Functional Verification for SystemC Descriptions Using Constraint Solving

Fabrizio Ferrandi
Politecnico di Milano
Dipartimento di Elettronica
e Informazione
ferrandi@elet.polimi.it

Michele Rendine
Politecnico di Milano
Dipartimento di Elettronica
e Informazione
mirendi@jumpy.it

Donatella Sciuto
Politecnico di Milano
Dipartimento di Elettronica
e Informazione
sciuto@elet.polimi.it

Abstract

This paper addresses the problem of test vectors generation starting from an high level description of the system under test, specified in SystemC. The verification method considered is based upon the simulation of input sequences. The system model adopted is the classical Finite State Machine model. Then, according to different strategies, a set of sequences can be obtained, where a sequence is an ordered set of transitions. For each of these sequences, a set of constraints is extracted. Test sequences can be obtained by generating and solving the constraints, by using a constraint solver (GProlog). A solution of the constraint solver yields the values of the input signals for which a sequence of transitions in the FSM is executed. If the constraints cannot be solved, it implies that the corresponding sequence cannot be executed by any test. The presented algorithm is not based on a specific fault model, but aims at reaching the highest possible path coverage.

1 Introduction

Functional verification is mainly based on the concept of simulation. There are a number of commercial simulators in the market which are able, with different degrees of precision and accuracy, of simulating the behavior of an hardware component, if we provide a specification with an hardware description language and a vector of input bits. The major problem of this technique is how to generate a set of input test vectors that is accurate enough to cover the largest possible set of behaviors of the component, using a limited amount of resources, typically represented by time. Manual and random approaches do not provide an accurate solution for the problem of verification, because there is no guarantee on the accuracy of the results that can be achieved in terms of bug-finding. This is why in the recent years the scientific community has focused its efforts upon this problem, trying to provide alternative techniques for the automatic generation of test cases. The algorithms that try to automatically generate test vectors, aimed at verification through simulation, are called ATPG (Automatic Test Pattern Generation). There are ATPGs based on the extended Finite State Machine model, such as [1], ATPGs based on genetic algorithms, such as [2] and [3], ATPG based on Binary Decision Diagrams, such as [4] and [5] and ATPG based on Assignment Decision Diagrams, such as [6]. The ATPG presented in this paper can be considered in the family of ATPGs based on controllability and observability through the solution of SAT problems. This approach has been developed also in [7], and [8]. Our algorithm improves these approaches by consideringing the solution of all constraints involving bits, bit vectors and integers that drive the control flow of the system under test. After the production of the test vectors, it is necessary to evaluate their effectiveness, and this goal can be achieved by using a coverage metric. Different criteria are commonly adopted to determine how good a test set is: statement coverage, branch coverage, condition coverage and path coverage. Path coverage is the most stringent of these criteria, the problem is that the number of paths is a generic description of an hardware model can be infinite. The algorithm we propose introduces the concept of sequence of transitions, where a sequence is a path of a given length in terms of time frames, and then generates all the test vectors necessary to exercise these sequences. In this way, we realize path coverage for all paths with a length that is less or equal to the fixed one.

2 Assumptions and Computational Model

The proposed test generation algorithm considers as internal description of the system under test the classical Finite State Machine computational model. A Finite State Machine, M, can be formalized as a 5-tuple

$$M = (I, U, S, S^0, R)$$

where $I \subseteq B^n$ represents the input alphabet, $U \subseteq B^m$ represents the output alphabet, $S \subseteq B^k$ is the set of states,

Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition (DATE'02)
1530-1591/02 $17.00 © 2002 IEEE

S^0 is the initial state and R is the global relation.
Naturally, the set B is defined as B=$\{0, 1\}$.
The use of FSMs can be exploited to represent concurrency: in fact, a complex system can always be described with a certain number of interconnected FSMs. To allow cooperation among interconnected FSMs, a communication and interaction mechanism is required: this is realized through FSMs composition. Through FSM composition we obtain a complex system, that is composed of a certain number of single FSMs, that however interact and exchange data among them. To realize such a system, a fundamental hypothesis must be satisfied: all FSMs must change state together. In this case it is possible to define the system state as the cartesian product of the states of all single FSMs composing the system. Moreover, we have considered only descriptions with a single clock signal.

3 The Test Generation Algorithm

3.1 Overview

The test generation approach proposed, aimed at verification, can be decomposed into 5 major tasks:

1. *Acquisition of Data*: In this phase all available information about the system is processed. These information are mainly constituted by four lists. These are the port list, the statement list, the conditional instruction list and the transition list. The lists are obtained from an analysis of the source SystemC code.

2. *Analysis of Transitions*: It consists in finding the initial and final state for each transition present in the transition list.

3. *Sequence Enumeration*: Sequence enumeration is the process of enumerating various potential execution paths through the SystemC program for which test vectors are required. The generation of these sequences can be performed according to different techniques. For instance, the breadth first approach with depth N is equivalent to consider all the possible execution paths that last N clock cycles.

4. *Analysis of Sequences and Production of Constraints*: For each considered sequence, a set of constraints is produced. This set of constraints corresponds to all the conditional instructions that have to result true during the execution of the sequence. The problem is that, in general, these constraints do not apply only to input ports, but also to variables and signals. Generally, it is not possible to force the value of a variable or a signal to a given value, but it is possible to backtrack through the code in order to obtain the direct dependence of the value of the variable or signal from a general set of input ports and constants. At the end of this phase, a set of different files is produced: each file is associated with a sequence, and reports the set of equations that must be verified for the execution of the sequence itself. These equations have as constraint variables only input ports, and also include domain constraints, that specify the domain from which the values of a constraint variable are drawn. All equations are written according to the GProlog format.

5. *Constraint Solving and Test Extraction*: Constraint Solving is the process of generating a solution, if one exists, to satisfy a set of constraints associated with a sequence. In our implementation we have used the GProlog constraint solver, but the technique used for constraint generation is independent of the constraint solver used. If the GProlog constraint solver does not find a solution, then no verification test can be generated for the sequence. If the constraint solver finds a solution, it is represented as a value assigned to each constraint variable. During the test extraction phase, we extract the test vectors from the solution written according to the GProlog output format, and arrange them in a suitable format for the SystemC simulator.

3.2 Acquisition of Data

During this phase, all the information about the topology of the system under test is collected. These information consists in four lists:

3.2.1 Port List

It is the list of all the input port of the system. The function of this list is to provide the data necessary during the equations expansion phase; moreover it is used also during the phase of constraint generation, in order to derive the correct upper and lower bounds for each constraint variable. If a port is declared as bit or bit vector, the only possible domain is the boolean one $\{0, 1\}$. If the port represents an integer, there is an additional information: in fact, if we declare a four bit integer, it can be both used to represent the numbers from 0 to 15, or from -8 to 7, and so on. In this case, also the upper and lower bound of the port, intended as the lower and the higher possible value that it is necessary to consider using N bits, is reported.

3.2.2 Statement List

This is the list containing the data about all the statements present in the code we want to test. A statement is defined as each operation the ends with that writing of a value over an operator, that is called target, and can be an output port or a variable or a signal. In general, the value written on the target depends on an undefined number of constants, input ports, variables and signals, but for the identified model only two kind of equations are allowed: unary equations, constituted by a single operator, and binary equations, constituted by two operators and one operation. In fact, it is possible to prove that each complex equation can be divided in a set of binary equations, through the introduction of a certain number of temporary variables. The adopted model allows different operations:

- Logic operations: NOT,AND,OR,XOR,NOR,NAND.
- Mathematical operations: Addition, Multiplication, Subtraction, Quotient of Division, Remainder of Division, Exponential.
- Operations on bit vectors: Concatenation, Selection of single bit, Selection of subvectors of bits.

This set of operations is sufficiently rich to allow the modeling of a large set of SystemC operations.

3.2.3 Conditional Instruction List

This list contains all the conditional instructions present in the code. Each conditional instruction corresponds to a condition that must be verified upon a variable or a signal or an input port. In our model, we have considered only two kind of conditional instructions:

- *Binary branch*: corresponds to a binary disjunction in the code. If the condition is true, a certain piece of code is executed, corresponding to the $then$ branch, otherwise another piece of code is executed, corresponding to the $else$ branch. This conditional instruction can be used to represent the if-then-else control instruction, but also more complex structures, such as the if-elseif-else control instruction.
- *Multiple branch*: corresponds to a disjunction of the code that presents more than two branches. The executed piece of code corresponds to the first condition that results to be true. This conditional instruction can be used to model the case instruction.

Naturally, the entire SystemC language presents a number of control instructions that is larger with respect to the model we have introduced; however, it is possible to prove that, if we use the FSM model, all the conditional instructions involving loops, as for example for, or $while$, can be translated in an appropriate FSM description. When a conditional instruction is defined, it is necessary to identify the left and the right operator, and the comparison criterion. There are six different types of comparison criteria: Equal, Not Equal, Greater than, Less than, Greater or Equal than, Less or Equal than.

3.2.4 Transitions List

This list contains all the possible transitions that are allowed in the system description. A transition is defined as the ordered set of all statements and conditional instructions that can be executed and verified during a single clock cycle. Each transition, in practice, corresponds to a path that can be followed in the system under test, if we limit the duration of the simulation to a single clock cycle. These transitions can be obtained by a recursive visit of the control data flow graph that represents the topology of the system under test. Let us consider the following example, called Example 1.

This piece of code represents a simple Finite State Machine. Figure 2 represents the Control Data Flow graph extracted from Example 1. The graph contains both the information about control conditions and about data flows (assignments on variables, signals and ports). If a single clock cycle is considered, each different possible path from the starting point of the CDFG to the final point of the CDFG represents a transition for the FSM. In the example, the number of possible paths is 8: this means that, during each clock cycle, the system exercises one of these 8 transitions.

3.3 Analysis of Transitions

After the acquisition of all lists, the second step of the methodology consists in analyzing each transition. The goal is to establish, for each of these transitions, the initial state and the final state. In fact, during the next step, all the available sequences of transitions will be considered, and a sequence is meaningful if and only if the final state of each transition composing the sequence corresponds to the initial state of the previous transition. The identification of the initial and final state of each transition is performed through the application of two algorithms:

- *Initial state*: the initial state can be obtained by the first multiple branch instruction on the state variable. In fact, if we refer to the classical FSM model, the backbone of the code is constituted by a case instruction on the value of the state variable. If the transition does not present any multiple branch instruction on the value of the state variable, it is assumed to have, as initial state, the reset state.
- *Final state*: the final state can be obtained by the last assignment statement on the state variable. If the transition does not present any assignment statement on the state variable, it is assumed to have, as final state, the same state that has been identified as initial state. It is important to notice that it is possible to find sequences with the same initial and final state; this, however, does not mean that the sequences can be considered the same one, because they are characterized by a different set of statements and conditional instructions.

```
#include <systemc.h>

SC_MODULE(fsm) {

  sc_in<bool> clock;
  sc_in<bool> reset;
  sc_in<bool> line;
  sc_out<bool> u;

  enum {A,B,C,D} state;

  SC_CTOR(fsm)
    {
    SC_METHOD(entry);
    sensitive_pos(clock);
    };
  void entry();
};

void fsm::entry() {
  sc_bit temp;
  if (reset.read()==true)
  {
    state=A;
    temp='0';
    u.write(false);
  }
  else
  {
    switch (state) {
    case A:
      temp='0';
      state=B;
      if (line.read()== false)
        temp=line.read();
      else
        temp=line.read();
    case B:
      if (temp=='0')
        state=C;
      else
        state=D;
      u.write(false);
    case C:
      if (line.read()==false)
        state=D;
      else
        state=A;
      u.write(false);
    case D:
      state=B;
      u.write((bool)temp);
    }
  }
}
```

Figure 1. Example 1

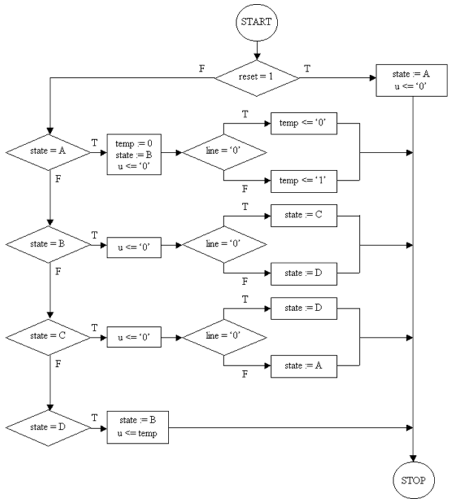

Figure 2. Control Data Flow graph

At the end of this phase, it is possible to obtain a graph, that shows all the possible transitions between the states of the system. Each arc between two states represents a transition, and is labeled with the transition identifier. According to what we said before, it is possible to have multiple arcs between two states. Figure 3 shows this graph for Example 1.

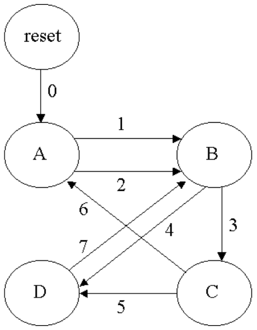

Figure 3. States and transitions graph of Example 1.

3.4 Sequence Enumeration

The following step is the enumeration of all theoretically possible sequences. A sequence is defined as an ordered set of transitions. A sequence is theoretically feasible if and only if the final state of each transition composing the sequence is equal to the initial state of the following transition. Moreover, each sequence always starts from the reset state (this means that the initial state of the first transition of each sequence is always the reset state). Once defined these simple rules, it is possible to define different criteria to select and enumerate the sequences. In the following, a number of these criteria are illustrated:

1. *Breadth first*: given a number N, the breadth first approach selects all the sequences that are composed by exactly N transitions. This strategy provides the higher possible coverage, because all the possible sequences with a length of N are considered. In practice, it corresponds to the path coverage, where each path lasts at most N clock cycles. However, this is also the most heavy strategy under the point of view of the computational complexity: in the worst case, if we have a fully connected graph, the number of generated sequences is $O(T_r^{(T_m)})$, where T_r is the average number of transitions starting from each state, and T_m is the number of considered time frames. If we consider an example where there are only two transitions starting from each state, the number of generated sequences will be exactly $2^{(T_m-1)}$.

2. *Breadth first plus transition selection*: this strategy is similar to the previous one; the user provides, as input, a set of transitions S. Then, the sequences are generated according to the breadth-first approach, but a sequence is accepted only if all transitions belong to set S. Through this approach, it is possible to discard a large number of sequences, and to save time and memory. Naturally this happens because the attention is focused on a smaller subset of transitions.

3. *Transition coverage*: this strategy is aimed at covering all transitions, but not necessarily all paths. A list of covered transitions is built, where for each transition it is reported if it has been covered or not. Then the sequences are generated according to the breadth first approach, but if a sequence is constituted by transitions that have been covered by previous sequences, it is discarded. In practice, a sequence is added to the list of the valid sequences if and only if it covers at least one transition that has not been previously covered by another sequence. The algorithm stops when each transition has been covered at least once.

At the end of this phase, a list of theoretically feasible sequences is obtained. Each one of these sequences is analyzed and processed during the following step.

3.5 Analysis of Sequences and Production of Constraints

The goal of this phase of the algorithm is to find, for each considered sequence, the set of constraints that corresponds to the execution of that specific sequence. This set of constraints is written according to the GProlog format. In general, each constraint derives from a control instruction, and the operators of a control instruction can be input ports, variables, signals or constraints. Let us consider the case of a variable x used in an if control instruction, as follows:

```
if (x==’0’) then
    {block 1}
else
    {block 2}
```

If the considered sequence contains the transition corresponding to the true branch of the if control instruction, it is possible to derive the following constraint:

$$x = 0,$$

x is not an input port, so it is not directly controllable. The solution of this constraint is not useful to identify the primary input values that control the variable. Moreover, let us consider the case where the variable x is subject to another assignment, and then used in another control instruction. It is necessary to distinguish the two different values of the same variable during two different moments of the execution. To solve these two problems, it is necessary to introduce first of all the information about time, then to apply an expansion algorithm, that is able to write constraints containing only input ports and constants.

- *Time*: The information about time can be easily derived from the structure of the sequences. Each transition corresponds to a different time frame, because in our model a transition is defined as the set of instructions and conditions that can be executed and verified in a single time frame. This means that it is possible to associate a mark to each variable, signal and input port. Variables, signals and input ports characterized by the same name, but a different time mark, could have different values. In the GProlog equations, the time mark is added to the name of the variable, signal or input port after a double underscore.

- *Expansion Algorithm*: Let us consider the following list of constraints:

  $$\begin{array}{l} reset\_\_1 = 1 \\ reset\_\_2 = 0 \\ state\_\_2 = A \\ reset\_\_3 = 0 \\ state\_\_3 = B \\ temp\_\_3 = 1 \end{array}$$

  While $reset$ is an input port, and is directly controllable, $state$ and $temp$ are two variables. We need to express these variables in terms of known signals (constants) or controllable signals (input ports). This operation can be obtained by a backward analysis of the code. Let us consider the previous example. The equations are allocated in memory as binary trees. All the leaves of these trees that are variables or signals could be subject to expansion. If we scan the list of statements executed during the sequence, starting from the last one to the first one, during the second clock cycle we find:

$$temp = line;$$

Due to the fact that this statement is executed during the second clock cycle, and this time mark is lower than the time mark of the variable temp in the equation trees, the substitution is executed. The equation tree changes, and this means that the constraint is modified into:

$$(line\_\_2) = 1$$

In practice, every time we find a statement having a variable as left operand, if the variable is also a leaf for the equation tree and the time mark of the variable in the tree is greater or equal to the time frame when the statement is executed, then the leaf of the tree is expanded. The same procedure is applied in case of signals. If each variable and signal of the system has been at least initialized, at the end of this phase all constraint equations should involve only known or directly controllable signals. If this does not happen, it means that there are variables or signals that have not been initialized, so we are dealing with a non-deterministic Finite State Machine.

- *Constraints production*: Once obtained the list of constraint equations for each sequence, it is necessary to translate these equations according to the GProlog format. Moreover, it is necessary to consider separately the case of bits, bit vectors and integers, because these three types belong to different domains, and are solved in different ways.

  – *Bit*: Table 1 shows the correspondence between the operations and comparators introduced previously and the GProlog equations. In this case, for each equation in the constraint equation list of the sequence, a single GProlog equation is produced. The domain of all constraint variables used in the equations is defined as the boolean domain $\{0, 1\}$.

Operation or comparator	GProlog
NOT	#\
AND	#/\
OR	#\/
XOR	##
NAND	#\/\
NOR	#\\/
Equal	# <=>
Not equal	#\ <=>

Table 1. Boolean operations and comparators

  – *Bit vector*: If at least one variable involved in a constraint equation is a bit vector, the situation is more complex. In fact, it is necessary to consider that more than a single GProlog equation can be extracted from a single constraint equation. Let us consider two bit vectors, $V1$ and $V2$, of the same size. If we consider the constraint equation:

$$V1 = V2$$

this implies that each bit of $V1$ must have a value equal to the corresponding bit of $V2$. In this case, it is possible to split a single constraint equation involving bit vectors in a series of GProlog equations involving a single bit. Each constraint equation between bit vectors involving an equivalence can be divided in a series of equivalences over the single bits of the bit vectors. This series of four equations in GProlog is shown below.

$$\begin{array}{c} V1\_0 \;\# <=> \; V2\_0 \\ V1\_1 \;\# <=> \; V2\_1 \\ V1\_2 \;\# <=> \; V2\_2 \\ V1\_3 \;\# <=> \; V2\_3 \end{array}$$

The number after the underscore represents the position of the bit in the array. The same procedure can be applied if the equivalence involves other logical operations, such as AND, OR, NOT, XOR, etc. A different approach must be used in case of inequality. Let us consider the following constraint:

$$V1 \neq V2$$

where $V1$ and $V2$ are bit vectors with a size of 4 bits. In this case, it is sufficient that at least one bit of $V1$ is different with respect to the corresponding bit of $V2$, in order to satisfy the constraint. In fact, each constraint equation

Operation	GProlog
Sum	$+$
Subtraction	$-$
Moltiplication	$*$
Quotient of integer division	$/$
Reminder of integer division	rem
Power	$**$
Equal	$\# =$
Not equal	$\#\backslash =$
Greater	$\# >$
Less	$\# <$
Greater or equal	$\# >=$
Less or equal	$\# =<$

Table 2. Mathematical operations and comparators

between bit vectors involving an inequality constraint can be translated into a single equation involving the single bits of the bit vectors. This equation can be written in GProlog language, using the table of equivalence presented before. The result is:

$$\begin{gathered}((\#\backslash(V1_0\#\#V2_0))\#/\backslash\\ (\#\backslash(V1_1\#\#V2_1))\#/\backslash\\ (\#\backslash(V1_2\#\#V2_2))\#/\backslash\\ (\#\backslash(V1_3\#\#V2_3)))\# <=> 0)\end{gathered}$$

– *Integer*: Integers represent a different kind of domain for GProlog equations. When a constraint variable is declared as an integer, it is necessary to specify also the upper and the lower bound of the integer. Table 2 represents the equivalence between the operations and comparators we have introduced previously and the corresponding GProlog instructions.

At the end of this process, for each sequence it is possible to obtain a set of GProlog equations. These equations are defined over variables belonging to the boolean or integer domain. Each constraint variable represents a bit of an input port, or a set of bits representing an integer. For the integer constraint variable, a range of finite values of existence is provided. This set of equations and domain constraints represents a SAT problem: if it is possible to find an assignment on the constraint variables that satisfies all constraints at the same, it is possible to determine an assignment over the input ports that exercise the sequence from which the constraints are originated.

3.6 Constraint Solving and Test Extraction

The last phase of the verification process consists in solving the constraint equations obtained for each sequence. There are two possible results:

1. The set of constraint equations obtained for a sequence cannot be satisfied. In this case, the sequence of transitions corresponding to the set of constraint equations is theoretically possible, but cannot be exercised in practice.

2. The set of constraint equations obtained for a sequence can be satisfied. In this case, the GProlog solver computes one possible solution for that set of constraints, and this solution is stored in a file.

Finally, the set of solutions obtained for all considered sequences is considered by a test vectors generator, that maps the values of the solutions over the input ports along the time frames in order to obtain a complete testbench.

3.7 Extension to Multiple Processes

The considered execution model for multiple processes is characterized by the fact that simulation time is advanced and signal values are updated only after all the processes are suspended. In this case, test generation must follow this model of execution. Consider the specification of n interacting FSMs, with $n \geq 1$, written using a single SystemC module. Each FSM defines a different process $P_1, P_2, P_3, ..., P_n$. If process Pi has xi possible transitions in a single clock cycle, then, since the processes can potentially execute independently of each other, there are $x_1 * x_2 * x_3 * ... * x_n$ potential execution paths in the program. Also in this case, some of these paths may be infeasible. This happens when processes influence each other's control flow by communicating through shared signals. Our approach to the problem consists in two phases: during the first phase, the algorithm is applied to each FSM independently. For each single process the sequences of transitions are generated, and for each sequence the constraint list is produced. Then, in a second phase, the interaction among different FSMs is considered. Let us consider the case of two interacting FSMs, F_1 and F_2. After the application of the algorithm, the number of sequences produced is respectively x_1 and x_2. This means that the total number of theoretically possible sequences is $x_1 * x_2$. Let us consider the following case: we want to generate the test vectors that will cause the execution of sequence $S_{1,i}$ for the Finite State Machine F_1 and sequence $S_{2,j}$ for the Finite State Machine F_2. The execution of $S_{1,i}$ imposes a certain number of constraints on the shared signals; these constraints must be added to the constraint list corresponding to sequence $S_{2,j}$

of F_2. At the same time, the constraint on the shared signal caused by the execution of the sequence $S_{2,j}$ of F_2 must be added to the constraint list corresponding to sequence $S_{1,i}$ of F_1. For instance, let us consider *out* an output port for F_1. The execution of the sequence $S_{1,i}$ provokes the fact that the value $'1'$ is assigned to this port during the first clock cycle. If this port represents a shared signal, there will be an input port of F_2 connected to the output port of F_1. Let us call this port in. In this case, the constraint to be added to the sequence $S_{2,j}$ is the following one:

$$(in\_\_2) = 1$$

In fact, when F_2 reads the value of the input port during the following clock cycle (the second one), it reads the value forced by F_1. In practice, the extension of the algorithm to multiple process architectures can be realized by adding the constraints over shared variables to each possible n-uple of theoretically possible sequences generated through the application of the basic test generation algorithm. We consider as shared variable each signal that is read from an output port for a process and than written on an input port of another process. The constraint is generated every time the output port is assigned to a value or an expression, and is added to the constraint list of the second process.

4 Experimental Results and Conclusion

The algorithm has been tested with a set of examples written in SystemC. These examples are translation in SystemC of a subset of benchmarks developed at Politecnico di Torino. The breadth first strategy has been used to produce the sequences of transitions. Table 3 shows the results of experiments: for each circuit, we have reported the depth of the breadth first algorithm, the number of theoretically possible sequences, the number of feasible sequences, the percentage of discarded sequences, and the path coverage reached through the test vectors generated by our algorithm. Moreover, we have also reported the path coverage reached by Commit [5], and Rage [2]. The results show that, in case of simple benchmarks without ports representing integers, the coverages are equivalent, otherwise the algorithm outperforms classical ATPG in terms of path coverage. This happens because the use of GProlog allows a faster and more efficient management of constraints involving integer variables. Future work will extend the experiments toward industrial SystemC descriptions, to deal with larger size examples.

Circ	Depth	T.seq	F.seq	Proposed	Commit	Rage
b01	10	512	512	100%	100%	100%
b02	11	50	50	100%	100%	100%
b03	7	26244	189	100%	100%	100%
b04	5	1296	100	46%	25%	35%
b06	7	2048	2048	100%	66%	96%
b10	5	108	108	87%	87%	87%
b11	8	372	205	68%	54%	68%

Table 3. Test generation results

References

[1] K. Cheng, A.S. Krishnakumar, *”Automatic generation of functional vectors using the extended finite state machine model”*, ACM Transactions on Design Automation of Electronic Systems, Volume 1, Issue 1, January 1996

[2] S. Chiusano, F. Corno, P. Prinetto, *”RT-level TPG exploiting high-level synthesis information”* Proceedings 17th IEEE VSLI Test Symposium, 1999

[3] F. Ferrandi, A. Fin, F. Fummi, D. Sciuto, *”An application of genetic algorithms and BDDs to functional testing”* Proceedings International Conference on Computer Design, 2000

[4] M.K. Ganai, A. Aziz, A. Kuehlmann, *”Enhancing simulation with BDDs and ATPG”*, Proceedings 36th Design Automation Conference, 1999

[5] F. Ferrandi, F. Fummi, D. Sciuto, *”Implicit test generation for behavioral VHDL models”*, Proceedings IEEE International Test Conference, 1998

[6] I. Ghosh, M. Fujita, *”Automatic test pattern generation for functional RTL circuits using Assignment Decision Diagrams”* Proceeding Design Automation Conference, 2000

[7] F. Fallah, P. Ashar, S. Devadas, *”Simulation vector generation from HDL descriptions for observability-enhanced statement coverage”*, Proceedings 35th Design Automation Conference, 1999

[8] R. Vemuri, R. Kalyanaraman, *”Generation of design verification test from behavioral programs using path enumeration and constraint programming”*; IEEE Transactions on Very Large Scale Integration Systems, Vol.3, No.2, June 1995