Test Generation based on CLP

Giuseppe Di Guglielmo, Franco Fummi,
Cristina Marconcini and Graziano Pravadelli
University of Verona
Italy

1. Introduction

The complexity of designs continues to rise, driven by technology advances, while time-to-market imposes always shorter time. Moreover, the increasing of design complexity implies that design verification becomes one of the most cost-dominating phase in design production.
Functional Automatic Test Pattern Generators (ATPGs) based on simulation (Corno et al., 2001; Fin & Fummi, 2003a) are fast, but generally, they are unable to cover corner cases, and they cannot prove untestability. On the contrary, functional ATPGs exploiting formal methods (Ghosh & Fujita, 2001; Zhang et al., 2003; Xin et al., 2005a), are exhaustive and cover corner cases, but they tend to suffer of the state explosion problem when adopted for verifying large designs. In this context, a functional ATPG is presented, that relies on the joint use of pseudo-deterministic simulation and constraint logic programming (CLP) (Jaffar & Maher, 1994), to generate high-quality test sequences for solving complex problems. Thus, the advantages of both simulation-based and static-based verification techniques are preserved, while their respective drawbacks are limited. In particular, CLP, a form of constraint programming in which logic programming is extended to include concepts from constraint satisfaction, is well-suited to be jointly used with simulation. In fact, information learned during design exploration by simulation can be effectively exploited for guiding the search of a CLP solver towards design under verification (DUV) areas not covered yet.
Therefore, this work is focused on the use of CLP for addressing corner cases during functional test pattern generation. In particular, a CLP-based fault-oriented ATPG engine is proposed to be adopted, after simulation, learning and random-walk/backjumping, as the last step of the incremental test generation flow showed in Figure 1.
According to such a flow, the ATPG framework is composed of three functional ATPG engines working on three different models of the same DUV: the hardware description language (HDL) model of the DUV, the set of concurrent EFSMs extracted from the HDL description, and the set of logic constraints modelling the EFSMs. The EFSM paradigm has been selected since it allows a compact representation of the DUV state space (Lee & Yannakakis, 1992) that limits the state explosion problem typical of more traditional FSMs.
In the proposed framework, the first engine is random-based, the second is transition-oriented, while the last is fault-oriented. This approach quickly covers the greater part of DUV faults, typically easy-to-detect faults. Then, the transition-oriented engine and fault-

Source: Micro Electronic and Mechanical Systems, Book edited by: Kenichi Takahata,
ISBN 978-953-307-027-8, pp. 572, December 2009, INTECH, Croatia, downloaded from SCIYO.COM

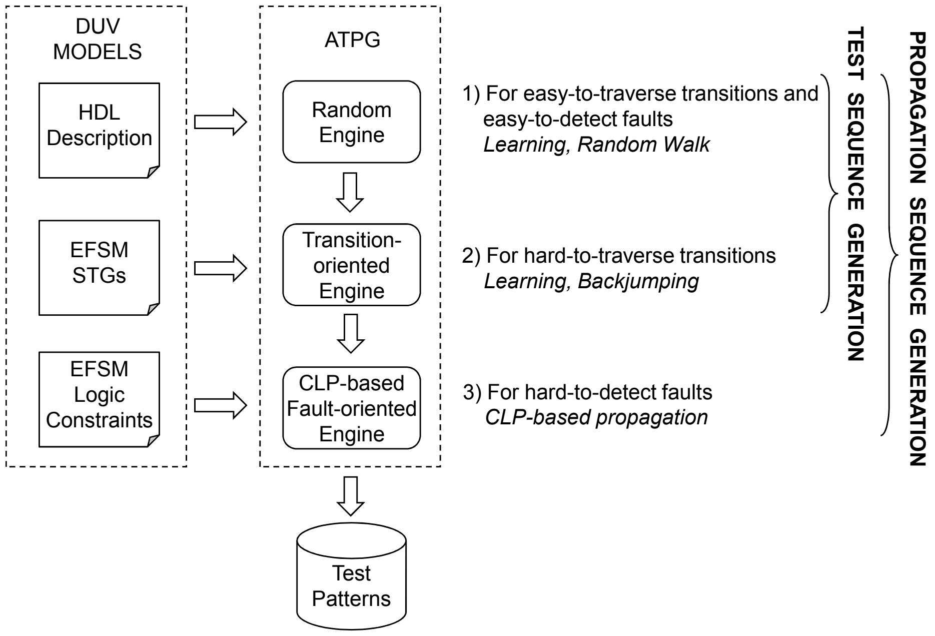
DUV MODELS
HDL Description
EFSM STGs
EFSM Logic Constraints
ATPG
Random Engine
Transition-oriented Engine
CLP-based Fault-oriented Engine
Test Patterns
1) For easy-to-traverse transitions and easy-to-detect faults
Learning, Random Walk
2) For hard-to-traverse transitions
Learning, Backjumping
3) For hard-to-detect faults
CLP-based propagation
TEST SEQUENCE GENERATION
PROPAGATION SEQUENCE GENERATION

Fig. 1. The incremental test generation flow.

oriented engine permit to deterministically generate test patterns for the remaining uncovered hard-to-detect faults. Therefore, the test generation is guided by means of transition coverage (Li & Wong, 2002) and fault coverage (Abramovici, 1993). In particular, 100% transition coverage is desired as a necessary condition for fault detection, while the bit coverage (Ferrandi et al., 1998a) functional fault model is used to evaluate the effectiveness of the generated test patterns by measuring the related fault coverage.

This Chapter is organized as follow. Section 2 reports the state of art of CLP techniques applied for test generation. Section 3 describes the EFSM as the adopted computational and the strategies defined for generating a particular type of EFSM suited for test generation. Section 4 presents the functional ATPG engine that relies on learning, random-walk/backjumping and constraint logic programming to deterministically generate test vectors for traversing all transition of the EFSM and activate the fault injected into the DUV. Section 5 describes the fault-oriented ATPG engine that exploits CLP techniques to propagate hard-to-detect faults activated but not observed to the outputs. Experimental results are presented in Section 6, and conclusions are drawn in Section 7.

2. Background

Constraint programming (Wallace, 1997) is a paradigm that is tailored to search problems. The main application areas are those of planning, scheduling, timetabling, routing, placement, investment, configuration, design and insurance. Constraint programming incorporates techniques from mathematics, artificial intelligence and operations research, and it offers significant advantages in these areas since it supports fast program development, economic program maintenance, and efficient runtime performance.

Constraint logic programming (CLP) combines logic, which is used to specify a set of possibilities explored via a very simple in-built search method, with constraints, which are used to minimize the search by eliminating impossible alternatives in advance.
The programmer can state the factors which must be taken into account in any solution (the constraints), state the possibilities (the logic program), and use the system to combine reasoning and search. The constraints are used to restrict and guide search. The whole field of software research and development has aims to optimize the task of specifying, writing and maintaining correct functioning programs.
CLP-based ATPGs have been already proposed in the literature (Vemuri & Kalyanaraman, 1995; Pauli et al., 2000; Xin and Harris, 2002; Ferrandi et al., 2002b), however, existent approaches differ in several aspects from the ones presented in this work. In (Pauli et al., 2000), CLP is used to generate test sequences according to a path coverage-based criterion. However, this approach is oriented only to the control part of circuits. Control flow paths are also the target of the approach presented in (Vemuri & Kalyanaraman, 1995; Ferrandi et al., 2002b). In (Vemuri and Kalyanaraman, 1995), constraints are derived from a preprocessing of a VHDL description to enumerate all the target paths. On the contrary, in (Ferrandi et al., 2002b), constraints are generated by enumerating paths of concurrent FSMs describing the DUV. However, path enumeration is a very hard and time-consuming task, since paths of sequential circuits are generally infinite. In (Xin and Harris, 2002), the authors propose to use CLP for generating test sequences targeting synchronization/timing faults in hardware/software models described as a network of co-design FSMs. This work identifies sequences to trigger synchronization faults but the observability of the fault effect is not considered.
Finally, the previous approaches propose neither strategies for completely modelling a design by means of CLP (just some paths are modelled), nor approaches for avoiding the risk of state explosion when a CLP solver is asked to generate a test sequence to activate the target path.
The CLP solver adopted and integrated into the proposed ATPG is ECLiPSe (ECLiPSe Common Logic Programming System) (Wallace & Veron, 1994; Apt & Wallace, 2007). ECLiPSe is a Prolog based system whose aim is to serve as a platform for integrating various Logic Programming extensions, in particular Constraint Logic Programming. The kernel of ECLiPSe is an efficient implementation of standard (Edinburgh-like) Prolog as described in basic Prolog texts. It is built around an incremental compiler which compiles the ECLiPSe source into WAM-like code, and an emulator of this abstract code. ECLiPSe is now an open-source project, with the support of Cisco Systems.

3. EFSM manipulation to exploit CLP based techniques

The EFSM model is a generalization of the classical FSM model that provides a compact representation of local data variables and preserve properties of the traditional state machine models.
In the current work, a digital system is represented as a set of concurrent EFSMs, one for each process of the DUV. In this way, according to the following Definition 1, the main characteristics of state-oriented, activity-oriented and structure-oriented models are captured (Gajski et al., 1997). In fact, the EFSM is composed of states and transitions, thus it is state-oriented, but each transition is extended with hardware description language (HDL) instructions that act on the DUV registers. In this sense, each transition represents a set of

activities on data, thus, the EFSM is a data-oriented model too. Finally, concurrency is intended as the possibility that each EFSM of the same DUV changes its state concurrently to the other EFSMs to reflect the concurrent execution of the corresponding processes. Data communication between concurrent EFSMs is guaranteed by the presence of common signals. In this way, structured models can be represented.

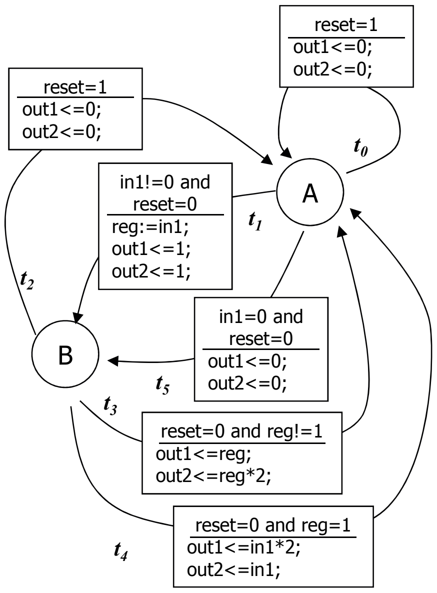

Fig. 2. State transition graph of an EFSM.

Definition 1 *An EFSM is defined as a 5-tuple M=<S,I,O,D,T> where: S is a set of states, I is a set of input symbols, O is a set of output symbols, D is a n-dimensional linear space* $D_1\times\ldots\times D_n$*, T is a transition relation such that* $T{:}S\times D\times I\rightarrow S\times D\times O$*. A generic point in D is described by a n-tuple* $x=(x_1,\ldots,x_n)$ *and models the values of the registers of the DUV. A pair* $<s,x>\in S\times D$ *is called configuration of M.*
An operation on *M* is defined in this way: if *M* is in a configuration *<s,x>* and it receives an input $i\in I$, it moves to the configuration *<t,y>* iff $((s,x,i),(t,y,o))\in T$ for $o\in O$.
The EFSM differs from the classical FSM, since each transition does not present only a label in the classical form *(i)/(o),* but it takes care of the register values too. Transitions are labelled with an enabling function *e* and an update function *u* defined as follows.
Definition 2 *Given an EFSM M=<S,I,O,D,T>,* $s\in S$, $t\in T$, $i\in I$, $o\in O$ *and the sets* $X=\{x\,|\,((s,x,i),(t,y,o))\in T \text{ for } y\in D\}$ *and* $Y=\{y\,|\,((s,x,i),(t,y,o))\in T \text{ for } x\in X\}$*, the enabling and update functions are defined respectively as:*

$$e\,(x,i)=\begin{cases}1 & \text{if } x\in X;\\ 0 & \text{otherwise.}\end{cases} \tag{1}$$

$$u(x,i)=\begin{cases}(y,o) & e(x,i)=1 \text{ and } ((s,x,i),(t,y,o))\in T;\\ \text{undef.} & \text{otherwise.}\end{cases}$$

An update function *u(x,i)* can be applied to a configuration $<s_1,x>$ if there is a transition $t:s_1 \rightarrow s_2$, labelled *e/u*, such that *e(x,i)*=1. In this case we say that *t* can be *traversed* by applying the input *i*. Figure 2 shows the state transition graph of a simple EFSM.

Many EFSMs can be generated starting from the same HDL description of a DUV. However, despite from their functional equivalence, they can be more or less easy to be traversed. Easiness of traversal is a mandatory feature for a computational model used in CLP-based test pattern generation, and it is a desirable condition to activate and propagate faults.

Stabilization of EFSMs improves the easiness of traversal (Lee & Yannakakis, 1992), but it can lead to the state space explosion. Thus, in (Di Guglielmo et al., 2006a), a set of theoretically-based automatic transformations has been proposed to generate a particular kind of semi-stabilized EFSM (S^2EFSM). This particular kind of EFSM allows the ATPG to easily explore the state space of the corresponding DUV reducing the risk of state explosion. The S^2EFSM presents the following characteristics:

- It is functionally and timing equivalent to the HDL description from which it is extracted, i.e., given an input sequence, the HDL description and the corresponding S^2EFSM provide the same result at the same time.
- The update functions contain only assignment statements. This implies that all the control information, needed by a CLP-based ATPG to traverse the DUV state space, resides in the enabling functions of the S^2EFSM.
- The S^2EFSM is partially stabilized to reduce the state explosion problem that may arise when stabilization is performed to remove inconsistent transitions. Only transitions not leading to state explosion are stabilized.

The S^2EFSM is referred simply as EFSM in the following.

4. CLP-based technique for generating test sequences

The ATPG engine proposed in this Section relies on random and pseudo-deterministic simulation and CLP techniques to generate test sequences for traversing the system represented as a collection of EFSM. A two step approach, implementing respectively the random engine and the transition-oriented engine, is depicted in Figure 1.

First the DUV state space is explored by performing a simulation-based random-walk (Section 4.3). This allows to quickly fire easy-to-traverse (ETT) transitions and, consequently, to quickly cover easy-to-detect (ETD) faults. However, the majority of hard-to-traverse (HTT) transitions remain, generally, uncovered.

Thus, the backjumping-based strategy is applied to cover the remaining HTT transitions by mean of transition-oriented ATPG (Section 4.4). Backjumping, also known as non-chronological backtracking, is a special kind of backtracking strategy which rollbacks from an unsuccessful situation directly to the cause of the failure. Thus, the engine deterministically backjumps to the source of failure when a transition, whose guard depends on previously set registers, cannot be traversed. Next it modifies the EFSM configuration to satisfy the condition on registers and successfully comes back to the target state to activate the transition.

The transition-oriented engine generally allows achieving 100% transition coverage. However, 100% transition coverage does not guarantee to explore all DUV corner cases, thus some hard-to-detect (HTD) faults can escape detection preventing the achievement of 100% fault coverage. Therefore, the CLP-based fault-oriented engine, as described in Section 5, is finally applied to focus on the remaining HTD faults.

4.1 ATPG architecture

The EFSM model is specially suited to be used with CLP-based ATPGs that generate test sequences by deterministically activating the enabling functions of the transitions. According to this observation, in this section the functional ATPG framework depicted in Figure 3 is described. The framework is composed of two main modules: the DUV-dependent component generator (DCG) and the run-time engine (RTE).

Given an HDL functional description of the DUV, the DCG modules generates the state-transition-graph (STG) representations of the corresponding EFSMs (EFSM STG Generator), the faulty description of the DUV and the related fault list (Fault Injector), and the file containing the constraints involved in the EFSM enabling and updating functions (Constraint Generator). The constraint generation for transition-oriented CLP-based ATPG is described in Section 4.2

The RTE module is composed of the EFSM navigator, the CLP Interface, and the Simulation Engine. The RTE navigates the STG representation of the EFSM to generate test sequences. An external CLP solver (ECLiPSe) is used to generate values for primary inputs of the DUV which allow firing the enabling functions of transitions that the EFSM navigator wants to traverse. Then, the generated test sequences are provided to the Simulation Engine, and the behaviour of the fault-free and faulty DUVs is compared. Test sequences that highlight discrepancies between the primary outputs of the fault-free and faulty DUVs constitute the final test patterns.

4.2 Constraint generation for transition-oriented CLP-based ATPG

Starting from the in-memory representation of the DUV, the Constraint Generator automatically creates the CLP-constraint file to allow the RTE module to evaluate the enabling functions when the EFSM is navigated. For example, Figure 4 shows the constraint representation for ECLiPSe solver related to transitions of the EFSM of Figure 2.

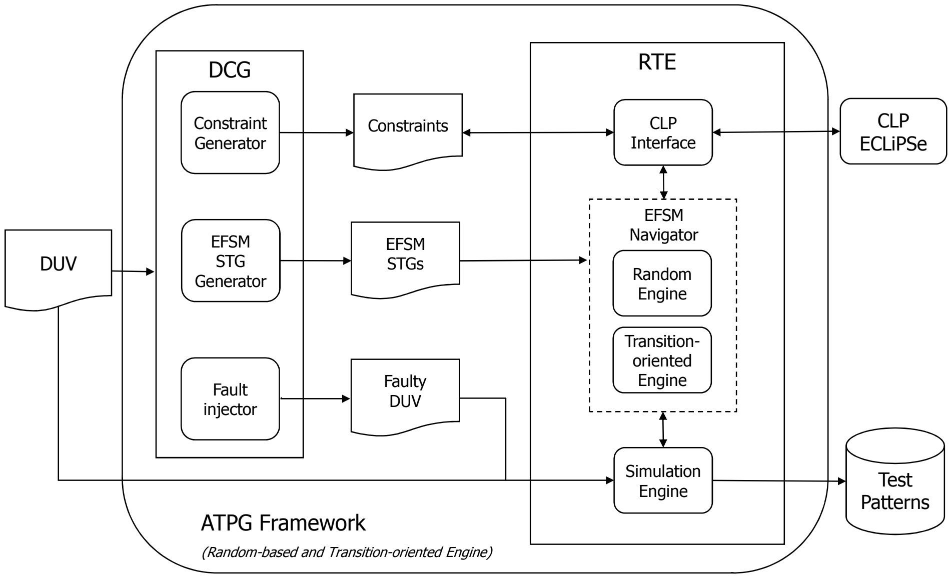

Fig. 3. The ATPG framework.

During the test pattern generation such constraints are exploited. In particular, each test vector is randomly initialized. Then, it is modified accordingly with the values provided by the CLP solver. In particular, if the enabling functions of the currently evaluated transition are satisfied, the part of the test vector related to the primary inputs involved in the enabling function is modified accordingly to the values returned by the CLP solver.
Moreover, the EFSM Navigator accesses to structures pointing to DUV internal registers. The actual values of the registers are used to instantiate the constraints in the current system status.

```
t0 :- reset::0..1,
    (reset#=1),
    indomain(reset,random).
t1 :- lb is -(2^32), rb is (2^32)-1,
    in1::lb..rb, reset::0..1,
    (in1#\=0)and(reset#=0),
    indomain(in1,random), indomain(reset,random).
t2 :- reset::0..1,
    (reset#=1),
    indomain(reset,random).
t3 :- lb is -(2^32), rb is (2^32)-1,
    reg::lb..rb, reset::0..1,
    (reg#\=1)and(reset#=0),
    indomain(in1,random), indomain(reset,random).
t4 :- lb is -(2^32), rb is (2^32)-1,
    reg::lb..rb, reset::0..1,
    (reg#=1)and(reset#=0),
    indomain(in1,random), indomain(reset,random).
t5 :- lb is -(2^32), rb is (2^32)-1,
    in1::lb..rb, reset::0..1,
    (in1#=0)and(reset#=0),
    indomain(in1, random), indomain(reset,random).
```

Fig. 4. Enabling function representations for ECLiPSe solver.

4.3 Random-walk

During the random-walk phase, the ATPG randomly walks across the transitions of the EFSMs representing the DUV. Thus, ETT transitions are very likely traversed.
Starting from a reset condition, the ATPG randomly selects a transition from each EFSM according to a scheduling policy. The EFSM-scheduling policy aims at maximizing the ATPG capability of exploring the whole state space. Then, it tries to satisfy the enabling function of each selected transition by exploiting the CLP solver invoked by providing it with the corresponding constraints. When it succeeds, the values for the primary inputs, provided by the solver, are used to generate a test vector. Finally, a simulation cycle is performed, by using the generated test vector, to update the internal registers of the DUV and to move to the destination state. Then, another transition is selected, and the cycle repeats.
Each time a test vector is generated, the traversed transition is labelled with the test sequence number and the test vector number. A list of pairs of parametric length is saved

for each transition. In this way, the backjumping mode can exploits such lists to quickly recover the prefix of a test sequence which allows the ATPG to move from the reset state to an already visited target state.

4.4 Backjumping

The ATPG automatically changes to the backjumping mode when the computation time assigned to the random-walk expires, or no coverage improvement is provided for long time. Thus, the transition-oriented ATPG works as represented in Figure 5. Let us assume t is a not fired transition, out-going from state S_t already visited during the random-walk phase. Let us also assume that the unsatisfiability of the enabling function of t depends on clauses involving a single register *reg*. If the unsatisfiability of t depends on more than one register, the backjumping procedure is repeated for each of them. Then extract an already visited transition t_u from the set of transitions T_u^{reg} whose update function updates *reg*. Load the test sequence, previously generated during random-walk mode, to move from the reset state to S_{tu} (source state of t_u). Thus, the ATPG backjumps from S_t to S_{tu}. Use the Dijkstra's shortest path search algorithm to provide a path π from S_{tu} to S_t starting with t_u. Satisfy the enabling function of t_u according to the constraints derived from the enabling function of t as follows. Let us suppose that e_{tu} is the enabling function of t_u and $e_t | reg_{tu}$ is the part of the enabling function of t which involves the clauses depending on *reg*, where each occurrence of *reg* has been substituted with the right-side expression of the assignment that updates *reg* in the update function of t_u. Invoking the CLP solver to satisfy the constraint $e_{tu} \wedge e_t | reg_{tu}$ allows us to obtain a test vector which satisfies e_{tu} and sets the value of *reg* in such a way that when simulation reaches transition t, following π, its enabling function will be correctly fired. The last observation may be false if there is a transition $t'_u \neq t_u \neq t$ in π, such that t'_u updates *reg* after t_u did. In this case, the ATPG moves the problem from t_u to t'_u requiring a solution for $e_{t'u} \wedge e_t | \ reg_{t'u}$. Finally satisfy the enabling function of transitions included in π by iteratively applying the constraint solver to generate the corresponding test vectors. The test sequence obtained by joining s, to move from the reset state to S_{tu}, and the test vectors generated to traverse π allows to fire t.

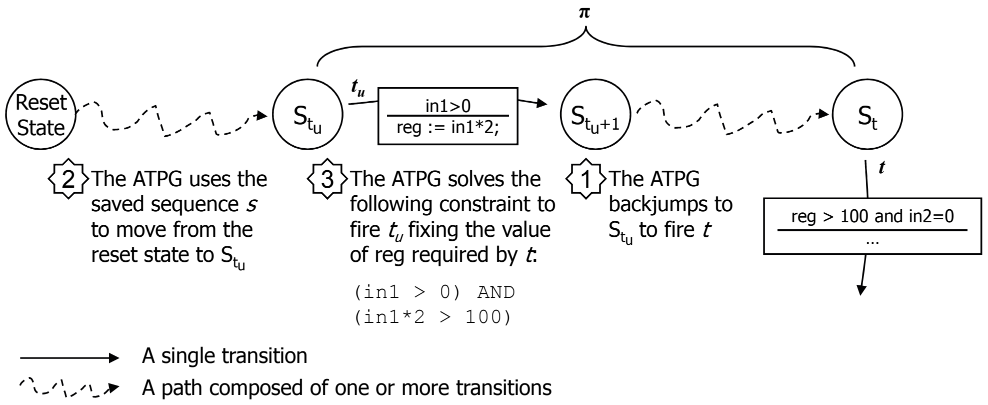
π
Reset State
S_tu
t_u
in1>0
reg := in1*2;
S_tu+1
S_t
t
reg > 100 and in2=0
...
2 The ATPG uses the saved sequence s to move from the reset state to S_tu
3 The ATPG solves the following constraint to fire t_u fixing the value of reg required by t:
(in1 > 0) AND
(in1*2 > 100)
1 The ATPG backjumps to S_tu to fire t
A single transition
A path composed of one or more transitions

Fig. 5. The backjumping strategy.

5. CLP-based technique for generating propagation sequences

In this section the CLP solver is used to deterministically search for sequences that propagate the faults observed, but not detected by means of the previously presented random and transition-oriented engines. In particular, Section 5.1 presents the technique defined to model the EFSM in CLP, Section 5.2 describes the implemented CLP-based fault-oriented ATPG engine, and Section 5.3 presents some optimization strategies defined to deal with the complexity typical of static techniques like CLP.

5.1 EFSM modelling by CLP

At first, the concept of time steps is introduced, required to model the EFSM evolution through time via CLP. Then, techniques are presented to model logical variables and constraints describing the enabling functions and update functions of the EFSM.

Hardware description languages easily allow modelling the DUV time evolution by means of processes, implicit or explicit wait statements, sensitivity lists and events. On the contrary, CLP does not provide an explicit mechanism to model the time. To overcome this limitation, a logical variable *N* is introduced to represents the total number of time frames on which the EFSM can evolve. The domain of *N* is *[1,Max]*, where *Max* is defined according to the sequential depth of the DUV. Then, the CLP variables, used to model the EFSM behaviour, are defined as arrays of size *N*. Thus, for example, let us consider a variable *V* defined in the HDL description of the DUV. When CLP is adopted, an array *V[]* is used to model the evolution in time of variable *V*. In this context, a CLP constraint of the form *V[T]#=0* indicates that at time *T* the variable *V* of the DUV has value *0*.

Three kinds of arrays of logical variables must be defined to describe, respectively, states, transitions, and registers of an EFSM.

- Each state of the EFSM is modelled by an array of boolean variables of size *N*. When the EFSM is in the state *S* at time *T*, the T^{th} element of the array *S[]*, corresponding to the state *S*, is assigned to *true*. For example, two arrays, *A[]* and *B[]*, are required to model states *A* and *B* of the EFSM of Figure 2. At every time step *T*, either *A[T]* or *B[T]* is *true*, thus indicating the current state of the EFSM.
- Each transition of the EFSM is associated to an array of boolean variables of size *N*. If a transition is fired at time *T*, the T^{th} element of the array corresponding to the transition is assigned to *true*.
- Registers are modelled as array of size *N* respecting their original data type.

The CLP code of Figure 6 exemplifies the constraints required to model logical variables for states, transitions, and registers of the EFSM of Figure 2. Moreover, in Figure 6, the predicate *bool(X,N)* defines the boolean data type used to model states and transitions. It means that the logical variable *X* is an array of size *N* (for modelling of time), whose elements can assume the values included in the list *Xlist*, i.e., *0* or *1*. In a similar way, the predicate *int32(X,N)* defines an arrays of *N* 32-bit integers used as data type to deal with primary inputs, primary outputs, and registers.

The functional behaviour of the EFSM is represented by means of enabling functions and update functions labelling the transitions between states. Thus finally, a way for modelling such functions and their relation with states and transitions is proposed. In particular, two kinds of constraints have been defined to model the current state of the EFSM, and the relation between the enabling function and the corresponding update function.

```
% data types
bool(X,N) :- dim(X,[N]),term_variables(X, Xlist), Xlist::[0,1].
int32(X,N) :- dim(X,[N]), X[1..N]:: -2147483648..2147483647.
% states
bool(A,N), bool(B,N),
% transition
bool(T0,N), bool(T1,N), bool(T2,N), bool(T3,N), bool(T4,N),bool(T5,N),
% primary inputs, primary outputs, and registers
int32(IN1,N), int32(REG,N), int32(OUT1,N), int32(OUT2,N).
```

Fig. 6. Constraints for modelling state, transition and register variables.

5.1.1 Current state modelling

Two constraints must be defined for each array of state variables to specify the current state of the EFSM. The first constraint specifies that, at each time step *T*, the *T* th element of an array *S[]*, modelling a state *S* of the DUV, is *true*, if and only the *T*th element of one of the transition arrays corresponding to the transitions in-going in *S* is *true*. The second constraint specifies the dual situation, i.e., if the *T*th element of the transition array is true at time *T*, then the *T*th element of the array associated to the destination state of the corresponding transition must be true at time step *T+1* (*NEXT_T*). For example, let us consider the EFSM of Figure 2. The constraints in Figure 7 must be defined for specifying that the current state of the EFSM at time step *T+1* is *A* , if and only if one of the transitions in-going in *A* has been fired at time *T*.

```
A[NEXT_T] #= T0[T] xor T2[T] xor T3[T] xor T4[T],
T0[T] xor T2[T] xor T3[T] xor T4[T] => A[NEXT_T].
```

Fig. 7. Constraints for modelling next-state relation.

Finally, a further constraint is introduced to explicitly force the system to be in a single state and transition at each time step. Thus, the *T*th element of arrays corresponding to states of the EFSM are put in *xor* each other as shown in Figure 8.

```
A[T] xor B[T],
T0[T] xor T1[T] xor T2[T] xor T3[T] xor T4[T] xor T5[T].
```

Fig. 8. Constraint for modelling transition and state mutual exclusion.

In fact, at a particular time step, only one transition of the EFSM can be traversed and, obviously, the EFSM can have only one state active. Designers implicitly include such a constraint, when they model the DUV by means of an HDL. However, the explicit presence of such a constraint, when the EFSM is provided to the CLP solver, allows the solver to immediately prune the solution space by ignoring configurations where more than one state variable is concurrently true, thus drastically reducing the number of backtracking steps.

5.1.2 Enabling and update function modelling

Firing a transition at time *T* implies that its enabling function is satisfied at time *T*, its update function is executed at time *T*, and the state of the EFSM at time *T* is the source of the transition. Thus, for example, if *Ti* is a transition out-going from state *S*, whose enabling function and update function are modelled, respectively, by the predicates *EF* and *UF* (described later), the constraints in Figure 9 are used to model *Ti*.

```
EF[T] and S[T] => Ti[T],
Ti[T] => EF[T] and S[T],
Ti[T] and (EF[T] and S[T]) => UF[T].
```

Fig. 9. Constraint for correlating the enabling and update functions to transitions.

The first two constraints represent a double implication for imposing that the transition variable *Ti[T]* is *true* (i.e., the transition *Ti* is fired at time *T*) if and only if the predicate of the corresponding enabling function *EF[T]* and the variable *S[T]*, associated to the state from which *Ti* is out-going, are *true*. On the contrary, the predicate of the update function *UF[T]* does not require a double implication, because it is possible that *UF[T]* is *true* even if *EF[T]* is *false*. However, in this case the transition is not fired and the update function is not executed. The predicate *EF[T]* is directly derived from the condition involved in the corresponding enabling function. Its modelling requires only a syntactical conversion from the syntax of the HDL used to model the DUV towards the syntax accepted by the CLP solver.
On the contrary, modelling the predicate *UF[T]* associated to an update function requires more attention. In particular, an update function involves assignments to registers and primary outputs. Let us use an example to show how to model such a kind of statement. Consider, for example, the statement *SIG := SIG + IN*, where *SIG* is an internal signal and *IN* is a primary input. The corresponding CLP constraint is *SIG[Next_T]#=SIG[T]+IN[T]*.
However, registers and primary outputs, that do not require to be updated, are not assigned in the update function when a design is modelled by using an HDL. Indeed, they implicitly preserve their previous value. Unfortunately, the CLP solver assigns random values to variables that are not explicitly assigned. Thus, when an update function is modelled by means of constraints, it has to ensured that a constraint is explicitly added to preserve the value of signals, registers and primary outputs that do not require to be updated.
According to the previous rules, for example, the transition t_4 in Figure 2 is modelled as depicted in Figure 10.

```
% Enabling function
( (REG[T] #=1) and B[T] )  => T4[T],
T4[T] => ( (REG[T] #= 1 ) and B[T] ),
% Update function
(T4[T] and ( (REG[T] #= 1) and B[T] ) =>( ( REG[Next_T] #= REG[T]) and
(OUT1[Next_T] #= IN1[T]*2) and (OUT2[Next_T] #= IN1[T])).
```

Fig. 10. Constraint for representing transition t_4 of the EFMS in Figure 2.

5.2 Fault-oriented CLP-based engine

The transition-oriented engine described in Section 4 pseudo-deterministically generates sequences for firing HTT transitions on EFSMs. In this way, the majority of faults are detected as a consequence of transition traversal, but some HTD faults can remain uncovered. On the contrary, the CLP-based fault-oriented engine exhaustively searches for test sequences targeting specific faults. It exploits the CLP-solver to explore the CLP-based representation of the DUV extracted from the EFSM model. The exhaustiveness, guaranteed by CLP, is paid in terms of execution time, but such an engine is applied to a small number of faults: those not detected neither by the random-based engine nor by the transition-oriented one.

Let us consider a fault *f* that has not been detected yet by these engines. This may depends on two different reasons:

1. the ATPG has been unable to find an activation sequence, i.e., in the case of the bit coverage fault model, a sequence that causes the bit (or the condition) affected by *f* to be set with the opposite value with respect to the one induced by *f* ;
2. the ATPG activated *f*, but it has been unable to find a propagation sequence, i.e., a sequence that propagates the effect of *f* to the primary outputs of the DUV.

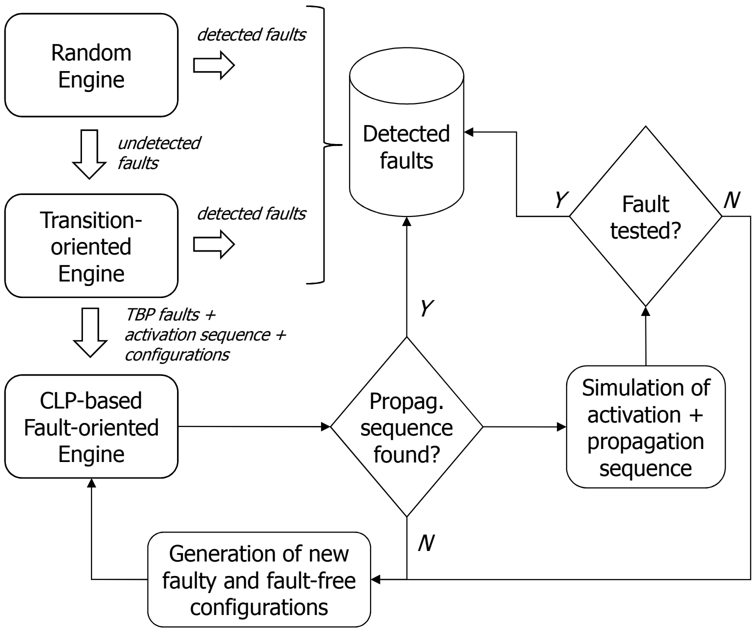
Random Engine
detected faults
undetected faults
Transition-oriented Engine
detected faults
TBP faults + activation sequence + configurations
CLP-based Fault-oriented Engine
Detected faults
Y
Fault tested?
Y
N
Propag. sequence found?
Simulation of activation + propagation sequence
N
Generation of new faulty and fault-free configurations

Fig. 11. The role of the CLP-based fault-oriented engine.

To distinguish between the previous alternatives, the ATPG observes the effect of each fault on both primary outputs and internal registers, during the simulation of test sequences generated by the random-based and transition-based engines. From this observation, the following well-known definitions derive.

Definition 3 *A fault f is said to be observable on primary outputs, i.e., detectable, if there exists a test sequence s such that, by concurrently applying s to the faulty and the fault-free DUVs, the value* *of at least one primary output in the fault-free DUV differs from the value of the corresponding primary output in the faulty DUV, at least once in time.*

Definition 4 *A fault f is said to be observable on a register if there exists a test sequence s such that, by concurrently applying s to the faulty and the fault-free DUVs, the value of at least one register in the fault-free DUV differs from the value of the corresponding register in the faulty DUV, at least once in time.*

According to the previous definitions, if the fault is observed on primary outputs, it is marked as detected and the corresponding test sequence is saved. Otherwise, if the fault is observed only on registers, the fault is marked as to be propagated (TBP). Finally, if the fault is observable neither on primary outputs nor on registers, this is due to the difficulty of finding an activation sequence. Thus, the fault is marked as to be activated (TBA).

The current work addresses TBP faults (see Figure 11); future works will address the problem of TBA faults. In the following, Section 5.2.1 describes how submit TBP problem to the CLP-solver, Section 5.2.2 introduces searching functions to generate test sequences and finally Section 5.2.3 proposes how to managing problem complexity.

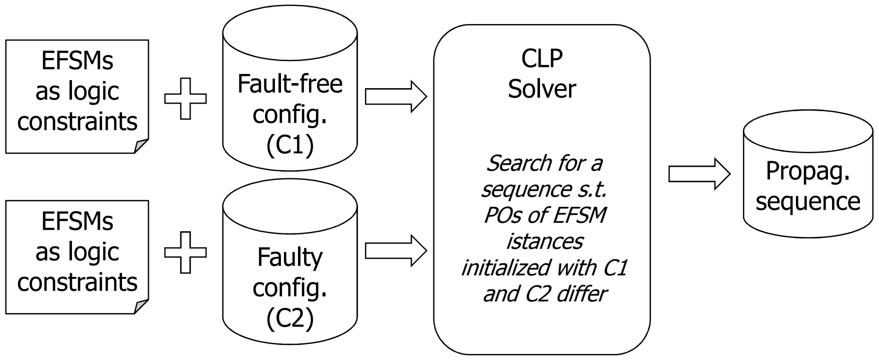
EFSMs as logic constraints
Fault-free config. (C1)
EFSMs as logic constraints
Faulty config. (C2)
CLP Solver
Search for a sequence s.t. POs of EFSM istances initialized with C1 and C2 differ
Propag. sequence

Fig. 12. Use of the CLP solver for finding propagation sequences.

5.2.1 Propagation sequence generation

The propagation sequence for a TBP fault is generated by providing the ATPG engine with two instances of the CLP-based representation of the DUV. This representation consists of the CLP model of the generated EFSM. The instances are initialized with the EFSM configurations (the faulty and the fault-free ones) that allow the random or the transition-based engine to observe the fault on at least one register.

Note that the two DUV instances, provided to the CLP solver, are exactly equal, but their registers are initialized with different values (the faulty and the fault-free ones). This is the only reason such instances should behave in different ways. In the following, the terms faulty and fault-free are used to distinguish the DUV instances initialized, respectively, with the faulty and the fault-free configuration. Such configurations consist of the values of registers (included the value of the state register) in the faulty and fault-free DUVs at the moment the fault has been observed. As an example, consider the constraints in Figure 13. They state that, at time *T*=1, *REG* has the same value (i.e., *5941*) in both the faulty and fault-free DUV instances, but the two EFSMs are in different states (i.e., the fault-free DUV is in state A, while the faulty DUV is in state B).

```
% Fault free
REG[1] #= 5941, A[1] #= 1, B[1] #= 0,
% Faulty
REG_F[1] #= 5941, A_F[1] #= 0, B_F[1] #= 1.
```

Fig. 13. How to specify faulty and fault-free configuration for EFSM models.

After the set-up of the faulty and fault-free configurations, the CLP-solver is asked to find a sequence that, starting from such configurations, propagates the effect of the fault towards the primary outputs (see Figure 12). Therefore, a constraint is defined that forces the arrays of primary outputs to be different at least in a position as shown in Figure 14.

```
~((OUT1 = OUT1_F) and (OUT2 = OUT2_F)).
```

Fig. 14. Asking for a propagation sequence.

If the solver finds a solution, it consists of a propagation sequence that can be appended to the activation sequence, previously generated by the random-based or transition-based engine to observe the target fault on the internal registers. The so obtained sequence is very likely a test sequence for the target fault, but this must be definitely proved via simulation (see Figure 11). In fact, the propagation sequence is generated by initializing a fault-free instance of the DUV with a faulty configuration, which is not the same as using a real faulty DUV instance directly affected by the fault. However, experimental results showed that in very few cases the propagation sequence generated by the CLP solver according to the proposed strategy, fails to propagate the corresponding fault when it is simulated on the faulty DUV.

5.2.2 Definition of search procedures

Constraints described in the previous subsection are used to set up the problem of finding a propagation sequence as a CLP problem. Then, some search procedures, that exploit search strategies and heuristics, must be defined to force the solver to provide the solution (i.e., the set of values to be assigned to the logic variables for satisfying all the problem's constraints), when it exists. Therefore, a predicate that exploits the *search/6* function of the ECLiPSe's IC library have been defined (Figure 15).

```
search_func(A), search_func(B), search_func(IN1),..
search_func(L):- search(L,0,input_order,indomain,complete,[]).
```

Fig. 15. A search procedure is defined for each variable of the DUV.

Such a function, whose signature is *search(Vars, Arg, Select, Choice, Method, Options)*, is a generic search routine which implements different partial search methods. It instantiates the variables *Vars* by performing a search based on the parameters provided by the user. In our case the search method performs a complete search routine which explores all alternative choices for each variable. The choice method *indomain* tries to find a solution by analyzing the variable values in increasing order, from the lower value in the variable range to the upper value. The predicate *search_func* is called on each variable of the DUV. In this way, if a solution exists, the solver provides a value for each variable for each time step, thus generating the required propagation sequence.

5.2.3 Managing the CLP complexity

Tools that exhaustively search for a solution of NP-hard problems frequently run out of resources when the state space to be analyzed is too large. The same happens for the CLP solver, when it is asked to find a propagation sequence on large sequential designs. To limit such a problem, heuristics is generally used for pruning the state space. However, this may prevent the solver from finding a solution (even if it exists), if the pruning is too restrictive. Thus, choosing a good heuristics is a very challenging task.

In this context, three strategies for managing the complexity of the CLP solver exploited by the fault-oriented engine have been defined. Two of them have been already presented in previous sections; however, for convenience of the reader, we summarize them here:

- The T^{th} element of arrays corresponding to states (and transitions) of the DUV are put in *xor* each other, to avoid that the solver wastes time to analyze configurations where more than one state variable is concurrently true. This drastically decreases the number of backtracking steps, especially for designs with many states and many registers.
- A constraint on DUV registers is defined to assure that at least one register of the faulty DUV differs from the corresponding register of the fault-free DUV at each time step T>1. On the contrary, the search is immediately stopped, and no solution is reported. Such a constraint avoids situations where the solver spends uselessly efforts, as it cannot lead to the observability on primary outputs if starting from different configurations, the faulty and fault-free DUVs evolve in the same configuration.

A further strategy for managing the complexity of the CLP solver, that can be jointly used with the previous ones, consists of asking the solver to find a solution (i.e., a propagation sequence) starting with a small state space, that is incrementally enlarged until a solution is found (or execution time expires). Thus, the state space to be analyzed by the solver is restricted by limiting the range of the DUV PIs, similarly to what has been proposed for limiting the size of binary decision diagrams in the test generation strategy proposed in (Ferrandi et al., 2002a). At the beginning, the ATPG statically fixes the values of all bits, but two, for each PI. In this way, only two bits can be changed by the CLP solver during the search, independently from the PIs range declared on the HDL description of the DUV. Then, the solver is asked to find a solution. If it fails, the ATPG opportunely increases the number of free bits of PIs. In particular, the ATPG engine searches for the constraints that induce the failure, and it frees the bits of the PIs involved in such constraints. Then, a new search session is launched. Such a process is iterated until a solution is found or execution time expires.

5.3 Optimizations exploiting EFSM model features

This section describes some heuristics to reduce the complexity problem for the solver. Two different approaches have been defined to improve the CLP-based ATPG engine by reducing the problem complexity.

The first approach is based on the EFSM manipulation as described in Section 5.3.1. A new EFSM is generated for the solver that has to deal with a reduced version. This technique exploits two phases. In the first phase, all the transitions that delete the observability property of the given configuration are removed. Then, all that parts of the EFSM that cannot be traversed are eliminated. This strategy removes all that constraints that are useless and could also avoid the propagation of observability on the primary outputs. In fact the new generated EFSMs are pruned by all that transitions are not needed to model any possible behaviour that can lead to fault observability. Note that the complexity of the proposed algorithm is linear with the number of transitions and that, above all, the number of transition is limited and is not affected by the state explosion problem with the EFSM model.

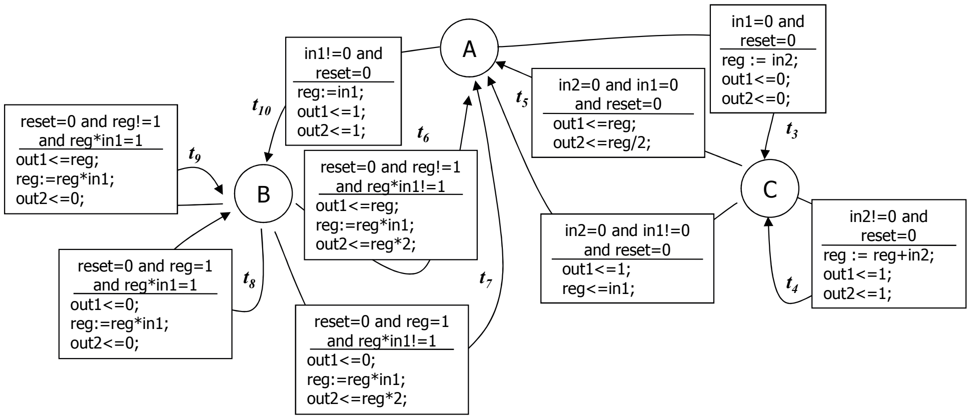
A
B
C
in1=0 and reset=0
reg := in2; out1<=0; out2<=0;
t_3
in1!=0 and reset=0
reg:=in1; out1<=1; out2<=1;
t_{10}
in2=0 and in1=0 and reset=0
out1<=reg; out2<=reg/2;
t_5
reset=0 and reg!=1 and reg*in1=1
out1<=reg; reg:=reg*in1; out2<=0;
t_9
t_6
reset=0 and reg!=1 and reg*in1!=1
out1<=reg; reg:=reg*in1; out2<=reg*2;
in2=0 and in1!=0 and reset=0
out1<=1; reg<=in1;
in2!=0 and reset=0
reg := reg+in2; out1<=1; out2<=1;
t_4
reset=0 and reg=1 and reg*in1=1
out1<=0; reg:=reg*in1; out2<=0;
t_8
t_7
reset=0 and reg=1 and reg*in1!=1
out1<=0; reg:=reg*in1; out2<=reg*2;

Fig. 16. Non-optimized EFSM example.

```
// let E be an EFSM and let ob_reg be the observed register
reduce_efsm (register ob_reg, efsm E) {
  //retrieve the sets of transitions and states of E
  T = transitions_set(E);
  S = states_set(E);
  for each transition t in T {
    // delete a transition if the conditions that preclude
    // observability are matched
    if (write_on_register(t, ob_reg) and not(read_register(t, ob_reg)))
      delete_transition(t);
  }
}
delete_transition(transition t) {
  // retrieve the in-going state for transition t
  state s= in_going_state(t);
  // to delete a state, it must be different from the state
  // of the current configuration and there are no other
  // transitions, except the current one t, going into it
  if (not(is_configuration_state(s)) and (number_of_ingoing_transitions(s) == 1)
    delete_state(s);
  // remove transition t from E
  t.remove();
}
```

Fig. 17. Algorithm for removal of EFSM transition precluding observability.

Then, another method is proposed in Section 5.3.2 to reduce the complexity problem for the solver. The idea is that part of the work performed by the solver to find a sequence could be done earlier, and then invoke the solver on the reduced constraints set. In this case, no constraints are actually removed, but a part of the solution is provided to the solver. In such way, different constraints are already satisfied at beginning, and this is equivalent to reduce the number of constraints.

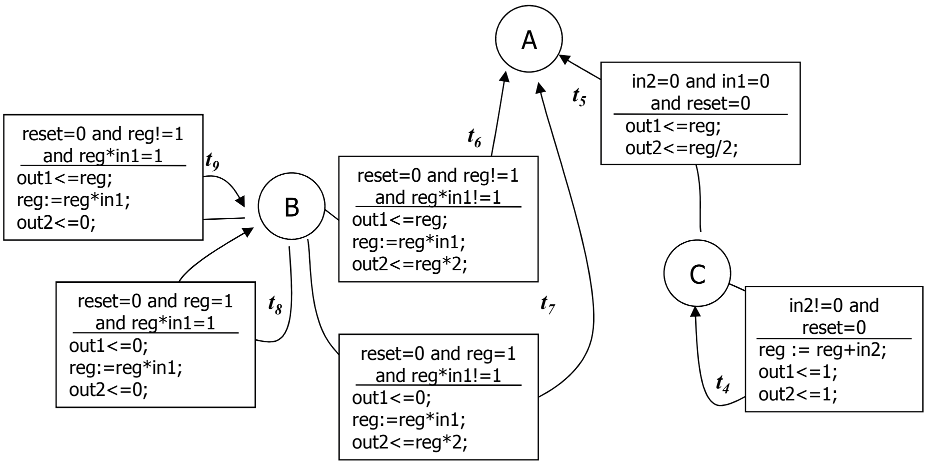

Fig. 18. Reduced EFSM.

5.3.1 Removal of useless transitions and of isolated states

Once a fault is observed on the registers, the ATPG saves a configuration. A configuration is composed by the state register and all the other registers of the design. Then, faulty and fault free configurations can be distinguished on the current state or on some registers value. Consider, for example, the EFSM represented in Figure 16 and that a particular fault *f* is observed on register *reg*. In faulty configuration *reg* is *0* and the EFSM in state *A*, and in the fault free configuration *reg* is 2 and state is always *A*. Then, if transition t_3 is traversed, a new value would be assigned to the register *reg*, losing the possibility of propagate the faulty configuration to the primary outputs. In fact, if the update function of transition t_3 is executed, then the faulty and fault free configuration would be equivalent.

Therefore, an algorithm has been defined to automatically prune this kind of transition from the EFSM model that is given in input to solver to generate a sequence. The algorithm is presented in Figure 17.

The algorithm uses the information collected during the learning phase to identify the transitions that cannot propagate a difference in the configuration and eliminates them.

Once the EFSM has been pruned from all the transitions that prevent the observability of registers configuration on primary outputs, some parts of the EFSM can remain isolated. Given the configuration state *s*, it is possible to be in a self-pointing state. This mean that there is no transitions outgoing from that state, but at most only transition in-going in *s*. Thus, the states of the EFSM that cannot be reached from the initial configuration are

```
// let E be a reduced EFSM
// let config_state be the configuration state
optimize (state config_state, efsm E) {
  state_set reached_states;
  // build the state set reached from the configuration state
  reached_states.insert(config_state);
  states_reached_from_state(config_state, reached_states);
  // remove all states that are not reached from the configuration
  for each state s in S {
    if ( not(reached_states.contains(s)) ) {
      s.remove();
      // remove all transitions moving out from state s
      transition tl = out_going_from_state(s);
      while ( not(tl.current_item() == NULL)) {
        tl.current_item().remove();
      }
    }
  }
}

// return the set of states reachable form sate s
states_reached_from_state(state s, state_set reached_states) {
  transition_list tl = out_going_from_state(s);
  while ( not(tl.current_item() == NULL)) {
    state in_state = in_going_state(t);
    // check whether current transition does not return to
    // current state and it does not reach a state which
    // has already been traversed in the visit
    if (not(in_state == s) and not(reached_states.contains(in_state))) {
      reached_states.insert(in_state);
      // continue exploring out-going paths
      // from state in_state
      states_reached_from_state(in_state, reached_states)
    }
    tl.move_to_next();
  }
}
```

Fig. 19. Algorithm for EFSM optimisation.

removed as they represent only constraints that cannot be satisfied. Figure 19 describes the algorithm that has been defined, to remove all the parts of the EFSM that are not reachable from the current configuration state. Let's consider the example in Figure 18 and say that configuration state is *B*. Then, starting form *B*, it is not possible to generate a sequence to reach state *C*. Therefore state *C* and all its out-going transition can be removed. The EFSM after the application of the optimization algorithm is depicted in Figure 20(a). Then, the Figure 20(b) presents the EFSM generated after optimization starting from state *C*.

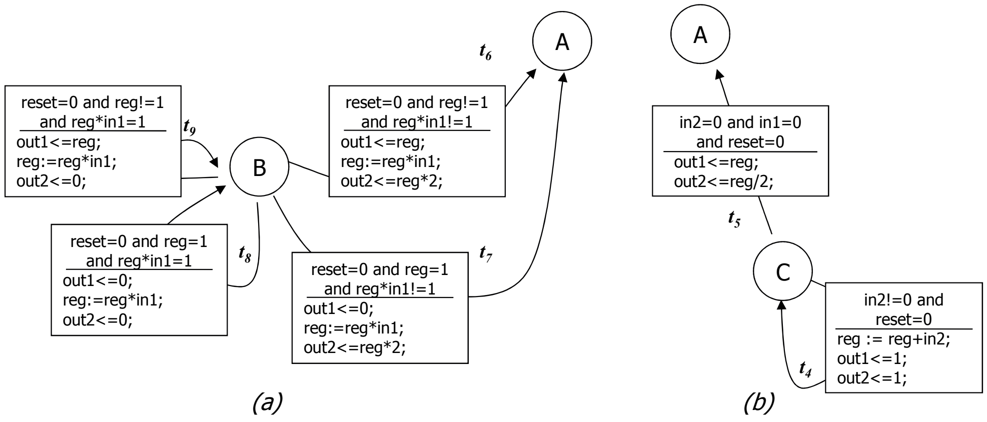
A
A
B
C
t_6
t_9
t_8
t_7
t_5
t_4
reset=0 and reg!=1 and reg*in1=1
out1<=reg; reg:=reg*in1; out2<=0;
reset=0 and reg!=1 and reg*in1!=1
out1<=reg; reg:=reg*in1; out2<=reg*2;
reset=0 and reg=1 and reg*in1=1
out1<=0; reg:=reg*in1; out2<=0;
reset=0 and reg=1 and reg*in1!=1
out1<=0; reg:=reg*in1; out2<=reg*2;
in2=0 and in1=0 and reset=0
out1<=reg; out2<=reg/2;
in2!=0 and reset=0
reg := reg+in2; out1<=1; out2<=1;
(a)
(b)

Fig. 20. Optimized from configuration's state B and C.

5.3.2 Pruning based on paths

The CLP solver tries to find a propagation sequence starting from the configuration that activates the fault. Such effort can be reduced if the solver is provided with a hint about the paths that are more profitable to be traversed. In this case, no constraints are actually removed from the CLP-based representation of the EFSM, but we provide the solver with part of the solution, thus reducing the number of satisfiable constraints.
Before invoking the solver to generate the propagation sequence, the ATPG applies an algorithm that searches for paths connecting the fault-free EFSM configuration with a state where at least one of update functions of its out-going transitions writes on the primary outputs. In fact, if a path allows updating the primary outputs, it is probable that the register value of the faulty configuration would be propagated on such outputs too. Then, all constraints related to the transition that are not involved in the generated paths are removed. These paths are generated such that they include transitions which allow the propagation of the values of registers involved in the faulty configuration towards the primary outputs. These transitions are identified and marked by learning phases performed during the EFSM generation and the subsequent removal of useless transitions. In particular, for every transition *t* marked as useful, a path is generated from the current configuration state to the out-going state of transition *t*.
The solver would try to check if the constraints are satisfiable and it would find a solution. If either it is not able to find a propagation sequence within a given timeout, or it returns that the problem is non satisfiable, a different path is generated and passed to the solver as initial configuration. Future works are related to associate a weight to each transition to generate paths maximizing the total weight of the involved transitions. Possible ideas for weighting transitions are: preferring transitions leading to a state, whose out-going transitions update the primary outputs, or transitions whose update functions use a large number of faulty registers, or transitions whose enabling functions consist of small conditions.

6. Experimental results

The CLP-based techniques for generating test sequences and propagation sequences has been applied to the benchmarks described in Table 1, where columns report the number of primary inputs (PIs), primary outputs (POs), flip-flops (FFs) and gates (Gates). Column Trns. shows the number of transitions of the EFSM modelling the DUV and GT (sec.) the time required to automatically generate the EFSM. Then, column BC reports the number of bit-coverage faults injected into the designs to check the fault coverage.

DUV	PIs	POs	FFs	Gates	Trns.	GT (sec.)	BC
b00	66	64	99	1692	7	0.1	1182
b04	13	8	66	650	20	0.3	408
b10	13	6	17	264	35	0.3	216
b11m	9	6	31	715	20	0.2	725
b00z	66	64	99	11874	9	0.2	1439
fr	34	32	100	1475	10	0.2	1041

Table 1. Benchmarks properties.

Such benchmarks have been selected because they present different characteristics which allow analyzing and confirming the effectiveness of the proposed approach. *b04, b10* have been selected from the well known ITC-99 benchmarks suite (ITC, 1999). *b11m* is a modified version of *b11*, included in the same suite, created by introducing a delay on some paths to make it harder to be traversed. The HDL descriptions of *b04, b10* and *b11m* contain a high number of nested conditions on signals and registers of different size. *b00, b00z* and *fr* contain conditional statements where one branch has probability $1-(1/(2^{-32}))$ of being satisfied, while the other has probability $1/(2^{-32})$. Thus, they are very hard to be tested by a random ATPG. In particular, *b00* and *b00z* are internal benchmarks, while *fr* is a real industrial case, i.e., it is a module of a face recognition system.

6.1 Test sequence generation

The effectiveness of the CLP-based transition oriented ATPG has been evaluated by comparing to a genetic algorithm-based high-level ATPG (Fin & Fummi, 2003a), which outperforms pure random-based ATPGs but it is not aware about the EFSM structure, and with a pseudo-deterministic ATPG, which uses only the random-walk mode to traverse the DUV state space. Stopping criterion is defined in term of the number and length of the generated test sequences. Table 2 reports the transition coverage (TC%), the statement coverage (SC%), the fault coverage (FC%), and the test generation time (T (sec.)), by using respectively the genetic algorithm-based ATPG (GA-ATPG), the pseudo-deterministic ATPG (RW-ATPG), and the proposed ATPG (RW+BJ-ATPG). It can be observed that RW+BJ-ATPG outperforms both the GA-ATPG and the RW-ATPG. The very low transition coverage achieved by the GA-ATPG for some benchmarks is due to the presence of transitions out-going from the initial states, whose enabling functions have an infinitesimal probability of being traversed by randomly fixing the values of primary inputs. Such a problem is partially solved by the RW-ATPG which is aware about the enabling functions of the EFSM, and definitely solved by the backjumping-based RW+BJ-ATPG that reaches 100% transition and statement coverage for all benchmarks. Then, also the achieved fault coverage for all benchmarks is sensibly increased.

6.2 Propagation sequence generation

The efficiency of the CLP-based fault oriented ATPG for propagation sequence generation has been evaluated by applying the testing flow of Figure 1.

	GA-ATPG				RW-ATPG				RW+BJ-ATPG			
DUV	**TC%**	**SC%**	**FC%**	**T (s.)**	**TC%**	**SC%**	**FC%**	**T (s.)**	**TC%**	**SC%**	**FC%**	**T (s.)**
b00	28.6	26.7	1.1	3.0	85.7	87.0	48.7	2.6	100.0	100.0	52.5	2.9
b04	80.0	90.2	94.9	23.2	85.0	95.0	99.0	8.7	100.0	100.0	99.0	9.1
b10	37.1	66.7	87.0	5.7	40.0	69.7	93.0	5.7	100.0	100.0	94.0	6.8
b11m	90.0	80.0	37.0	5.7	95.0	82.2	39.0	5.1	100.0	100.0	54.6	16.3
b00z	22.2	31.0	13.7	4.1	66.6	75.9	44.3	5.0	100.0	100.0	51.8	12
fr	20.0	13.3	0.86	10.3	80.0	86.7	70.4	4.9	100.0	100.0	84.0	5.2

Table 2. Comparison between a GA-based ATPG, a pseudo-deterministic ATPG and proposed CLP-based approach.

	RW+BJ-ATPG					CLP		CLP pure		RW+BJ-ATPG+CLP		
DUV	**FC%**	**TBP**	**TBA**	**SL**	**T (s.)**	**Prop**	**T (s.)**	**Prop**	**T (s.)**	**FC%**	**SL**	**T (s.)**
b00	52.5	64	498	3	2.9	84	2.5	0	aborted	59.6	7	5.9
b04	99.0	4	1	6	9.1	4	3.6	0	aborted	99.8	10	13.4
b10	94.0	13	0	11	6.8	12	3.3	0	aborted	99.5	18	10.1
b11m	54.6	117	124	59	16.3	313	36.7	0	aborted	66.8	71	56.7
b00z	13.8	131	497	6	12	613	9.1	0	aborted	56.4	30	18.5
fr	84.0	63	82	42	5.2	22	6.0	0	aborted	86.1	80	11.2

Table 3. Experimental results of fault-oriented ATPG.

Columns RW+BJ-ATPG of Table 3 report results achieved by applying transition-oriented ATPG of the incremental test generation flow of Figure 1. In particular, these columns show the achieved fault coverage (FC%), the number of faults observed but not detected (TBP), the number of faults not activated (TBA), the average length of the generated test sequences (SL) and the test generation time (T (sec.)).

Then, the CLP-based fault-oriented engine has been applied to find propagation sequences for TBP faults. Columns Prop. and T (sec.), below CLP, reports, respectively, the number of TBP faults for which the CLP-based engine was able to generate a propagation sequence, and the corresponding execution time.

The column CLP pure reports the results achieved by using the CLP-based engine without applying the strategies described in Section 5.2.3 for managing the CLP complexity. In this case, all TBP faults were aborted, since the CLP solver always run out of resources. This highlights the effectiveness of the strategies proposed for managing the CLP complexity.

Finally, the last three columns of the table report, respectively, the total fault coverage (FC%), the average length of test sequences (SL) and the total generation time T (sec.) obtained by adopting all steps of the incremental testing flow shown in Figure 1.

Results show that the fault-oriented engine increased the fault coverage for all benchmarks without requiring long computation time. No fault has been aborted (i.e., the engine never run out of resources), even if some TBP faults remained untested, because no propagation sequence was found. The analysis of TBP faults not propagated highlighted the fact that many configurations allow TBP faults to be observed on internal registers (i.e., there exist many activation sequences), but very few of them allow TBP faults to be propagated. Moreover, such few configurations are difficult to be generated by using the transition-oriented ATPG, since they are not fault-oriented. To solve such a problem, in the future, the

CLP-based fault-oriented engine will be extended for the generation of activation sequences too.

The effectiveness of the optimization strategies, proposed in Section 5.3, are summarized in Table 4. This methodology has been applied also to another benchmark, *Prawn*, that is a RISC processor with the instruction set having been enhanced to include interrupt handling and conditional branches.

Columns St. and T. report, respectively, the number of states and transitions of the corresponding EFSMs. Column TBP shows the number of faults to be propagated which have been activated by using the RW+BJ-ATPG. Column TOut s. shows the timeout provided to the CLP solver for finding a propagation sequence. Columns PSEQ, Abort and Time s. under No optimization shows, respectively, the number of propagation sequences generated by the CLP solver, the number of faults aborted (i.e., the number of faults for which the solver was unable to provide a response, either positive or negative), and the total time required for generating the CLP constraints to model the EFSM and running the search. The same parameters have been computed after applying the optimization techniques presented in Section 5.3 (Optimized column). Experimental results show that the proposed optimization techniques sensibly improve the effectiveness of the solver in searching for propagation sequences. The improvement is particularly evident in the case of *Prawn*, whose EFSM is very large. Without optimizations the solver always aborted, while after optimizations were applied, it succeeded in generating propagation sequences for all the TBP faults. Moreover, it can be observed that the sequence generation time was sensibly decreased, for all benchmarks, but *b04*. In the case of *b04*, optimizations did not provide benefits, since its EFSM is composed of very few states that cannot be further reduced by applying the proposed optimizations.

					No Optimization			Optimized		
DUV	**St.**	**T.**	**TBP**	**TOut s.**	**PSEQ**	**Abort**	**Time s.**	**PSEQ**	**Abort**	**Time s.**
b04	3	20	4	10	4	0	16.23	4	0	18.51
b10	11	35	13	10	13	0	26.08	13	0	7.43
b11m	9	20	117	10	117	0	63.17	117	0	16.68
prawn	61	160	66	14	0	66	998.12	66	0	105.89

Table 4. Experimental results of fault-oriented ATPG with optimization.

7. Conclusions

This work defines a functional ATPG framework that exploits a particular kind of EFSM which has been theoretically showed to allow a more uniform traversing of the DUV state space. Determinism is obtained by interfacing with CLP solver that adopts formal methods to solve the conditions of the enabling functions.

The effectiveness of the proposed ATPG compared with a genetic-based ATPG is evident. It greatly benefits from the fact that, by using the EFSM model, all conditional statements included in the DUV are under its control. The adoption of the EFSM model joint to the learning/random-walk/backjumping-based mechanisms allows to accurately addressing hard-to-traverse transitions. Then the fault-oriented engine has been proposed together with. This is the first work addressing the problems of entirely modelling an EFSM by means of CLP, and generating functional test patterns by combining the use of EFSM and

CLP. Moreover, some strategies have been implemented to manage the CLP complexity, and experimental results showed that, in this way, the proposed engine is able to generate propagation sequences and increase the fault coverage without running out of resources.

8. References

Abramovici, M. (1993). Dos and don'ts in computing fault coverage. In *Proc. of IEEE ITC*.

Apt, K. R. & Wallace, M. G. (2007). *Constraint Logic Programming using Eclipse*. Cambridge Univeristy Press.

Cheng, K. & Krishnakumar, A. (1996). Automatic generation of functional vectors using the extended finite state machine model. *ACM Transactions on Design Automation of Electronic Systems*, 1(1):57–59.

Corno, F., Cumani, G., Reorda, M. S. & Squillero, G. (2001). Effective techniques for high-level atpg. In *Proc. of IEEE ATS*, pages 225–230.

Di Guglielmo, G., Fummi, F., Marconcini, C. & Pravadelli, G. (2006a). EFSM Manipulation to Increase High-Level ATPG Efficiency. In *Proc. of IEEE ISQED*, pages 57–62.

Di Guglielmo, G., Fummi, F., Marconcini, C. & Pravadelli, G. (2006b). Fate: a functional atpg to traverse unstabilized efsms. In *Proc. of IEEE ETS*.

Di Guglielmo, G., Fummi, F., Marconcini, C. & Pravadelli, G. (2006c). Improving Gate-Level ATPG by Traversing Concurrent EFSMs. In *Proc. of IEEE VTS*.

Di Guglielmo, G., Fummi, F., Marconcini, C. & Pravadelli, G. (2007). Improving high-level and gate-level testing with FATE: a functional ATPG traversing unstabilized EFSMs. *IEE Computers and Digital Techniques*, 1(3):187–196.

Dijkstra, E. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271.

Ferrandi, F., Fummi, F. & Sciuto, D. (1998a). Implicit test generation for behavioral vhdl models. In *Proc. of IEEE ITC*, pages 587–596.

Ferrandi, F., Fummi, F. & Sciuto, D. (1998b). Implicit test generation for behavioral vhdl models. In *Proceedings of IEEE International Test Conference (ITC)*, pages 436–441.

Ferrandi, F., Fummi, F. & Sciuto, D. (2002a). Test generation and testability alternatives exploration of critical algorithms for embedded applications. *IEEE Transactions on Computers*, C-51(2):200–215.

Ferrandi, F., Rendine, M. & Sciuto, D. (2002b). Functional verification for systemc descriptions using constraint solving. In *Proceedings of IEEE Design Automation and Test in Europe (DATE)*, pages 744–751.

Fin, A. & Fummi, F. (2003a). Genetic algorithms: the philosophers stone or an effective solution for high-level TPG? In *Proc. of IEEE HLDVT*, pages 163–168.

Fin, A. & Fummi, F. (2003b). Genetic Algorithms: the Philosopher's Stone or an Effective Solution for High-Level TPG? In *Proc. of IEEE HLDVT*, pages 163–168.

Fummi, F., Harris, I. G., Marconcini, C., and Pravadelli, G. (2007). A CLPbased Functional ATPG for Extended FSMs. In *Proc. of IEEE MTV*.

Gajski, D., Zhu, J. & Domer, R. (1997). Essential issue in codesign. Thecnical report ICS-97-26, University of California, Irvine.

Ghosh, I. & Fujita, M. (2001). Automatic test pattern generation for functional register-transfer level circuits using assignment decision diagrams. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 20(3):402–415.

Guarnieri, V., Fummi, F., Marconcini, C. & Pravadelli, G. (2008). An Optimized CLP-based Technique for Generating Propagation Sequences. In *Proc. of IEEE EWDTS*, pages 25–28.

Ier, M., Parthasarathy, G. & Cheng, K.-T. (2005). Efficient conflict-based learning in an RTL circuit constraint solver. In *Proc. of IEEE DATE*, pages 666–671.

ITC (1999). High time for high-level test generation. Panel at IEEE ITC.

Jaffar, J. & Maher, M. J. (1994). Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581.

Kowalski, R. (1979). Algorithm = logic + control. In *Communications of the ACM*, pages 424–436.

Lee, D. & Yannakakis, M. (1992). Online minimization of transition systems. In *Proc. of ACM STOC*, pages 264–267.

Li, J. & Wong, W. (2002). Automatic test generation from communicating extended finite state machine (cefsm)-based models. In *Proc. of IEEE ISORC*, pages 181–185.

Lin, X., Pomeranz, I. & Reddy, S. M. (1999). Techniques for improving the efficiency of sequential circuit test generation. In *Proc. of ACM/IEEE ICCAD*, pages 147–151.

Lingappan, L., Ravi, S. & Jha, N. (2003). Test generation for non-separable RTL controller-datapath circuits using a satisfiability based approach. In *Proc. of IEEE ICCD*, pages 187–193.

Marconcini, C. (2008). A Functional ATPG as a bridge between Functional Verification and Testing. In *Ph.D. Thesis*.

Myers, G. (1979). *The Art of Software Testing*. Wiley - Interscience, New York.

Myers, G. (1999). *The Art of Software Testing*. Wiley - Interscience.

Padmanabhuni, S. (1999). Extended analysis of intelligent backtracking algorithms for the maximal constraint satisfaction problem. In *Proc. of IEEE CCECE*, pages 1710–1715.

Pauli, C., Nivet, M. L. & Santucci, J. F. (2000). Use of constraint solving in order to generate test vectors for behavioral validation. In *Proc. of IEEE HLDVT*, pages 15–20.

Russel, S. & Norvig, P. (2002). *Artificial Intelligence: A Modern Approach*. Prentice Hall.

Vemuri, R. & Kalyanaraman, R. (1995). Generation of design verification tests from behavioral vhdl programs using path enumeration and constraint programming. *IEEE Trans. Very Large Scale Integr. Syst.*, 3(2):201–214.

Wallace, M. & Veron, A. (1994). Two problems-two solutions: one system-ECLIPSE. In *IEE Colloquium on Advanced Software Technologies for Scheduling*, pages 1–3.

Wallace, M. G. (1997). Constraint programming. In *The Handbook of Applied Expert Systems*. CRC Press.

Wu, Q. & Hsiao, M. (2004). Efficient ATPG for design validation based on partitioned state exploration histories. In *Proc. of IEEE VTS*, pages 389–394.

Xin, F., Ciesielski, M. & Harris, I. (2005a). Design validation of behavioral vhdl descriptions for arbitrary fault models. In *Proc. of IEEE ETS*, pages 156–161.

Xin, F., Ciesielski, M. & Harris, I. (2005b). Design validation of behavioral VHDL descriptions for arbitrary fault models. In *Proc. of IEEE ETS*, pages 156–161.

Xin, F. & Harris, I. G. (2002). Test generation for hardware-software covalidation using non-linear programming. In *Proc. of IEEE HLDVT*, pages 175–180.

Zhang, L., Ghosh, I. & Hsiao, M. (2003). Efficient Sequential ATPG for Functional RTL Circuits. In *Proc. of IEEE ITC*, pages 290–298.