УДК 004.05

# Метод генерации функциональных тестов для HDL-описаний на основе проверки HLDD-моделей

М.С. Лебедев, С.А. Смолов

Институт системного программирования РАН, {lebedev, smolov}@ispras.ru

***Аннотация* — Автоматизированные методы являются перспективным направлением в области проверки корректности (верификации) цифровой микроэлектронной аппаратуры. В настоящее время предложен ряд методов генерации функциональных тестов, основанных на использовании моделей. В данной статье предложен новый метод, основанный на автоматическом извлечении моделей в форме высокоуровневых решающих диаграмм и генерации тестов, обеспечивающих высокое покрытие кода, с помощью средств проверки моделей. Проведено сравнение с существующими методами, основанными на извлечении моделей из исходного кода. Результаты экспериментов демонстрируют преимущества предложенного подхода.**

***Ключевые слова* — проектирование аппаратуры, функциональная верификация, статический анализ, генерация тестов, охраняемое действие, высокоуровневая решающая диаграмма, расширенный конечный автомат, проверка моделей.**

## I. Введение

Проверка функциональной корректности (верификация) является дорогостоящим и трудоемким этапом проектирования цифровой микроэлектронной аппаратуры [1]. Автоматизированные методы верификации часто используют математические абстракции свойств и поведения целевых систем, называемые *моделями*. Источниками сведений для построения моделей могут быть проектная документация (в таком случае модели разрабатываются вручную), либо исходный код целевой системы. В статье рассматриваются методы верификации аппаратуры, основанные на автоматизированном извлечении моделей из исходного кода на *языках описания аппаратуры* (Hardware Description Language, HDL) [2]. Примерами таких моделей являются конечные автоматы, решающие диаграммы и сети Петри [3].

Одним из способов определения, выполняются ли заданные формальные требования (спецификации) на модели целевой системы, является *проверка моделей* (model checking) [4]. Генерация функциональных тестов с применением инструментов проверки моделей традиционно осуществляется путем автоматического построения *контрпримеров* – последовательностей входных воздействий, приводящих к поведению, противоречащему спецификации.

При использовании инструментов проверки моделей для верификации аппаратных систем актуальной проблемой является доказательство эквивалентности целевой системы и её модели. Такая проверка не требуется, если модель автоматически извлекается из исходного кода системы и транслируется в формат инструмента проверки.

В данной работе описывается метод генерации функциональных тестов для модулей цифровой аппаратуры, основанный на автоматическом извлечении моделей в виде *высокоуровневых решающих диаграмм* (High-Level Decision Diagram, HLDD) [5] из исходного кода. Извлеченные модели автоматически транслируются в формат инструмента проверки моделей nuXmv [6]. В роли спецификаций выступают условия срабатывания переходов *расширенных конечных автоматов* (Extended Finite State Machine, EFSM), также построенных по коду. Контрпримеры, построенные инструментом nuXmv, транслируются в формат тестов, которые могут быть выполнены на HDL-симуляторе (HDL testbench).

Статья организована следующим образом. В разделе II делается обзор методов генерации функциональных тестов на основе моделей, извлеченных из исходного кода HDL-описаний. В разделе III определяются основные понятия, используемые в работе. В разделе IV описывается метод построения HLDD-моделей. В разделе V приводятся результаты экспериментов. Раздел VI завершает статью.

## II. Обзор работ

Идея построения моделей по коду HDL-описаний и последующей генерации тестов не является новой. В работе [7] представлен прототип инструмента CV проверки моделей для VHDL-описаний. Работа инструмента включает пять основных этапов. На первом этапе осуществляется синтаксический разбор VHDL-описания и построение внутреннего представления. На втором этапе выполняется построение модели на основе *двоичных решающих диаграмм* (Binary Decision Diagram, BDD). Третий этап включает разбор спецификации, выраженной в терминах *логики деревьев вычислений* (Computational Tree Logic, CTL) на специализированном языке, синтаксис которого описан в статье. На четвертом этапе выполняется подача спецификации и BDD-модели инструменту проверки CBMC [8].

Завершающий этап состоит в трансляции вывода инструмента в формат тестов, исполнимых на HDL-симуляторе. В работе подчеркивается, что компактность построенной BDD-модели является ключевым фактором для её дальнейшей проверки. Для решения проблемы «взрыва состояний» (state explosion) предлагается использовать эвристики, сокращающие размер BDD-моделей, однако сами эвристики в статье не приводятся.

В работе [9] описан метод извлечения EFSM-моделей и генерации тестовых последовательностей, обеспечивающих покрытие всех переходов автомата. Информация о том, какие переменные целевой системы задают состояние автомата, а какие – являются сигналами синхроимпульса – считается заданной. Построение EFSM-модели осуществляется в несколько этапов, заключающихся в последовательном упрощении структуры переходов между состояниями автомата (устранение вложенных условных операторов, объединение совместимых переходов, устранение зависимостей по данным между охранными условиями переходов и переменными состояния). Генерация тестов осуществляется путем обхода графа состояний с помощью техник случайного обхода и *переходов с возвратами* (backjumping).

В работе [10] предложен усовершенствованный вариант метода [9]: вместо приближенного метода проверки достижимости пути выполнения используется построение *слабейшего предусловия* [11]; при наличии нескольких тестов, покрывающих один и тот же переход графа состояний, производится их фильтрация по покрываемым переходам. Также в работе [10] предложен новый метод генерации функциональных тестов, основанный на методе [12] извлечения EFSM-моделей из HDL-описаний. Метод [12] не использует дополнительной информации о семантике переменных HDL-описания: для выявления переменных состояния и сигналов синхроимпульса используются эвристики, основанные на анализе зависимостей по данным. Результаты экспериментов показывают высокие результаты в терминах покрытия исходного кода тестами, сгенерированными методом [10] при сокращении длины тестов.

## III. Основные понятия

Предполагается, что модели, описанные в данном разделе, функционируют в дискретном времени, что неявно предполагает наличие часов. Будем называть *часами* (clock) множество типов событий $C = \{c_1,\ldots,c_k\}$, где тип события $c = \{signal, edge\}$ – это пара, включающая однобитный сигнал *signal*, называемый *синхросигналом*, и тип его регистрации *edge*: передний фронт (изменение значения с 0 на 1) или задний фронт (изменение значения с 1 на 0). Тик часов определяется наступлением события, относящегося к одному из указанных типов.

Пусть $V$ — множество переменных. Функция, которая каждой переменной ставит в соответствие значение соответствующего типа, называется *означиванием* (valuation). Пусть $Dom_V$ — множество всех означиваний на множестве переменных $V$. Охранным условием (*guard*) называется булева функция, определенная на множестве означиваний (отображение вида $Dom_V \to \{true, false\}$); действием (*action*) — преобразование означиваний (отображение вида $Dom_V \to Dom_V$). Пара $\gamma \to \delta$, где $\gamma$ — охранное условие, а $\delta$ — действие, называется *охраняемым действием* (Guarded Action, GA). В дальнейшем будем считать, что помимо семантики охранных условий и действий (задаваемых отображениями указанных видов) известен их синтаксис (что позволяет совершать над ними символические манипуляции).

Охраняемые действия будем называть *синхронизированными* [13], если с каждым из них связаны часы. Система $\{\langle C^{(i)}, \gamma^{(i)} \to \delta^{(i)}\rangle\}_{i=1,l}$ синхронизированных охраняемых действий может быть представлена в виде ориентированного ациклического графа $G = (N, E, C)$, называемого *решающей диаграммой системы охраняемых действий* (Guarded Actions Decision Diagram, GADD). Здесь $N$ – множество вершин графа, $E$ – множество ребер, $C$ – часы. Множество вершин $N$ содержит два непересекающихся подмножества (возможно, пустых): множество $N_s$ внутренних (non-terminal) вершин, помеченных выражениями $\gamma(n_s)$, $n_s \in N_s$; и множество $N_t$ листовых (terminal) вершин, помеченных действиями $a(n_t)$, $n_t \in N_t$. Ребра $e \in E$ графа $G$ помечены множествами $Val(e, n)$ допустимых значений $\gamma(n)$, где $n$ – начальная вершина ребра $e$ (здесь e – исходящее ребро вершины n, $e \in Out(n)$). Особый случай составляют ребра, помеченные меткой *default* – прохождение по ним означает, что выражение $\gamma(n)$ приняло значение, не присутствующее ни в одном из множеств, которыми помечены ребра, исходящие из $n$. В предположении, что GADD содержит ровно одну *корневую* вершину $n_{root}$ (не имеющую входных ребер, $In(n_{root}) = \varnothing$), множество путей из корневой вершины в листовые задает систему синхронизованных охраняемых действий. Так, $i$-й путь GADD, заданный вершинами $n_1^{(i)},\ldots,n_m^{(i)}$ и ребрами $e_1^{(i)},\ldots e_{m-1}^{(i)}$ ($n_1^{(i)} \equiv n_{root}$, $n_m^{(i)} \in N_t$, $e_k^{(i)} \in Out(n_u^{(i)}) \cap In(n_{u+1}^{(i)})$, $u = 1,\ldots,m\text{-}1$), задает охраняемое действие с охранным условием в виде конъюнкции $p_1^{(i)}\ldots p_{m-1}^{(i)}$ ($p_r^{(i)} = \mathrm{AND}(\gamma(n_r) == q)$, $r = 1,\ldots,m\text{-}1$, $q \in Val(e_r, n_r)$) и действием $a(n_m^{(i)})$. Часы охраняемого действия являются подмножеством часов GADD.

В работе [5] дается определение высокоуровневой решающей диаграммы (HLDD) и показывается, что каждую переменную HDL-описания можно представить в виде функции $v = f(v_1,\ldots,v_n) = f(V)$, заданной HLDD $H_v$. Пусть $Z(v)$ – конечное множество возможных значений переменной $v \in V$. *Высокоуровневой решающей диаграммой* для переменной $v$ будем называть ориентированный ациклический граф $H_v = (M, \Gamma, V)$, где $M$ – множество вершин, $\Gamma$ – отображение $M \to 2^M$. Будем называть вершину $m_2 \in M$ *последующей* по отношению к

вершине $m_1 \in M$, а $m_1$ – *предыдущей* по отношению к $m_2$, если $m_2 \in \Gamma(m_1)$. Обозначим $\Gamma(m)$ множество всех *последующих* вершин вершины $m \in M$, а $\Gamma^{-1}(m)$ – множество всех *предыдущих* вершин вершины $m$. Множество $M$ содержит два непересекающихся подмножества: внутренних вершин $M_n$ и листовых вершин $M_t$ ($M_n \cup M_t = M$, $M_n \cap M_t = \varnothing$). Назовем вершину $m_0$ графа $H_v$ *начальной*, если множество ее предыдущих вершин пусто: $\Gamma^{-1}(m_0) = \varnothing$. Внутренние вершины $m_c \in M_n$ помечены переменными $v(m_c) \in V$ и удовлетворяют следующему условию: $2 \le |\Gamma(m_c)| \le |Z(v(m_c))|$. Листовые вершины $m_k \in M_t$ помечены либо функциями $v(m_k) = f_k(V_k)$, $V_k \subseteq V$, либо переменными $v_k \in V$, либо константами. Ребра HLDD помечены множествами допустимых значений переменных, как и в GADD, либо могут быть помечены ключевым словом *default* при сходных условиях.

Для значения переменной $v(m) = e$, $e \in Z(v(m))$, активируется соответствующая дуга из вершины $m \in M$ в последующую вершину $m^e \in \Gamma(m)$. Дуги, активированные вектором $V^t$ из всех переменных $v \in V$, формируют активированный путь $l(m_0, m_k)$ из начальной вершины $m_0$ в одну из листовых вершин $m_k$, помеченную функцией $f_k(V_k)$.

## IV. Метод построения HLDD-модели и генерации тестов

Предлагаемый метод генерации тестов состоит из следующих шагов:

1) синтаксический анализ HDL-описания и генерация GADD-модели;

2) построение HLDD-модели по GADD-модели;

3) трансляция HLDD-модели и набора проверяемых свойств (спецификации) в формат инструмента проверки моделей nuXmv (SMV-модель);

4) проверка SMV-модели с помощью nuXmv, трансляция результатов в формат исполнимых тестов.

Реализация шага 1 описана в работе [12], поэтому сразу перейдем к шагу 2. Отметим, что для упрощения дальнейших преобразований все действия, которыми помечены листовые вершины GADD-модели, представлены в форме *однократных присваиваний* (static single assignment, SSA) [14].

### A. *Построение HLDD-модели*

GADD-модель и HLDD-модель сохраняют иерархическую структуру модулей целевого HDL-описания без изменений. Процессы HDL-описания в GADD-модели представляются в виде отдельных диаграмм. Псевдокод алгоритма построения набора HLDD по GADD G = (*N*, *E*, *C*) процесса HDL-описания представлен ниже:

```
proto = new;
for node ∈ N do
  hldd_node = transform_node(node);
  proto.add(hldd_node);
end
copy_edges(E, proto);
for (v : non_input_variables(G)) do
  hldd = proto.keep_assigns(v);
  hldd.add_missing_terminals();
  hldd.transform_identical_assigns();
end
```

На первом шаге алгоритма создается прототип *proto*, в который помещаются вершины HLDD, построенные из вершин GADD $G$ с помощью метода *transform_node*. Листовые вершины GADD $n_t \in N_t$ трансформируются в листовые вершины HLDD $m_k \in M_t$. Каждая листовая вершина $n_t$, помеченная действием $a(n_t)$ из $s$ присваиваний, преобразуется в последовательность $s$ вершин, каждая из которых помечена единственным присваиванием $a_d(n_t)$, $d$=1,…,$s$. Каждая построенная на данном этапе вершина HLDD помечается целевой переменной $v_k$ (левая часть присваивания $a_d(n_t)$) и функцией $f_k(V_k)$ (правая часть присваивания $a_d(n_t)$).

Внутренние вершины GADD $n_s \in N_s$ трансформируются во внутренние вершины HLDD $m_c \in M_n$. Условие $\gamma$ в вершине $n_s$ заменяется новой переменной *guard*($m_c$) в вершине $m_c$. Для переменной *guard*($m_c$) строится HLDD, состоящая из единственной листовой вершины, помеченной условием $\gamma$ (метод *create_variable_from_switch*).

Ребра GADD преобразуются в ребра HLDD с сохранением значений (метод *copy_edges*).

Затем для каждой моделируемой внутренней переменной $v$ создается HLDD *hldd*, копирующая прототип *proto*. Из множества листовых вершин $M_t$ прототипа удаляются вершины, не помеченные данной переменной (метод *keep_assigns*). К тем ребрам, у которых не осталось последующих листовых вершин, с помощью метода *add_missing_terminals* присоединяются новые листовые вершины, помеченные $f(v) = v$. Это означает, что в случае активации любого пути к данной листовой вершине итоговое значение переменной $v$ не меняется.

В *hldd* ищутся внутренние вершины $m_c$, у которых все достижимые листовые вершины помечены одной функцией $f_k(v_k)$. Метод *transform_identical_assigns* заменяет каждую найденную вершину $m_c$ и подграф, достижимый из неё, листовой вершиной, помеченной $f_k(v_k)$.

Рассмотрим пример построения HLDD-модели для описания на языке VHDL. Описание содержит модуль с единственным процессом. Интерфейс модуля содержит входные сигналы *clk*, *rst*, *x*, *y* и выходной сигнал *res* (все сигналы размерности 1 бит). В процессе объявлены переменные *cnt* (битовый вектор длины 1) и *state* (целое число, принимающее значения 0 и 1). Ниже приведен код процесса указанного модуля:

```
process (clk, rst, x, y)
  variable cnt: std_logic;
  variable state: integer range 0 to 1;
begin
if (rst = '1') then
```

```
  cnt := '0';
  state := 0;
elsif (clk = '1') then
  if (state = 1) then
    cnt := x or y;
    state := 0;
  elsif (state = 0) then
    cnt := x and y;
    state := 1;
  end if;
  res <= cnt;
end if;
end process;
```

На рис. 1 приведена GADD-модель данного процесса. Внутренние вершины GADD изображены в виде ромбов и соответствуют операторам *ветвления* в исходном коде HDL-описания, а листовые – в виде прямоугольников и соответствуют *базовым блокам* HDL-описания (непрерывным последовательностям присваиваний). У ветвлений ровно одна входящая дуга и не менее двух исходящих, у базовых блоков ровно одна входящая дуга и не более одной исходящей. Ребра, выходящие из внутренних вершин, помечены множествами допустимых значений выражений в ветвлениях. Заметим, что не все пути в GADD-модели являются достижимыми (к таковым не относится, например, путь по ребру *default*, исходящему из внутренней вершины, помеченной переменной *state*). На рис. 2 показан прототип HLDD. Условия во внутренних вершинах заменены на переменные *guard0*, *guard1*, *guard2*. Серым цветом выделены листовые вершины, помеченные переменной *cnt*.

Рассмотрим построение HLDD для переменной *cnt*. Вначале создается корневая вершина диаграммы, помеченная *cnt*. Затем создается ребро без пометок, соединяющее корневую вершину HLDD с корневой вершиной прототипа. Листовые вершины прототипа, помеченные *cnt*, получают новые значения пометок, соответствующие правым частям присваиваний (значения, присвоенные переменной *cnt*). Листовые вершины прототипа, не помеченные переменной *cnt*, удаляются. К каждому ребру прототипа, конечная вершина которого была удалена, добавляется листовая вершина, помеченная *cnt* (что означает, что на данном пути выполнения значение целевой переменной не изменяется). Результат построения представлен на рис. 3. Аналогичные диаграммы строятся для всех переменных HDL-описания, не являющихся входными сигналами.

### *B. Построение SMV-модели и ее проверка*

Полученная HLDD-модель транслируется в описание на языке SMV. Структура модулей и процессов цифрового устройства при этом сохраняется. Если в HDL-описании содержатся какие-либо ограничения на значения переменных или указаны их начальные значения, то эта информация также добавляется в SMV-модель.

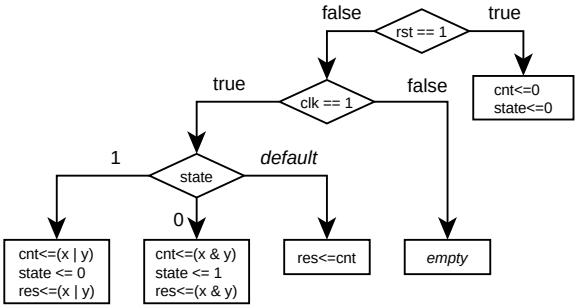


**Рис. 1. GADD-модель**

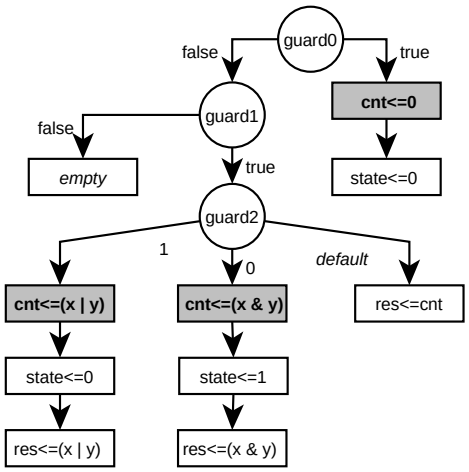


**Рис. 2. Прототип HLDD**

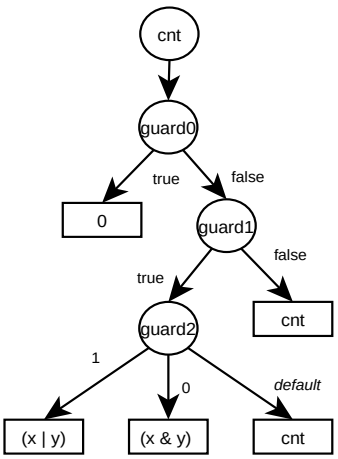


**Рис. 3. HLDD для переменной CNT**

Источником сведений для построения спецификаций в данной работе является EFSM-модель HDL-описания (формальное определение приводится в работе [12]). Здесь для краткости приведем неформальное определение. Расширенный конечный автомат является частным случаем классического *конечного автомата* (Finite State Machine, FSM),

содержащим, во-первых, множество переменных (входных, внутренних и выходных). Во-вторых, в EFSM-модели переходы между состояниями снабжены охранными условиями на значения переменных (входных и внутренних) и действиями по изменению значений переменных (внутренних и выходных). Переход EFSM может сработать, только если выполнено его охранное условие; при срабатывании перехода выполняется соответствующее действие. В предложенном методе в SMV-модель добавляются спецификации в виде отрицаний условий срабатывания переходов EFSM-модели, построенной по целевому HDL-описанию методом [12]. Отрицание используется для того, чтобы инструмент проверки моделей мог построить контрпример – последовательность входных воздействий, приводящую к состоянию данных, при котором соответствующий переход срабатывает.

Построенная SMV-модель с добавленными спецификациями проверяется с помощью инструмента nuXmv. Полученные контрпримеры транслируются в набор тестов, нацеленных на покрытие достижимых переходов EFSM-модели.

Ниже приведен результат трансляции высокоуровневой решающей диаграммы для переменных *cnt* и *guard0* в формат SMV, поддерживаемый инструментом проверки моделей nuXmv. Данное SMV-описание состоит из секции объявления переменных (VAR) и секции присваиваний (ASSIGN). Конструкция *init* определяет начальное значение переменной. Конструкция *next* определяет значение переменной в следующем состоянии модели. Присваивание (":=") определяет значение переменной в текущем состоянии модели. Представленные в примере числовые значения соответствуют битовым векторам и в формате SMV представляются в виде конструкций "0<тип><разрядность>_<значение>".

```
VAR
  cnt : word[1];
  guard0 : boolean;
…
ASSIGN
  init(cnt) := 0d1_0;
…
ASSIGN
  next(cnt) :=
    case
      (guard0 = TRUE) : 0d1_0;
      (guard0 = FALSE) :
        case
          (guard1 = TRUE) :
            case
              (guard2 = 0sd32_1) : (x | y);
              (guard2 = 0sd32_0) : (x & y);
              TRUE : cnt;
            esac;
          (guard1 = FALSE) : cnt;
        esac;
      esac;
…
guard0 := (rst = 0d1_1);
```

Приведем также пример спецификации:

**LTLSPEC** ! **F** ((state = 0sd32_0) & (clk = 0d1_1) & !(rst = 0d1_1));

В условие достижимости перехода входят ограничение на переменную *state*, определяющее начальное состояние перехода, а также охранное условие перехода, зависящее от переменных *clk* и *rst*.

## V. Результаты экспериментов

Метод генерации тестов с помощью проверки HLDD-моделей был реализован программно в инструменте HDL Retrascope 0.2.1 [15]. Разработка выполнена на языке программирования Java с использованием библиотеки средств разрешения ограничений Fortress [16]. В качестве примеров использовались HDL-описания из пакета тестов ITC'99 [17].

Для проверки моделей использовались два метода: *символический* (Symbolic Model Checking) и *ограниченный* (Bounded Model Checking) [19]. В некоторых случаях использование символической проверки моделей требовало слишком много времени и ресурсов из-за слишком большого пространства состояний модели (например, для модулей B04, B10, B11). Ограниченная проверка моделей позволила существенно сократить используемые ресурсы за счет обхода модели на определенную глубину, называемую *границей* (bound). Так как размер границы влияет на результат проверки (не для всех спецификаций при заданной границе можно построить контрпример), то в некоторых случаях пришлось итеративно увеличивать его для получения контрпримеров.

Построенные тесты сравнивались с тестами, сгенерированными с помощью существующих методов (FATE [9], RETGA [10], случайное тестирование) по следующим критериям эффективности – суммарной длине тестов и достигнутому уровню покрытия кода целевого HDL-описания. Измерение покрытия кода проводилось в терминах инструкций (statement) и ветвлений (branch) с помощью HDL-симулятора QuestaSim [18].

В табл. 1 представлено количество строк кода в целевых HDL-описаниях и полученных SMV-моделях (без учета спецификаций).

Таблица 1

*Размер HDL-описаний и SMV-моделей*

| ***Описание*** | ***HDL*** | ***SMV*** |
|---|---|---|
| B01 | 102 | 207 |
| B02 | 70 | 143 |
| B03 | 134 | 637 |
| B04 | 101 | 809 |
| B06 | 127 | 442 |
| B07 | 92 | 370 |
| B08 | 88 | 315 |
| B09 | 100 | 263 |
| B10 | 167 | 755 |
| B11 | 118 | 368 |

В табл. 2 представлены длины полученных тестов в тактах. Для метода случайного тестирования указана длина теста, при которой прекращается рост покрытия исходного кода HDL-описания (при максимально допустимой длине теста в 1 000 000 тактов).

Таблица 2

*Длина тестов в тактах*

| ***Описание*** | ***FATE*** | ***RETGA*** | ***SMV*** | ***Random*** |
|---|---|---|---|---|
| B01 | 115 | 49 | 69 | 300 |
| B02 | 62 | 33 | 47 | 80 |
| B03 | - | - | 504 | 2000 |
| B04 | 104 | 36 | 67 | 200 |
| B06 | 198 | 76 | 88 | 700 |
| B07 | 246 | 166 | 249 | 1000 |
| B08 | 31 | 52 | 31 | 1000000 |
| B09 | 19 | 231 | 84 | 1000000 |
| B10 | 173 | 135 | 134 | 650000 |
| B11 | 101 | 721 | 194 | 1000000 |

Для 6 примеров из 10 тесты, полученные с помощью описываемого метода, оказалась короче тестов, полученных методом FATE. В остальных случаях тесты либо имеют одинаковую длину, либо тесты, полученные методом FATE, достигают меньшего уровня покрытия (см. табл. 3-4). Сравнение с тестами, полученными методом RETGA, на заданном наборе примеров не приводит к однозначному выводу о преимуществе одного из методов над другим.

Заметим, что, в отличие от методов FATE и RETGA, рассматриваемый метод не основан на обходе расширенного конечного автомата. В случае описания B03 это привело к тому, что предлагаемый метод успешно сгенерировал тест, в отличие от FATE и RETGA (извлекаемая из этого модуля EFSM-модель оказалась сложна для обхода).

В табл. 3 показан процент покрытия HDL-кода по строкам в сравнении с методами FATE, RETGA и методом случайного тестирования.

Таблица 3

*Покрытие кода по строкам*

| ***Описание*** | ***FATE*** | ***RETGA*** | ***SMV*** | ***Random*** |
|---|---|---|---|---|
| B01 | 97,14% | 100% | 100% | 100% |
| B02 | 100% | 100% | 100% | 100% |
| B03 | - | - | 100% | 100% |
| B04 | 100% | 100% | 100% | 100% |
| B06 | 100% | 100% | 100% | 100% |
| B07 | 93,93% | 93,93% | 93,93% | 84,85% |
| B08 | 81,81% | 100% | 100% | 90,91% |
| B09 | 35,29% | 100% | 100% | 61,77% |
| B10 | 95,94% | 100% | 100% | 97,29% |
| B11 | 69,23% | 94,87% | 94,87% | 87,18% |

В табл. 4 приведен процент покрытия HDL-кода по ветвлениям в сравнении с методами FATE, RETGA и методом случайного тестирования.

Как видно из представленных результатов, описанный метод позволяет добиться покрытия кода, не худшего, чем метод RETGA. Заметим, что на примерах B07 и B11 ни один из предложенных методов не позволяет достичь полного покрытия по строкам и ветвлениям – это связано с наличием недостижимого кода в указанных HDL-описаниях.

Таблица 4

*Покрытие кода по ветвлениям*

| ***Описание*** | ***FATE*** | ***RETGA*** | ***SMV*** | ***Random*** |
|---|---|---|---|---|
| B01 | 96,15% | 100% | 100% | 100% |
| B02 | 100% | 100% | 100% | 100% |
| B03 | - | - | 100% | 100% |
| B04 | 100% | 100% | 100% | 100% |
| B06 | 100% | 100% | 100% | 100% |
| B07 | 94,73% | 94,73% | 94,73% | 73,69% |
| B08 | 76,92% | 100% | 100% | 84,62% |
| B09 | 35,71% | 100% | 100% | 57,15% |
| B10 | 90,47% | 100% | 100% | 97,61% |
| B11 | 71,87% | 96,87% | 96,87% | 90,63% |

## VI. Заключение

В работе представлен метод генерации функциональных тестов для HDL-описаний, основанный на автоматическом построении HLDD-моделей по исходному коду и их проверке с помощью инструмента nuXmv. Основным преимуществом представленного подхода является его универсальность в определении целей тестирования. Выбор покрытия переходов EFSM в качестве цели тестирования обусловлен необходимостью сравнения с существующими методами генерации тестов (FATE, RETGA). Сформулировав и проверив любую другую спецификацию, можно получить тест, покрывающий соответствующую функциональность целевого HDL-описания.

Эксперименты по сравнению тестов, сгенерированных с помощью предложенного подхода, с существующими аналогами, не продемонстрировали сокращения их длины при сохранении уровня покрытия. На наш взгляд, простые оптимизации (например, фильтрация тестов по покрытию переходов EFSM) могут привести к изменению данной картины.

В качестве направления для дальнейших исследований рассматривается применение подхода к более сложным примерам HDL-описаний (в том числе на языке Verilog). Показателями сложности в данном случае следует считать количество путей выполнения в отдельных процессах, а также число самих процессов и модулей. В настоящее время ведется разработка средств *декомпозиции* сложных процессов HDL-описаний на более простые с учетом

*зависимостей по данным* (dataflow), а также средств генерации тестов с использованием *предикатной абстракции* (Predicate Abstraction) [20].


## Поддержка

Исследование выполнено при финансовой поддержке РФФИ в рамках научного проекта № 15-07-03834.


## Литература


[1] Bergeron J. Writing Testbenches: Functional Verification of HDL Models. Springer, 2003. 478 p.

[2] Navabi Z. Languages for Design and Implementation of Hardware. W.-K. Chen (Ed.). The VLSI Handbook. CRC Press, 2007. 2320 p.

[3] Лазарев В.Г., Пийль Е.И. Синтез управляющих автоматов. М.: Энергоатомиздат, 1989. 328 с.

[4] Кларк Э.М., Грамберг О., Пелед Д.А. Верификация моделей программ. Model Checking. М.: МЦНМО, 2002. 416 с.

[5] Ubar R., Raik J., Jutman A., Jenihhin M. Diagnostic Modeling of Digital Systems with Multi-Level Decision Diagrams. // Design and Test Technology for Dependable Systems-on-Chip. 2011. P. 92-118.

[6] Cavada D., Cimatti A., Dorigatti M., Griggio A., Mariotti A., Micheli A., Mover S., Roveri M., Tonetta S. The nuXmv Symbolic Model Checker. // Proceedings of the 16th International Conference on Computer Aided Verification (CAV). 2014. № 8559. P. 334-342.

[7] Deharbe D., Shankar S., Clarke E.M. Model Checking VHDL with CV. // Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design (FMCAD). 1998. P. 508-514.

[8] URL: http://www.cprover.org/cbmc/ (дата обращения: 28.03.2016).

[9] Guglielmo G., 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. 2011. № 27(2). P. 137-162.

[10] Melnichenko I., Kamkin A., Smolov S. An Extended Finite State Machine-Based Approach to Code Coverage-Directed Test Generation for Hardware Designs. // Proceedings of the Institute for System Programming. 2015. № 27(3). P. 161-182.

[11] Dijkstra E. A Discipline of Programming. // Prentice Hall. 1976. 217 p.

[12] Smolov S., Kamkin A. A Method of Extended Finite State Machines Construction From HDL Descriptions Based on Static Analysis of Source Code. // St. Petersburg State Polytechnical University Journal. Computer Science, Telecommunications. № 1(212). 2015. P. 60-73.

[13] Brandt J., Gemünde M., Schneider K., Shukla S., Talpin J.-P. Integrating System Descriptions by Clocked Guarded Actions. // Forum on Design Languages. 2011. P. 1-8.

[14] Cytron R., Ferrante J., Rosen B.K., Wegman M.N., Zadeck F.K. Efficiently computing static single assignment form and the control dependence graph. // ACM Transactions on Programming Languages and Systems. 1991. № 13(4). P. 451-490.

[15] URL: http://forge.ispras.ru/projects/retrascope/ (дата обращения: 18.03.2016).

[16] URL: http://forge.ispras.ru/projects/solver-api (дата обращения: 02.04.2016).

[17] URL: http://www.cad.polito.it/tools/itc99.html/ (дата обращения: 18.03.2016).

[18] URL: https://www.mentor.com/products/fv/questa/ (дата обращения: 23.03.2016).

[19] Clarke E., Biere A., Raimi R., Zhu Y. Bounded Model Checking Using Satisfiability Solving. // Formal Methods in System Design. 2001. Vol. 19 Iss. 1. P. 7-34.

[20] Clarke E., Talupur M., Veith H., Wang D. SAT Based Predicate Abstraction for Hardware Verification. // Lecture Notes in Computer Science. 2004. Vol. 2919. P. 78-92.


# A Method of Functional Test Generation for HDL Descriptions Based on Model Checking of High-Level Decision Diagrams


M.S. Lebedev, S.A. Smolov

Institute for System Programming of the Russian Academy of Sciences, {lebedev, smolov}@ispras.ru



***Keywords* — hardware design; functional verification; static analysis; test generation; guarded action; high-level decision diagram; extended finite state machine; model checking.**


## Abstract


Functional verification is an expensive and time-consuming stage of the hardware design process. Automated methods are one of the promising directions in this research area. A number of functional test generation methods based on model extraction have been proposed recently. Automated verification methods often use abstract models of hardware. Properties of a model may be represented in the form of specifications. Model checking is one of the techniques for proving these properties. Functional tests can be obtained by parsing the counterexamples – sequences of states and variable values which violate the specification and are generated by the model checking tool.

Actual problems of functional test generation methods based on symbolic model checking techniques are state explosion and the necessity to translate the device under verification to the model checker format. The first one remains a challenging task though bounded model checking and predicate abstraction techniques can be helpful. As for the second problem, automated methods should be proposed to extract testable models from HDL descriptions and to analyze them with model checking tools.

The method proposed in this paper is based on automated extraction of high-level decision diagrams (HLDD) and extended finite-state machines (EFSM) from the HDL description. For the specified variable its high-level decision diagram is a directed acyclic graph which represents the variable's values at every condition specified in the description. A set of HLDDs for all non-input variables forms a HLDD model of the process.

Another kind of abstractions that are used in the proposed method is an EFSM model. EFSM can be treated as a finite state machine (FSM) including both a set of internal variables and Boolean conditions (called guards) on transitions between states and sequences of statements (actions) which should be executed when corresponding guards are fired. For every transition of the EFSM that is extracted from the HDL description the constraint is created that encodes its feasibility condition (safety property).

HLDD model and specifications that are extracted from EFSM model are then translated into SMV language format. SMV model is checked by the nuXmv model checker. Counterexamples generated by nuXmv are translated into HDL-based tests and can be simulated by a HDL simulator.

The proposed method was implemented as a part of the HDL Retrascope tool. Some designs from the ITC'99 benchmark were used as examples. Experimental results were compared to FATE, RETGA and random test generation methods. Comparison shows that tests obtained by the proposed method are shorter than produced by the FATE method and much shorter than generated by random test generation method. Tests generated by the RETGA method have comparable lengths to the new ones.

All tests were simulated by the QuestaSim HDL simulator. Statement and branch coverage of the original HDL descriptions was collected. Coverage results show that the proposed method provides better coverage than the FATE and random generation methods and not worse than the RETGA method.

## REFERENCES


[1] Bergeron J. Writing Testbenches: Functional Verification of HDL Models. Springer, 2003. 478 p.

[2] Navabi Z. Languages for Design and Implementation of Hardware. W.-K. Chen (Ed.). The VLSI Handbook. CRC Press, 2007. 2320 p.

[3] Lazarev V.G., Piil' E.I. Sintez upravlyayushchikh avtomatov (Control automata synthesis). Moscow, Energoatomizdat, 1989. 328 p. (in Russian).

[4] Clarke E.M., Grumberg O., Peled D.A. Verifikacija modelej programm. Model Checking (in Pussian). Moscow, MCNMO, 2002. 416 p.

[5] Ubar R., Raik J., Jutman A., Jenihhin M. Diagnostic Modeling of Digital Systems with Multi-Level Decision Diagrams. // Design and Test Technology for Dependable Systems-on-Chip. 2011. P. 92-118.

[6] Cavada D., Cimatti A., Dorigatti M., Griggio A., Mariotti A., Micheli A., Mover S., Roveri M., Tonetta S. The nuXmv Symbolic Model Checker. // Proceedings of the 16th International Conference on Computer Aided Verification (CAV). 2014. No. 8559. P. 334-342.

[7] Deharbe D., Shankar S., Clarke E.M. Model Checking VHDL with CV. // Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design (FMCAD). 1998. P. 508-514.

[8] CBMC model checker, available at http://www.cprover.org/cbmc/ (accessed 04.04.2016).

[9] Guglielmo G., 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. 2011. No. 27(2). P. 137-162.

[10] Melnichenko I., Kamkin A., Smolov S. An Extended Finite State Machine-Based Approach to Code Coverage-Directed Test Generation for Hardware Designs. // Proceedings of the Institute for System Programming. 2015. No. 27(3). P. 161-182.

[11] Dijkstra E. A Discipline of Programming. // Prentice Hall. 1976. 217 p.

[12] Smolov S., Kamkin A. A Method of Extended Finite State Machines Construction From HDL Descriptions Based on Static Analysis of Source Code. // St. Petersburg State Polytechnical University Journal. Computer Science, Telecommunications. No. 1(212). 2015. P. 60-73.

[13] Brandt J., Gemünde M., Schneider K., Shukla S., Talpin J.-P. Integrating System Descriptions by Clocked Guarded Actions. // Forum on Design Languages. 2011. P. 1-8.

[14] Cytron R., Ferrante J., Rosen B.K., Wegman M.N., Zadeck F.K. Efficiently computing static single assignment form and the control dependence graph. // ACM Transactions on Programming Languages and Systems. 1991. No. 13(4). P. 451-490.

[15] HDL Retrascope toolkit, available at http://forge.ispras.ru/projects/retrascope/ (accessed 04.04.2016).

[16] Fortress library, available at http://forge.ispras.ru/projects/solver-api (accessed 04.04.2016).

[17] ITC'99 benchmark, available at http://www.cad.polito.it/tools/itc99.html/ (accessed 04.04.2016).

[18] QuestaSim simulator, available at https://www.mentor.com/products/fv/questa/ (accessed 23.03.2016).

[19] Clarke E., Biere A., Raimi R., Zhu Y. Bounded Model Checking Using Satisfiability Solving. // Formal Methods in System Design. 2001. Vol. 19 Iss. 1. P. 7-34.

[20] Clarke E., Talupur M., Veith H., Wang D. SAT Based Predicate Abstraction for Hardware Verification. // Lecture Notes in Computer Science. 2004. Vol. 2919. P. 78-92.