An Extended Finite State Machine-Based Approach to Code Coverage-Directed Test Generation for Hardware Designs

[1] *I. Melnichenko <igor.melnitxenko@gmail.com>,*
[2] *A. Kamkin <kamkin@ispras.ru>,*
[2] *S. Smolov <smolov@ispras.ru>,*
[1] *INEUM, 24 Vavilova st., Moscow, 119334, Russian Federation*
[2] *Institute for System Programming of the Russian Academy of Sciences,*
25 Alexander Solzhenitsyn st., Moscow, 109004, Russian Federation

Abstract. Model-based test generation is widely spread in functional verification of hardware designs. The extended finite state machine (EFSM) is known to be a powerful formalism for modelling digital hardware. As opposed to conventional finite state machines, EFSM models separate datapath and control, which makes it possible to represent systems in a more compact way and, in a sense, reduces the risk of state explosion during verification. However, EFSM state graph traversal problem seems to be nontrivial because of guard conditions that enable model transitions. In this paper, a new EFSM-based test generation approach is proposed and compared with the existing solutions. It combines random walk on a state graph and directed search of feasible paths. The first phase allows covering "easy-to-fire" transitions. The second one is aimed at "hard-to-fire" cases; the algorithm tries to build a path that enables a given transition; it is carried out by analyzing control and data dependencies and applying symbolic execution techniques. Experiments show that the suggested approach provides better transition coverage with shorter test sequences comparing to the known methods and achieves a high level of code coverage in terms of statements and branches. Out future plans include some optimizations aimed at method's applicability to industrial hardware designs.

Keywords: hardware design; hardware description language; simulation-based verification; test generation; modelling; extended finite state machine; graph traversal; random walk; backjumping; symbolic execution; constraint solving

DOI: 10.15514/ISPRAS-2015-27(3)-12

For citation: Melnichenko I., Kamkin A., Smolov S. An Extended Finite State Machine-Based Approach to Code Coverage-Directed Test Generation for Hardware Designs. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 3, 2015, pp. 161-182. DOI: 10.15514/ISPRAS-2015-27(3)-12.

1. Introduction

Functional verification is a labor-intensive and time-consuming stage of the hardware design process. According to [1], it spends about 70% of the effort, while the number of verification engineers is usually twice the number of designers. Moreover, the "verification gap", i.e. a difference between verification needs and capabilities, seems to grow over time [2]. In such a situation, improvement of the existing verification methods and development of new ones is of high value and importance. Simulation-based verification, often referred to as testing, is a widely accepted approach to hardware verification. It requires a testbench [1], a special environment that generates inputs, so-called stimuli, vectors or patterns, and optionally observes the outputs, so-called reactions.

Among the methods for stimulus generation, model-based approaches are of interest. Being formal representations of designs under test, models serve as a valuable source of "testing knowledge". There are a lot of model types used for specifying hardware: finite state machines (FSM) [3], extended FSM (EFSM) [4], Petri nets [5], etc. The key distinction of the EFSM formalism is clear separation of data and control flows. It is worth mentioning that EFSM models can be automatically extracted from HDL descriptions making it possible to generate code coverage-directed tests [6].

This article advances the FATE approach to EFSM-based functional test generation (FTG) [7]. The main feature of FATE is backjumping: if an EFSM traverser fails to cover a transition, it tries to detect a cause of the failure (that is, a transition which must be traversed in order to enable the target one) and constructs a path directly from the found transition. Another important part of the approach is a special heuristic addressing counters and loops. However, FATE is hardly applicable to hardware designs with complicated data and control dependencies.

The rest of the paper is organized as follows. Section II defines the EFSM model and briefly describes an EFSM extraction method having been used. Section III considers the original FATE approach, while Section IV introduces a number of improvements to it. Section V proposes a new EFSM-based FTG method and shows how it works by the example of two simple EFSMs. Section VI contains an experimental comparison of the abovementioned approaches. Section VII concludes the paper and outlines directions for future improvement of the suggested algorithm.

2. EFSM Model and HDL-to-EFSM Extraction

Let V be a set of *variables*. A *valuation* is a function that associates each variable with a value from the corresponding domain. The set of all valuations over V is denoted as D_V. A *guard* is a Boolean function defined on valuations ($D_V \to \{true, false\}$). An *action* is a transformation of valuations ($D_V \to D_V$). A pair $\gamma \to \delta$, where γ is a guard and δ is an action, is called a *guarded action*. When we speak about a function, it is implied that there is a *description* of the function in some formal language (thus, we can reason about the function's syntax, not only the semantics).

An EFSM is a tuple $M = \langle S_M, V_M, T_M \rangle$, where S_M is a set of *states*, $V_M = (I_M \cup O_M \cup R_M)$ is a set of *variables*, consisting of *inputs* (I_M), *outputs* (O_M) and *registers* (R_M), and T_M is a set of *transitions* (all sets are supposed to be finite). Each transition $t \in T_M$ is a tuple $(s_t, \gamma_t \rightarrow \delta_t, s'_t)$, where s_t and s'_t are respectively the *initial* and the *final state* of t, whereas γ_t and δ_t are respectively the *guard* and the *action* of t. A valuation $\nu \in D_{V_M}$ is referred to as a *context*, while a pair $(s, \nu) \in S_M \times D_{V_M}$ is called a *configuration*. A transition t is said to be *enabled* for a configuration (s, ν) if $s_t = s$ and $\gamma_t(\nu) = true$.

Given a *clock* C (a periodic event generator) and an *initial configuration* (s_0, ν_0), the EFSM operates as follows. In the beginning, it resets (initializes) the configuration: $(s, \nu) \leftarrow (s_0, \nu_0)$. On every "tick" of C, it computes the set of enabled transitions $E \leftarrow \{t \in T_M \mid s_t = s \wedge \gamma_t(\nu) = true\}$. A single transition $t \in E$ (chosen non-deterministically) fires; the EFSM changes the configuration (updates the context and moves from the initial state to the final one) $(s, \nu) \leftarrow (s'_t, \delta_t(\nu))$.

In this paper, we do not discuss in detail the way the EFSM models are extracted. At the experimental phase, we use an implementation of the method introduced in [8]. The method deals with HDL descriptions written in synthesizable subsets of VHDL and Verilog [9]. The major advantage of the approach is high automation – it requires no information except HDL code. The method uses heuristics for identifying states and clock signals and extracts the EFSM from the control flow graph-based representation. For every process defined in the HDL description, a single EFSM is usually built; all EFSM models of the description are defined over the same set of variables. It should be emphasized that EFSM actions have the "flat" syntax, which means that each action is a linear sequence of assignments.

We have enhanced the cited method by adding a new heuristic aimed at recognizing the *initial configuration*. A guarded action $\gamma_r \rightarrow \delta_r$ is said to be *resetting* if the following properties hold: (1) γ_r depends on exactly one clock signal, which is called a *reset*; (2) δ_r consists solely of assignments of the kind $v = c$, where $v \in (O_M \cup R_M)$ and c is a constant expression. Provided that there is only one resetting action, that action is supposed to lead to the initial EFSM configuration.

3. The Original FATE Algorithm

The aim of the FATE algorithm is to generate a test that covers all transitions of a given multi-EFSM system. A *test* is a set of *test sequences*, i.e. sequences of test vectors. A *test vector* is a valuation over the joint set of the EFSMs' inputs. The algorithm includes three phases: an *EFSM analysis*, a *random traversal* and a *directed traversal*.

3.1 EFSM Analysis

In the beginning, for each EFSM of the system, data and control dependencies between its transitions are derived. Let t and τ be transitions and v be a variable. v is said to be *defined* in t ($v \in Def_t$) if δ_t contains an assignment to v; v is said to be *used* in τ ($v \in Use_\tau$) if v appears either in γ_τ ($v \in Use_{\gamma_\tau}$) or in the right hand side of δ_τ ($v \in Use_{\delta_\tau}$). It is said that τ is *data dependent* on t (via v) if there exists a variable v such that $v \in (Def_t \cap Use_{\delta_\tau})$ and there exists a path $P = \{t_i\}_{i=1}^{n}$ from t to τ ($s'_t = s_{t_1}$ and $s'_{t_n} = s_\tau$) that does not define v. To keep the data dependency between τ and t, if $v \in Def_\tau$, there should be δ_τ's assignment with v in the right hand side that precedes the assignments to v. It is said that τ is *control dependent* on t (via v) if there exists a variable v such that $v \in (Def_t \cap Use_{\gamma_\tau})$ and there exists a path from t to τ that does not define v.

The derived data and control dependencies are represented by the directed graphs whose vertices are the transitions and arcs are the dependencies. Thus, each EFSM is associated with two such graphs (one is for the control dependencies; another is for the data dependencies).

The second step of the analysis is *counter detection*. A register r is said to be a *counter* if there is a loop in the EFSM such that: (1) there is a transition t that defines r; (2) r is defined recurrently (the current value depends on the previous one); (3) there is a transition t' that is control dependent on t via r. For each counter, all data dependency loops are saved.

Let us consider an EFSM M with $R_M = \{x, y\}$ such that there is a loop which consists of the following transitions:

1. $\gamma \equiv true$; $\delta \equiv \{x = y\}$;
2. $\gamma \equiv true$; $\delta \equiv \{y = x + 1\}$;
3. $\gamma \equiv true$; $\delta \equiv \{x = 1\}$;
4. $\gamma \equiv (y = 3)$; $\delta \equiv \{\}$.

In this example, y is considered as a counter with a data dependency loop consisting of transitions 1 and 2.

3.2 Random Traversal

After the analysis, the random traversal phase is launched. The phase is parameterized with two values, L and N, where L is the length of a test sequence and N is the number of test sequences in the test. The random traversal is described by the following pseudo-code ($\{M_i = \langle S_i, V, T_i \rangle\}_{i=1}^{m}$ are the EFSMs being tested; *result* is the generated test):

```
result ← ∅
coverage ← ∅
while |result| < N ∧ coverage ≠ ∪i Ti do
  reset({Mi})
```

```
    sequence ← ∅
    while |sequence| < L do
      vector ← ∅
      for i ∈ {1, ..., m} do
        out ← {t ∈ $T_i$ | $s_t$ = $s_i$}
        while out ≠ ∅ do
          t ← choose(out)
          out ← out \ {t}
          constraint ← refine($\gamma_t$, vector ∪ ν)
          if isSAT(constraint) then
            vector ← vector ∪ solve(constraint)
            coverage ← coverage ∪ {t}
            break
          end
        end // while out
      end // for i
      apply(vector, {$M_i$})
      sequence ← sequence · {vector}
    end // while sequence
    result ← result ∪ {sequence}
  end // while result
```

The pseudo-code above is based on the following functions: *reset*({M_i}) initializes the configurations of the models {M_i}; *choose*(T) returns a random item of the non-empty set T; *refine*(γ, ν) replaces variables of the formula γ with their values according to the partial valuation ν; *isSAT*(γ) checks whether the constraint γ is satisfiable; *solve*(γ) returns a valuation ν such that $\gamma(\nu) = 1$; *apply*(ν, {M_i}) assigns the inputs of the models {M_i} according to the partial valuation ν and executes the enabled transitions (uninitialized inputs are randomized). The symbols s_i and ν denotes respectively the current state of the model M_i and the context (shared among all models).

Being defined over the same set of variables, the EFSM models may affect each other while being co-executed. To minimize the influence, the following technique is applied. Each EFSM M_i is supplied with two parameters, F_i and A_i, where F_i is a constant inversely proportional to the number of inputs used in the M_i's guards (the more such inputs M_i has, the more models are expected to be affected by M_i) and A_i is a so-called *aging factor* (initially set to zero). The sum $(F_i + A_i)$ is supposed to be the priority for choosing the model M_i. The priorities specify the order in which the models are handled (**for** $i \in \{1, ..., m\}$ **do** ... **end**). The main idea with the aging factor is as follows. If test vector generation for M_i fails (*isSAT*(*constraint*) returns *false* for an outgoing transition), A_i is increased by a constant ΔA. Note that [7] has

no particular definition of ΔA; we use the value $\Delta A = min_{i=1,m} F_i$. After the model selection loop, the aging factor of the most priority model is set to zero.

3.3 Directed Traversal

If there are uncovered transitions after the random traversal, FATE proceeds with the directed generation. Before describing the phase, let us make a remark. The procedure below, applies Dijkstra's algorithm for finding a shortest path in a graph [10]; it is assumed that an arc weight is the number of registers used in the transition's guard. The directed traversal is performed separately for each EFSM. Here is the pseudo-code (*M* is the EFSM being tested; *result* is the generated test):

```
  targets ← $T_M$ \ coverage
  while targets ≠ ∅ do
    t ← choose(targets)
    covered = false
    for prefix ∈ reach(M, $s_t$) do
      reset(M)
      sequence ← ∅
      for vector ∈ prefix do
        apply(vector, M)
        sequence ← sequence · {vector}
      end // for vector
      constraint ← refine($\gamma_t$, ν)
      if isSAT(constraint) then
        vector ← solve(constraint)
        apply(vector, M)
        sequence ← sequence · {vector}
        result ← result ∪ {sequence}
        coverage ← coverage ∪ {t}
        covered ← true
        break
      end
    end // for prefix
    if ¬covered then
      if ¬process(M, t) then
        warning "The transition t cannot be reached"
      end
    end
    targets ← targets \ {t}
  end // while targets
```

Besides the auxiliary functions defined above, this pseudo-code uses *reach*(*M*, *s*), which returns the set of known test sequences reaching the state *s* of the model *M*, and *process*(*M*, *t*), which tries to cover the transition *t* of the model *M* by taking into account the control dependencies (it will be described later on). Note that if *targets* includes transitions outgoing from the covered states, *choose*(*targets*) returns one of them; transitions whose initial states has not been reached are selected only if there are no others. Here is the description of *process*(*M*, *t*):

```
registers ← $R_M \cap Use_{\gamma t}$
for reg ∈ registers do
  defines ← {t ∈ $T_M$ | reg ∈ $Def_t$}
  for def ∈ defines do
    for prefix ∈ reach(M, $s_{def}$) do
      reset(M)
      sequence ← ∅
      for vector ∈ prefix do
        apply(vector, M)
        sequence ← sequence · {vector}
      end
      path ← shortestPath(M, $s'_{def}$, $s_t$)
      path ← path · {t}
      if isCounter(reg) then
        constraint ← refine($\gamma_{def}$, ν)
        vector ← solve(constraint)
        apply(vector, M)
        sequence ← sequence · {vector}
        loop ← processCounter(M, $s'_{def}$, t, reg)
        if loop = null then
          return false
        end
        path ← loop · path
      else
        path ← {def} · path
      end
      covered ← true
      for p ∈ path do
        if reg ∉ $Def_p$ ∨ p = t then
          γ ← γp
        else
          γ ← $\gamma_p \wedge \gamma_{t|reg}[\delta_p]$
        end
        constraint ← refine(γ, ν)
        if isSAT(constraint) then
```

```
          vector ← solve(constraint)
          apply(vector, M)
          sequence ← sequence · {vector}
        else
          covered ← false
          break
        end
      end // for p
      if covered then
        result ← result ∪ {sequence}
        coverage ← coverage ∪ {t}
        return true
      end
    end // for prefix
  end // for def
end // for reg
return false
```

The following notations are used: *shortestPath*(*M*, *s*, *s*′) finds the shortest path between the states *s* and *s*′ of the *M*’s state graph using Dijkstra’s algorithm; *isCounter*(*reg*) checks whether the register *reg* is a counter; $\gamma_{|v}$ denotes the minimal sub-constraint of the constraint γ that depends on the variable *v* such that $\gamma \rightarrow \gamma_{|v}$ holds; γ[δ] stands for the constraint produced from γ by applying the substitution corresponding to the action δ.

Let $\gamma \equiv (x = const_1 \wedge y = const_2)$ and $\delta \equiv \{x = z\}$, where *x*, *y*, and *z* are variables, while $const_1$ and $const_2$ are constants. In this case, $\gamma_{|x} \equiv (x = const_1)$ and $\gamma[\delta] \equiv (z = const_1 \wedge y = const_2)$.

Here is the pseudo-code for *processCounter*(*M*, *s*, *t*, *reg*).

```
if $\gamma_{t|reg}$(ν) then
  return {}
end
loop ← null
loopIterator ← createLoops(M, s, reg)
while $\neg\gamma_{t|reg}$(ν) do
  while hasNext(loopIterator) do
    tempContext ← ν
    tempSequence ← sequence
    loop ← next(loopIterator)
    for l ∈ loop do
      constraint ← refine($\gamma_l$, ν)
      if isSAT(constraint) then
```

```
            vector ← solve(constraint)
            apply(vector, M)
            sequence ← sequence · {vector}
          else
            ν ← tempContext
            sequence ← tempSequence
            loop ← null
            break
          end
          if loop ≠ null ∧ $\gamma_{t|reg}$(ν) then
            return loop
          end
        end // for loop
      end // while hasNext
    end // while ¬γ
    return null
```

The pseudo-code utilizes three special functions: *createLoops*(M, s, r) constructs all possible elementary loops in the M's state graph that start from the state s and include transitions dependent via the register r and returns the iterator that combines a *bounded* number of elementary loops into complex ones (the elementary loops are constructed by using Dijkstra's algorithm to connect dependent transitions); *hasNext*(i) checks whether the iterator i can produce more loops; *next*(i) returns the next loop and updates the iterator i. Note that the limit on the loop length is chosen individually for each design.

4. The FATE+ Algorithm

We have implemented a slightly modified version of the original FATE algorithm, so-called FATE+. Let us consider the changes having been made.

4.1 Transition Selection

In FATE+'s random traversal, *choose*(T), where T is a non-empty set of transitions, works a bit differently. If there exist uncovered transitions, the function randomly chooses one of them; otherwise, it returns an arbitrary item of T. Our experiments show that this minor change significantly increases the effectiveness of the random generation phase.

4.2 Symbolic Execution

FATE implements an approximate method for checking whether a given path is feasible (**for** $p \in$ *path* **do** ... **end**). Let P be a path, t be the last transition of P, r be a register used in γ_t, and ν be a context. Given a transition p of P, the algorithm checks whether p defines r. If it does, the following constraint is constructed and

tried to be satisfied: $\gamma \leftarrow \gamma_p \wedge \gamma_{t|r}[\delta_p]$. It is worth reminding that $\gamma_{t|r}$ is the minimal conjunctive member of γ_t that includes all occurrences of r, while $\gamma_{t|r}[\delta_p]$ is the formula produced from $\gamma_{t|r}$ by applying the forward substitution corresponding to the action δ_p. The method looks inadequate in the sense that if γ is unsatisfiable for some p, it does not really mean that P is infeasible.

We suggest replacing the approximate approach with full-scale symbolic execution that takes into consideration all the variables defined and used along the path. To be more precise, we suggest using the well-known method for computing the weakest precondition of a loop-free program, i.e. a sequence of guarded actions, with respect to a postcondition [11]. The main idea is as follows. Let $\gamma \equiv$ *true*. Starting from the end of P, for each transition p, including t, the following transformation of γ is performed: $\gamma \leftarrow \gamma_p \wedge \gamma[\delta_p]$. Note that the input variables are renamed in such a way that each transition refers to a unique copy of the inputs. As soon as P is processed, all occurrences of the registers are replaced by the values taken from ν: $\gamma \leftarrow$ *refine*(γ, ν). P is feasible if and only if γ is satisfiable. A test sequence can be constructed by solving the constraint.

Let us consider an EFSM M with $I_M = \{i0, i1, i2\}$ and $R_M = \{x, y, z\}$ such that there is a path which consists of the following transitions:

1. $\gamma \equiv$ *true*; $\delta \equiv \{z = i0\}$;
2. $\gamma \equiv (i1 = 1)$; $\delta \equiv \{x = z\}$;
3. $\gamma \equiv$ *true*; $\delta \equiv \{y = i2\}$;
4. $\gamma \equiv (x = 4 \wedge y = 2)$; $\delta \equiv \{\}$.

For this path, $\gamma \equiv (i0[0] = 4 \wedge i1[1] = 1 \wedge i2[2] = 2)$ is produced.

4.3 Test Reduction

In FATE, there is a frequent situation where multiple test vectors cover the same transition. To overcome the issue, we have introduced a simple test reduction technique. While generating tests, each test sequence is associated with the transitions having been covered. At the end of the process, the set of test sequences W and the set of covered transitions T_{cov} are available. The technique is as follows. First, the transitions reached by unique test sequences are identified. Each test sequence that covers at least one such transition is moved from W to the reduced test R; all transitions covered by the sequence are excluded from T_{cov}. Then, while T_{cov} is not empty, the following actions are performed. The test sequences that cover largest subsets of T_{cov} are determined; among them, a shortest one is chosen. The selected sequence is moved from W to R, while the covered transitions are removed from T_{cov}.

5. The RETGA Algorithm

The algorithm proposed in this paper is called RETGA (Retrascope EFSM-based Test Generation Algorithm). It has the same phases as FATE; moreover, the EFSM analysis phase is identical to FATE's one. As FATE+, it uses the modified *choose*(T) function and applies the test reduction. Let us consider the main phases in more detail.

5.1 Random Traversal

As in FATE, the EFSM models are processed one-by-one; though a different arbitration principle is used. The priority of a model depends on the coverage having been achieved: the better the coverage is, the less the priority is. Such a strategy is to avoid a situation when a covered EFSM of the highest priority prevents generating inputs for poorly covered models.

The pseudo-code for the random traversal is as follows (as before, $\{M_i = \langle S_i, V, T_i \rangle\}_{i=1}^{m}$ are the EFSMs being tested; *result* is the generated test):

```
result ← ∅
coverage ← ∅
ignored ← 0
L ← ($\Sigma_i$ |$T_i$|) / ($\Sigma_i$ |$S_i$|)
while ignored ≤ L ∧ coverage ≠ $\cup_i$ $T_i$ do
  reset({$M_i$})
  sequence ← ∅
  usefulSequence ← false
  transitions ← ∅
  buffer ← ∅
  while |buffer| ≤ L do
    vector ← ∅
    usefulVector ← false
    for i ∈ {1, ..., m} do
      out ← {t ∈ $T_i$ | $s_t$ = $s_i$}
      while out ≠ ∅ do
        t ← choose(out)
        out ← out \ {t}
        constraint ← refine($\gamma_t$, vector ∪ ν)
        if isSAT(constraint) then
          vector ← vector ∪ solve(constraint)
          if t ∉ coverage then
            usefulSequence ← true
            coverage ← coverage ∪ {t}
          end
          if t ∉ transitions then
            usefulVector ← true
            transitions ← transitions ∪ {t}
          end
          break
        end
      end // while out
    end // for i
    apply(vector, {$M_i$})
    buffer ← buffer · {vector}
    if usefulVector then
      sequence ← sequence · buffer
      buffer ← ∅
    end
  end // while sequence
  if usefulSequence then
    result ← result ∪ {sequence}
  else
    ignored ← ignored + 1
  end
end // while result
```

5.2 Directed Traversal

Before describing the directed traversal phase, let us give some definitions. A *piecewise path* is a sequence of paths, so-called *pieces*, for which there is a path including all of the pieces (with no overlaps) in the given order. Given a register r, a *partial definition path* is a piecewise path that *propagates* at least one input to r and has no transitions not taking part in the propagation.

The *propagation* of an input to a register is inductively defined as follows. If there exist a transition t and a variable r^* such that δ_t contains an assignment to r^* that involves x, then x is said to be *propagated* to r^* along the piecewise path $\{\{t\}\}$. If (1) x is *propagated* to r^* along the path P, (2) τ is data dependent on t, the last transition of the last piece of P, via r^*, and (3) δ_τ contains an assignment to r which involves r^*, then x is said to be *propagated* to r along the path $P \cdot \{\{\tau\}\}$.

The directed traversal is performed separately for each EFSM. Here is the pseudo-code (M is the EFSM being tested; *result* is the generated test):

```
targets ← {t ∈ ($T_M$ \ coverage) | reach(M, $s_t$) ≠ ∅}
while targets ≠ ∅ do
  t ← choose(targets)
  path ← shortestPath$^*$(M, $s_t$)
  path ← path · {t}
  if isFeasible(M, path) then
    sequence ← solve(M, path)
    result ← result ∪ {sequence}
```

```
      coverage ← coverage ∪ {t}
    else
      if ¬process(M, t) then
        warning "The transition t cannot be reached"
      end
    end
    targets ← (targets \ {t}) ∪ {$\tau$ ∈ $T_M$ | $s_\tau$ = $s'_t$}
  end // while targets
```

Here, *shortestPath**(*M*, *s*) returns a shortest (in terms of the number of transitions) path from the initial state of the model *M* to the state *s*; *isFeasible*(*M*, *P*) constructs the weakest precondition of the path *P* with respect to *true* and checks whether it is satisfiable in the initial context of the model *M*; *solve*(*M*, *P*) satisfies the constraint and converts the solution to the test sequence (uninitialized inputs are randomized). The *process*(*M*, *t*) function looks as follows:

```
  for counter ∈ {r ∈ $R_M$ ∩ $Use_{\gamma t}$ | isCounter(r)} do
    loops ←
      {{{$t_i$}}$_i$ | {$t_i$}$_i$ ∈ dataDepLoops(M, counter)}
    if processLoops(M, t, counter, loops) then
      return true
    end
  end // for counter
  for define ∈ partialDefPaths(M, $R_M$ ∩ $Use_{\gamma t}$) do
    if processPieces(M, t, define) then
      return true
    end
  end // for define
  return false
```

In the pseudo-code above, *dataDepLoops*(*M*, *c*) denotes the set of data dependency loops for the counter *c* of the model *M* (each loop starts with the transition that *defines* the counter). As you can see, *loops* is the set of piecewise paths relating to the data dependency loops. *partialDefPaths*(*M*, *R*) returns the set of partial definition paths for *M*'s registers of the set *R*. Here is the description of *processLoops*(*M*, *t*, *counter*, *loops*):

```
  groups ← groupLoops(loops, counter)
  for group ∈ groups do
    loopIterator ← init(M, group)
    while hasNext(loopIterator) do
      loop ← next(loopIterator)
      if processPieces(loop · {{t}}) then
        return true
      end
    end //while hasNext
```

```
  end // for group
  return false
```

Here, *groupLoops*(*L, counter*) splits the set of loops (piecewise paths) *L* into disjoint subsets according to the first transition (which defines the *counter* register). The loop iteration scheme is similar to FATE's one, though each result is a piecewise path. The pseudo-code for *processPieces*(*M*, *t*, $\{P_i\}_{i=1}^{k}$) is shown below:

```
  if reach(M, $s_t$) = ∅ then
    return false
  end
  path ← shortestPath*(M, start($P_1$))
  for i ∈ {1, ..., k-1} do
    path ← path · $P_i$
    if ¬isFeasible(M, path) then
      return false
    end
    path′ ←
      path · shortestPath(M, end($P_i$), start($P_{i+1}$))
    failed ← true
    if isFeasible(M, path′) then
      path ← path′
      failed ← false
    else
      for bridge ∈ paths(M, end($P_i$), start($P_{i+1}$)) do
        path′ ← path · bridge
        if isFeasible(M, path′) then
          path ← path′
          failed ← false
          break;
        end
      end // for bridge
    end // if isSAT
    if failed then
      return false
    end
  end // for i
  path ← path · $P_k$
  if ¬isFeasible(M, path) then
    return false
  end
  sequence ← solve(M, path)
  result ← result ∪ {sequence}
```

```
    coverage ← coverage ∪ {t}
    return true
```

In the pseudo-code, *start*(*P*) and *end*(*P*) return respectively the initial and the final state of the piecewise path *P*; *paths*(*M*, *s*, *s*′) returns the list of cycle-free paths between *M*'s states *s* and *s*′ sorted by length.

5.3 Examples

Let us consider how the RETGA algorithm works on the example of two models, namely EFSM-1 and EFSM-2. Both models correspond to the cases that are difficult for FATE.

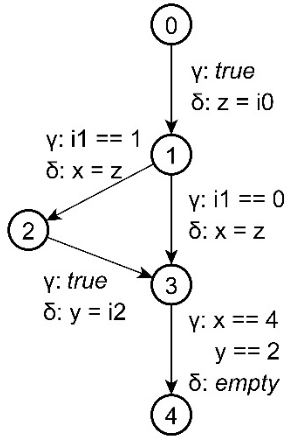

Fig. 1. EFSM-1

In EFSM-1 (see Fig. 1), the random traversal is unlikely to cover the transition 3→4 as it requires, first, walking through the path 0→1→2→3 and, second, assigning *i*0 ← 4 (while traversing 0→1) and *i*2 ← 2 (while traversing 2→3). The random traversal is most likely produce two input sequences that cover 0→1→2→3 and 0→1→3. As for the directed traversal of 3→4, the following partial definition paths are found for the registers x and y used in the transition's guard:

1. 0→1→3 (*i*0 is propagated to *x* via *z*);
2. 0→1→2 (*i*0 is propagated to *x* via *z*);
3. 2→3 (*i*2 is directly assigned to *y*).

The first path does not initialize *y* and has no continuations that could do that. For the second one, the pieces {0→1→2, 3→4} are composed and supplemented by the only "bridge" 2→3. For the third path, the "prefix" 0→1→2 explored at the random traversal phase is put before the partial definition path. In both cases, the path 0→1→2→3→4 is constructed. To check whether the path is feasible, the weakest precondition is computed: $i0[1] = 4 \wedge i1[2] = 1 \wedge i2[3] = 2$ (the indices in the square

brackets refer to the positions of the test vectors in the test sequence). It is satisfiable; the solution is as follows:

1. *i*0 = 4; *i*1 and *i*2 are randomly valued;
2. *i*1 = 1; *i*0 and *i*2 are randomly valued;
3. *i*2 = 2; *i*0 and *i*1 are randomly valued;
4. *i*0, *i*1 and *i*2 are randomly valued.

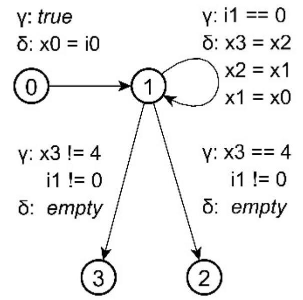

Fig. 2 EFSM-2

In EFSM-2 (see Fig. 2), a transition of the interest is 1→2. The shortest path that reaches the transition is 0→1→1→1→2 with the assignment *i*0 ← 4 on the first step. There is only one partial definition path for *x*3, namely 0→1→1→1. The path can be supplemented only with the target transition, which gives 0→1→1→1→2. The weakest precondition is $i0[1] = 4 \wedge i1[2] = 0 \wedge i1[3] = 0 \wedge i1[4] = 0 \wedge i1[5] \neq 0$ and it is satisfiable.

6. Experimental Results

The RETGA algorithm has been implemented as a part of the Retrascope [12] project. It uses the Fortress [14] library together with the Z3 [15] solver for representing expressions and solving constraints. To compare the algorithm with FATE and FATE+, the ITC'99 benchmark [13] was utilized.

Table I shows the characteristics of the EFSMs extracted from some ITC'99's designs. As it has been already said, we used the extended variant of the method described in [8] to build the models, though all of the presented approaches do not depend on the way EFSMs are produced.

Table I. Characteristics of the Extracted EFSMs

Design	Number of States	Number of Transitions
b01	8	24
b02	7	17
b04	3	29
b06	7	33

Design	Number of States	Number of Transitions
b07	8	21
b08	4	12
b10	11	38

Table II and Table III show the test generation results. All generators achieve 100% coverage for b01, b02, b04 and b06 and 95% coverage for b07 (there is an infeasible transition). The difference in coverage reached by RETGA and FATE / FATE+ for b08 is due to the fact that FATE and FATE+ handle data dependencies in a simpler way; in particular, they do not try different “bridges”. The difference in coverage reached by FATE and FATE+ for b08 and b10 demonstrates the advantage of the symbolic execution over the simplified approach used in FATE. The difference in size of the tests generated by FATE and FATE+ relates to the test reduction technique applied in FATE+. The RETGA’s tests are usually shorter since it rejects redundant random vectors.

It is significant to note that the *L* and *N* parameters (which are related to the random traversal phase of FATE and FATE+) were set to $\sum_{i=1}^{m} |S_i|$ and $\sum_{i=1}^{m} |T_i| \,/\, \sum_{i=1}^{m} |S_i|$ respectively. The loop iteration limit (which is relevant for all of the generators) was set to 8 (this value is enough for b07 and b08, whereas other designs have no counters).

Table II. Number of Test Vectors in the Tests

	FATE	FATE+	RETGA
b01	115	70	49
b02	62	48	33
b04	104	104	36
b06	198	100	76
b07	246	208	166
b08	31	31	52
b10	173	170	135

Table III. Transition Coverage Achieved by the Tests

	FATE	FATE+	RETGA
b01	100%	100%	100%
b02	100%	100%	100%
b04	100%	100%	100%
b06	100%	100%	100%
b07	95%	95%	95%
b08	75%	83%	100%
b10	89%	100%	100%

The tests generated by RETGA were applied to the designs by using the Questa simulator [16]. The source code coverage having been achieved is presented in Table IV (each column corresponds to some metric of the Questa coverage report). It can be seen that the code coverage is rather high.

Table IV. Source Code Coverage Reached by RETGA

	Statements	Branches	FSM States	FSM Transitions
b01	100%	100%	100%	100%
b02	100%	100%	100%	100%
b04	100%	100%	100%	100%
b06	100%	100%	100%	100%
b07	93.93%	94.73%	100%	100%
b08	100%	100%	100%	100%
b10	100%	100%	100%	100%

7. Conclusion

In this paper, an EFSM-based test generation algorithm has been proposed. The approach allows reaching better transition coverage with less number of test vectors than the known methods. However, the research is still in progress; there are many issues to be solved. Let us mention some of them. First, the approach is hardly applicable to complex hardware designs involving a great number of tightly connected EFSMs. It uses a simple coverage-based heuristic to decide which EFSM to handle next, whereas advanced techniques are expected to rely on the semantics of a system under test. Second, the method for searching “bridges” needs to be optimized. Being irrelevant for simple EFSMs (as ones presented in Section VI), this issue is of high value and importance for real-life hardware. Third, in the current implementation, each guard (each constraint, in general) is viewed as an indivisible entity and solved as a whole. It is not an issue as long as the goal is to cover EFSM transitions, but it may lead to poor expression coverage as there are many ways to satisfy a constraint. Finally, the quality of testing strongly depends on the models being used. It seems to be useful to formalize a notion of a “good” model.

References

[1]. Bergeron J. Writing Testbenches: Functional Verification of HDL Models, *Kluwer Academic Publishers*, 2003.
[2]. Blyler J. Are Best Practices Resulting in a Verification Gap? (http://chipdesignmag.com/sld/blog/2014/03/04/are-best-practices-resulting-in-a-verification-gap).
[3]. Jusas V., Neverdauskas T. FSM Based Functional Test Generation Framework for VHDL. *Proceedings of International Conference on Information and Software Technologies (ICIST)*, 2012. pp. 138-148.
[4]. Duale A.Y., Uyar M.U. A Method Enabling Feasible Conformance Functional Test Sequence Generation for EFSM Models. *IEEE Transactions on Computers*, 53(5), 2004. pp. 614-627.
[5]. Lazarev V.G., Pijl' E.I. Sintez upravljajushhih avtomatov. *Energoatomizdat*, Moscow, 1989. 328 p. (in Russian)
[6]. Cheng K.T., Krishnakumar A.S. Automatic Generation of Functional Vectors Using the Extended Finite State Machine Model. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 1996. pp. 57–79.
[7]. Di Guglielmo G., Di Guglielmo L., Fummi F., Pravadelli G. Efficient Generation of Stimuli for Functional Verification by Backjumping Across Extended FSMs. *Journal of Electronic Functional testing: Theory and Application*, 27(2), 2011. pp. 137–162.
[8]. Kamkin A. Smolov S. The Method of EFSM Extraction from HDL: Application to Functional Verification. *Proceedings of the Conference on Problems of Perspective Micro- and Nanoelectronic Systems Development*, Part II, 2014. pp. 113-118.
[9]. Navabi Z. Languages for Design and Implementation of Hardware. W.-K. Chen (Ed.). The VLSI Handbook. *CRC Press*, 2007. 2320 p.
[10]. Dijkstra E.W. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1, 1959, pp. 269–271.
[11]. Dijkstra E.W. A Discipline of Programming. *Prentice Hall*, 1976, 217 p.
[12]. Retrascope toolkit. http://forge.ispras.ru/projects/retrascope
[13]. ITC'99 benchmark. http://www.cad.polito.it/tools/itc99.html
[14]. Fortress library. http://forge.ispras.ru/projects/solver-api
[15]. Z3 solver. http://z3.codeplex.com
[16]. Questa simulator. http://www.mentor.com/products/fv/questa/

Подход к генерации тестов, нацеленных на покрытие кода HDL-описаний аппаратуры, на основе расширенных конечных автоматов

[1] И. Мельниченко <igor.melnitxenko@gmail.com>,
[2] А. Камкин <kamkin@ispras.ru>,
[2] С. Смолов <smolov@ispras.ru>,
[1] ОАО «Институт электронных управляющих машин им. И.С. Брука», 119334, Москва, ул. Вавилова, 24
[2] Институт системного программирования РАН, 109004, Москва, ул. Александра Солженицына, 25

Аннотация. Генерация тестов по моделям широко используется для функциональной верификации аппаратуры. Расширенные конечные автоматы (extended finite state machines, EFSM) — удобный формализм для моделирования цифровых устройств. В отличие от обычных конечных автоматов, в EFSM-моделях управляющие сигналы и данные разделены, что позволяет описывать системы в более компактной форме, уменьшая в некотором смысле риск комбинаторного взрыва при верификации. Однако обход графа состояний EFSM-модели является нетривиальной задачей из-за наличия условий на выполнимость переходов. В данной статье представлен метод генерации тестов по EFSM-моделям и проведено его сравнение с другими подходами. Предлагаемый метод сочетает случайный обход графа состояний автомата и направленный поиск реализуемых путей. Первая из указанных фаз направлена на покрытие «простых» переходов, вторая — «сложных». Под сложностью переходов здесь понимается наличие зависимостей охранных условий переходов от внутренних переменных. При направленном поиске используется информация о зависимостях по данным и управлению между переходами автомата и задействуется символическое исполнение. Было выполнено сравнение предлагаемого метода с существующими аналогами путем сопоставления параметров тестов, сгенерированных для заданного набора описаний модулей цифровой аппаратуры. Во всех случаях в качестве входных данных использовались EFSM-модели, автоматически извлеченные из кода. Полученные данные показывают, что в сравнении с другими подходами метод обеспечивает лучшие показатели покрытия исходного кода более короткими тестами. В будущем планируется реализовать ряд оптимизаций, направленных на применение метода к промышленным HDL-описаниям.

Ключевые слова: проектирование аппаратуры; язык описания аппаратуры; имитационная верификация; генерация тестов; моделирование; расширенный конечный автомат; обход графа; случайный обход; поиск с возвратами; символическое исполнение; разрешение ограничений.

DOI: 10.15514/ISPRAS-2015-27(3)-12

Для цитирования: Мельниченко И.В., Камкин А.С., Смолов С.А. Подход к генерации тестов, нацеленных на покрытие кода HDL-описаний аппаратуры, на основе расширенных конечных автоматов. Труды ИСП РАН, том 27, вып. 3, 2015 г., стр. 161-182 (на английском языке). DOI: 10.15514/ISPRAS-2015-27(3)-12.

Список литературы

[1]. Bergeron J. Writing Testbenches: Functional Verification of HDL Models, *Kluwer Academic Publishers*, 2003.
[2]. Blyler J. Are Best Practices Resulting in a Verification Gap? (http://chipdesignmag.com/sld/blog/2014/03/04/are-best-practices-resulting-in-a-verification-gap).
[3]. Jusas V., Neverdauskas T. FSM Based Functional Test Generation Framework for VHDL. *Proceedings of International Conference on Information and Software Technologies (ICIST)*, 2012. pp. 138-148.
[4]. Duale A.Y., Uyar M.U. A Method Enabling Feasible Conformance Functional Test Sequence Generation for EFSM Models. *IEEE Transactions on Computers*, 53(5), 2004. pp. 614-627.
[5]. Лазарев В.Г., Пийль Е.И. Синтез управляющих автоматов. *Энергоатомиздат*, 1989. 328 с.
[6]. Cheng K.T., Krishnakumar A.S. Automatic Generation of Functional Vectors Using the Extended Finite State Machine Model. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 1996. pp. 57–79.
[7]. Di Guglielmo G., Di Guglielmo L., Fummi F., Pravadelli G. Efficient Generation of Stimuli for Functional Verification by Backjumping Across Extended FSMs. *Journal of Electronic Functional testing: Theory and Application*, 27(2), 2011. pp. 137–162.
[8]. Kamkin A. Smolov S. The Method of EFSM Extraction from HDL: Application to Functional Verification. *Proceedings of the Conference on Problems of Perspective Micro- and Nanoelectronic Systems Development*, Part II, 2014. pp. 113-118.
[9]. Navabi Z. Languages for Design and Implementation of Hardware. W.-K. Chen (Ed.). The VLSI Handbook. *CRC Press*, 2007. 2320 p.
[10]. Dijkstra E.W. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1, 1959, pp. 269–271.
[11]. Dijkstra E.W. A Discipline of Programming. *Prentice Hall*, 1976, 217 p.
[12]. Инструмент Retrascope. http://forge.ispras.ru/projects/retrascope
[13]. Тестовый набор ITC'99. http://www.cad.polito.it/tools/itc99.html
[14]. Библиотека Fortress. http://forge.ispras.ru/projects/solver-api
[15]. Решатель ограничений Z3. http://z3.codeplex.com
[16]. Симулятор Questa. http://www.mentor.com/products/fv/questa/