurement tool), this gives confidence in STTVC correctness. Furthermore, as could be expected, the assembler testsuite covers only 72 percents of the test paths generated by STTVC.

Conclusion and perspectives

Taking into account that there is a need of ISS validation, we suggest a novel approach to test generation for functional processor models, based on the idea of solving constraints stores, each one representing a path in the control flow of the processor cycle. For this objective we have developed a new constraint solver specialy tuned for our test generation application and a translator between our input language and a C program which generates the test vectors.

Implementation results are very recent and does not illustrate clearly the tools efficiency in terms of generation time. However the case study on the STM7 microcontroller shows coverage measures on the ISS simulation with the test vectors generated with a first alpha version of the tool. With only 1922 test vectors, thus only 1922 simulation cycles, the statement coverage reaches the same percentage than a classical assembler testsuite, and the better path coverage justifies the proposed approach.

Figure 4 shows a screen-shot of the graphical interface for visualizing the path coverage of the assembler testsuite with a coloring of the graph representing all the paths detected by the STTVC tool.

More experiments are needed to assess the capabilities of this test generation methodology, but we are already working in four directions to improve its correctness and efficiency :

- in order to improve time generation, we are working on STCS solver for domain propagations and constraint store analysis ;
- for each test vector we plan to implement several mutations [12] in order to increase the efficiency of the errors detection ;
- in order to improve false path detection, symbolic manipulations of constraints are needed ;
- comparing coverage of the VHDL model with functional coverage of the corresponding ISS, using the same test vectors, remains to be done.

Acknowledgment. we thank Emanuele Baggetta for the development of a graphical interface to STTVC.

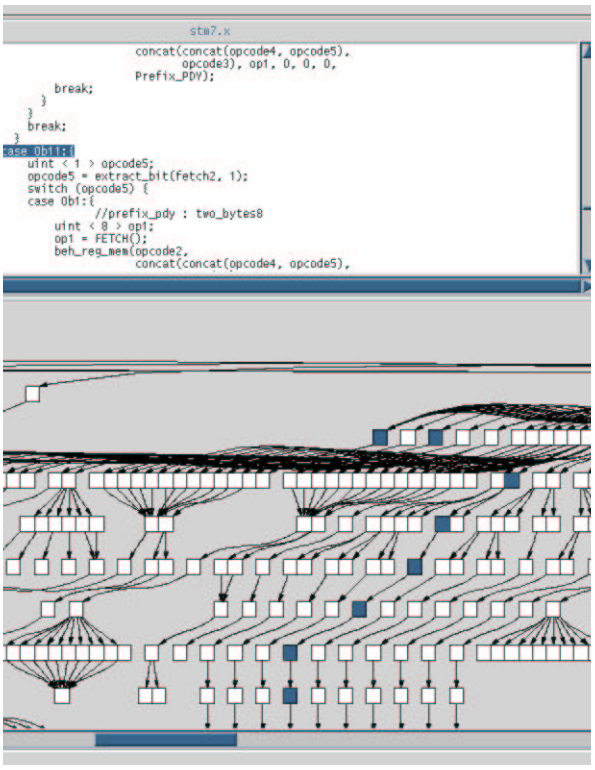

Figure 4. Tool screen-shot for path coverage

References

1. F. Casaubieilh, A. McIsaac, M. Benjamin, M. Bartley, F. Pogodalla, F. Rocheteau, M. Belhadj, J. Eggleton, G. Mas, G. Barrett, and C. Berthet, "Functional verification methodology of chameleon processor," in *Design Automation Conference, DAC'96*, (Las Vegas, Nevada, USA), pp. 421–426, 1996.
2. A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, C. Metzger, M. Molcho, and G. Shurek, "Test program generation for functional verification of powerpc processors in IBM," in *Design Automation Conference*, pp. 279–285, 1995.
3. S. Rubin, M. Levinger, R. Pratt, and william Moore, "Fast construction of test-program generators for digital signal processors," in *ICASPP'99 Conference Proceedings*, 1999.
4. H. Zhu, P. A. Hall, and J. H. May, "Software Unit Test Coverage and Adequacy," *ACM Computing Surveys*, vol. 29, pp. 366–427, December 1997.
5. J. Jaffar and M. J. Maher, "Constraint logic programming: A survey," *The Journal of Logic Programming*, vol. 19 & 20, pp. 503–582, 1994.
6. P. v. H. e. a. V. Saraswat, "Constraint programming," *ACM Computing Surveys*, vol. 28, Dec 1996.
7. R. Barták, "Constraint Programming: In Pursuit of the Holy Grail," in *Proceedings of WDS99 (invited lecture)*, (Prague), June 1999.
8. P. Codognet and D. Diaz, "Compiling Constraints in clp(FD)," *Journal of Logic Programming*, vol. 27, June 1996.
9. P. Codognet and D. Diaz, "The gnu prolog system and its implementation," *Journal of Functional and Logic Programming*, vol. 6, Oct 2001.
10. F. Ferrandi, M. Rendine, and D. Sciuto, "Functional Verification for SystemC Descriptions Using Constraint Solving," in *Design Automation and Test in Europe (DATE'02)* (C. D. Kloos and J. da Franca, eds.), (Paris, France), pp. 744–751, 4-8 March 2002.
11. C. Paoli, M. Nivet, and J. Santucci, "Use of constraint solving in order to generate test vectors for behavioral validation," in *High-Level Design Validation and Test Workshop. Proceedings. IEEE International*, pp. 15 – 20, 2000.
12. R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, pp. 34–41, April 1978.