УДК 004.05

DOI 10.5862/JCTCS.212.6

С.А. Смолов, А.С. Камкин

МЕТОД ПОСТРОЕНИЯ РАСШИРЕННЫХ КОНЕЧНЫХ АВТОМАТОВ ПО HDL-ОПИСАНИЮ НА ОСНОВЕ СТАТИЧЕСКОГО АНАЛИЗА КОДА

S.A. Smolov, A.S. Kamkin

A METHOD OF EXTENDED FINITE STATE MACHINES CONSTRUCTION FROM HDL DESCRIPTIONS BASED ON STATIC ANALYSIS OF SOURCE CODE

Сложность цифровой микроэлектронной аппаратуры неуклонно возрастает, что существенно затрудняет ее верификацию – проверку корректности. Чрезвычайно актуальными оказываются методы автоматизированной верификации. Подобные методы, как правило, основаны на использовании моделей – формализованных представлений проектируемой аппаратуры, удобных для генерации тестов и/или формальной проверки свойств. Часто модели строятся вручную, что чревато ошибками и может приводить к неадекватным результатам верификации.

Описан метод автоматического извлечения моделей, имеющих форму расширенных конечных автоматов, непосредственно из проектных описаний аппаратуры. Приведены экспериментальные данные по применению предложенного метода.

ЦИФРОВАЯ АППАРАТУРА; ФУНКЦИОНАЛЬНАЯ ВЕРИФИКАЦИЯ; ЯЗЫК ОПИСАНИЯ АППАРАТУРЫ; СТАТИЧЕСКИЙ АНАЛИЗ; ГЕНЕРАЦИЯ ТЕСТОВ; ПРОВЕРКА МОДЕЛЕЙ; ЛОГИЧЕСКИЙ СИНТЕЗ; РАСШИРЕННЫЙ КОНЕЧНЫЙ АВТОМАТ; ОХРАНЯЕМОЕ ДЕЙСТВИЕ.

The complexity of digital microelectronic hardware grows steadily, which complicates its functional verification and makes the methods of automated functional verification extremely important. Such methods usually use models that are formal representations of hardware descriptions. Such models are suitable for functional test generation and/or property checking. These models are often manually built, which can cause errors or unexpected behavior.

This paper comes up with a new method of automated extraction of extended finite-state machine models from hardware descriptions. The key feature of the method is automated detection of hardware module's registers that encode the module's state. The experimental results of the method's application are also presented in the paper.

DIGITAL HARDWARE; FUNCTIONAL VERIFICATION; HARDWARE DESCRIPTION LANGUAGE; STATIC ANALYSIS; FUNCTIONAL TEST GENERATION; MODEL CHECKING; LOGIC SYNTHESIS; EXTENDED FINITE-STATE MACHINE; GUARDED ACTION.

Функциональная верификация является одним из наиболее трудоемких и дорогостоящих этапов в процессе проектирования цифровой микроэлектронной аппаратуры [1]. Для автоматизации верификации широко используются модели – математические абстракции, описывающие структуру и/или поведение разрабатываемой системы. Примерами типов моделей, широко применяемых при проектировании аппаратуры, являются конечные автоматы и сети Петри [2]. Модели могут строиться на основе анализа требований (технического задания, внутренней документации и т. п.) либо извлекаться из проектных описаний аппаратуры, выполненных на специализированных языках (Hardware Description Language –

HDL), например, VHDL или Verilog [3]. В статье рассматривается метод второго типа и его применение к верификации. Для моделирования аппаратуры используется формализм расширенных конечных автоматов (Extended Finite State Machine – EFSM) [4].

Расширенный конечный автомат (называемый также EFSM-моделью) устроен следующим образом. Во-первых, в дополнение к конечному множеству состояний, имеющемуся в классическом автомате (Finite State Machine – FSM), он содержит множество переменных (входных, внутренних и выходных). Во-вторых, в EFSM-модели переходы между состояниями снабжены охранными условиями (guards) на значения переменных (входных и внутренних) и действиями (actions) по изменению значений переменных (внутренних и выходных). Переход расширенного конечного автомата может сработать, только если выполнено его охранное условие; при срабатывании перехода выполняется соответствующее действие.

В EFSM-моделях управляющая логика (control logic) естественным образом отделяется от функций преобразования данных (datapath), как это принято при проектировании цифровой аппаратуры [5]. Являясь адекватным формализмом для моделирования широкого класса систем (компьютерных протоколов, систем управления и др.), расширенные конечные автоматы активно используются в верификации: для построения тестовых наборов, проверяющих соответствие системы требованиям [6]; для генерации тестовых последовательностей, нацеленных на маловероятные ситуации в работе системы [7] (в которых могут проявляться трудно обнаруживаемые ошибки проектирования [8]); для формальной проверки свойств системы [9].

В работе предлагается метод извлечения EFSM-моделей из исходного кода HDL-описаний, ориентированный на решение задач верификации. Разработка подхода мотивировалась следующими соображениями. Во-первых, автоматическое построение модели по исходному коду позволяет избежать ошибок, имеющих место при ручном моделировании; упрощает поддержку систем верификации (модель и некоторые части тестового окружения могут быть автоматически перестроены при изменении кода проекта); повышает точность и нацеленность верификации. Во-вторых, расширенные конечные автоматы являются хорошо изученными математическими объектами, для которых разработаны эффективные методы анализа (представляется перспективным адаптировать имеющийся арсенал методов и инструментов для их применения к HDL-описаниям). В-третьих, EFSM-модели представляются удобным средством для интеграции различных техник верификации аппаратуры как имитационных (simulation-based), так и формальных.

Статья является расширенной версией работы [10], представленной нами на конференции МЭС-2014: в ней уточнен алгоритм извлечения EFSM-моделей из HDL-описаний и, кроме того, приведены экспериментальные данные по применению предложенного метода.

Обзор работ

Несмотря на то, что имеется большое число работ, посвященных использованию EFSM-моделей для верификации программных и аппаратных систем, существует не так много подходов, в которых такие модели извлекаются непосредственно из исходного кода HDL-описаний. Алгоритмы построения EFSM-моделей по исходному коду известны и широко применяются в современных САПР (на них, в частности, базируются методы логического синтеза [11]), однако получаемые при их использовании модели не всегда адекватны для целей верификации [7]. Состояниям в таких моделях соответствуют операторы (точки) в исходном коде, в которых выполняется ожидание входных событий; при выделении состояний никак не учитываются заданные в коде соотношения между переменными (условия ветвления, выражения в присваиваниях и т. п.).

В работе [12] рассмотрена среда мутационного тестирования (mutation testing)

FAST. Мутационное тестирование – это метод оценки адекватности тестовых наборов, основанный на внесении небольших изменений (мутаций) в исходный код: если тесты не в состоянии обнаружить такие изменения, они считаются неполными. HDL-описания с внедренными в них ошибками (так называемые мутанты) автоматически транслируются средой FAST в более абстрактные, но событийно эквивалентные модели уровня транзакций (Transaction Level Model – TLM). Цель этого преобразования состоит в ускорении прогона тестов для измененных описаний (это актуальная задача, поскольку число мутантов, как правило, велико, а тесты имеют значительную длину). Ключевой частью работы является метод абстракции – преобразования HDL-описаний уровня регистровых передач (Register Transfer Level – RTL) в TLM-представления, – в основе которого используются EFSM-модели. Анализируя извлеченный из HDL-описания автомат, среда FAST идентифицирует операции над данными (computational phases) – пути в графе состояний, включающие действия по получению входной информации, ее обработке и вычислению выходного результата. Абстракция осуществляется путем объединения состояний и переходов автомата, относящихся к одному этапу одной операции.

Ряд работ (например, [6–8, 13]) посвящен проблеме генерации функциональных тестов на основе EFSM-моделей. Если модель извлекается из исходного кода HDL-описания, сгенерированные тесты обеспечивают высокий уровень покрытия кода (code coverage). Основным подходом к построению тестов на основе автоматных моделей является обход (traversal, exploration) графа состояний – построение пути (или набора путей), содержащего все состояния и переходы автомата [14]. В отличие от классических конечных автоматов, при обходе EFSM-моделей имеется сложность, связанная с наличием у переходов охранных условий. Если условия зависят не только от входных переменных, но и от внутренних, определение достижимости состояний становится вычислительно трудной задачей. Существуют техники преобразования расширенных конечных автоматов, позволяющие устранять (или минимизировать) такие зависимости (см., например, [6, 13, 15]), но они, вообще говоря, приводят к комбинаторному взрыву числа состояний. Альтернативу им составляют подходы на основе поиска с возвратом (backtracking, backjumping) [7].

В работе [7] описан метод извлечения «простых для обхода» (easy-to-traverse) EFSM-моделей из HDL-описаний. Метод состоит из четырех этапов. На первом этапе (для каждого процесса, заданного в описании), используя известный алгоритм [11], строится начальная (референсная) EFSM-модель (Reference EFSM – REFSM). В общем случае полученная модель «трудна для обхода» (hard-to-traverse), в частности, из-за того, что содержит условные операторы в действиях переходов. На втором этапе в REFSM-модель добавляются промежуточные состояния, а переходы декомпозируются таким образом, чтобы их действия не содержали ветвлений. Полученная модель (Largest EFSM – LEFSM), строго говоря, не эквивалентна исходной модели – один шаг работы REFSM может соответствовать нескольким шагам в LEFSM. Для обеспечения временной эквивалентности REFSM и LEFSM в последней модели выполняется расщепление промежуточных состояний и объединение совместимых переходов. В результате образуется SEFSM-модель (Smallest EFSM). На завершающем этапе, используя метод [13], выполняется частичная стабилизация SEFSM-модели, нацеленная на устранение зависимостей охранных условий переходов от переменных, кодирующих состояния. Результатом является S^2EFSM-модель (Semi-Stabilized EFSM), которая эквивалентна исходной модели и, по утверждению авторов, является «простой для обхода».

Результаты экспериментов, представленные в работе [7], демонстрируют эффективность подхода для решения задач нацеленной генерации тестов, однако процедура построения EFSM-модели по исходному коду HDL-описания вызывает вопросы. Во-первых, представляется слиш-

ком жестким то ограничение, что для одного процесса HDL-описания строится одна EFSM-модель. С одной стороны, один логический блок аппаратуры (по сути, автомат) может быть определен с помощью нескольких процессов (такие процессы используют общие переменные и работают в режиме взаимного исключения). С другой стороны, в рамках одного процесса могут быть описаны действия, относящиеся к разным логическим блокам (возможно, это не самый хороший стиль кодирования, но он допускается HDL-языками). Во-вторых, процесс построения модели представляется чрезмерно усложненным: аналогичных результатов можно добиться более простыми средствами, если с самого начала определить, какие внутренние переменные описывают состояние автомата.

Основные понятия

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 γ – охранное условие, а δ – 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).

Расширенным конечным автоматом (*EFSM-моделью*) называется тройка $\langle S, V, T\rangle$, где S – конечное множество состояний; $V = I \cup O \cup R$ – конечное множество переменных, состоящее из *входных сигналов* (I), *выходных сигналов* (O) и *внутренних регистров* (R); T – конечное множество *переходов*: каждый переход $t \in T$ – это кортеж вида $(s_t, \gamma_t \to \delta_t, s'_t)$, где s_t и s'_t – соответственно начальное и конечное состояния перехода, а γ_t и δ_t – соответственно охранное условие и действие. Означивание $\nu \in Dom_V$ называется *контекстом* автомата, а пара $(s, \nu) \in S \times Dom_V$ – его *конфигурацией*. Переход t называется *разрешенным* в конфигурации (s, ν), если $s_t = s$ и $\gamma_t(\nu) = true$.

Расширенный конечный автомат функционирует в дискретном времени, что неявно предполагает наличие часов, по тикам которых срабатывают переходы. Под *часами* (clock) в данной работе понимается вполне конкретный объект – непустое множество *событий*, где событие – это пара, включающая однобитный сигнал, называемый *синхросигналом*, и тип его регистрации: *передний фронт* (изменение значения с нуля на единицу) или *задний фронт* (изменение значения с единицы на нуль). Тик часов определяется наступлением события. Для заданных часов C и начальной конфигурации (s_0, ν_0) расширенный конечный автомат работает следующим образом. Вначале выполняется *сброс* (reset) – устанавливается начальная конфигурация автомата: $(s, \nu) \leftarrow (s_0, \nu_0)$. На каждом такте (промежутке времени между двумя тиками) определяется множество разрешенных переходов: $E \leftarrow \{t \in T \mid s_t = s \wedge \gamma_t(\nu) = true\}$. Если множество E не пусто, срабатывает переход $t \in E$, выбранный из него недетерминированным образом. При выполнении перехода конфигурация обновляется соответствующим образом: $(s, \nu) \leftarrow (s'_t, \delta_t(\nu))$.

Извлечение EFSM-модели

Предлагаемый метод извлечения EFSM-модели (точнее, системы EFSM-моделей, каждая из которых описывает отдельный процесс) из исходного кода HDL-описания состоит из следующих шагов:

1) синтаксический анализ HDL-описания и построение *дерева абстрактного синтаксиса*;

2) обход дерева абстрактного синтаксиса и построение *внутреннего представления*:

- идентификация *синхросигналов*;
- выявление *неявных переменных состояния*;

3) трансформация внутреннего представления в систему охраняемых действий;

4) анализ зависимостей между охраняемыми действиями и идентификация *пере-*

менных состояния;

5) анализ условий на переменные состояния и построение *пространства состояний* EFSM-модели;

6) построение *отношения переходов* EFSM-модели.

Построение системы охраняемых действий. Результатом предварительной обработки является система охраняемых действий, с каждым из которых связаны часы ($\{\langle C^{(i)}, \gamma^{(i)} \to \delta^{(i)}\rangle\}$) – такие действия называются *синхронизированными* (clocked) [16].

Способы реализации **шага 1** широко известны [17], поэтому сразу перейдем к **шагу 2**. Одной из его целей является идентификация синхросигналов с целью построения часов. Для решения этой задачи предлагается следующая эвристика. Считается, что переменная ν является *синхросигналом*, если выполнены следующие условия:

- ν является входным однобитным сигналом;
- ν присутствует в списке чувствительности (определение дано ниже) хотя бы одного из процессов (или в операторе ожидания событий *wait*);
- ν не используется в присваиваниях (ни в левых, ни в правых частях).

Напомним, что *списком чувствительности* (sensitivity list) процесса называется набор типов событий, задающих условие активации процесса: процесс запускается каждый раз, когда возникает событие, относящееся к одному из указанных типов (и не запускается в иных ситуациях).

На шаге 2 также определяются неявные переменные состояния и добавляются во внутреннее представление. Под *неявными переменными состояния* понимаются внутренние регистры, явно не присутствующие в коде, но необходимые для корректного представления автомата, специфицированного в HDL-описании (обычно такие переменные искусственно вводятся инструментами логического синтеза [11]). Цель выявления и добавления неявных переменных состояния состоит в декомпозиции сложных, многотактных процессов на однотактные микрооперации. С каждым процессом p связана неявная переменная состояния (обозначим ее r_p), а с каждой операцией w типа *wait* внутри процесса p (включая операцию активации процесса) – определенное значение этого регистра (обозначим его ν_w). В графе потока управления процесса p анализируются пути между парами операций *wait*, при этом проводится ряд преобразований. Процесс p удаляется из внутреннего представления; вместо него для каждого пути (точнее, ациклического подграфа с одним истоком и одним стоком) π (пусть это будет путь между w_i и w_j) строится новый процесс p_π. Условие активации p_π совпадает с условием операции w_i, а тело имеет следующий вид: **if** $r_p = \nu_{wi}$ **then** π; $r_p := \nu_{wj}$ **end if**. В целом, реализация этого шага совпадает с алгоритмом, описанным в [11].

На **шаге 3** для каждого элементарного процесса (микрооперации) p, построенного на шаге 2, строится множество всех входящих в него синхронизированных охраняемых действий $\{\langle C_p^{(i)}, \gamma_p^{(i)} \to \delta_p^{(i)}\rangle\}_{i=1,n}$. В простейшем случае это множество устроено следующим образом:

$C_p^{(i)}$ содержит все синхросигналы, задействованные в процессе p;

$\gamma_p^{(i)}$ определяет условие $i^{\text{й}}$ ветви условного оператора верхнего уровня (если такого оператора нет, то $n = 1$ и $\gamma_p^{(1)} \equiv$ *true*);

$\delta_p^{(i)}$ содержит все действия $i^{\text{й}}$ ветви (все тело процесса p, если $n = 1$).

В общем случае после выполнения шага 2 возможны ситуации, когда охраняемые действия содержат вложенные условные операторы. Для упрощения последующего анализа эти операторы «поднимаются» на уровень охранных условий (с учетом зависимостей от предшествовавших им операторов присваивания). Для этого используются техники символического выполнения, включая классический метод обратных подстановок [18], позволяющий вычислить слабейшее предусловие (weakest precondition) ациклической программы для заданного постусловия (в нашем случае – условия ветвления). Это приводит к расщеплению охраняемых действий: каждому пути в графе потока управления процесса, связывающему начальную и конечную вершины, ставится в соответствие свое охраняемое действие.

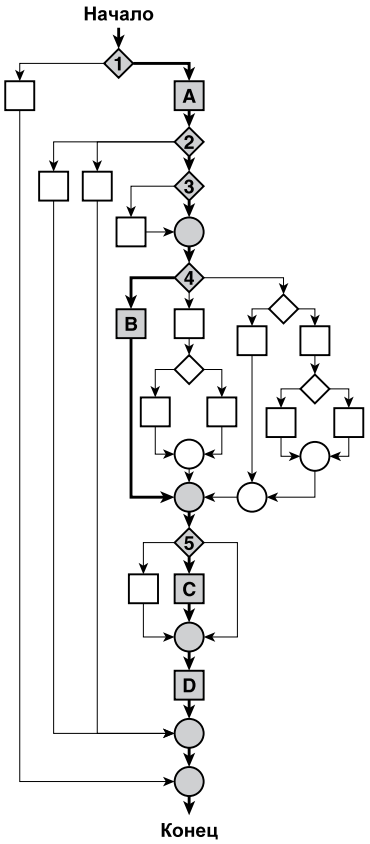

Рис. 1. Пример графа потока управления

Здесь следует сделать два замечания. Во-первых, предложенный метод (в том виде, как он описан) не применим к HDL-описаниям, содержащим циклы с переменным числом итераций в рамках одной микрооперации (заметим, что такие описания являются редкостью). Во-вторых, он предполагает, что процессы HDL-описаний имеют сравнительно небольшую сложность (обычно это так: сложность аппаратуры достигается не сложностью отдельных процессов, а их огромным числом).

Для того чтобы проиллюстрировать построение системы охраняемых действий, рассмотрим HDL-описание b04 из пакета тестов (benchmark) ITC'99 [19]. Это арифметико-логическое устройство, позволяющее вычислять сумму, среднее арифметическое, минимум и максимум набора целых чисел. На рис. 1 изображен граф потока управления описания b04. Квадратные вершины соответствуют *базовым блокам*, ромбические – *ветвлениям*, круглые – *соединениям*. У базовых блоков одна входящая дуга и одна выходящая; у ветвлений одна входящая дуга и не менее двух выходящих; у соединений не менее двух входящих и одна выходящая.

В графе на рис. 1 выделен путь из начальной вершины в конечную, для которого в табл. 1 приведено охраняемое действие. В левой колонке таблицы представлены фрагменты описания b04 на языке VHDL, соответствующие выделенным вершинам графа (латинскими буквами обозначены базовые блоки, цифрами – ветвления). Правая колонка содержит охраняемое действие, полученное в результате анализа выделенного пути. Отметим, что для данного охраняемого действия (и процесса в целом) в качестве синхросигналов были идентифицированы CLOCK (передний фронт) и RESET (задний фронт).

На рис. 2 показана система охраняемых действий описания b04, полученная путем «подъема» условных операторов. Выделенный путь соответствует рассмотренному охраняемому действию. Видно, что условия (ромбические вершины) образуют иерархию и разделяются разными охраняемыми действиями. Сами действия представляют собой последовательности, составленные из исходных базовых блоков (квадратных вершин). Таким образом, система охраняемых действий одного процесса представляется в виде структуры данных, близкой к высокоуровневой решающей диаграмме (High-Level Decision Diagram – HLDD) [20]. Мы называем эту структуру *решающей диаграммой системы охраняемых действий* (Guarded Actions Decision Diagram – GADD) или просто *GADD-диаграммой*.

Всего для HDL-описания b04 с помощью предложенного метода строится 27 охраняемых действий. Это число меньше общего количества путей между начальной и конечной вершинами графа потока управления, т. к. для некоторых путей получаемые охранные условия являются невыполнимыми.

Построение EFSM-модели. На **шаге 4** осуществляется анализ зависимостей между охраняемыми действиями. Пусть x и y –

Таблица 1

Построение охраняемого действия

	Фрагмент HDL-описания	Охраняемое действие
1	**not** (RESET = '1') **and** (CLOCK'**event and** CLOCK = '1')	**Охранное условие**: (stato == sC) **and** (DATA_IN < RMIN) **and not** (DATA_IN > RMAX) **and not**(RESTART = '1') **and not**(ENABLE = '1')
A	RES := RESTART; ENA := ENABLE; AVE := AVERAGE;	
2	stato<= sC	**Действие**: RES := RESTART; ENA := ENABLE; AVE := AVERAGE; DATA_OUT <= RLAST; RMIN := DATA_IN; REG4 := REG3; REG3 := REG2; REG2 := REG1; REG1 := DATA_IN; stato := sC;
3	**not** (ENA = '1')	
4	**not** (RES = '1') **and not** (ENA = '1')	
B	DATA_OUT <= RLAST;	
5	**not** (DATA_IN > RMAX) **and** (DATA_IN < RMIN)	
C	RMIN := DATA_IN;	
D	REG4 := REG3; REG3 := REG2; REG2 := REG1; REG1 := DATA_IN; stato := sC	**Часы**: CLOCK (передний фронт), RESET (задний фронт)

охраняемые действия, а v – переменная. Говорят, что переменная v *определяется* в охраняемом действии *x* (и обозначают это как $v \in Def_x$), если действие *x* содержит присваивание переменной v. Говорят, что переменная v *используется* в охраняемом действии *y* (и обозначают это как $v \in Use_y$), если v присутствует в охранном условии *y* или в правой части некоторого присваивания его действия. В зависимости от того, как именно используется переменная, в охранном условии или в действии, различают *зависимости по управлению* и *по данным*.

На основе анализа зависимостей между охраняемыми действиями осуществляется идентификация переменных состояния. Под *переменными состояния* (на этом шаге все переменные являются явными – см. шаг 2) понимаются внутренние регистры, используемые для организации потоков управления процессов. Для идентификации таких переменных используется следующая эвристика. Переменная v является *переменной состояния*, если выполнены следующие условия:

- v не является входным сигналом;
- в системе охраняемых действий процесса существует хотя бы одно охраняемое действие, которое зависит от v по управлению, и в котором v определяется (прямо или косвенно);
- ни одно из выражений, стоящих в правых частях операторов присваивания в v, не содержит входных сигналов.

Следует особенно подчеркнуть, что приведенные эвристики (эвристика идентификации синхросигналов и эвристика идентификации переменных состояния) не зафиксированы методом, а являются его параметрами. В зависимости от целей и/или ресурсов их можно усиливать или ослаблять. Конструируемые для разных эвристик EFSM-модели могут различаться структурой (числом состояний и переходов), но обязаны быть функционально эквивалентными (доказательство этого утверждения выходит за рамки настоящей статьи).

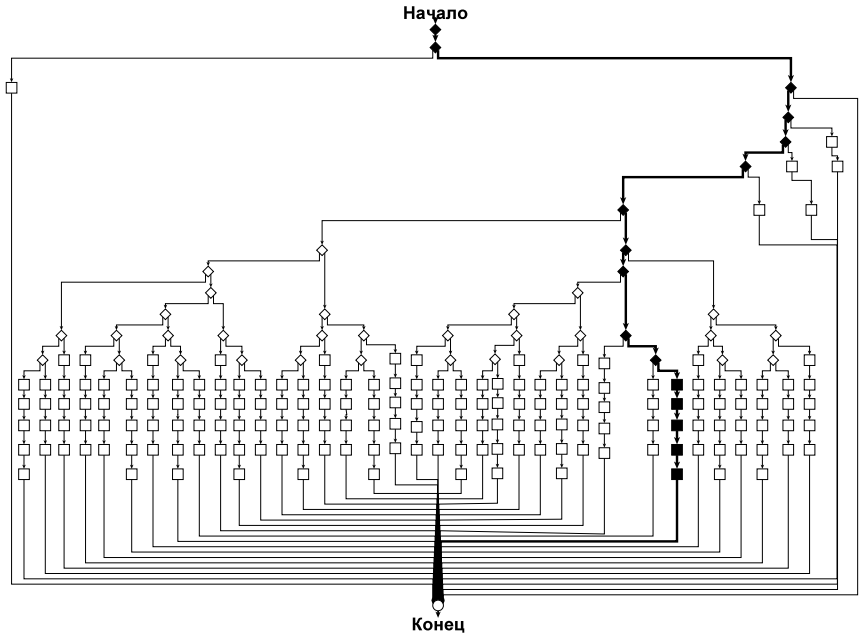

Рис. 2. Пример системы охраняемых действий (GADD-диаграммы)

На **шаге 5** строится пространство состояний EFSM-модели. Ключевая идея этого шага состоит в следующем: множество всех ограничений на переменные состояния разбивается (путем уточнения некоторых ограничений) на попарно несовместные условия (этот процесс называется *ортогонализацией* [21]); каждому из полученных условий сопоставляется уникальное состояние EFSM-модели. Реализация этой идеи осуществляется в два этапа:

1) построение условий на переменные состояния;

2) ортогонализация условий на переменные состояния.

Для построения условий на переменные состояния исходная GADD-диаграмма трансформируется: ветвления «расщепляются» таким образом, чтобы отделить переменные состояния от прочих переменных (будем называть такие переменные *контекстными*). Поясним, какой смысл вкладывается в термин «расщепление». Узел диаграммы, задающий ветвление вида **if-then-else**, описывается кортежем $\langle \lambda, \varphi, \lambda_{true}, \lambda_{false} \rangle$, где λ – метка узла, φ – булевское условие ветвления, λ_{true} и λ_{false} – метки узлов, в которые ведут дуги, помеченные соответственно константами *true* и *false*. Если узел имеет вид $\langle \lambda, (\varphi \vee \psi), \lambda_{true}, \lambda_{false} \rangle$, он заменяется на $\langle \lambda, \varphi, \lambda_{true}, \lambda' \rangle$ и $\langle \lambda', \psi, \lambda_{true}, \lambda_{false} \rangle$. Если узел имеет вид $\langle \lambda, (\varphi \wedge \psi), \lambda_{true}, \lambda_{false} \rangle$, он заменяется на $\langle \lambda, \varphi, \lambda', \lambda_{false} \rangle$ и $\langle \lambda', \psi, \lambda_{true}, \lambda_{false} \rangle$. Другие логические связки обрабатываются аналогичным образом. Заметим, что если в условие ветвления входят только либо переменные состояния, либо контекстные переменные, узел остается неизменным. «Расщепление» осуществляется только в том случае, когда есть возможность отделить переменные состояния от контекстных переменных (в условие ветвления входят переменные разных видов, но в нем присутствуют логические подформулы, содержащие только либо переменные состояния, либо контекстные переменные).

Построение множества условий на переменные состояния осуществляется путем обхода полученной диаграммы в глубину. Для этого заводится вспомогательный список формул, пустой в начале обхода. При первичном посещении нетерминального узла проверяется, содержит ли соответствующая формула переменные состояния, и если содержит, она добавляется в список. При достижении терминального узла, во множество условий на переменные со-

стояния добавляется конъюнкция формул, входящих в сформированный список. При повторном посещении узла (при выполнении возврата) формула удаляется из списка. Заметим, что построенные условия на переменные состояния, хотя и соответствуют разным путям диаграммы, не обязаны быть ортогональными (попарно несовместными): каждое такое условие получено из полной формулы пути удалением некоторых конъюнктивных членов – ограничений на контекстные переменные. Также отметим, что объявления переменных в HDL-описаниях могут включать *инварианты* (ограничения на возможные значения): перечисления, диапазоны и т. п. Во все условия на переменные состояния добавляются соответствующие инварианты (если они не тривиальны).

Ортогонализация условий на переменные состояния осуществляется с использованием аппарата булевой алгебры. Каждому конъюнктивному члену каждого условия ставится в соответствие буква – булева переменная с отрицанием (¬) или без него: заведомо эквивалентным формулам (в частности, совпадающим текстуально) сопоставляются одинаковые буквы; заведомо противоположным (например, когда одна формула является отрицанием другой) — противоположные (имеющие вид ν и ¬ν). Таким образом, каждому условию на переменные состояния ставится в соответствие слово – конъюнкция букв. Для выделения заведомо эквивалентных и заведомо противоположных формул используется процедура *канонизации*, унифицирующая (до некоторой степени) запись формул (для этого используются правила перезаписи (rewriting) вида $(\varphi \neq \psi) \rightarrow \neg(\varphi = \psi)$, $(\varphi \geq \psi) \rightarrow \neg(\varphi < \psi)$ и т. п.).

Ортогонализация выполняется на уровне булевых слов по следующей схеме (для наглядности знаки операции конъюнкции опущены; запись вида $\neg b$, где $b = \neg\nu$ следует понимать как ν) [21]:

1) если два слова содержат противоположные буквы, они ортогональны;

2) в противном случае в одном из слов (обозначим его w) присутствуют буквы $(b_1, ..., b_n)$, чьи переменные не входят в другое слово (w'):

- $w = a_1...a_m b_1...b_n$ остается без изменений;
- $w' = a_1...a_m c_1...c_k$ заменяется на следующие слова:

$a_1...a_m c_1...c_k \neg b_1$;
$a_1...a_m c_1...c_k b_1 \neg b_2$;
...
$a_1...a_m c_1...c_k b_1...b_{n-1} \neg b_n$.

В результате ортогонализации получается расширенное множество слов. Каждому из них ставится в соответствие условие на переменные состояния и собственно состояние EFSM-модели. Заметим, что некоторые условия, полученные таким образом, могут оказаться противоречивыми. Такие условия обнаруживаются с помощью средств проверки выполнимости формул (нами используется SMT-решатель Z3 [22]) и исключаются из рассмотрения.

Завершающий **шаг 6** заключается в построении отношения переходов EFSM-модели. Для каждой пары состояний и каждого охраняемого действия проверяется совместимость охранного условия с первым состоянием пары, а результата действия – со вторым; при положительных результатах проверок соответствующий переход добавляется в конструируемую модель:

1) для каждого состояния s и каждого охраняемого действия x, если охранное условие γ_x совместимо с состоянием s (совместимость проверяется с помощью SMT-решателя), создается *протопереход* $t_{s,x} = (s, \gamma_x \rightarrow \delta_x, -)$, чье конечное состояние (состояния) еще не определено;

2) для каждого состояния s' и каждого протоперехода $t_{s,x}$ методом обратных подстановок [18] строится слабейшее предусловие действия δ_x для постусловия s' (обозначим его $wp(\delta_x, s')$); если $wp(\delta_x, s')$ и s совместимы, переход $(s, (\gamma_x \wedge wp(\delta_x, s')) \rightarrow \delta_x, s')$ добавляется во множество переходов;

3) переходы между одними и теми же парами состояний, имеющие одинаковые охранные условия, «объединяются» (заменяются на один переход с объединенным действием).

Этапы 1 и 2 содержат вложенные циклы, внутри которых выполняются сложные манипуляции. Для оптимизации перебора ис-

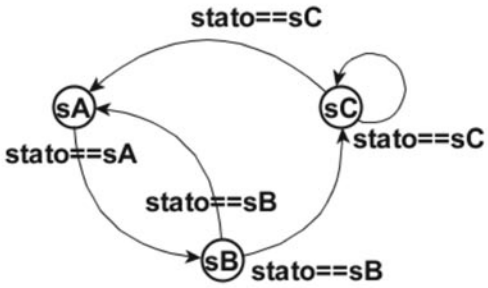

Рис. 3. Пример EFSM-модели

пользуется следующее соображение. Если выполнима импликация $\gamma_x \rightarrow s$, то начальным состоянием для охраняемого действия x может быть только s, и проверка других состояний не имеет смысла. То же касается $wp(\delta_x, s')$ и s. Для поиска кандидата на роль начального состояния для заданного условия анализируется вложенность формул, а также используются эвристики.

На рис. 3 схематично изображена EFSM-модель для HDL-описания b04, построенная с помощью описанного выше метода. В качестве переменной состояния была выделена переменная stato, а в качестве состояний – ограничения вида stato == c, где c – константа.

Состояния модели помечены соответствующими константами, а переходы – ограничениями на переменные состояния, входящими в состав охранных условий. Два перехода ($sB \rightarrow sA$ и $sC \rightarrow sA$) соответствуют сбросу состояния. Еще в двух переходах происходит изменение состояния ($sA \rightarrow sB$ и $sB \rightarrow sC$). Петля в вершине sC соответствует 23 переходам, реализующим вычисления.

Результаты экспериментов

Описанный метод извлечения EFSM-моделей из исходного кода HDL-описаний был реализован в прототипе инструмента HDL Retrascope. Разработка выполнена на языке программирования Java с использованием средств Z3 (SMT-решатель) [22], JUNG (библиотека работы с графами) [23], zamiaCAD (платформа для разработки и анализа HDL-описаний) [24] и Fortress (библиотека работы с формулами) [25]. Прототип позволяет анализировать описания цифровой аппаратуры на синтезируемых подмножествах (synthesizable subsets) языков VHDL и Verilog, строить и визуализировать систему расширенных конечных автоматов, моделирующих процессы HDL-описаний (в текущей версии не поддерживается анализ описаний, имеющих иерархическую структуру, а также описаний повышенной сложности).

В рамках работы проанализированы HDL-описания, входящие в пакет тестов ITC'99 [19]. Результаты применения метода к тестам пакета представлены в табл. 2. EFSM-модели были извлечены для 13 описаний из 22; остальные используют конструкции языка VHDL, не поддержанные в прототипе на момент проведения экспериментов (9 описаний). Для всех описаний (из 13 успешно обработанных) были извлечены синхросигналы, и во всех случаях их множества включали CLOCK и RESET. Для всех описаний все переменные, в имени которых присутствует подстрока STAT, были идентифицированы как переменные состояния (такие имена позволяют предположить, что они используются инженерами-проектировщиками для кодирования состояний управляющих автоматов).

Отметим, что для всех проанализированных описаний число состояний в извлеченных автоматах относительно невелико и растет с ростом числа выявленных переменных состояния. Число переходов также растет с увеличением числа состояний и числа путей выполнения (путей в GADD-диаграмме).

Расширенные конечные автоматы активно используются в области имитационной и формальной верификации цифровой аппаратуры. Известно множество подходов к генерации тестов и формальной проверке свойств, основанных на анализе EFSM-моделей (см. [6–9, 11–13, 15]). В статье рассмотрен метод построения расширенных конечных автоматов по исходному коду HDL-описаний. Отличительной чертой метода является автоматическое выделение внутренних переменных, представляющих состояния

Таблица 2

Результаты применения метода на пакете тестов ITC'99

Описание	Число строк	Синхросигналы	Переменные состояния	Число состояний	Число переходов
b01	102	clock, reset	stato	8	24
b02	70	clock, linea, reset	stato	7	17
b03	134	clock, reset	stato, fu1, fu2, fu3, fu4, coda0	23	467
b04	101	clock, reset	stato	3	29
b05	310	clock, reset, start	stato, flag, mar	9	700
b06	127	clock, cont_eql, reset, eql	state	7	33
b07	92	clock, reset, start	stato, mar	8	21
b08	88	clock, reset, start	stato, mar	5	13
b09	100	clock, reset	stato, d_in, old	8	22
b10	167	clock, reset, rts, rtr, test, start	stato	11	33
b11	118	clock, stbi, reset	stato, cont, cont1	13	46
b13	296	clock, dsr, reset, eoc	s1, s2, conta_tmp, mpx, next_bit, tx_conta, itfc_state	30	98
b15	671	clock, ready_n, reset, hold, na_n, bs16_n	state, state2, instqueuerd_addr, flush	24	360

устройства, а также использование техник символического выполнения для построения множества состояний и отношения переходов. Эксперименты показали применимость подхода к HDL-описаниям небольшой сложности (до 1000 строк кода). В ближайшем будущем планируется улучшить и испытать созданный прототип для описаний средней и повышенной сложности (порядка 10 000 строк кода) с учетом иерархической структуры модулей, а также провести сравнение метода с другими подходами.

Еще одним направлением исследований является развитие методов верификации на основе EFSM-моделей. Сюда относятся анализ *зависаний* (deadlocks) и *конфликтов доступа к данным* (races, hazards) [26], которые могут возникать при наличии нескольких параллельно работающих EFSM-моделей над общим множеством переменных. Здесь же следует упомянуть *конколическое тестирование* (concolic [concrete & symbolic] testing), основанное на комбинировании статических и динамических техник верификации [27]. Кроме того, извлеченную EFSM-модель можно использовать для получения сведений о порядке подачи воздействий на HDL-описание и приеме реакций от него (т. е. о *протоколе взаимодействия* с устройством). Эти сведения могут использоваться как для формальной верификации, так и для генерации шаблонов тестовых систем, осуществляющих имитационную верификацию.

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

СПИСОК ЛИТЕРАТУРЫ

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

2. **Лазарев В.Г., Пийль Е.И.** Синтез управляющих автоматов. М.: Энергоатомиздат, 1989. 328 с.

3. **Botros N.M.** HDL Programming Fundamentals: VHDL and Verilog. Charles River Media, 2005. 506 p.

4. **Holzmann G.J.** Design and Validation of Computer Protocols. Prentice Hall, 1990. 512 p.

5. **Баранов С.И., Майоров С.А., Сахаров Ю.П., Селютин В.А.** Автоматизация проектирования цифровых устройств. Л.: Судостроение, 1979. 264 с.

6. **Duale A.Y., Uyar M.U.** A Method Enabling Feasible Conformance Functional Test Sequence Generation for EFSM Models // IEEE Transactions on Computers. 2004. No. 53(5). Pp. 614–627.

7. **Guglielmo G., Guglielmo L., Fummi F., Pravadelli G.** Efficient Generation of Stimuli for Functional Verification by Backjumping Across Extended FSMs // J. of Electronic Testing. 2011. No. 27(2). Pp. 37–162.

8. **Fummi F., Marconcini C., Pravadelli G.** Functional Verification based on the EFSM Model // IEEE International High Level Design Validation and Test Workshop. 2004. Pp. 69–74.

9. **Guglielmo G., Fummi F., Pravadelli G., Soffia S., Roveri M.** Semi-formal Functional Verification by EFSM Traversing via NuSMV // IEEE International High Level Design Validation and Test Workshop. 2010. Pp. 58–65.

10. **Камкин А.С., Смолов С.А.** Метод извлечения EFSM-моделей из HDL-описаний: применение к функциональной верификации // Проблемы разработки перспективных микро- и наноэлектронных систем: сб. трудов. М.: ИППМ РАН, 2014. Ч. II. С. 113–118.

11. **Giomi J.-C.** Finite State Machine Extraction from Hardware Description Languages // ASIC Conference and Exhibition, 1995. Pp. 353–357.

12. **Bombieri N., Fummi F., Guarnieri V.** FAST: An RTL Fault Simulation Framework based on RTL-to-TLM Abstraction // J. of Electronic Testing. 2012. No. 28(4). Pp. 495–510.

13. **Cheng K.-T., Krishnakumar A.S.** Automatic Generation of Functional Vectors Using The Extended Finite State Machine Model // ACM Transactions on Design Automation of Electronic Systems. 1996. No. 1(1). Pp. 57–79.

14. **Бурдонов И.Б., Косачев А.С., Кулямин В.В.** Неизбыточные алгоритмы обхода ориентированных графов. Детерминированныйслучай // Программирование. 2003. № 29(5). С. 11–30.

15. **Hierons R.M.,Kim T.-H., Ural H.** Expanding an Extended Finite State Machine to Aid Testability // Computer Software and Applications Conference. 2002. Pp. 334–339.

16. **Brandt J., Gemьnde M., Schneider K., Shukla S., Talpin J.-P.** Integrating System Descriptions by Clocked Guarded Actions // Forum on Design Languages. 2011. Pp. 1–8.

17. **Ахо А.В., Лам М.С., Сети Р., Ульман Д.Д.** Компиляторы: принципы, технологии и инструментарий. М.: ИД «Вильямс», 2011. 1184 с.

18. **Floyd R.W.** Assigning Meaning to Programs // Symp. on Applied Mathematics, 1967. Pp. 19–32.

19. ITC'99 [электронный ресурс] / URL: http://www.cad.polito.it/downloads/tools/itc99.html

20. **Raik J., Reinsalu U., Ubar R., Jenihhin M., Ellervee P.** Code Coverage Analysis Using High-Level Decision Diagrams // IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems. 2008. Pp. 1–6.

21. **Закревский А.Д.** Параллельные алгоритмы логического управления. Мн.: Ин-т технической кибернетики АН Беларуси, 1999. 202 с.

22. Z3 [электронный ресурс] / URL: http://z3.codeplex.com

23. JUNG [электронный ресурс] / URL: http://jung.sourceforge.net

24. zamiaCAD [электронный ресурс] / URL: http://zamiacad.sourceforge.net

25. Fortress [электронный ресурс] / URL: http://forge.ispras.ru/projects/solver-api

26. **Kumar R., Tahar S.** Formal Verification of Pipeline Conflicts in RISC Processors // Proc. of the Conf. on European Design Automation. 1994. Pp. 284–289.

27. **Sen K.** Concolic Testing // IEEE/ACM Internat. Conf. on Automated Software Engineering. 2007. Pp. 571–572.

REFERENCES

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

2. **Lazarev V.G., Piyl Ye.I.** *Sintez upravlyayushchikh avtomatov* [*Synthesis of control machines*]. Moscow: Energoatomizdat Publ., 1989, 328 p. (rus)

3. **Botros N.M.** *HDL Programming Fundamentals: VHDL and Verilog.* Charles River Media, 2005, 506 p.

4. **Holzmann G.J.** *Design and Validation of*

Computer Protocols. Prentice Hall, 1990, 512 p.

5. **Baranov S.I., Mayorov S.A., Sakharov Yu.P., Selyutin V.A.** *Avtomatizatsiya proyektirovaniya tsifrovykh ustroystv* [*Computer-aided design of digital devices*]. Leningrad: Sudostroyeniye Publ., 1979, 264 p. (rus)

6. **Duale A.Y., Uyar M.U.** A Method Enabling Feasible Conformance Functional Test Sequence Generation for EFSM Models, *IEEE Transactions on Computers*, 2004, No. 53(5). Pp. 614–627.

7. **Guglielmo G., Guglielmo L., Fummi F., Pravadelli G.** Efficient Generation of Stimuli for Functional Verification by Backjumping Across Extended FSMs, *Journal of Electronic Testing*, 2011, No. 27(2). Pp. 37–162.

8. **Fummi F., Marconcini C., Pravadelli G.** Functional Verification based on the EFSM Model, *IEEE International High Level Design Validation and Test Workshop*, 2004, Pp. 69–74.

9. **Guglielmo G., Fummi F., Pravadelli G., Soffia S., Roveri M.** Semi-formal Functional Verification by EFSM Traversing via NuSMV, *IEEE International High Level Design Validation and Test Workshop*, 2010, Pp. 58–65.

10. **Kamkin A.S., Smolov S.A.** Metod izvlecheniya EFSM-modeley iz HDL-opisaniy: primeneniye k funktsionalnoy Verifikatsii [Extraction method EFSM-models of HDL-descriptions: application to functional verification], *Problemy razrabotki perspektivnykh mikro- i nanoelektronnykh sistem*. Moscow: IPPM RAN Publ., 2014, Vol. II, Pp. 113–118. (rus)

11. **Giomi J.-C.** Finite State Machine Extraction from Hardware Description Languages, *ASIC Conference and Exhibition*, 1995, Pp. 353–357.

12. **Bombieri N., Fummi F., Guarnieri V.** FAST: An RTL Fault Simulation Framework based on RTL-to-TLM Abstraction, *Journal of Electronic Testing*, 2012, No. 28(4), Pp. 495–510.

13. **Cheng K.-T., Krishnakumar A.S.** Automatic Generation of Functional Vectors Using The Extended Finite State Machine Model, *ACM Transactions on Design Automation of Electronic Systems*, 1996, No. 1(1), Pp. 57–79.

14. **Burdonov I.B., Kosachev A.S., Kulyamin V.V.** Neizbytochnyye algoritmy obkhoda oriyentirovannykh grafov. Determinirovannyy sluchay 14 [Non-redundant traversal algorithms directed graphs. Deterministic case], *Programmirovaniye*, 2003, No. 29(5). Pp. 11–30. (rus)

15. **Hierons R.M.,Kim T.-H., Ural H.** Expanding an Extended Finite State Machine to Aid Testability, *Computer Software and Applications Conference*, 2002, Pp. 334–339.

16. **Brandt J., Gemьnde M., Schneider K., Shukla S., Talpin J.-P.** Integrating System Descriptions by Clocked Guarded Actions, *Forum on Design Languages*, 2011, Pp. 1–8.

17. **Akho A.V., Lam M.S., Seti R., Ulman D.D.** *Kompilyatory: printsipy, tekhnologii i instrumentariy* [*Compilers: Principles, Technologies and Tools*]. Moscow: Vilyams Publ., 2011, 1184 p. (rus)

18. **Floyd R.W.** Assigning Meaning to Programs, *Symposium on Applied Mathematics*, 1967. Pp. 19–32.

19. ITC'99. Available: http://www.cad.polito.it/downloads/tools/itc99.html

20. **Raik J., Reinsalu U., Ubar R., Jenihhin M., Ellervee P.** Code Coverage Analysis Using High-Level Decision Diagrams, *IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, 2008, Pp. 1–6.

21. **Zakrevskiy A.D.** *Parallelnyye algoritmy logicheskogo upravleniya* [*Parallel algorithms for logic control*]. Mn.: Institut tekhnicheskoy kibernetiki AN Belarusi Publ., 1999, 202 p. (bel)

22. Z3. Available: http://z3.codeplex.com

23. JUNG. Available: http://jung.sourceforge.net

24. zamiaCAD. Available: http://zamiacad.sourceforge.net

25. Fortress. Available. http://forge.ispras.ru/projects/solver-api

26. **Kumar R., Tahar S.** Formal Verification of Pipeline Conflicts in RISC Processors, *Proceedings of the Conference on European Design Automation*, 1994, Pp. 284–289.

27. **Sen K.** Concolic Testing, *IEEE/ACM International Conference on Automated Software Engineering*, 2007, Pp. 571–572.

СМОЛОВ Сергей Александрович – *младший научный сотрудник Института системного программирования РАН.*

109004, Россия, Москва, ул. Александра Солженицына, д. 25.

E-mail: smolov@ispras.ru

SMOLOV, Sergey A. *Institute for System Programming of the Russian Academy of Sciences.*

109004, Alexander Solzhenitsyn Str. 25, Moscow, Russia.

E-mail: smolov@ispras.ru

КАМКИН Александр Сергеевич – *старший научный сотрудник Института системного программирования РАН, кандидат физико-математических наук.*
109004, Москва, ул. Александра Солженицына, д. 25.
E-mail: kamkin@ispras.ru

KAMKIN, Alexander S. *Institute for System Programming of the Russian Academy of Sciences.*
109004, Alexander Solzhenitsyn Str. 25, Moscow, Russia.
E-mail: kamkin@ispras.ru