Обзор методов извлечения моделей из HDL-описаний

С.А. Смолов <smolov@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. В статье дается обзор существующих методов извлечения моделей из описаний цифровой аппаратуры, разработанных на языках семейства HDL (Hardware Description Language). Методы извлечения моделей используются для решения многих задач, связанных с процессом проектирования и обеспечения качества программных и аппаратных систем. В данной работе затрагиваются методы решения следующих актуальных задач – оптимизация кода, оптимизация логического синтеза, абстракция, функциональная верификация. В статье рассматриваются методы извлечения таких семейств моделей, как графы потока и зависимостей, а также автоматные модели. Подробно рассматриваются методы построения программных срезов, конечных автоматов и расширенных конечных автоматов.

Ключевые слова: языки описания аппаратуры; извлечение моделей; статический анализ; оптимизация кода; абстракция; логический синтез; функциональная верификация; программные срезы; графы потока; графы зависимостей; конечные автоматы; расширенные конечные автоматы.

DOI: 10.15514/ISPRAS-2015-27(1)-6

Для цитирования: Смолов С.А. Обзор методов извлечения моделей из HDL-описаний. Труды ИСП РАН, том 27, вып. 1, 2015 г., стр. 97-124. DOI: 10.15514/ISPRAS-2015-27(1)-6.

1. Введение

В настоящее время цифровая аппаратура повсеместно используется во многих сферах человеческой деятельности – производственной, научной, бытовой и так далее. Появление все новых научно-технических задач и необходимость автоматизации производства приводят к постоянному росту сложности отдельных аппаратных компонентов и увеличению их количества в рамках аппаратных комплексах. Временны́е и финансовые затраты на разработку многокомпонентных аппаратных систем также возрастают, что на текущем этапе развития технологий привело к унификации процесса проектирования цифровой аппаратуры. Процесс проектирования включает следующие основные этапы: (1) составление требований, (2) архитектурное проектирование, (3) детальное проектирование, (4) логическое

проектирование (логический синтез), (5) физическое проектирование (физический синтез) [1]. Результатами каждого из этапов соответственно являются: (1) спецификация, (2) программный симулятор, (3) модель уровня регистровых передач (RTL, Register Transfer Level), (4) фотошаблон, (5) готовая микросхема.

В данной статье рассматриваются методы и инструментальные средства анализа моделей уровня регистровых передач (далее – RTL-моделей), полученных на этапе детального проектирования. Причиной служит тот факт, что последующие этапы (логический и физический синтез) автоматизированы средствами современных САПР и, следовательно, детальное проектирование является ключевым и критически важным этапом в контексте обеспечения корректности разрабатываемого аппаратного комплекса. На этапе разработки RTL-модели замеченные ошибки могут быть легко исправлены, чего нельзя сказать о результатах применения процедур синтеза. Пропущенные ошибки могут привести к летальным последствиям и серьезным материальным и финансовым потерям [2].

Модели уровня регистровых передач разрабатываются на специализированных языках описания аппаратуры (HDL, Hardware Description Language). Средствами данных языков можно описывать структуру и поведение аппаратных компонентов [3] (последнее – с потактовой точностью). Примерами языков описания аппаратуры являются Verilog [4] и VHDL [5].

В рамках процесса детального проектирования актуальными являются следующие задачи: (1) ко-симуляция (совместное исполнение моделей, разработанных на разных языках описания, а также совместное исполнение программных и аппаратных моделей), (2) оптимизация кода (например, для упрощения последующего этапа логического синтеза), (3) верификация (проверка соответствия модели её спецификации). В связи с ростом сложности аппаратных систем данные задачи не могут эффективно решаться вручную (методом экспертизы кода), что делает актуальной задачу разработки соответствующих автоматизированных средств. Распространенный подход при разработке методов и инструментальных средств анализа моделей, реализованных на языках описания аппаратуры (в дальнейшем – HDL-описаний), состоит в использовании моделей – абстрактных представлений аппаратных систем. В зависимости от поставленной задачи, уровень абстракции модели может варьироваться от воспроизведения поведения HDL-описания с потактовой точностью, до, например, предоставления информации об интерфейсе или отдельном режиме работы устройства.

В работе представлен обзор существующих методов извлечения моделей из HDL-описаний цифровой аппаратуры. Статья организована следующим образом. В Разделе 2 представлены необходимые базовые сведения о структуре HDL-описаний. В Разделе 3 делается обзор методов извлечения графов потоков и зависимостей. Раздел 4 описывает методы извлечения автоматных моделей. Раздел 5 завершает статью.

2. Структура HDL-описаний

Языки семейства HDL во многом схожи с языками императивного программирования, такими как C и Ada. В частности, используется аналогичный аппарат декларирования и использования процедур и функций, типов данных (представлены булевский и целочисленный типы, битовые векторы и массивы фиксированной длины), арифметических и логических операций (в том числе побитовых). Остановимся на основных отличиях HDL-описаний в контексте структуры и исполнения.

Основным компонентом HDL-описания является модуль (module). Модули в HDL-описаниях соответствуют элементам аппаратных систем. Каждый модуль имеет интерфейс, содержащий входные (input) и выходные (output) сигналы модуля. Как правило, интерфейс модуля содержит специальные входные сигналы, называемые сигналами синхроимпульса (clk, он же сигнал тактовой частоты) и сброса (reset). Функциональная логика модуля описывается в его теле (architecture). Тело модуля может содержать фиксированное количество экземпляров других модулей (тогда HDL-описание называется иерархическим) а также фиксированное количество процессов (process). Процессы – это сущности, исполняемые в рамках одного модуля в параллельном режиме. Каждый процесс может иметь так называемый список чувствительности (sensitivity list), являющийся подмножеством входного интерфейса модуля, а также включать исполняемый код. Ряд типов инструкций исполняемого кода имеет традиционную для императивных языков программирования семантику; это справедливо для последовательных (блокирующих) присваиваний, условных операторов и операторов цикла. Специфические для HDL-описаний инструкции, как правило, представляют сведения о временны́х ограничениях на выполнение процесса. Часто используемыми специфическими инструкциями являются инструкции ожидания (wait), приостанавливающие выполнение соответствующего процесса до выполнения определенного условия или возникновения определенного события, а также параллельные (не блокирующие) присваивания. Под событиями в HDL-описаниях понимают изменения уровня входного сигнала; различают события по приходу переднего фронта сигнала (rising edge), заднего фронта сигнала (falling edge) и произвольного фронта сигнала (event).

Важной особенностью HDL-описаний является специфический режим выполнения процессов. Во время исполнения (симуляции) каждый процесс постоянно находится в ожидании прихода события. Если возникшее в системе событие принадлежит списку чувствительности некоторого процесса, то выполняется содержащийся в нем код и процесс вновь переходит в состояние ожидания. Отличия от описанного режима исполнения возникают при наличии в коде специфических инструкций. В частности, если нить исполнения процесса достигает инструкции ожидания, то исполнение приостанавливается до тех пор, пока не произойдет событие для данной

инструкции ожидания. Что касается блоков параллельных присваиваний, то означивание определяемых в них переменных происходит параллельно, а их обновленные значения оказываются доступны только на следующей итерации исполнения кода процесса. Наличие специфических инструкций, а также парадигма исполнения HDL-описаний обуславливают необходимость разработки собственных методов извлечения моделей и затрудняют использование методов анализа программ.

Пример. 1 HDL-описание на языке VHDL

01:	**library** IEEE;	используемая библиотека
02:	**use** IEEE.std_logic_1164.all;	используемый пакет библиотеки
03:	**entity** SAMPLE **is**	модуль
04:	**port** (	интерфейс модуля
05:	PRESET : **in integer**;	входной целочисленный сигнал
06:	RST, CLK, D : **in std_logic**;	входные однобитные сигналы
07:	Q : **out std_logic**);	выходной однобитный сигнал
08:	**end** SAMPLE;	
09:	**architecture** BODY **of** SAMPLE **is**	тело модуля
10:	**begin**	
11:	**process** (RST, CLK)	объявление процесса и списка чувствительности
12:	**begin**	
13:	**if** (RST = '0') **then**	условный оператор
14:	**if** (PRESET = 0) **then**	условный оператор
15:	Q <= '0';	параллельное присваивание
16:	**else**	
17:	Q <= '1';	параллельное присваивание
18:	**end if**;	
19:	**wait on** D;	оператор ожидания
20:	Q <= D;	параллельное присваивание
21:	**end if**;	
22:	**end process**;	
23:	**end** BODY;	

Отметим, что не весь код HDL-описаний может быть обработан с помощью средств логического синтеза. Так, например, не синтезируемыми (хотя и исполнимыми с помощью специальных компонентов, называемых HDL-симуляторами) являются инструкции, выполняющие вычисления над рациональными числами и функции отладочной печати (для них не могут быть сгенерированы аппаратные компоненты, реализующие их логику). В дальнейшем будет предполагаться, что методы анализа HDL-описаний применяются на синтезируемом подмножестве инструкций данных языков. Не

синтезируемые конструкции, как правило, используются в целях отладки, в тестовых системах (testbench) или модельных примерах для оценки эффективности работы инструментов анализа и синтеза (benchmark).

Синтаксис HDL-описания на языке VDHL приведен в прим. 1.

В современных аппаратных системах размер кода HDL-описаний может составлять сотни тысяч строк. В дальнейшем такие описания, имеющие, как правило, иерархическую структуру, мы будем называть *описаниями повышенной сложности*; именно их анализ, оптимизация и верификация представляют наибольший интерес. В Разделах 3-4 делается обзор методов извлечения моделей с указанием того, имеется ли у методов инструментальная поддержка и были ли эти методы апробированы на HDL-описаниях повышенной сложности.

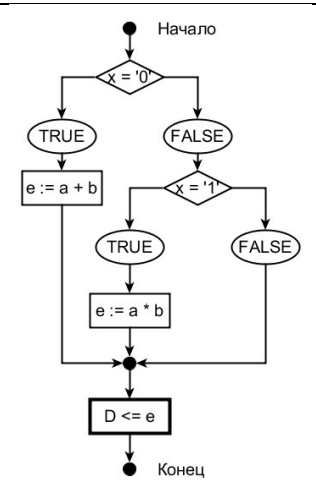
3. Графы потоков и зависимостей

Графы потоков и графы зависимостей – традиционные представления, используемые в области анализа программ [6]. Распространенными их разновидностями являются *графы потока управления* (множества всех возможных путей исполнения) и *графы зависимостей по данным* (множества переменных с наложенными на них отношениями зависимости по данным). В силу широкого распространения в программной инженерии эти модели одними из первых были адаптированы для анализа HDL-описаний. В прим. 2 приведен процесс на языке VHDL и соответствующий ему граф потока управления (CFG, Control Flow Graph). Прямоугольные вершины графа соответствуют базовым блокам, базовый блок с параллельным (не блокирующим) присваиванием выделен утолщенной рамкой. Ромбические вершины соответствуют операторам ветвления, овальные – значениям условного оператора. Круглые черные вершины соответствуют операторам слияния, начальной и конечной вершинам.

Одной из ранних работ в области статического анализа HDL-описаний, в которой фигурируют графы потока управления, является [7]. В ней описывается система SAVE (Static Analysis for VHDL Evaluation) оценивания описаний аппаратуры на языке VHDL. Система позволяет анализировать код с точки зрения эффективности его симуляции, а также оценивать сравнительную сложность различных модулей. Сравнительная эффективность симуляции различных инструкций определяется на основе жестко заданных в системе правил, а те, в свою очередь, строятся на основе обобщения экспериментальных данных. Хотя в работе указывается, что метрики определения сложности HDL-описаний должны отличаться от аналогичных метрик, используемых для программных модулей, сами отличия в статье не описаны. Графы потока управления используются в системе SAVE для визуализации HDL-описаний.

Пример 2. HDL-описание и его граф потока управления

```
01:  process(a, b, x)
02:    variable e : integer range 0 to 9;
03:  begin
04:    if (x = '0') then
05:      e := a + b;
06:    elsif (x = '1') then
07:      e := a * b;
08:    end if;
09:    D <= e;
10:  end process;
```



В работе [8] рассматриваются примеры использования методологий статического анализа, используемых в области программного обеспечения, для анализа HDL-описаний. В частности, для выявления *зависаний* (deadlocks) в наборах однотипных процессов предлагается использовать *информационные графы* (information graph) и графы зависимостей. Вершины информационного графа соответствуют процессам, а ребра – зависимостям по внутренним переменным (если переменная в одном процессе читается, а в другом - определяется). Если информационный граф не содержит циклов, то считается, что целевое HDL-описание может содержать зависания, так как процессы не могут вычислять новые значения переменных состояния на основе обновленных значений входных сигналов. Вершинами графа зависимостей являются сигналы, которыми обмениваются процессы, а ребрами – зависимости по событиям между сигналами. Наличие циклов в таком графе также сигнализирует о возможном наличии зависаний.

В статье также представлены некоторые техники анализа графа потока HDL-описаний. *Временной анализ* (timing analysis) направлен на вычисление для каждой переменной максимальной и минимальной задержек по присваиванию им значений. Для определения этих величин анализируются блоки вида *after x*, где x – величина задержки (например, указанная в наносекундах). Величина задержки присваивания значений переменной определяется как сумма величин задержек на данном пути в графе потока управления. Полученные значения задержек используются для формирования временных ограничений для генерации случайных тестовых последовательностей.

Второй техникой анализа является поиск *неявных элементов памяти* (implicit memory elements) и *прозрачных переменных* (transparent). Для этого для

каждой переменной каждого процесса HDL-описания на основе анализа всех путей в графе потока строится регулярное выражение, каждый элемент которого сигнализирует об операции над переменной. Всего существует три типа операций – присваивание (assignment), чтение (reading), ожидание (waiting). Неявные элементы памяти и прозрачные переменных определяются посредством установления соответствия между извлеченными регулярными выражениями и заданными шаблонами (например, в случае прозрачных переменных необходимо, чтобы регулярные выражения содержали только использования их значений). По словам авторов статьи, определение неявных элементов памяти и прозрачных переменных способно упростить процесс верификации HDL-описаний.

Предложенные в статье методы поясняются на модельных примерах небольшого размера и не имеют инструментальной поддержки.

В статье [9] описывается метод предварительной оптимизации HDL-описаний на языке VHDL, основанный на построении *структуры данных системы* (SDS, System Data Structure). Структура данных системы состоит из *графа потока данных* (Data Flow Graph), *графа управления временными свойствами* (Control Timing Graph) и *связей* (bindings) между ними. Процедура построения представления, предварительно оптимизированного для последующего логического синтеза, содержит следующие шаги:

- построение графа потока управления (используется инструмент VAUL [10]) и выделение базовых блоков;
- анализ потоков данных (на первом этапе – выявление зависимостей по чтению и записи переменных на уровне инструкций для каждого базового блока, на втором этапе – на уровне графа);
- построение графов потоков данных для каждой переменной и графа управления временными свойствами для всего HDL-описания;
- составление очередности выполнения инструкций (scheduling).

Метод реализован в прототипе инструмента VHDL2SDS. Инструмент поддерживает следующие алгоритмы очередности выполнения инструкций: «как можно раньше» (As Soon As Possible, ASAP), «как можно позже» (As Late As Possible, ALAP). Прототип инструмента был апробирован на простых описаниях на языке VHDL.

В работе [11] предложен метод проверки выполнения свойств (в частности, определения недостижимых путей в графе потока управления) для HDL-описаний на языке Verilog. Для этого строится вспомогательная структура данных, называемая графом зависимостей путей (Path Dependency Graph). В граф зависимостей путей попадают только те пути, в которых читаются или

определяются переменные, присутствующие в проверяемом свойстве. Затем методом обратных подстановок [12] (подход предлагается для выделенного класса HDL-описаний, содержащих ровно один процесс), определяется условие попадания в данный путь выполнения.

В работе [13] описывается метод поиска состояний гонки (races) в описаниях цифровой аппаратуры на языке SystemC [14]. Метод основан на построении графов потока управления и анализе процессов на коммутативность с помощью техники предикатной абстракции CEGAR [15] и инструмента проверки моделей Cadence SMV [16]. Инструментальная реализация SCOOT [17] предложенного метода основана на особенностях языка SystemC и его компонента выполнения SystemC Sheduler, поэтому адаптация данного метода для анализа описаний на других языках представляется трудоемкой. Инструмент SCOOT не апробировался на описаниях повышенной сложности.

В работе [18] описывается метод масштабируемой генерации тестовых стимулов на основе гибридной верификации HDL-описаний. Под гибридной верификацией в данном случае понимается комбинация статического анализа кода и имитационного верификации. В качестве входных данных методу подается целевой Verilog-модуль и длина тестовой последовательности. Предполагается, что модуль не содержит гонок и имеет только один входной сигнал синхроимпульса. Под отсутствием гонок следует понимать условие на определения переменных: каждая переменная должна определяться не более одного раза за один такт работы модуля. Длина тестовой последовательности задается вручную; её конкретное значение может быть задано на основе анализа покрытия кода или требований имеющимися тестовыми последовательностями, реализованными, например, методом случайной генерации.

Генерация тестовой последовательности осуществляется на модели вида графа потока управления. В предположении, что все пути выполнения в графе потока однотактные (каждый путь может быть пройден за один такт синхроимпульса), метод строит упорядоченную последовательность графов потока. Количество копий графов потока в последовательности задается равным длине тестовой последовательности; таким образом, для каждой тестовой последовательности, выполняемой за n тактов, существует путь в упорядоченной связанной последовательности графов потока.

На начальном этапе выполняется случайная генерация значений входных сигналов целевого модуля. Для сгенерированных значений выполняется симуляция модуля с параллельным сохранением информации о том, какой путь выполнения был покрыт. Для покрытого пути составляется ограничение в терминах текущих значений переменных (для блокирующих присваиваний) и значений переменных, которые будут им присвоены на следующем такте (для не блокирующих присваиваний). Информация об ограничениях и номере текущего такта сохраняется посредством инструментирования HDL-описания. Ограничения хранятся в стеке (вершиной стека является ограничение,

построенное в результате анализа последней инструкции HDL-описания, а, соответственно, дном стека является ограничение на первую инструкцию). Далее методом поиска в глубину выполняется инвертирование ограничений, начиная с вершины стека. Инвертируются только ограничения, извлеченные из условных блоков, ограничения на переменные, определяемые в блоках присваиваний, не изменяются, а служат только для генерации корректных тестовых воздействий. Если модифицированное таким образом ограничение оказывается неразрешимым, то оставляется исходное ограничение и инвертируется следующий элемент стека ограничений.

Узким местом данного подхода является необходимость частых обращений к решателю ограничений для проверки выполнимости каждого пути. Кроме того, в HDL-описаниях повышенной сложности количество путей выполнения может быть весьма велико и приводить к «комбинаторному взрыву» [19], т.е. делать их перебор чрезмерно длительной процедурой.

Для уменьшения числа путей выполнения, необходимых для генерации покрывающих тестовых последовательностей, предлагается использовать символические состояния. Символические состояния – это наборы ограничений на все внутренние переменные целевого модуля. Если на некотором такте две сгенерированные тестовые последовательности приводят к одному и тому же символическому состоянию, то избыточно генерировать для каждого из них все возможные пути дальнейшего выполнения. В методе предлагается для каждого построенного пути выполнения на каждом такте хранить достигнутое символическое состояние. Если для последующего пути на некотором такте будет достигнуто состояние, уже достигнутое ранее, то дальнейшее построение пути прекращается.

Большое внимание в данной работе уделяется вопросам эффективного хранения символических состояний и их сравнения без использования решателей. Предлагается хранить состояния в виде множеств инструкций ветвления в графе потока, по которым прошел путь, достигающий данного состояния. Тогда сравнение символических состояний сводится к сравнению множеств и может быть выполнено без использования решателя. В работе отмечается, что такой подход не позволяет выявлять эквивалентные ограничения, построенные путем обхода разных путей выполнения. Дополнительные оптимизации, описанные в методе (анализ зависимостей по данным для построения упрощенных ограничений, выявление заведомо противоречивых ограничений при инвертировании), направлены на уменьшение числа обращений к решателю. Метод реализован в проекте STAR [20] (STatic Analysis of RTL) и апробирован на HDL-описаниях небольшой сложности (не более 1500 строк кода в каждом). Эксперименты показали, что тесты, сгенерированные инструментом STAR, обеспечивают более высокое покрытие ветвей и путей, чем случайная генерация при меньшей длине тестов (в ряде случаев отмечено уменьшение длины тестовых последовательностей на три порядка).

В работе [21] предложен гибридный подход к генерации утверждений (assertion) с помощью техник статического анализа кода и анализа данных (data mining). В данном случае под анализом данных следует понимать выявление статистических закономерностей в изменениях значений выходных сигналов. На первом этапе метода выполняются прогон набора случайных тестов на целевом HDL-описании и сбор сведений о значениях, принимаемых выходными сигналами. На втором этапе выполняется анализ зависимостей по данным между переменными. В результате для каждой переменной, для которой предполагается сгенерировать утверждение, строится так называемый конус влияния [22] (cone-of-influence) – множество переменных и входных сигналов, значения которых определяют значение переменной. На третьем этапе выполняется статистический анализ данных, полученных в результате выполнения случайных трасс, с целью выявления статистически достоверных взаимосвязей между элементами конуса влияния и заданными переменными. Выявленные взаимосвязи формулируются в виде утверждений в терминах линейной темпоральной логики. Метод был реализован в инструменте GoldMine [23], построенные утверждения были проверены с помощью коммерческого инструмента формальной верификации Cadence Incisive [24], а также использованы для создания регрессионных тестов. Эксперименты демонстрируют применимость предложенного подхода и его программной реализации для верификации HDL-описаний повышенной сложности. Однако построенные инструментом утверждения могут содержать ошибки, т.к. они извлекаются из исходного кода, а не из спецификаций.

3.1 Программные срезы

Важным направлением в области статического анализа программного кода вообще и HDL-описаний в частности является извлечение и анализ программных срезов (program slicing). Срезом программы является подмножество её инструкций, которые могут привести программу к набору состояний (точек), называемому критерием среза (slicing criterion). Программные срезы можно рассматривать как подграфы графа потока управления исходного HDL-описания.

Исторически, первыми были разработаны техники извлечения срезов для программ на процедурных языках [25]. Одной из первых работ по извлечению программных срезов HDL-описаний является [26]. В ней предложен способ адаптации техник построения срезов процедурных программ для описаний на языке VHDL. Авторами статьи была разработана программная реализация подхода (с использованием коммерческого инструмента Codesurfer [27]), а также предложены варианты его дальнейшего использования для проектирования, тестирования (имитационной верификации, основанной на анализе результатов исполнения HDL-описания в симулируемом окружении) и формальной верификации (математически строгого доказательства выполнения свойств).

В работе [28] предложен подход к извлечению срезов для описаний на языке Verilog. Программная реализация подхода представляет собой надстройку над коммерческим Verilog-симулятором, получающую на вход HDL-описание и критерий среза, задаваемый пользователем. Компонент анализирует Verilog-код с помощью интерфейса VPI [29], строит исполнимый срез, генерирует упрощенное Verilog-описание и передает его симулятору для исполнения. Инструмент был апробирован на описаниях, часть кода которых содержали управляющие конечные автоматы (FSM, Finite State Machine). В качестве критериев среза задавались внутренние переменные, определяющие состояние. Полученные на выходе автоматы состояний вручную сравнивались со спецификациями соответствующих аппаратных компонентов.

В работе [30] предложено развитие подходов к извлечению срезов в HDL-описаниях посредством наложения дополнительных ограничений (в виде предикатов логики первого порядка) на начальные значения выбранных переменных или входных сигналов целевой системы. В таком случае критерий среза называется условным критерием среза (conditioned slicing criterion). С точки зрения верификации, интересным классом условий являются требования на входные стимулы и возвращаемые реакции, выраженные в терминах линейной темпоральной логики [31] (LTL, Linear Temporal Logic). Для такого класса условий вводится понятие предшествующего условного среза (antecedent conditioned slice) как последовательности точек программы, в каждой из которых выполнен условный критерий среза. Предложенный метод был апробирован на Verilog-реализации протокола USB 2.0. По спецификации протокола был составлен набор свойств, заданных в терминах логики LTL. Для каждого свойства извлекался предшествующий условный срез, который проверялся на выполнимость с помощью инструмента проверки моделей Cadence SMV [16]. Использование техники извлечения условных срезов существенно сократило время верификации и не исказило конечный результат.

Статья [32] посвящена использованию срезов для локализации ошибок. Предполагается, что для некоторого иерархического HDL-описания уже имеется набор функциональных тестов, причем каждый тест был выполнен на искомом HDL-описании хотя бы один раз и завершился успешно или с ошибкой. Предложенный метод локализации ошибок основан на извлечении срезов из графа экземпляров (IG, Instantiation Graph) HDL-описания и анализе покрытия инструкций среза имеющимися тестами при динамической верификации. Метод реализован в открытой расширяемой среде проектирования и верификации ZamiaCAD [33] и был применен для выявления ошибочных инструкций в коде процессора ROBSY.

4. Автоматные модели

В теории алгоритмов под абстрактным автоматом понимается пятерка $A = (S,I,O,\delta,\lambda)$, где S – конечное множество состояний, I и O – соответственно

входной и выходной алфавиты, из символов которых формируются строки, считываемые и выдаваемые автоматом, $\delta : S \times X \rightarrow S$ – функция переходов, $\lambda : S \times X \rightarrow Y$ – функция выходов [34]. Автоматные модели широко применяются в верификации программных систем, в том числе таких критически важных, как реализации телекоммуникационных протоколов [35] и операционные системы реального времени [36]. В области аппаратных систем автоматные модели (в частности, конечные автоматы, речь о которых пойдет далее) активно используются в качестве эталонных моделей при имитационной верификации [37]. В данном разделе речь пойдет о методах извлечения автоматных моделей из HDL-описаний.

Здесь уместно отметить, что существующие коммерческие инструменты поддержки проектирования цифровой аппаратуры зачастую тоже используют техники построения моделей. В частности, особое внимание уделяется извлечению моделей вида конечных автоматов (FSM, Finite State Machine). Как правило, данные техники основаны на выявлении в коде специальных шаблонов и конструкций, заранее определенных в документации для описания конечно-автоматных моделей. Например, в работе [38] представлено описание стиля программирования конечных автоматов, принятого в компании Cisco. Использование аналогичных подходов с одной стороны, позволяет эффективным образом строить и анализировать модели даже для описаний повышенной сложности. С другой стороны, использование коммерческих инструментов приводит к необходимости следовать жестко заданным правилам проектирования во всех реализуемых проектах и порождает зависимость процесса проектирования от выбранной технологии.

4.1 Конечные автоматы

В данном разделе представлены существующие методы извлечения моделей типа конечных автоматов. Абстрактный автомат называется конечным, если конечно его множество состояний [39].

В работе [40] предлагается новый метод логического синтеза HDL-описаний в более низкоуровневое представление в виде списка соединений (netlist). В качестве промежуточных стадий метод содержит построение графа потока управления для каждого процесса, а также извлечение конечного автомата (далее – FSM-модели) для каждого графа потока управления. Извлечение FSM-моделей возможно только в том случае, если граф потока управления процесса содержит инструкции wait, зависящие от одного и того же события, связанного с одним и тем же сигналом синхроимпульса. Каждой инструкции wait, обладающей указанным свойством, в FSM-модели сопоставляется состояние. Переходы конечного автомата строятся на основе наличия путей между wait-инструкциями в графе потока управления.

Работа [41] также посвящена решению задачи логического синтеза HDL-описаний в список соединений. В качестве промежуточного этапа используется выявление блоков типа FSM-моделей в исходном коде. Авторы

подхода экспериментально доказывают, что список соединений, сгенерированный на основе комбинации блоков типа конечных автоматов, и блоков, содержащих комбинационную логику, обладает меньшим временем выполнения. Блоки типа конечных автоматов определяются путем сопоставления с шаблонами, заданными авторами подхода. Всего представлено три шаблона конечно-автоматных блоков, каждых из которых содержит циклические зависимости по управлению между блоками, определяющими переменную состояния, и блоками, выполняющими промежуточные вычисления (комбинационная логика). Метод дает нетривиальные результаты, если в исходном HDL-описании есть процессы, имеющие непустые списки чувствительности, а также содержащие ровно одну переменную состояния. Поддерживается иерархическая структура HDL-описаний, никаких других ограничений на исходный код не накладывается.

В работе [42] предложен метод оптимизации логического синтеза за счет извлечения конечных автоматов из описаний на языках VHDL и Verilog. В основе метода лежит классификация процессов целевого HDL-описания по следующим множествам: процессы, выполняемые при возникновении событий над синхросигналами (clock); процессы, содержащие переходы между состояниями конечного автомата (transition); процессы, генерирующие выходные сигналы (output). Предполагается, что конечный автомат может быть «размыт» по процессам различных классов. На код HDL-описания накладывается существенное ограничение: описание должно содержать только FSM-модель. Также должны быть заранее известны переменные состояния, синхросигналы и сигналы сброса.

В работе [43] предложен поход к выявлению в коде HDL-описаний внутренних переменных, задающих состояния конечных автоматов. Авторы утверждают, что протоколы взаимодействия между компонентами индустриальных HDL-описаний зачастую удобно выражать именно в виде FSM-моделей. Предполагается использовать найденные в исходном коде конечные автоматы для декомпозиции HDL-описаний на более простые для верификации подсистемы, а также для оптимизации логического синтеза и, возможно, снижения энергопотребления синтезированных аппаратных систем. Предложенный метод определения переменных состояния состоит из следующих этапов:

- построение абстрактного дерева синтаксиса [14];
- построение графа потока данных на уровне сигналов;
- построение графа зависимостей между внутренними переменными;
- определение переменных состояния.

На первом этапе используется классический подход [44]. На втором этапе строится граф потоков данных между параллельно выполняемыми

компонентами HDL-описания. Вершинами графа являются компоненты, ребрами – зависимости. Существует четыре типа ребер, соответствующих зависимостям по *управлению* (control), по *данным* (data), по *синхросигналу* (clock) и по *сбросу* (reset). В работе не описано, как идентифицируются зависимости по синхросигналу и по сбросу. Зависимости по управлению определяются как использования переменных, присутствующих в условных операторах. В остальных случаях зависимости трактуются как зависимости по данным. На третьем этапе строится граф зависимостей между внутренними переменными (это уточнение графа потока данных до уровня отдельных сигналов). Четвертый этап заключается в выявлении в графе переменных, удовлетворяющих следующим свойствам:

- хотя бы одна исходящая дуга для данной переменной является циклической;
- хотя бы одна исходящая дуга для данной переменной задает зависимость по управлению;
- все входные зависимости по данным являются зацикленными на себя (self-loop), либо являются зависимостями от констант.

Если для некоторой внутренней переменной все условия выполнены, то переменная является переменной состояния FSM.

Предложенный метод был реализован на языке C++ в инструменте Asynchronous Verilog Synthesizer [45] и апробирован на нескольких промышленных Verilog-описаниях. Утверждается, что метод дает ложные срабатывания (в качестве состояний иногда определяются переменные, таковыми не являющиеся). Однако, процент таких срабатываний невысок (на трех проанализированных системах он не превышал 10%). Процедура проверки корректности извлеченных переменных состояния в работе детально не описана.

В работе [46] описан метод извлечения конечных автоматов из описаний на языке Verilog. Предлагается анализировать построенные конечные автоматы автоматически с помощью инструмента проверки моделей NuSMV [47]. Метод позволяет преобразовывать только HDL-описания, реализующие логику FSM. Концептуальной особенностью метода является автоматическое определение внутренних переменных, задающих состояние конечного автомата. В качестве таковых определяются переменные, чтение которых выполняется синхронно с возникновением события над синхросигналом (например, приход переднего фронта), а значения циклическим образом зависят от них самих в смысле зависимостей по данным.

4.2 Расширенные конечные автоматы

Значимым в контексте применимости к моделированию аппаратных комплексов частным случаем автоматных моделей являются *расширенные конечные автоматы*. В дополнение к свойствам конечного автомата, расширенный конечный автомат (EFSM, Extended Finite State Machine) имеет собственные переменные (входные, выходные и внутренние), а его переходы имеют особую структуру. Каждый переход содержит *охранное условие* (булевский предикат) и *действие* (последовательность инструкций). Переход срабатывает только в том случае, если выполнено охранное условие, тогда выполняется действие.

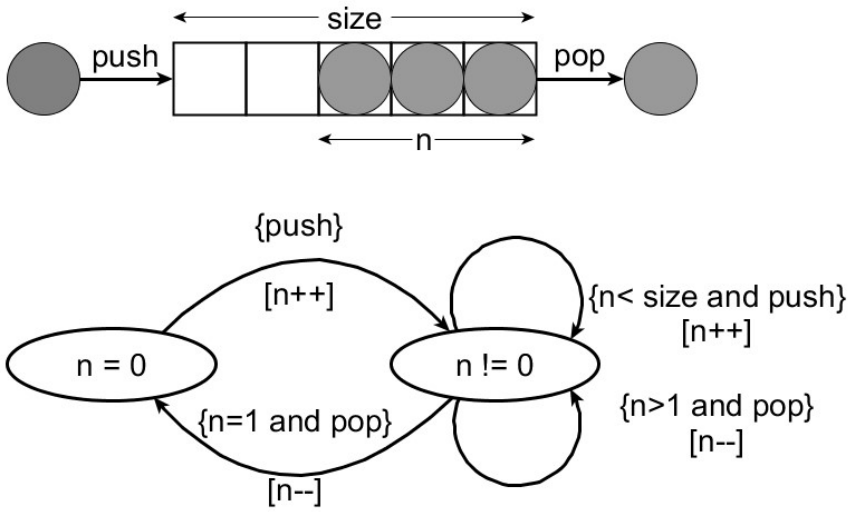


Рис. 1. Пример расширенного конечного автомата

На рис. 1 представлен пример простой очереди (FIFO, First In First Out) и расширенного конечного автомата, моделирующего её поведение. Очередь может принимать на вход воздействия типа push (добавление элемента) и pop (удаление элемента). Размер очереди равен size, а текущее значение степени заполнения – n. Расширенный конечный автомат имеет два состояния, соответствующих пустой и непустой очереди. Для каждого перехода указаны охранные условия (в фигурных скобках) и действия (в квадратных скобках).

В работе [48] описан метод автоматической генерации тестовых последовательностей для HDL-описаний на языках VHDL и BESTMAP-C [49]. Тестовые последовательности нацелены на выявление *постоянных ошибок* (stuck-at faults) типа разрыва цепи, заключающихся в постоянной генерации неверных значений сигналов HDL-описания. Процедура генерации состоит из двух фаз. На первой фазе выполняется автоматизированное извлечение

расширенных конечных автоматов из исходного кода; на второй фазе производится генерация тестовых последовательностей, покрывающих состояния и ветви полученного автомата. Процедура извлечения EFSM-моделей работает при следующих ограничениях:

- заранее определены переменные, задающие состояние HDL-описания;
- тела процессов анализируются по отдельности, возможная иерархическая структура HDL-описаний никак не учитывается, равно как и список чувствительности процесса;
- каждый путь исполнения в HDL-описании содержит две непрерывных (возможно, пустых) и непересекающихся последовательности блоков – условных операторов и операторов присваивания соответственно.

В статье детально обсуждаются процедуры *ортогонализации* охранных условий (enabling function) на пути HDL-описания и стабилизации (т.е. сокращения до одного элемента) множества выходных состояний для каждого перехода EFSM-модели. Однако способы сравнения состояний (для определения эквивалентных) и выявления недостижимых переходов авторами опускаются. Метод генерации функциональных векторов описан неформально. Недостатком метода является экспоненциальный рост числа состояний при стабилизации.

В работе [50] предложен метод автоматизированной генерации модульных функциональных тестов для HDL-описаний на языке VHDL. В основе метода лежит генерация по исходному коду моделей в виде EFSM и генерации по ним тестов методом обхода с помощью техники переходов с возвратами (backjumping). Предполагается наличие дополнительной информации о том, какими переменные задают состояние искомого модуля, а какие – являются синхросигналами.

Первый шаг метода состоит в извлечении исходной модели (REFSM, Reference EFSM), функционально эквивалентной HDL-описанию. Исходная модель всегда содержит ровно одно символическое состояние, а все переходы модели (соответствующие путям выполнения HDL-описания с точностью до выбора ветви в операторах switch/case) являются замкнутыми на него. Следующий шаг метода направлен на устранение условных операторов типа if-then-else из действий переходов. Это свойство достигается путем расщепления действий переходов на более простые, не содержащие условные операторы. Расщепление состоит во введении дополнительных символических состояний, поэтому построенная на данном этапе модель носит название наибольшей модели (LEFSM, Largest EFSM). Помимо увеличения числа состояний, наибольшая модель также обладает следующим недостатком – она не является функционально эквивалентной HDL-описанию. Устранение

указанных недостатков происходит на следующем шаге метода, где выполняется объединение совместных переходов. Под совместными переходами понимаются такие, что их охранные условия могут принимать истинное значение независимо друг от друга, а также не имеющие общих переменных. В данном случае под общими переменными для пары переходов подразумеваются такие, которые определяются в действии первого перехода и используются в охранном условии второго перехода. Если данные условия не выполнены, и переходы объединять нельзя, однако для достижения функциональной эквивалентности это необходимо, то допускается исправление кода HDL-описания вручную с тем, чтобы комбинируемые переходы не содержали общих переменных. Последний шаг метода состоит в построении полу-стабилизированной EFSM-модели (S2EFSM, semi-stabilized EFSM). На данном шаге из охранных условий переходов выделяются ограничения на переменные состояния; эти ограничения сопоставляются построенным символическим состояниям. Предложенный метод был использован для построения на его основе генератора модульных функциональных тестов и был апробирован на HDL-описаниях небольшой сложности.

В работе [51] предлагается использовать EFSM-модели для выполнения полуформальной (semi-formal) верификации HDL-описаний. Подход использует техники имитационной верификации с использованием случайной генерации стимулов на основе ограничений (constrained random simulation), обхода состояний EFSM-модели и анализа достижимости граничных случаев с помощью инструмента проверки свойств NuSMV.

Целью извлечения моделей из HDL-описаний может являться абстрагирование – представление аппаратуры на более высоком уровне, чем потактовое описание поведения. Техника абстрагирования применяется для ускорения симуляции [52], а также для ко-симуляции представлений аппаратуры, описанных на разных уровнях абстракции. В работе [53] представлен метод трансляции HDL-описаний в представление уровня транзакций (TLM, Transaction-level Modeling) [54] с целью их совместной симуляции с модулями, уже реализованными в терминах TLM. Основными примитивами в TLM-представлении являются сообщения (наборы сигналов), которыми модули обмениваются по установленным между ними каналам. Последовательности обменов сообщениями носят название транзакций. Предложенный метод трансляции HDL-описаний основан на построении EFSM-моделей в качестве промежуточного представления (для этого используется метод [49]). Полученные расширенные конечные автоматы разделяются на под-фазы, соответствующие чтению входных сигналов (input subphase), выполнению промежуточных вычислений (elaboration subphase) и генерации выходных сигналов (output subphase). Разделение выполняется по результатам анализа потоков данных. Полученные под-фазы преобразуются в макро-состояния EFSM-модели (логика охранных условий и действий в

переходах под-фаз сохраняется в действиях переходов EFSM-модели), которые затем автоматически транслируется в TLM-представление. На завершающем этапе генерации TLM-представления во входное макро-состояние добавляется примитив TLM-транзакции чтения, а в выходное макро-состояние – примитив TLM-транзакции записи. Метод реализован в виде компонента среды HIFSuite [55] и апробирован на описаниях небольшой сложности. Результаты демонстрируют уменьшение времени симуляции при значительном сокращении времени разработки TLM-модели по сравнению с разработкой вручную.

Работа [56] содержит описание метода извлечения EFSM-моделей из описаний на языках Verilog и VHDL. Метод принимает на вход только HDL-описание, никакой дополнительной информации для анализа не требуется. На первом этапе строится граф потока управления. На втором этапе граф преобразуется в систему охраняемых действий посредством «подъема» внутренних условных операторов на верхний уровень. Под охраняемым действием в данном случае понимается пара «охранное условие - действие»; данное представление активно используется для описания асинхронных систем [57]. Затем с помощью эвристик определяются переменные синхроимпульса и переменные состояния. Состояния EFSM-модели строятся в виде ограничений на переменные состояния, построенные путем ортогонализации охранных условий. Переходы EFSM-модели строятся путем проверки совместности условий охраняемых действий (для начальных вершин переходов) и слабейших предусловий (для конечных вершин) с условиями, задающими состояния. Предложенный метод был реализован в инструменте Retrascope [58] и апробирован на HDL-описаниях небольшой сложности.

4.3 Прочие автоматные модели

Статья [59] посвящена техникам абстракции представлений цифровой аппаратуры. Предлагаемый подход строит на основе конечного автомата и сведений о переменных автомата более компактную структуру данных, называемую автоматом потока управления (ECFM, Extracted Control Flow Machine). Под сведениями о переменных следует понимать классификацию на переменные управления (задающие состояние автомата) и регистры данных. Предполагается, что такие сведения предоставляются разработчиком HDL-описания. Регистры данных, как правило, имеют большую размерность, и именно они зачастую являются причиной экспоненциального роста числа состояний в конечных автоматах. Построенный автомат потока управления предлагается использовать для оценки покрытия имеющихся тестовых наборов (в данной работе речь идет не о функциональных тестах, а о тестах готовой продукции). Используются две основные метрики покрытия – покрытие состояний ECFM-модели и переходов ECFM-модели. Если обнаруживается переход или состояние ECFM-модели, не покрываемые существующим тестовым набором, то генерируется тест, нацеленный на

покрытие указанного компонента. Поскольку тесты, построенные для более абстрактной, нежели исходное HDL-описание, ECFM-модели, в общем случае не могут быть корректно применены к исходной модели, задача генерации нацеленных тестов возлагается на сторонние инструменты (в статье предлагается использовать HITEC [60]).

Работа [61] тоже посвящена проблеме абстракции HDL-описаний. В частности, в ней описан метод трансформации представлений вида конечных автоматов в так называемые семантические конечные автоматы (SFSM, Semantic Finite State Machine). По сравнению с FSM-моделями, SFSM-модель обладает следующей особенностью: каждый переход между состояниями снабжен булевским условием на входные сигналы и переменные состояния (enabling function); функцией вычисления конечного состояния, зависящей от входных сигналов и переменных состояния (update function); функцией вычисления значений выходных сигналов, зависящей от входных сигналов и переменных состояния (action function). Состояния SFSM-модели называются семантическими и определяются наборами переменных состояния исходной FSM-модели. Поскольку SFSM-модель не хранит сведений о конкретных значениях входных сигналов и переменных состояния, размер пространства состояний в ней не превосходит (а в ряде экспериментов оказывается значительно меньше) размера пространства состояний FSM-модели, что потенциально ускоряет её функциональное тестирование на основе обхода графа состояний. Метод построения SFSM-модели по FSM-модели состоит из следующих шагов:

- построение *дерева инструкций* (statement tree);
- построение *семантических состояний*;
- вычисление булевских условий переходов;
- построение переходов SFSM-модели.

Рассмотрим шаги метода более подробно. Построение дерева инструкций проводится посредством обработки *синхронной секции* (synchronous section) FSM-модели, включающей функции вычисления следующего состояния и выходных сигналов. Нетерминальные вершины дерева содержат взаимоисключающие условия на входные сигналы и переменные состояния. Листья дерева содержат функции вычисления переменных состояния и выходных сигналов. На этапе построения семантических состояний анализируются функции вычисления, заданные в листьях дерева инструкций. Если для нескольких листовых вершин соответствующие функции эквивалентны, то листовые вершины объявляются соответствующими одному и тому же семантическому состоянию. Условия переходов вычисляются как конъюнкции условий, заданных в нетерминальных вершинах дерева, для каждого пути в дереве. Построение SFSM-модели завершается тем, что для

каждого семантического состояния определяются допустимые исходящие переходы, посредством проверки их булевских условий. Переходы с противоречивыми условиями отбрасываются.

Приведенные результаты экспериментов показывают, что тестовые наборы, построенные методом обхода графа состояний SFSM-модели, могут обеспечивать более высокое покрытие инструкций исходного кода, чем наборы случайных тестов, даже при условии существенно бóльшего размера последних.

5. Заключение

В статье сделан обзор существующих методов извлечения моделей из исходного кода описаний цифровой аппаратуры (HDL-описаний). Рассмотренные методы сгруппированы по типам моделей: программные срезы, графы потока, конечные автоматы. Возможна также классификация методов по иному критерию, а именно - по решаемым с их помощью задачам (оптимизация кода ([7], [9], [40], [41], [42], [43]), абстракция ([53], [59], [61]), тестирование ([8], [11], [13], [18], [21], [26], [28], [30], [32], [38], [43], [46], [48], [50], [51], [56])). Ключевыми проблемами существующих подходов можно считать необходимость получения дополнительной информации от пользователя для построения моделей, более адекватно представляющих те или иные аспекты целевой аппаратной системы, а также трудности при работе с HDL-описаниями повышенной сложности. В некоторых случаях программные реализации методов построения моделей являются существенно зависимыми от особенностей языка описания аппаратуры ([13]).

Вместе с тем нельзя не отметить тот факт, что разные типы моделей демонстрируют разную эффективность в решении разных задач. К примеру, в контексте функциональной верификации методы построения программных срезов демонстрируют высокие результаты в задачах генерации нацеленных тестов, в то время как методы построения расширенных конечных автоматов позволяют генерировать тестовые последовательности, обеспечивающие полное покрытие исходного кода. Несомненно, актуальной задачей является разработка программных систем, обеспечивающих интеграцию различных подходов к построению и преобразованию моделей цифровой аппаратуры.

Список литературы

[1]. А.С. Камкин, А.М. Коцыняк, С.А. Смолов, А.А. Сортов, А.Д. Татарников, М.М. Чупилко. Средства функциональной верификации микропроцессоров. Труды ИСП РАН, т. 26, вып. 1, 2014 г., стр. 149-200. doi: 10.15514/ISPRAS-2014-26(1)-5

[2]. Statistical Analysis of Floating Point Flaw: Intel White Paper Section 3. http://www.intel.com/support/processors/pentium/sb/CS-013007.htm. Дата публикации: 08.07.2004.

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

[4]. IEEE Standard for Verilog Hardware Description Language, 1364-2005, IEEE, 2006, 560 p.
[5]. IEEE Standard VHDL Language Reference Manual, 1076-2008, IEEE, 2009, 626 p.
[6]. A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman. Compilers: Principles, Technologies, and Tools (2nd Edition). Addison Wesley, 2006. 1000 p.
[7]. A. Balboni, M. Mastretti, M. Stefanoni. Static Analysis for VHDL Model Evaluation. Proceedings of the European Design Automation Conference (EURO-DAC '94), IEEE Computer Society Press, Los Alamitos, 1994, pp. 586-591.
[8]. L. Baresi, C. Bolchini, D. Sciuto. Software Methodologies for VHDL Code Static Analysis based on Flow Graphs. Proceedings of the European Design Automation Conference (EURO-DAC '94), IEEE Computer Society Press, Los Alamitos, 1994, pp. 406-411.
[9]. D.-C. Peixoto, D. Silva Jr., J.-M. Mata, C.-N. Coelho Jr., A.-O. Fernandes. Translation of hardware description languages to structured representation: a tool for digital system analysis. Proceedings of 13th Symposium on Computer Architecture and High Performance Computing (SBAC - PAD), 2001, Pirenypolis.
[10]. VAUL. A VHDL Analyzer and Utility Library. http://www-dt.e-technick.uin-dortmund.de/~mvo/vaul/ University of Dortmund, Department of Electrical Engineering, AG SIV, 1994.
[11]. M. Zaki, Y. Mokhtari, S. Tahar. A Path Dependency Graph for Verilog Program Analysis. Proceedings of the IEEE Northeast Workshop on Circuits and Systems (NEWCAS'03), Montreal, 2003, pp. 109-112.
[12]. R. Floyd. Assigning meanings to programs. Proceedings of Symposia in Applied Mathematics, vol. 19, 1967, pp. 19-32.
[13]. N. Blanc, D. Kroening. Race Analysis for SystemC using Model Checking. ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 15 issue 3, 2010, pp. 356 – 363. doi: 10.1109/ICCAD.2008.4681598
[14]. IEEE Standard SystemC Language Reference Manual, 1666-2005, IEEE, 2003, 428 p.
[15]. E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Counterexample-Guided Abstraction Refinement. Computer Aided Verification, Lecture Notes in Computer Science vol. 1855, 2000, pp. 154-169.
[16]. Cadence SMV Symbolic Model Checker. http://www.kenmcmil.com/smv.html. Дата обращения: 02.03.2015.
[17]. SCOOT A Tool for the Static Analysis of SystemC. http://www.cprover.org/scoot/. Дата обращения: 04.03.2015.
[18]. L. Liu, S. Vasudevan. Scaling Input Stimulus Generation through Hybrid Static and Dynamic Analysis of RTL. ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 20 issue 1, 2014. doi:10.1145/2676549
[19]. P. Godefroid. Compositional dynamic test generation. Proceedings of the 34th Annual ACM Symposium on Principles of Programming Languages (SIGPLAN-SIGACT), 2007, pp. 47-54. doi: 10.1145/1190216.1190226
[20]. STAR. http://users.crhc.illinois.edu/liu187/. Дата обращения: 04.03.2015.
[21]. S. Hertz, D. Sheridan, S. Vasudevan. Mining Hardware Assertions With Guidance From Static Analysis. Computer-Aided Design of Integrated Circuits and Systems, Transactions on IEEE, vol. 32, issue 6, 2013, pp. 952-965. doi: 10.1109/TCAD.2013.2241176
[22]. E. Clarke, O. Grumberg, D. Peled. Model checking. The MIT Press, 1999, 314 p.
[23]. GoldMine. http://goldmine.csl.illinois.edu/. Дата обращения: 04.03.2015.

[24]. Cadence Incisive. http://www.cadence.com/products/fv/iv_kit/pages/default.aspx. Дата обращения: 04.03.2015.
[25]. M. Weiser. Program slicing. IEEE Transactions on Software Engineering, vol. 10, issue 4, 1984, pp. 352–357.
[26]. E.M. Clarke, M. Fujita, S.P Rajan, T. Reps, S. Shankar. Program Slicing of Hardware Description Languages. Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods, pp. 298-302.
[27]. Codesurfer. http://www.grammatech.com/research/technologies/codesurfer. Дата обращения: 02.03.2015
[28]. T. Li, Y. Guo, S.-K. Li. Automatic Circuit Extractor for HDL Description Using Program Slicing. Journal of Computer Science and Technology vol. 19, issue 5, pp. 718-728. doi: 10.1007/BF02945599
[29]. C. Dawson, S.K. Pattanam, D. Roberts. The Verilog Procedural Interface for the Verilog Hardware Description Language. Proceedings of Verilog HDL Conference, 1996, pp. 17-23. doi: 10.1109/IVC.1996.496013
[30]. S. Vasudevan, E. A. Emerson, J. A. Abraham. Efficient Model Checking of Hardware Using Conditioned Slicing. Proceedings of the Fouth International Workshop on Automated Verification of Critical Systems (AVoCS), Vo1. 28, issue 6, 2005, pp. 279–294. doi: 10.1016/j.entcs.2005.04.017
[31]. M. Huth, M. Ryan. Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge University Press, New York, 2004, 440 p.
[32]. A. Tepurov, V. Tihhomirov, M. Jenihhin, J. Raik, G. Bartsch, J.H. Meza Escobar, H. Wuttke. Localization of Bugs in Processor Designs Using ZamiaCAD Framework. Proceedings of 13th International Workshop on Microprocessor Test and Verification (MTV), 2012, pp. 41-47.
[33]. ZamiaCAD. http://zamiacad.sourceforge.net/web/. Дата обращения: 06.03.2015.
[34]. J. Hopcroft, R. Motwani, J. Ullman. Introduction to Automata Theory: Languages, and Computation. Pearson/Addison Wesley, 2007, 535 p.
[35]. M. Yuang. Survey of protocol verification techniques based on finite state machine models. Proceedings of the Computer Networking Symposium, 1988, pp. 164-172.
[36]. R. Obermaisser, C. El-Salloum, B. Huber, H. Kopetz. Modeling and Verification of Distributed Real-Time Systems using Periodic Finite State Machines. Journal of Computer Systems Science & Engineering, vol. 22, No. 6, 2007.
[37]. А.С. Камкин. Метод формальной спецификации аппаратуры с конвейерной организацией и его приложение к задачам функционального тестирования. Труды ИСП РАН, Вып. 16, 2009 г., стр. 107-128.
[38]. T.-H. Wang, T. Edsall. Practical FSM Analysis for Verilog. Proceedings of Verilog HDL Conference and VHDL International Users Forum, 1998, pp. 52-58. doi:10.1109/IVC.1998.660680
[39]. A. Gill. Introduction to the Theory of Finite-state Machines. McGraw-Hill, 1962, 207 p.
[40]. J.-C. Giomi. Method of Extracting Implicit Sequential Behavior from Hardware Description Languages. 5.774.370, USA, 531.996, 18.09.1995, 30.06.1998, pp. 1-44.
[41]. C.-N. Liu, J.-Y. Jou. A FSM Extractor for HDL Description at RTL Level. Proceedings of the 2nd International Symposium on Quality Electronic Design, 2001, p. 372.
[42]. M. E. J. Gilford, G. N. Walker, J. L. Tredinnick, M. W. P. Dane, M. J. Reynolds. Recognition of a State Machine in High-Level Integrated Circuit Description Language Code. 7.152.214 B2, USA, 10/736.967, 15.12.2003, 19.12.2006, pp. 1-32.

[43]. W. Song, J. Garside. Automatic Controller Detection for Large Scale RTL Designs. Euromicro Conference on Digital System Design (DSD), Los Alamitos, CA. 2013, pp. 844-851. doi: 10.1109/DSD.2013.94
[44]. G. D. Micheli. Synthesis and Optimization of Digital Circuits. McGraw-Hill Science/Engineering/Math, 1994, 576 p.
[45]. Asynchronous Verilog Synthesizer. http://wsong83.github.io/index.html. Дата обращения: 05.03.2015.
[46]. A. Höller, C. Preschern, C. Steger, C. Kreiner, A. Krieg, H. Bock, J. Haid. Automatized High-Level Evaluation of Security Properties for RTL Hardware Designs. Proceedings of the Workshop on Embedded Systems Security, No. 6, 2013, pp. 1-8. Doi:10.1145/2527317.2527323
[47]. NuSMV. http://nusmv.fbk.eu/. Дата обращения: 05.03.2015.
[48]. K.-T. Cheng, A.S. Krishnakumar. Automatic Generation of Functional Vectors Using the Extended Finite State Machine Model. ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 1 issue 1, 1996, pp. 57-79. doi: 10.1145/225871.225880
[49]. J-Y. Jou, S. Rothweiler, R. Ernst, S. Sutarwala, A. Prabhu. BESTMAP: Behavioral Synthesis from C. In International Workshop on Logic Synthesis (Research Triangle Park, NC, May), 1998.
[50]. G. D. Guglielmo, L. D. Guglielmo, F. Fummi, G. Pravadelli. Efficient Generation of Stimuli for Functional Verification by Backjumping Across Extended FSMs. Journal of Electronic Testing, vol. 27, issue 2, 2011, pp. 137-162. Doi: 10.1007/s10836-011-5209-8
[51]. G. Di Guglielmo, F. Fummi, G. Pravadelli, S. Soffia, M. Roveri. Semi-formal functional verification by EFSM traversing via NuSMV. IEEE International High Level Design Validation and Test Workshop (HLDVT), 2010, pp. 58-65. doi: 10.1109/HLDVT.2010.5496660
[52]. D. Kim, M. Ciesielski, K. Shim, S. Yang. Temporal Parallel Simulation: A Fast Gate-level HDL Simulation Using Higher Level Models. Proceedings of the Conference on Design, Automation and Test in Europe (DATE), 2011. doi: 10.1109/DATE.2011.5763251
[53]. N. Bombieri, F. Fummi, G. Pravadelli. Automatic Abstraction of RTL IPs into Equivalent TLM Descriptions. IEEE Transactions on Computers, vol. 60, issue 12, 2011, pp. 1730-1743. doi: 10.1109/TC.2010.187
[54]. OSCI TLM-2.0. http://www.accellera.org/resources/videos/tlm20andsubset/. Дата создания: 22.02.2010.
[55]. HIFSuite. https://www.edalab.it/. Дата обращения: 05.05.2015.
[56]. А.С. Камкин, С.А. Смолов. Метод извлечения EFSM-моделей из HDL-описаний: применение к функциональной верификации. Сборник трудов конференции «Проблемы разработки перспективных микро- и наноэлектронных систем» под общ. ред. академика РАН А.Л. Стемпковского, часть 2, 2014, стр. 113-118.
[57]. Brandt J., Gemünde M., Schneider K., Shukla S., Talpin J.-P. Integrating System Descriptions by Clocked Guarded Actions. Forum on Design Languages, 2011, pp. 1-8.
[58]. Retrascope. http://forge.ispras.ru/projects/retrascope/. Дата обращения: 05.03.2015.
[59]. D. Moundanos, J.A. Abraham, Y.V. Hoskote. Abstraction Techniques for Validation Coverage Analysis and Test Generation. IEEE Transactions on Computers, IEEE, 1998, vol. 47, issue 1, pp. 2-14. doi:10.1109/12.656068

[60]. T.M. Niermann, J.H. Patel. HITEC: A Test Generation Package for Sequential Circuits. Proceedings of International European Design Automation Conference (EDAC), 1996, pp. 875-884. doi:10.1109/EDAC.1991.206393
[61]. C.-N. Jimmy Liu, J.-Y. Jou. An Efficient Functional Coverage Test for HDL Descriptions at RTL. Proceedings of International Conference on Computer Design (ICCD), Austin, TX, 1999, pp. 325-327. doi:10.1109/ICCD.1999.808561

A Survey of Methods for Model Extraction from HDL Descriptions

S.A. Smolov <smolov@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. In this paper a survey of existing methods of model extraction from hardware system descriptions written in Hardware Description Languages (like Verilog and VHDL) is presented. There are many tasks in hardware and software design where models are applied. The most actual tasks that are mentioned in this paper are: code optimization, logical synthesis optimization, model abstraction, and functional verification. The model categories that are mostly described here are flow or dependency graph models and automata models. As for flow graphs or dependency graphs, the methods of program slices extraction are described in details. Program slices can be characterized as suitable enough for directed test generation. Almost all the described automata models are finite state machine models and extended finite state machine models and so methods of such models extraction are the most popular for logical synthesis optimization and for functional test generation. The tests that can be generated from automata models shows high coverage of the target description. The key problems of existing model extraction methods are: the complexity of an application to industrial hardware descriptions (because of their complex structure), lack of automation (sometimes the hardware designer's knowledge is needed), the absence of open-source implementations. Also it is an actual task to create extendible frameworks for integration of different model extraction and analysis methods. Such framework can help in development of effective hybrid methods for hardware synthesis and verification.

Keywords: hardware description languages; model extraction; static analysis; code optimization; model abstraction; logical synthesis; functional verification; program slicing; flow graphs; dependency graphs; finite state machines; extended finite state machines.

DOI: 10.15514/ISPRAS-2015-27(1)-6

For citation: Smolov S.A. A Survey of Methods for Model Extraction from HDL Descriptions. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 1, 2015, pp. 197-124 (in Russian). DOI: 10.15514/ISPRAS-2015-27(1)-6

References

[1]. A. Kamkin, A. Kotsynyak, S. Smolov, A. Sortov, A. Tatarnikov, M. Chupilko. Sredstva funktsionalnoy verifikatsii mikroprotsessorov [Tools for Functional Verification of Microprocessors]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 26, issue 1, 2014, pp. 149-200. doi: 10.15514/ISPRAS-2014-26(1)-5

[2]. Statistical Analysis of Floating Point Flaw: Intel White Paper Section 3. http://www.intel.com/support/processors/pentium/sb/CS-013007.htm. Access date: 08.07.2004.

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

[4]. IEEE Standard for Verilog Hardware Description Language, 1364-2005, IEEE, 2006, 560 p.

[5]. IEEE Standard VHDL Language Reference Manual, 1076-2008, IEEE, 2009, 626 p.

[6]. A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman. Compilers: Principles, Technologies, and Tools (2nd Edition). Addison Wesley, 2006. 1000 p.

[7]. A. Balboni, M. Mastretti, M. Stefanoni. Static Analysis for VHDL Model Evaluation. Proceedings of the European Design Automation Conference (EURO-DAC '94), IEEE Computer Society Press, Los Alamitos, 1994, pp. 586-591.

[8]. L. Baresi, C. Bolchini, D. Sciuto. Software Methodologies for VHDL Code Static Analysis based on Flow Graphs. Proceedings of the European Design Automation Conference (EURO-DAC '94), IEEE Computer Society Press, Los Alamitos, 1994, pp. 406-411.

[9]. D.-C. Peixoto, D. Silva Jr., J.-M. Mata, C.-N. Coelho Jr., A.-O. Fernandes. Translation of hardware description languages to structured representation: a tool for digital system analysis. Proceedings of 13th Symposium on Computer Architecture and High Performance Computing (SBAC - PAD), 2001, Pirenypolis.

[10]. VAUL. A VHDL Analyzer and Utility Library. http://www-dt.e-technick.uin-dortmund.de/~mvo/vaul/ University of Dortmund, Department of Electrical Engineering, AG SIV, 1994.

[11]. M. Zaki, Y. Mokhtari, S. Tahar. A Path Dependency Graph for Verilog Program Analysis. Proceedings of the IEEE Northeast Workshop on Circuits and Systems (NEWCAS'03), Montreal, 2003, pp. 109-112.

[12]. R. Floyd. Assigning meanings to programs. Proceedings of Symposia in Applied Mathematics, vol. 19, 1967, pp. 19-32.

[13]. N. Blanc, D. Kroening. Race Analysis for SystemC using Model Checking. ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 15 issue 3, 2010, pp. 356 – 363. doi: 10.1109/ICCAD.2008.4681598

[14]. IEEE Standard SystemC Language Reference Manual, 1666-2005, IEEE, 2003, 428 p.

[15]. E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Counterexample-Guided Abstraction Refinement. Computer Aided Verification, Lecture Notes in Computer Science vol. 1855, 2000, pp. 154-169.

[16]. Cadence SMV Symbolic Model Checker. http://www.kenmcmil.com/smv.html. Access date: 02.03.2015.

[17]. SCOOT A Tool for the Static Analysis of SystemC. http://www.cprover.org/scoot/. Access date: 04.03.2015.

[18]. L. Liu, S. Vasudevan. Scaling Input Stimulus Generation through Hybrid Static and Dynamic Analysis of RTL. ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 20 issue 1, 2014. doi:10.1145/2676549

[19]. P. Godefroid. Compositional dynamic test generation. Proceedings of the 34th Annual ACM Symposium on Principles of Programming Languages (SIGPLAN-SIGACT), 2007, pp. 47-54. doi: 10.1145/1190216.1190226

[20]. STAR. http://users.crhc.illinois.edu/liu187/. Access date: 04.03.2015.

[21]. S. Hertz, D. Sheridan, S. Vasudevan. Mining Hardware Assertions With Guidance From Static Analysis. Computer-Aided Design of Integrated Circuits and Systems, Transactions on IEEE, vol. 32, issue 6, 2013, pp. 952-965. doi: 10.1109/TCAD.2013.2241176

[22]. E. Clarke, O. Grumberg, D. Peled. Model checking. The MIT Press, 1999, 314 p.

[23]. GoldMine. http://goldmine.csl.illinois.edu/. Access date: 04.03.2015.

[24]. Cadence Incisive. http://www.cadence.com/products/fv/iv_kit/pages/default.aspx. Access date: 04.03.2015.

[25]. M. Weiser. Program slicing. IEEE Transactions on Software Engineering, vol. 10, issue 4, 1984, pp. 352–357.

[26]. E.M. Clarke, M. Fujita, S.P Rajan, T. Reps, S. Shankar. Program Slicing of Hardware Description Languages. Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods, pp. 298-302.

[27]. Codesurfer. http://www.grammatech.com/research/technologies/codesurfer. Access date: 02.03.2015

[28]. T. Li, Y. Guo, S.-K. Li. Automatic Circuit Extractor for HDL Description Using Program Slicing. Journal of Computer Science and Technology vol. 19, issue 5, pp. 718-728. doi: 10.1007/BF02945599

[29]. C. Dawson, S.K. Pattanam, D. Roberts. The Verilog Procedural Interface for the Verilog Hardware Description Language. Proceedings of Verilog HDL Conference, 1996, pp. 17-23. doi: 10.1109/IVC.1996.496013

[30]. S. Vasudevan, E. A. Emerson, J. A. Abraham. Efficient Model Checking of Hardware Using Conditioned Slicing. Proceedings of the Fouth International Workshop on Automated Verification of Critical Systems (AVoCS), Vo1. 28, issue 6, 2005, pp. 279–294. doi: 10.1016/j.entcs.2005.04.017

[31]. M. Huth, M. Ryan. Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge University Press, New York, 2004, 440 p.

[32]. A. Tepurov, V. Tihhomirov, M. Jenihhin, J. Raik, G. Bartsch, J.H. Meza Escobar, H. Wuttke. Localization of Bugs in Processor Designs Using ZamiaCAD Framework. Proceedings of 13th International Workshop on Microprocessor Test and Verification (MTV), 2012, pp. 41-47.

[33]. ZamiaCAD. http://zamiacad.sourceforge.net/web/. Дата обращения: 06.03.2015.

[34]. J. Hopcroft, R. Motwani, J. Ullman. Introduction to Automata Theory: Languages, and Computation. Pearson/Addison Wesley, 2007, 535 p.

[35]. M. Yuang. Survey of protocol verification techniques based on finite state machine models. Proceedings of the Computer Networking Symposium, 1988, pp. 164-172.

[36]. R. Obermaisser, C. El-Salloum, B. Huber, H. Kopetz. Modeling and Verification of Distributed Real-Time Systems using Periodic Finite State Machines. Journal of Computer Systems Science & Engineering, vol. 22, No. 6, 2007.

[37]. A. Kamkin. Metod formalnoy spetsifikatsii apparaturyi s konveyernoy organizatsiey i ego prilozhenie k zadacham funktsionalnogo testirovaniya [A Method of Pipelined Hardware Specification and its Application to Functional Verification]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 16, 2009, pp. 107-128 (in Russian).

[38]. T.-H. Wang, T. Edsall. Practical FSM Analysis for Verilog. Proceedings of Verilog HDL Conference and VHDL International Users Forum, 1998, pp. 52-58. doi:10.1109/IVC.1998.660680
[39]. A. Gill. Introduction to the Theory of Finite-state Machines. McGraw-Hill, 1962, 207 p.
[40]. J.-C. Giomi. Method of Extracting Implicit Sequential Behavior from Hardware Description Languages. 5.774.370, USA, 531.996, 18.09.1995, 30.06.1998, pp. 1-44.
[41]. C.-N. Liu, J.-Y. Jou. A FSM Extractor for HDL Description at RTL Level. Proceedings of the 2nd International Symposium on Quality Electronic Design, 2001, p. 372.
[42]. M. E. J. Gilford, G. N. Walker, J. L. Tredinnick, M. W. P. Dane, M. J. Reynolds. Recognition of a State Machine in High-Level Integrated Circuit Description Language Code. 7.152.214 B2, USA, 10/736.967, 15.12.2003, 19.12.2006, pp. 1-32.
[43]. W. Song, J. Garside. Automatic Controller Detection for Large Scale RTL Designs. Euromicro Conference on Digital System Design (DSD), Los Alamitos, CA. 2013, pp. 844-851. doi: 10.1109/DSD.2013.94
[44]. G. D. Micheli. Synthesis and Optimization of Digital Circuits. McGraw-Hill Science/Engineering/Math, 1994, 576 p.
[45]. Asynchronous Verilog Synthesizer. http://wsong83.github.io/index.html. Access date: 05.03.2015.
[46]. A. Höller, C. Preschern, C. Steger, C. Kreiner, A. Krieg, H. Bock, J. Haid. Automatized High-Level Evaluation of Security Properties for RTL Hardware Designs. Proceedings of the Workshop on Embedded Systems Security, No. 6, 2013, pp. 1-8. Doi:10.1145/2527317.2527323
[47]. NuSMV. http://nusmv.fbk.eu/. Access date: 05.03.2015.
[48]. K.-T. Cheng, A.S. Krishnakumar. Automatic Generation of Functional Vectors Using the Extended Finite State Machine Model. ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 1 issue 1, 1996, pp. 57-79. doi: 10.1145/225871.225880
[49]. J-Y. Jou, S. Rothweiler, R. Ernst, S. Sutarwala, A. Prabhu. BESTMAP: Behavioral Synthesis from C. In International Workshop on Logic Synthesis (Research Triangle Park, NC, May), 1998.
[50]. G. D. Guglielmo, L. D. Guglielmo, F. Fummi, G. Pravadelli. Efficient Generation of Stimuli for Functional Verification by Backjumping Across Extended FSMs. Journal of Electronic Testing, vol. 27, issue 2, 2011, pp. 137-162. Doi: 10.1007/s10836-011-5209-8
[51]. G. Di Guglielmo, F. Fummi, G. Pravadelli, S. Soffia, M. Roveri. Semi-formal functional verification by EFSM traversing via NuSMV. IEEE International High Level Design Validation and Test Workshop (HLDVT), 2010, pp. 58-65. doi: 10.1109/HLDVT.2010.5496660
[52]. D. Kim, M. Ciesielski, K. Shim, S. Yang. Temporal Parallel Simulation: A Fast Gate-level HDL Simulation Using Higher Level Models. Proceedings of the Conference on Design, Automation and Test in Europe (DATE), 2011. doi: 10.1109/DATE.2011.5763251
[53]. N. Bombieri, F. Fummi, G. Pravadelli. Automatic Abstraction of RTL IPs into Equivalent TLM Descriptions. IEEE Transactions on Computers, vol. 60, issue 12, 2011, pp. 1730-1743. doi: 10.1109/TC.2010.187
[54]. OSCI TLM-2.0. http://www.accellera.org/resources/videos/tlm20andsubset/. Access date: 22.02.2010.
[55]. HIFSuite. https://www.edalab.it/. Access date: 05.05.2015.

[56]. A. Kamkin, S. Smolov. 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.
[57]. Brandt J., Gemünde M., Schneider K., Shukla S., Talpin J.-P. Integrating System Descriptions by Clocked Guarded Actions. Forum on Design Languages, 2011, pp. 1-8.
[58]. Retrascope. http://forge.ispras.ru/projects/retrascope/. Дата обращения: 05.03.2015.
[59]. D. Moundanos, J.A. Abraham, Y.V. Hoskote. Abstraction Techniques for Validation Coverage Analysis and Test Generation. IEEE Transactions on Computers, IEEE, 1998, vol. 47, issue 1, pp. 2-14. doi:10.1109/12.656068
[60]. T.M. Niermann, J.H. Patel. HITEC: A Test Generation Package for Sequential Circuits. Proceedings of International European Design Automation Conference (EDAC), 1996, pp. 875-884. doi:10.1109/EDAC.1991.206393
[61]. C.-N. Jimmy Liu, J.-Y. Jou. An Efficient Functional Coverage Test for HDL Descriptions at RTL. Proceedings of International Conference on Computer Design (ICCD), Austin, TX, 1999, pp. 325-327. doi:10.1109/ICCD.1999.808561