Code-Based Test Generation for Validation of Functional Processor Descriptions

Fabrice Baray[1,2], Philippe Codognet[2], Daniel Diaz[3], and Henri Michel[1]

[1] ST-Microelectronics, Central R&D, France
{Fabrice.Baray,Henri.Michel}@st.com
[2] University of Paris 6, LIP6, France
Philippe.Codognet@lip6.fr
[3] University of Paris 1, France
Daniel.Diaz@univ-paris1.fr

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 this paper we address the problem of automatic generation of high quality test-suites for microprocessor functional models validation. We present the design and implementation 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. The test vectors are computed with a dedicated constraint solver designed to handle specific constraints related to typical constructs found in microprocessor descriptions. Results are illustrated with a case study.

Keywords: Code-based test generation, functional hardware verification, constraint solving techniques

1 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.

H. Garavel and J. Hatcliff (Eds.): TACAS 2003, LNCS 2619, pp. 569–584, 2003.

1.1 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 (change 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.

A second distinction is between the style of models, it can be an *hardware model* written in an high level hardware description language such as Verilog or VHDL, in this case the model will be used as input to synthesis tools to finally produce a network of transistors which constitutes the chip. Even with *high level* description languages such models bear some structural similarity with the hardware, and as a consequence are complex and slow. A different kind of models are *software models* which may have very little internal structural similarity with the hardware, but provide a good estimate (if not perfect) of the hardware behavior. Software models should be easier to write, and much faster to run (2 to 3 orders of magnitude is a common requirement).

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 the 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 stable architectural state. In this view there is no notion of parallel execution or instructions whose execution may involve several cycles.

1.2 Use of a Simulation Model for Hardware Verification

Though formal verification methods are standard for microprocessor verification, the complete verification flow for a complex microprocessor still includes pseudo-random generation of very large test-suites (see for example [3] 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 code exhibit the same behavior on the HDL model (typically VHDL or Verilog) and on the functional simulator.

In this method small sequences of assembler code are produced, with random (but directed) generators. A typical tool for this task is the Genesys tool ([2], [14]), 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.

1.3 Differences between Bugs in ISS and in HDL Models

Experience in the verification of recent microprocessors or DSP design projects seems to indicate that the curves of rate of new bugs discovery always exhibit the same shape (see figure 1).

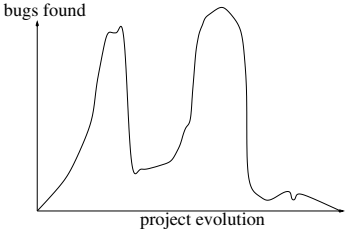

Fig. 1. Rate of new bugs discovery over project life

The rate of bugs discovery increases to a first peak, which corresponds to the time where

- the test generation machinery is in place
- the Hardware designers have implemented most instructions: the processor may not run many programs, but at least it can run simple programs.
- The architecture is mastered by both the hardware designers and the developers of the ISS model

Then, the rate of bug discoveries decreases. All complex microarchitecture mechanisms may not be implemented, for example the verification engineers are explicitly requested not to generate fragments of assembler with 2 consecutive *load* instructions because the memory control unit is far from mature. In this phase, many tests may be explicitly disabled from the test-suite because there is no point of hitting too much blocks where there are already many bugs to fix.

The second peak corresponds to the moment where all complex hardware models are in place and the verification engineers are requested to stress the hardware model as much as possible. At this point the bugs become more and more subtle ([10], and the time to solve a bug is much larger, debugging typically involves analyzing complex *waveforms* of signals : a very painful process.

It is a particularly annoying experience at this moment of the project to find out that discrepancy of behaviors between the Hardware model and the reference simulator are due to bugs in the (much less complex) functional model. Unfortunately it is not uncommon to find bugs in the ISS late in the design project.

The justification of the work presented in this paper is to make sure that the ISS reaches complete maturity in the first phase of the project (before the first peak in the bugs curve). At this moment it is crucial to have a test suite which has stressed the ISS in all possible ways (even if it has not stressed the Hardware model in all possible ways).

The automatically generated ISS test-suite 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

In the remainder of this paper, section 2 introduces the semantic aspects addressed in our input language and the test generation methodology. Section 3 gives some key points on the input language analysis and the constraint problem generation. Section 4 describes the constraint solving general domain and why a new constraint solver was developed to solve our specific problem of test vector generation. Finally, we will mention one case study on a ST micro-controller with quantitative and comparative results.

2 Context

2.1 Input Language

Many languages exist to describe microprocessors architecture. They are used for either educational purposes [13], reference documentation of an instruction set (the Sparc Reference manual for example [18]), or they are real languages for which a compiler exists to convert the architectural description into an executable ISS. Many companies have used proprietary languages to develop microprocessors, (AMD RTL language for example [15]). STMicroelectronics also is using a language (*idl*) to derive several flavors of an ISS as part of the *flexsim* technology. Today SystemC avoids the need of a special compiler, all useful specific constructs useful to write a simulator being implemented as C++ classes.

In order to propose an open approach and avoid syntactic details from particular languages, we define in this section a small language called x which focuses only on key abstract semantic aspects. Note that this language is in fact the *functional* subset of SystemC, we can compile x source with a C++ compiler and the SystemC libraries.

An x description defines registers, memory and functions to represent execution of a functional cycle of the microprocessor. By functional cycle we mean the behavior to execute one instruction without taking into account hardware

clock cycles. This cycle consists in fetching the instruction bytes from the memory, decoding the instruction and executing the effective operational part with the operand values. The x language, syntactically based on C, introduces the minimal structures to represent this kind of program, that is roughly : integer types, scalar and array variables, if-then-else and switch constructs and function definitions. The following table details the terminal symbols of the x grammar.

Terminal symbol	**Meaning**
K, V, T, F	constant, variable, type and function
uop, *bop*	unary and binary operator

Types T are integral types (either *signed* or *unsigned*). Since this is a language to model the behavior of processors, an integral type explicitly specifies the number of bits used to represent it. Signed entities are represented with a 2's complement encoding, other alternative integer encoding disappeared long ago among processors and it is highly unlikely that future processors adopt a different encoding for integers. For compatibility reasons with C++ we adopt a template notation, *signed int* $< 5 >$ for a signed integer type on 5 bits for example.

The language has support for unidimensional arrays. Elements in an array all have the same type. Elements are indexed with an integer starting at 0. The size s of an array is specified at declaration, the valid range of array indices being then $0..s-1$.

We implemented a subset of all operators of the C language, with the following restriction: all operators which are described as *implementation-defined* in C language reference [11] are either absent from our language or their semantics is more precise than in C. In particular $++$ $--$ (post or pre increment or decrement) are absent because the semantics of expressions which contain multi side-effects is *implementation-defined*. Similarly calls to the same function in the same expression, and arguments evalation are done from left to right.

Unary operators *uop* are composed of logical not !, bitwise not $\sim$ and arithmetic negation $-$. Binary operators *bop* are composed of classical arithmetic operators ($+$ $-$ $*$ /), classical bitwise operators (& | $\wedge$ $>>$ $<<$), bit concatenation of two operands (*concat*), bit extraction (*extract*) and classical relational, equality and logical operators ($==$ $\neq$ $>$ $\geq$ $<$ $\leq$ && ||).

Non terminal symbol	**Meaning**
D_v, D_f	variable and function definitions
E, S, P	expressions, statements and program

The abstract grammar is defined by the following rules ([...] is an optional construction) :

$D_v := T_0\ V_0, \ldots, V_n$ *variable definition*
$D_f := T\ F(T_0\ V_0, \ldots, T_n\ V_n)\ \{\ D_v^0; \ldots; D_v^m\ ;\ S_0 \ldots S_l\ \}$ *function definition*

$$
\begin{array}{lll}
E := K & & \textit{constant value} \\
\quad | \; V & & \textit{variable access} \\
\quad | \; (E_1) & & \\
\quad | \; E_1 \; bop \; E_2 & & \textit{binary operator} \\
\quad | \; uop(E_1) & & \textit{unary operator} \\
\quad | \; (T) \; E_1 & & \textit{casting operator} \\
\quad | \; F(E_0, \ldots, E_n) & & \textit{function call} \\
S := \{ \; [D_v^0; \ldots; D_v^n \; ; \; S_0; \ldots; S_m] \; \} & & \textit{nested block} \\
\quad | \; V \; = \; E & & \textit{assignment} \\
\quad | \; V \; := \; E & & \textit{delayed assignment} \\
\quad | \; \textit{if} \; (E) \; S_1 \; [\textit{else} \; S_2] & & \textit{selection statements} \\
\quad | \; \textit{switch} \; (E) & & \\
\qquad \textit{case} \; K_1 \; : \; [\ldots \; \textit{case} \; K_o] \; : \; S_1^1 \; \ldots \; S_1^k & & \\
\qquad \ldots & & \\
\qquad \textit{case} \; K_i \; : \; [\ldots \; \textit{case} \; K_m] \; : \; S_n^1 \; \ldots \; S_n^k & & \\
\qquad \textit{default} \; : \; S_{n+1} \; \ldots \; S_{n+1}^k & & \\
\quad | \; \textit{return} \; (E) & & \textit{return of function} \\
P := D_v^0; \ldots; D_v^n; D_f^0 \; \ldots D_f^p \; \{ \; S_0; \ldots; S_m \; \} & & \textit{global program}
\end{array}
$$

In addition, the following restrictions apply : no nested function definitions, no recursive function call and functions need to be defined before being used.

It is worth noticing the existence of a delayed assignment which performs the assignment of a new value at the end of the execution cycle. This typical hardware functionality is mainly used to simplify the description which normally uses temporary variables to store intermediate results before assignment to registers and memory at the end of the cycle.

2.2 Test Generation Methodology

In a program P, the global variable definitions denotes the input values of the description. A test vector can be viewed as a couple $< v, P(v) >$, where v contains the input values for each global variable definition, and $P(v)$ contains the expected output values. In fact, the processor description is mainly composed of the decoding statements, leading to many *switch* instructions on the *codeop*. Therefore, this *branch specificity* naturally induces a *path coverage* criterion. *Path coverage* criterion is the closest quantitative approximation of the quality of the tests in this context. Therefore, the test generation strategy is decomposed into two phases :

1. the first one generates from the code as many *constraint stores* as paths in the description, this step will be explained in Section 3 ;
2. the second one analyzes the store to generate values for the test vector corresponding to this path, this step will be detailed in Section 4.

3 Constraint Store Generation

The method for generating constraints from a description in the x language is depicted on Figure 2, which illustrates the development tool called **STTVC** (ST Test Vectors Constructor).

We want to create a *code explorer* : a program which computes the test vectors according to the error types we want to detect in the ISS functional model. This is done by translating the hardware description in x into a C program, which is then linked to the STCS solver library. This translations is a purely static analysis, dealing mainly with type analysis.

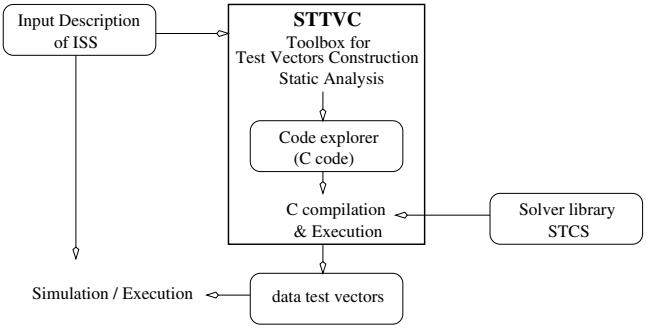

Fig. 2. Tool architecture

Let us now present through examples the path analysis induced by the path coverage criterion and then detail the generation of constraints associated to expressions, bit manipulations and array management.

3.1 Paths Analysis

Consider the two following fragments of code:

```
uint<8> X,Y,Z;

void cycle() {
  if (X==5) {          // first
    if (Y==5) {        // second
    }
  }
  if (Z>4 && Z<10) { // third
  }
}
```

(a)

```
uint<8> X,Y,Z;

uint<8> f() {
  if (Z<10) return ..;
  return ..;
}

void cycle() {
  if (Z>4 && f()!=0) ;
}
```

(b)

Fig. 3. Code examples

Test vectors have to be found to distinguish between each path in this code. A path can be completely defined by giving a truth value for each conditional statement. For instance, a unique path is defined by considering, in the fragment (a), the first boolean expression being true, the second false and the third true.

Such a program can be represented by a control flow graph (CFG). A CFG is a directed graph where each node represents a statement and the arcs (eventually labeled with conditions on variables) represent the computational flow, together with a starting and an ending node. All function calls are inlined, creating a single CFG for the main function (in other words, the code is flattened). The CFGs of fragments (a) and (b) are shown in Figure 4.

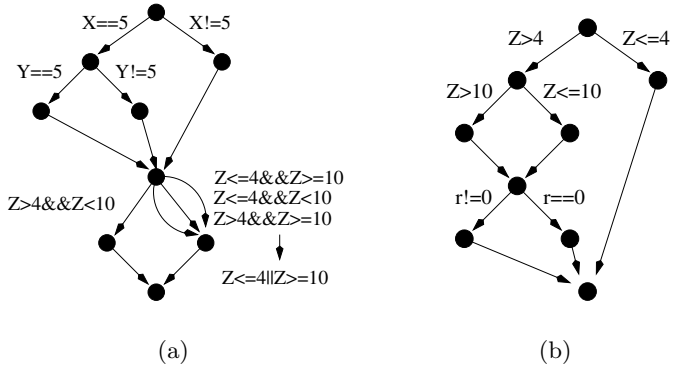

Fig. 4. Control Flow Graph

Exploring all paths of the code comes down to compute all paths in the graph from the starting node to the ending node. When a conditional expression uses a boolean operator (*and*, *or*) one has to generate one path for each combination of truth values for the involved sub-expressions. Therefore for a condition with n conjunctions, 2^n arcs join in the graph the *if* node and the nodes corresponding to the *then* and *else* statements. Simple optimizations can avoid spurious paths like in example (b) of figures 3 and 4 where the function call f is only evaluated when $Z > 4$.

From an implementation point of view, all truth values associated to conditional expressions can be grouped into a single boolean vector, called the *control flow vector* (CFV for short). Exploring all paths thus consists in generating all possible values for the CFV. Now, in order to generate the correct constraints, we simply have to rewrite the initial code by replacing all conditional expressions by a (boolean) test on the corresponding element of CFV and add the associated constraint to the current store.

Exploring all paths thus consists in enumerating all CFV exhaustively. An interesting optimization to avoid exploring useless paths is as follows : whenever the conditions on the labels of two arcs in a beginning of a path are inconsistent, all paths with this prefix can be discarded.

3.2 Variables, Expressions, and Assignments

The global and uninitialized variables of the program (like X, Y or Z in the previous example) are considered input variables. Solving the accumulated constraints on these variables provides a set of possible input values for each path.

All conditional expressions in the x code implicitly assume these constraints on the variables. Complex expressions are decomposed into primitive operations (see section 4) with temporary variables for the intermediate results.

Each temporary variable created for a binary operator has its own type which depends on the types of the operands. For instance, the addition of two 8-bit unsigned numbers gives an unsigned variable on 9 bits ; the extraction of 4 bits from a variable gives an unsigned variable on 4 bits. Details of the typing rules are strongly dependent from the input language semantics.

As observed, only conditional statements give rise to constraints, not assignment statements. An assignment should rather influence the following statements, as classically resolved by transforming code into a Static Single Assignment form (SSA) [6] – a process also called *spatial incarnation* [17]. For instance, an assignment *X=5* cannot simply constrain X to be equal to 5 because it would prevent any further assignment of X to different values. Instead, all future uses of X should thereafter reference a new variable bound to the constant 5. Delayed assignments (executed at the end of a cycle execution) are also possible in the x language and in our translation method these delayed assignments do not generate constraints either.

Furthermore, notice that boolean expressions used in assignments do not generate constraints directly. The statement *B=(X< Y)* does not imply that X is less than Y, but only that the value of B depends on the value of the condition. These kinds of constraints are classically called *reified constraints*.

3.3 Array Constraints

In [9,17], array manipulations are performed using equivalent constructions of SSA form. The key idea is to generate, for each assignment of an array element, a set of *element* constraints of the underlying Constraint Logic Programming language. A single assignment inside an array of size n thus introduces $2n-1$ *element* constraints (2 constraints for $n-1$ indices which are not assigned plus 1 for the assigned one). In the functional description of the ST processor that will be detailed later, memory is represented as an array which is addressed with 24 bits index registers. An address register of size v implies a 2^v memory size, the approach of [9] would lead therefore to a huge number of *element* constraints (proportional to number of write access $\times\ 2^v$). This is unsolvable in practice. We have therefore developed another approach that is introduced by the following simple example :

```
uint<8> mem[256]
uint<8> X,Y;
void cycle() {
  if (mem[X]==5) { ... } // X index named X1
  mem[Y]=2;
  if (mem[X]==5) { ... } // X index named X2
}
```

The first idea consists in introducing directly in the solver an array type with two new constraints for read and write access to its elements (*tabRead* and *tabWrite*). These constraints are responsible for maintaining the consistency of the array values by enforcing the following rules. These two rules are equivalent,

but the solver implementation takes into account the two propagations : from the indices relationship to the value relationship and vice versa.

$$\begin{array}{ll} i = j & \Rightarrow T[i] = T[j] \\ T[i] \neq T[j] & \Rightarrow i \neq j \end{array}$$

For instance, if Y is equal to X, the second access to $mem[X]$ (identified by $X2$ below) should give the value 2. Two *tabRead* and one *tabWrite* constraints have to be added to the store but they are order-dependent as the order of accesses in the initial code must be guaranteed. This violates the classical constraint programming paradigm where constraint solving is order-independent. Moreover, static dependency analysis between variables is very limited in this approach because indices are only known dynamically.

Another strategy to handle array manipulations can be defined by dealing with this problem in two different steps. Since the complexity of array manipulations directly depends on the equality or inequality of indices, the first step can be to fix relationships between indices, and then push in a second step constraints enforcing such relationships.

For instance, among the three array accesses in the previous example, some may be *aliases* (refer to the same memory location, because the indices are the same). The number of possible relationships between n indices corresponds to the number of partitions of an n-set, which is the n^{th} *Bell number* b_n [1].

example of relations		
(X_1, Y)	(X_1, X_2)	(Y, X_2)
$=$	$=$	$=$
$=$	$\neq$	$\neq$
$\neq$	$=$	$\neq$
$\neq$	$\neq$	$=$
$\neq$	$\neq$	$\neq$

formula of Bell numbers

$$b_0 = 1$$

$$b_{n+1} = \sum_{k=0}^{n} C_k^n b_k \text{ where } n \geq 0$$

$$C_k^n = \frac{n!}{k!(n-k)!} \text{ (binomial coeffs)}$$

The exponential complexity of the Bell numbers does not seem to be a problem considering that only a correct combination for indices is searched for. An *alias coverage* criterion could be defined to estimate the coverage of the different choices of indices relationship. This new coverage criterion makes sense to verify self reference to memory such as *mem[mem[i]] where mem[i]=i*. This second approach for array manipulation has the following advantages:

- variable independence can be analyzed in order to split the constraint store into independent groups of variables ;
- further graph analysis can be performed to deduce heuristics for variable ordering to solve the constraints with improved performances.

The main drawback of this solution is the fact that the choice of relationships between indices can be inconsistent with other constraints in the store. For instance, it is not possible to choose $Y = X_2$ and to generate a test for the path

satisfying the second test. We rely on the backtracking mechanism of the constraint solver to recover from such failures. Moreover in many cases inconsistent choices are avoided with a symbolic solver.

4 Constraint Solving

Constraint Programming has proved to be very successful for Problem Solving and Combinatorial Optimization applications by combining the declarativity of a high-level language with the efficiency of specialized algorithms for constraint solving, borrowing sometimes techniques from Operational Research and Numerical Analysis, see [16] for a general survey and [12] for a complete description of the paradigm of Constraint Logic Programming on which this work has been based. We will first introduce the basic notions of classical constraint solving and then detail the specific constraint solver that has been developed, and its specific features.

4.1 Classical Constraint Solvers

A constraint is a logical relationship 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 [5]. 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 $[0 \ldots C]$. However for much more constrained variables, the interval representation is only an approximation of its 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 a value 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 concerning the order in which variables are considered and the order in which values are considered in the variable domains.

4.2 Specific Constraint Solver

The first idea to solve our test generation problem was to use the existing solver GNU Prolog [4], like in [8]. However such an approach has several drawbacks because the constraints we are dealing with include some peculiar operations derived from the hardware description language:

- 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. For instance, let Y and Z be unsigned variables on 8 bit which are constrained to be strictly greater than 0, and X be an unsigned 9-bit variable. The constraint $X = Y + Z$ reduces the domain of X to be greater than or equal to 2. But if an 8-bit unsigned variable T is equal to the casting of X, its value can be 0 or 1 due to the loss of the 9^{th} bit of X. Moreover, 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 composed 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). [8] proposes to consider variables as bit vectors and constraints on variable bits as several independent constraints on individual bits. This solution however cannot be combined with arithmetic constraints on the same variable and thus prevents possible domain reductions.
- 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 contributions of this work concerns the development of a dedicated constraint solver, named STCS. STCS was mainly developed with the experience of GNU Prolog, but also introduces new specific constraints : *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) (X, Y, Z being variables, C a constant).

The solver implements these specific constraints with two internal representations of the domain of each variable : a simple interval representation for the arithmetic constraints (*min_value, max_value* values) and a *bit* representation in order to handle efficiently the *bit* based constraints. This bit representation maintains the information of known bits in the bit-vector. In the solver, the propagation is decomposed into bit propagation and interval propagation and coherence between the two representations is maintained by enforcing the following rules :

- at each interval modification, the bit representation is checked for modification (propagation of most significant bits, 1 for *min_value* and 0 for *max_value*, into the bit representation ;
- at each bit modification, a new *min_value, max_value* interval is computed according to the known bits.

Moreover, this new solver was developed as a general library, and is not limited to solve microprocessor test generation problems; it can be reused in other domains or other applications.

5 Case Study: STM7 Micro-Controller

In order to evaluate the performances and applicability of this innovative test generation method and the STTVC tool presented 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 main instructions, its description in x includes about 3000 lines of code. Combining the addressing modes with the instruction set gives about 700 various instructions.

Starting from an x functional description (issued from the IDL ST description), test vectors are generated with the STTVC tool described earlier and are simulated with three platforms : x, IDL and VHDL. 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 instruction to the test and saves the context after each test cycle. For each simulation, the main result is composed of the set of test vectors which don't give the same results on all platforms. This summarizes the differences between the different models of the same micro-controller. On the other hand, efficiency results in terms of percentage of coverage and simulation time can be computed.

5.1 Evaluating the Number of Paths

The number of possible paths for the specification can be statically evaluated on the control flow graph. Each sequence of test statements introduces a multiplicative relationship between number of "before paths" and "after paths". Each nesting of test statements introduces an additive relationship and, for switch statements, m multiple cases with the same statements containing n paths (*case* without *break* in C) induce globally mn paths.

On the STM7 case study, the description defines 21 functions. The total number of potential paths is about 143 millions, which is a relatively frightening number. The *cut* in the path search space must be very efficient to handle such a description. In fact, the number of evaluated paths by STTVC tool, after constraint store consistency analysis and reduction of the search space, is 18457 which is a good result. This drastic reduction can be explained by analyzing the microprocessor description as follows: although each function has a great number of internal paths and can be called many times, each call activates only a small number of these paths.

In order to validate our *cut* implementation, STTVC was run without *cut* optimization during more than 2 days on the same example; we generated exactly the same test vectors.

5.2 Tests Generation

Table 1 presents statistics about the execution of STTVC on a SUN Ultra-SPARC-II 450 MHz processor with enough memory (only about 5Mo of memory

used). The main point is that all impractical paths of the description are detected without labeling the variables (detecting an impractical path by labeling could be very time consuming).

This good result is mainly due to the detection by the domain inconsistency of impractical paths, but the symbolic solver also has an important impact. The role of the symbolic task is the same as the domain inconsistency (to avoid impractical paths) but the inconsistency relationship between variables which is handled symbolically is not detected by the basic domain propagation algorithm.

Array fails indicates the number of choices of relations (equalities, inequalities) among couples of indices which are not consistent with other constraints, but not deduced before the labeling. This stresses one drawback of this solution for handling arrays.

Table 1. Paths statistics

Checked paths	18457
Impractical paths detected by domain inconsistency	16299
Impractical paths detected by symbolic solver	236
Impractical paths detected by labeling	0
Array fails	56
Test vectors generated	1922
Global CPU Time	1 hour 05 minutes

In terms of generation time, the results vary a lot. Ninety percent of tests are trivial for the solver and can be computed in a time less than one second. For a small, but non predictable, number of paths, the constraint solver takes a lot of time to label variables to find a correct solution (the worst test case takes 13 minutes to compute). This result is not surprising. Due to the NP completeness of the constraint solving problem some constraint stores are more complex to solve.

5.3 Tests Execution

Given our generated test-suite with 1922 test vectors, the next step consists in executing the tests for acquiring error detection and coverage percentages in order to compare with coverage calculated with the simulation of classical manually written assembler test-suite.

In terms of cycle number, the STM7 assembler test suite is ten times bigger than our test vectors simulation with 18941 cycles and takes more time to execute, compared with only 1922 cycles for 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 2 compares the assembler test-suite 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 max) and IBB for implicit basic blocks (48 max) which correspond to the *else* part of an *if* statement without *else* or to the *default* part of a *switch* statement without *default*.

Table 2. Statement coverage results

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

With this criterion, the results are quite similar. For the assembler test-suite 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 test-suite 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, this gives confidence in STTVC correctness. Furthermore, as could be expected, the assembler test suite covers only 72 percent of the test paths generated by STTVC.

6 Conclusion and Perspectives

We have presented a code-based test generation method for the validation of functional microprocessor description. It is based on the idea of solving constraint stores, each one representing a path in the control flow of the processor cycle. Due to the domain specific aspects, the solver contains dedicated primitive constraints to handle efficiently the bit manipulation operations of hardware descriptions.

Performances of this approach have been measured on a real case study, the STM7 microprocessor, in order to evaluate the quality of the test generation. A comparison with a classical assembler test suite shows the good results of our method in terms of both simulation cycles and path coverage percentage. Current and future research could concern the development of

- an enhanced propagation mechanism between numerical calculus and bit manipulation in order to avoid useless labeling ;
- a general framework for the symbolic solver in order to have a better detection of unpracticable paths ;
- tests generation for a better coverage criterion for the detection of human errors in the functional model, by using mutation aspects [7] ;
- a comparison between our constraint solver with bit oriented constraint and SAT techniques.

Acknowledgment. We thank Emanuele Baggetta for the development of a graphical interface to STTVC, and Jean-Claude Bauer who lead the STM7 ISS development effort.

References

1. www-gap.dcs.st-and.ac.uk/~history/Miscellaneous/StirlingBell/bell.html.
2. Aharon Aharon, Dave Goodman, Moshe Levinger, Yossi Lichtenstein, Yossi Malka, Charlotte Metzger, Moshe Molcho, and Gil Shurek. Test program generation for functional verification of powerpc processors in IBM. In *Design Automation Conference*, pages 279–285, 1995.
3. Françoise Casaubieilh, Anthony McIsaac, Mike Benjamin, Mike Bartley, François Pogodalla, Frédéric Rocheteau, Mohamed Belhadj, Jeremy Eggleton, Gérard Mas, Geoff Barrett, and Christian Berthet. Functional verification methodology of chameleon processor. In *Design Automation Conference, DAC'96*, pages 421–426, Las Vegas, Nevada, USA, 1996.
4. P. Codognet and D. Diaz. The gnu prolog system and its implementation. *Journal of Functional and Logic Programming*, 6, Oct 2001.
5. Philippe Codognet and Daniel Diaz. Compiling Constraints in clp(FD). *Journal of Logic Programming*, 27(3), June 1996.
6. R. Cytron, J. Ferrante, B.K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *Transactions on Programming Languages and Systems*, 4:451–490, Oct 1991.
7. Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, pages 34–41, April 1978.
8. Fabrizio Ferrandi, Michele Rendine, and Donatella Sciuto. Functional Verification for SystemC Descriptions Using Constraint Solving. In Carlos Delgado Kloos and Jose da Franca, editors, *Design Automation and Test in Europe (DATE'02)*, pages 744–751, Paris, France, 4-8 March 2002.
9. Arnaud Gotlieb. *Génération automatique de cas de test structurel avec la programmation logique par contraintes.* PhD thesis, Université de Nice-Sophia Antipolis, 2000.
10. Richard C. Ho, C. Han Yang, Mark A. Horowitz, and David L. Dill. Architecture validation for processors. In *ISCA*, pages 404–413, 1995.
11. ISO/IEC 9899. *Programming languages – C*, first edition, 1990.
12. Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19 & 20:503–582, 1994.
13. D. A. Patterson and J. L. Henessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1990.
14. Shai Rubin, Moshe Levinger, Randal Pratt, and william Moore. Fast construction of test-program generators for digital signal processors. In *ICASPP'99 Conference Proceedings*, 1999.
15. David M. Russinoff and Arthur Flatau. Rtl: A verrified floating-point multiplier. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer Aided Reasoning: ACL2 Case Studies*, pages 199–200. Kluwer Academic Press, 2000.
16. P. van Hentenryck et al. V. Saraswat. Constraint programming. *ACM Computing Surveys*, 28, Dec 1996.
17. R. Vemuri and R. Kalyanaraman. Generation of design verification tests from behavioral vhdl programs using path enumeration and constraint programming. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 3:201–214, 1995.
18. David L. Weaver and Tom Germond. *The SPARC architecture manual (version 9).* Prentice-Hall PTR, Upper Saddle River, NJ 07458, USA, 1994.