Использование контрактных спецификаций для автоматизации функционального тестирования моделей аппаратного обеспечения

А.С. Камкин (kamkin@ispras.ru)

Аннотация. Контрактные спецификации в форме пред- и постусловий широко используются в программной инженерии для формального описания интерфейсов программных компонентов. Такие спецификации, с одной стороны, удобны для разработчиков, поскольку хорошо привязываются к архитектуре системы, с другой стороны, на их основе можно автоматически генерировать тестовые оракулы, проверяющие соответствие поведения целевой системы требованиям, описанным в спецификациях. В работе предлагается использовать контрактные спецификации для автоматизации функционального тестирования моделей аппаратного обеспечения, разработанных на таких языках, как VHDL, Verilog, SystemC, SystemVerilog и др. В статье подробно описаны особенности спецификации аппаратного обеспечения, приводится сравнение предлагаемого подхода с существующими методами спецификации, применяемыми в тестировании аппаратуры. В качестве базового подхода используется технология тестирования UniTESK, разработанная в Институте системного программирования РАН.

1. Введение

Современный мир не мыслим без огромного разнообразия электронных устройств. Мобильные телефоны, цифровые фотокамеры и переносные компьютеры давно стали неотъемлимыми атрибутами жизни человека. Специальные устройства управляют работой бытовой техники, контролируют бортовые системы самолетов и космических спутников, управляют медицинскими системами жизнеобеспечения. В основе практически всех этих систем лежит полупроводниковая аппаратура — кристаллы интегральных схем, состоящие из миллионов связанных друг с другом микроскопических транзисторов, которые, пропуская через себя электрические токи, реализуют требуемые функции.

Чтобы убедиться, что аппаратура работает *правильно*, то есть реализует именно те *функции*, которые от нее ожидают пользователи, на практике используют *функциональное тестирование*. Требования, предъявляемые к качеству тестирования аппаратного обеспечения, очень высоки. Это связано

не только с тем, что аппаратура лежит в основе всех информационных и управляющих вычислительных систем, в том числе достаточно критичных к сбоям и ошибкам. Большое влияние на формирование высоких требований оказывают также экономические факторы. В отличие от программного обеспечения, в котором исправление ошибки стоит сравнительно дешево, ошибка в аппаратном обеспечении, обнаруженная несвоевременно, может потребовать перевыпуск и замену продукции, а это сопряжено с очень высокими затратами. Так, известная ошибка в реализации инструкции FDIV микропроцессора Pentium[1] [1], заключающаяся в неправильном делении некоторых чисел с плавающей точкой, обошлась компании Intel в 475 миллионов долларов [2, 3]. С другой стороны, требования к срокам тестирования также очень высоки. Важно не затягивать процесс и выпустить продукт на рынок своевременно, пока он не потерял актуальность, и на него существует спрос.

Как разработать качественный продукт своевременно, используя ограниченные ресурсы? В настоящее время для проектирования аппаратного обеспечения используются языки моделирования высокого уровня, которые позволяют значительно ускорить процесс разработки за счет автоматической трансляции описания аппаратуры на *уровне регистровых передач* (*RTL*, *register transfer level*) в описание аппаратуры на *уровне логических вентелей* (*gate level*). Такие языки называются *языками описания аппаратуры* (*HDL*, *hardware description languages*), а модели, построенные на их основе — *HDL-моделями* или *RTL-моделями*[2]. Языки описания аппаратуры позволяют значительно повысить продуктивность разработки аппаратного обеспечения, но они не страхуют от всех ошибок, поэтому функциональное тестирование по-прежнему остается актуальной и востребованной задачей.

При современной сложности аппаратного обеспечения[3] невозможно разработать приемлимый набор тестов вручную за разумное время. Необходимы технологии автоматизированной разработки тестов. В настоящее время разработка таких технологий и поддерживающих их инструментов выделилась в отдельную ветвь *автоматизации проектирования электроники* (*EDA*, *electronic design automation*) — *автоматизацию тестирования* (*testbench automation*). Основной задачей тестирования является проверка соответствия поведения системы предъявляемым к ней требованиям. Для возможности автоматизации такой проверки требования к системе должны быть представлены в форме, допускающей автоматическую обработку. Такую

1 Pentium — торговая марка нескольких поколений микропроцессоров семейства x86, выпускаемых компанией Intel с 22 марта 1993 года.

2 Именно такие модели являются предметом исследования настоящей работы.

3 Число транзисторов в современных микросхемах достигает сотней миллионов. Согласно закону Мура (Moore) это число возрастает примерно вдвое через каждые 18-24 месяцев.

форму представления требований называют *формальными спецификациями* или просто *спецификациями*.

В работе рассматривается определенный вид спецификаций — *контрактные спецификации* (*contract specifications*). Контрактные спецификации и процесс проектирования на их основе (*DbC*, *Design-by-Contract*) были введены Бертраном Майером (Bertrand Meyer) в 1986 году в контексте разработки программного обеспечения [4, 5]. Центральная метафора подхода заимствована из бизнеса. Компоненты системы взаимодействуют друг с другом на основе взаимных *обязательств* (*obligations*) и *выгод* (*benefits*). Если компонент предоставляет окружению некоторую функциональность, он может наложить *предусловие* (*precondition*) на ее использование, которое определяет обязательство для клиентских компонентов и выгоду для него. Компонент также может гарантировать выполнение некоторого действия с помощью *постусловия* (*postcondition*), которое определяет обязательство для него и выгоду для клиентских компонентов.

Почему в своих исследованиях мы выбрали именно контрактные спецификации? Контрактные спецификации, с одной стороны, достаточно удобны для разработчиков, поскольку хорошо привязываются к архитектуре системы, с другой стороны, в силу своего представления стимулируют усилия по созданию независимых от реализации *критериев корректности целевой системы* [6]. Основное же их преимущество состоит в том, что они позволяют автоматически строить *тестовые оракулы*, проверяющие соответствие поведения целевой системы требованиям, описанным в спецификациях.

Несколько слов о том, как организована статья. Во втором, следующем за введением, разделе даются общие сведения о моделях аппаратного обеспечения и типичной организации аппаратуры. В третьем разделе описывается предлагаемый подход к спецификации и проверке требований к аппаратному обеспечению. Четвертый раздел содержит краткий обзор технологии тестирования UniTESK и инструмента разработки тестов CTESK. В нем также описан способ использования инструмента для спецификации аппаратуры. В пятом разделе приводится сравнение предлагаемого подхода с существующими методами спецификации аппаратного обеспечения. Шестой раздел описывает опыт практического применения подхода. Наконец, в последнем, седьмом разделе делается заключение и очерчиваются направления дальнейших исследований.

2. Модели аппаратного обеспечения

Перед тем как описывать предлагаемый подход, рассмотрим особенности моделей аппаратного обеспечения на таких языках, как VHDL [7], Verilog [8], SystemC [9], SystemVerilog [10] и др. Знание этих особенностей позволяет адекватно адаптировать контрактные спецификации в форме пред- и постусловий для функционального тестирования моделей аппаратного обеспечения.

2.1. Особенности моделей аппаратного обеспечения

Модели аппаратного обеспечения представляют собой системы из нескольких взаимодействующих *модулей*. Как и в языках программирования, модули используются для декомпозиции сложной системы на множество независимых или слабо связанных подсистем. У каждого модуля имеется *интерфейс* — набор входов и выходов, через которые осуществляется соединение модуля с окружением, и *реализация*, определяющая способ обработки модулем входных сигналов: вычисление значений выходных сигналов и изменение внутреннего состояния.

Обработка модулем входных сигналов инициируется *событиями* со стороны окружения. Под событиями в моделях аппаратного обеспечения понимаются любые изменения уровней сигналов. Поскольку обычно рассматриваются двоичные сигналы, выделяются два основных вида событий: *фронт сигнала* (*posedge*, *positive edge*) — изменение уровня сигнала с низкого на высокий – и срез сигнала (*negedge*, *negative edge*) — изменение уровня сигнала с высокого на низкий[4].

Как правило, каждый модуль состоит из нескольких статически созданных *параллельных процессов*[5], каждый из которых реализует следующий цикл: сначала осуществляется ожидание одного или нескольких событий из заданного набора событий, затем производится их обработка, после чего цикл повторяется. Набор событий, ожидаемых процессом для обработки, называется *списком чувствительности* (*sensitive list*) процесса. Будем называть процесс *пассивным*, если он находится в состоянии ожидания событий, и *активным* в противном случае.

Важной особенностью моделей аппаратного обеспечения является наличие в них понятия *времени*. Время моделируется целочисленной величиной; можно задавать физический смысл единицы времени. Для описания причинно-следственных отношений между событиями, происходящими в одну единицу модельного времени используется понятие *дельта-задержки* (*delta delay*). События, между которыми есть дельта-задержка, выполняются последовательно одно за другим, но в одну и ту же единицу модельного времени.

Для выполнения моделей аппаратного обеспечения с целью анализа их поведения обычно используется *симуляция по событиям* (*event-driven simulation*). В отличие от *симуляции по интервалам времени* (*time-driven simulation*), в которой значения сигналов и внутренние состояния модулей вычисляются через регулярные интервалы времени, в этом способе

[4] Мы не рассматриваем здесь разного рода неопределенные значения, часто используемые в моделировании аппаратного обеспечения.

[5] В дальнейшем будем называть такие процессы *модельными процессами*, чтобы отличать их от процессов операционной системы.

модель рассматривается только в те моменты времени, когда происходят некоторые события.

Работа *событийного симулятора* (*event-driven simulator*) состоит в следующем. В начале симуляции модельное время устанавливается в ноль. Далее в цикле, пока есть активные процессы[6], выбирается один из них и выполняется до тех пор, пока он не станет пассивным. После того как выполнены все активные процессы, симулятор проверяет, есть ли события, запланированные на текущий момент времени через дельта-задержку или на будущие моменты времени. Если такие события есть, симулятор изменяет модельное время на время ближайшего события, реализует события, запланированные на этот момент времени, перевычисляет множество активных процессов, после чего цикл повторяется. Если таких событий нет, симуляция заканчивается.

2.2. Типичная организация модулей аппаратного обеспечения

В дальнейшем будем считать, что спецификация и тестирование моделей аппаратного обеспечения осуществляется на уровне отдельных модулей.

В типичном случае работа модуля аппаратного обеспечения управляется *сигналом тактового импульса*, который для краткости будем называть *тактовым сигналом* или просто *часами*. Фронты (или срезы) тактового сигнала разбивают непрерывное время на дискретный набор интервалов, называемых *тактами*. Поведение модуля на текущем такте определяется значениями входных сигналов и внутренним состоянием модуля. Как правило, часть входов модуля определяет *операцию*, которую модулю следует выполнить; такие входы будем называть *управляющими* (*control*). Другая часть входов определяет *аргументы операции*; такие входы будем называть *информационными* (*informative*). Среди операций, реализуемых модулем, обычно присутствует специальная операция *NOP* (*no operation*), означающая бездействие модуля.

Модули аппаратного обеспечения могут быть организованы разными способами. В соответствии с длительностью операций, выполняемых модулем, эти операции бывают *однотактными* и *многотактными*. По способу организации выполнения операций модули делятся на *модули с поочередным выполнением операций*, *модули с конвейерным выполнением операций* и *модули с параллельным выполнением операций*.

Рассмотрим, как осуществляется выполнение модулем однотактной операции. До начала очередного такта окружение устанавливает на соответствующих входах модуля код операции и аргументы. Выполнение операции начинается модулем с началом такта. За этот такт модуль производит необходимые

[6] В начале симуляции активными являются процессы, осуществляющие инициализацию.

вычисления, изменяет внутреннее состояние и устанавливает значения выходных сигналов, которые окружение может использовать, начиная со следующего такта (Рис. 1).

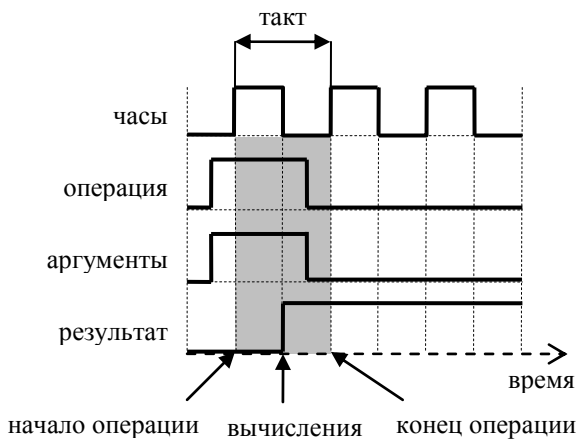

Рис. 1. Временна́я диаграмма сигналов для однотактной операции.

В отличие от однотактной операции, результат многотактной операции вычисляется постепенно, такт за тактом. Пусть операция f выполняется модулем за n тактов, тогда на каждом такте $\tau \in \{1, ..., n\}$ модуль выполняет некоторую *микрооперацию* f_τ, а окружение после окончания каждого такта получает некоторый частичный результат. Представление многотактовой операции f в виде последовательности микроопераций $(f_1, \ldots, f_n)$ будем называть *временно́й декомпозицией f*.

Теперь несколько слов о способах организации выполнения операций. В модулях с поочередным выполнением операций, как видно из названия, очередную операцию можно подавать на выполнение только после того, как полностью завершена предыдущая. В модулях с конвейерным выполнением операций операции можно подавать последовательно друг за другом, не дожидаясь завершения предыдущей операции. В модулях с параллельным выполнением операций несколько операций можно подавать одновременно. Модули с поочередным и конвейерным выполнением операций объединим общим термином — *модули с последовательным выполнением операций*, поскольку и в том, и в другом случае операции подаются на выполнение последовательно одна за другой.

В дальнейшем будем считать, что модули организованы таким образом, что одновременно выполняемые операции не вступают в *конфликты* (*hazards*), то есть не влияют друг на друга. Если взаимное влияние все-таки возможно,

требования должны описывать, как подавать операции на выполнение, чтобы избежать возникновения конфликтов.

3. Спецификация и проверка требований

Как отмечалось во введении, для возможности автоматизации проверки соответствия поведения системы требованиям они должны быть представлены в форме, допускающей автоматическую обработку. Такая форма представления требований называется *формальными спецификациями* или просто *спецификациями*. Рассмотрим разновидности требований к аппаратному обеспечению.

3.1. Требования к модулям аппаратного обеспечения

В общем случае операции являются многотактными, то есть выполняются модулем за несколько тактов. Требования на такие операции бывают двух основных типов: *требования на операцию в целом*, которые не накладывают ограничений на то, на каком именно такте выполняется та или иная микрооперация, и *требования на временну́ю композицию операции*, в которых фиксируется, на каких тактах выполняются конкретные микрооперации.

Требования на операцию в целом допускают определенную свободу в реализации модуля. Не важно, на каком такте выполняется некоторая микрооперация, важно, чтобы после завершения всей операции результат этой микрооперации был доступен окружению. В процессе тестирования требования на операцию в целом проверяются после завершения операции.

Требования на временну́ю композицию операции являются более жесткими. В них указаны такты, на которых выполняются микрооперации. Обычно при тестировании имеет смысл проверять не то, что микрооперация была выполнена на определенном такте τ_0, а то, что в конце этого такта соответствующим выходам модуля были присвоены требуемые значения, неважно на каком именно такте $\tau \in \{1, \ldots, \tau_0\}$.

При такой трактовке требования на операцию в целом являются частным случаем требований на временну́ю композицию; поэтому в дальнейшем мы не будем различать эти типы требований — просто будем считать, что каждому требованию соответствует номер такта, в конце которого его следует проверять.

3.2. Спецификация требований

Предлагаемый подход к представлению требований основан на использовании контрактных спецификаций в форме пред- и постусловий. В отличие от классического Design-by-Contract, когда контракты определяются на уровне операций, мы предлагаем определять контракты для отдельных микроопераций, а контракт для операции в целом получать путем *временно́й композиции* контрактов отдельных микроопераций (Рис. 2.).

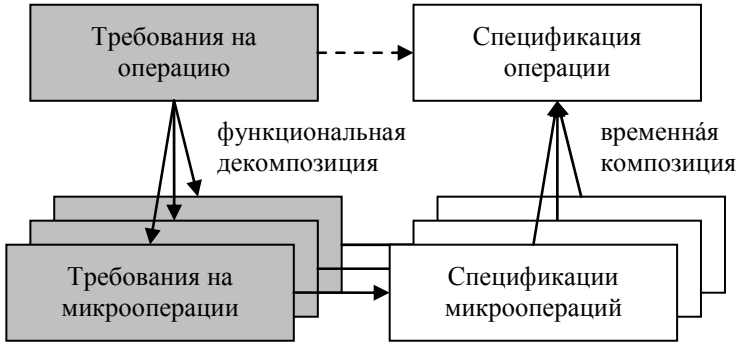

Рис. 2. Построение спецификации отдельной операции.

Здесь под микрооперациями мы понимаем некоторым образом выделенные аспекты функциональности операций, реализуемые за один такт работы модуля, а под временно́й композицией контрактов — спецификацию, в которой для каждой микрооперации указан номер такта, в конце которого должен выполняться соответствующий контракт.

В общих чертах процесс спецификации требований к отдельной операции состоит в следующем. Определяется предусловие, ограничивающее ситуации, в которых операцию можно подавать на выполнение. На основе анализа документации производится *функциональная декомпозиция* операции на набор микроопераций. Для каждой микрооперации определяется постусловие, описывающее требования к ней. После этого производится временна́я композиция спецификаций —постусловие каждой микрооперации помечается номером такта, в конце которого оно должно быть выполнено. Таким образом, контракт операции *f*, для которой выделено *n* микроопераций, формализуется структурой $C=(pre, \{(post_i, \tau_i)_{i=1,n}\})$[7].

Для контракта $C=(pre, \{(post_i, \tau_i)_{i=1,n}\})$ введем следующие обозначения. Через pre_C будем обозначать предусловие операции (*pre*), через $post_{C,i}$ — постусловие *i*-ой микрооперации ($post_i$), через $\tau_{C,i}$ — номер такта, в конце которого должно выполняться постусловие *i*-ой микрооперации (τ_i), через $Post_C(\tau)$ — конъюнкцию постусловий микроопераций, помеченных тактом τ, то есть $\wedge\{ post_{C,i} \mid \tau_{C,i}=\tau \}$.

[7] Для наглядности мы не вводим модель данных и не уточняем сигнатуры пред- и постусловий.

3.3. Проверка требований

После того как требования к модулю формализованы, проверка поведения модуля на соответствие им может осуществляться в процессе тестирования автоматически.

Предположим, что в некоторый момент времени t тестируемый модуль выполняет m операций $f_1, \ldots, f_m$, которые были поданы на выполнение раньше на $\tau_1, \ldots, \tau_m$ соответственно ($\tau_i \geq 1$, i=1, …, m). Пусть $C_1, \ldots, C_m$ — контракты операций $f_1, \ldots, f_m$ соответственно, и в моменты подачи операций $f_1, \ldots, f_m$ были выполнены предусловия $pre_{C1}, \ldots, pre_{Cm}$; тогда для проверки правильности поведения модуля в момент времени t необходимо проверить выполнимость предиката $Post_{C1}(\tau_1) \wedge \ldots \wedge Post_{Cm}(\tau_m)$.

Понятно, что для проверки соответствия поведения модуля требованиям также важно уметь строить хорошие тестовые последовательности, но рассмотрение этого вопроса выходит за рамки данной работы.

4. Технология тестирования UniTESK

В качестве базового подхода в работе используется технология тестирования UniTESK [11], разработанная в Институте системного программирования РАН [12]. Характерными чертами технологии являются использование контрактных спецификаций в форме пред- и постусловий интерфейсных операций и *инвариантов типов данных* для спецификации требований, а также применение *обобщенных конечно-автоматных моделей* для построения тестовых последовательностей.

4.1. Архитектура тестовой системы UniTESK

Архитектура тестовой системы UniTESK [13] была разработана на основе многолетнего опыта тестирования промышленного программного обеспечения из разных предметных областей и разной степени сложности. Учет этого опыта позволил создать гибкую архитектуру, основанную на следующем разделении задачи тестирования на подзадачи:

- построение тестовой последовательности, нацеленной на достижение нужного покрытия;
- создание единичного тестового воздействия в рамках тестовой последовательности;
- установление связи между тестовой системой и реализацией целевой системы;
- проверка правильности поведения целевой системы в ответ на единичное тестовое воздействие.

Для решения каждой из этих подзадач предусмотрены специальные компоненты тестовой системы (Рис. 3): для построения тестовой последовательности и создания единичных тестовых воздействий — *обходчик*

и *итератор тестовых воздействий*; для проверки правильности поведения целевой системы — *тестовый оракул*; для установления связи между тестовой системой и реализацией целевой системы — *медиатор*. Рассмотрим подробнее каждый из указанных компонентов.

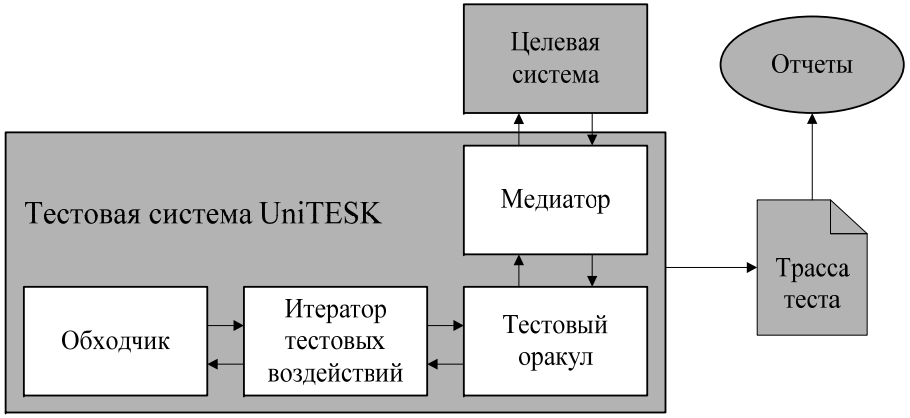

Рис. 3. Архитектура тестовой системы UniTESK.

Обходчик является библиотечным компонентом тестовой системы UniTESK и предназначен вместе с итератором тестовых воздействий для построения тестовой последовательности. В основе обходчика лежит алгоритм обхода графа состояний *обобщенной конечно-автоматной модели* целевой системы (конечного автомата, моделирующего целевую систему на некотором уровне абстракции). Обходчики, реализованные в библиотеках инструментов UniTESK, требуют, чтобы обобщенная конечно-автоматная модель целевой системы, была детерминированной[8] и имела сильно-связный граф состояний.

Итератор тестовых воздействий работает под управлением обходчика и предназначен для перебора в каждом достижимом состоянии конечного автомата допустимых тестовых воздействий. Итератор тестовых воздействий автоматически генерируется из тестового сценария, представляющего собой неявное описание обобщенной конечно-автоматной модели целевой системы.

Тестовый оракул оценивает правильность поведения целевой системы в ответ на единичное тестовое воздействие. Он автоматически генерируется на основе формальных спецификаций, описывающих требования к целевой системе в виде пред- и постусловий интерфейсных операций и инвариантов типов данных.

[8] Исключение составляет обходчик *ndfsm* [16], позволяющий обходить графы состояний для некоторого класса недетерминированных конечных автоматов.

Медиатор связывает абстрактные формальные спецификации, описывающие требования к целевой системе, с конкретной реализацией целевой системы. Медиатор преобразует единичное тестовое воздействие из спецификационного представления в реализационное, а полученную в ответ реакцию — из реализационного представления в спецификационное. Также медиатор синхронизирует состояние спецификации с состоянием целевой системы.

Трасса теста отражает события, происходящие в процессе тестирования. На основе трассы можно автоматически генерировать различные отчеты, помогающие анализировать результаты тестирования.

4.2. Инструмент разработки тестов CTESK

Инструмент CTESK [11], который используется в описываемой работе, является реализацией концепции UniTESK для языка программирования C. Для разработки компонентов тестовой системы в нем используется язык SeC (specification extension of C), являющийся расширением ANSI C. Инструмент CTESK включает в себя транслятор из языка SeC в C, библиотеку поддержки тестовой системы, библиотеку спецификационных типов и генераторы отчетов.

Компоненты тестовой системы UniTESK реализуются в инструменте CTESK с помощью специальных функций языка SeC, к которым относятся:

- *спецификационные функции* — содержат спецификацию непосредственной реакции целевой системы в ответ на единичное тестовое воздействие, а также определение структуры тестового покрытия;
- *функции отложенных реакций* — содержат спецификацию отложенных реакций целевой системы;
- *медиаторные функции* — связывают спецификационные функции с тестовыми воздействиями на целевую систему, а также реакции целевой системы с функциями отложенных реакций;
- *функция вычисления обобщенного состояния* — вычисляет состояние обобщенной конечно-автоматной модели целевой системы;
- *сценарные функции* — описывают набор тестовых воздействий для каждого достижимого обобщенного состояния.

В работах [14, 15] подробно описано, как базовая архитектура тестовой системы UniTESK может быть расширена для функционального тестирования моделей аппаратного обеспечения, разработанных на языках Verilog и SystemC. Там же приводятся технические детали, связанные с использованием инструмента CTESK для функционального тестирования таких моделей.

4.3. Использование CTESK для спецификации аппаратуры

Инструмент разработки тестов CTESK предоставляет достаточно универсальные средства для спецификации систем с асинхронным

интерфейсом [16]. Эти средства были адаптированы для спецификации модулей аппаратного обеспечения. Рассмотрим подробнее процесс разработки спецификаций.

Для каждой операции, реализуемой модулем, пишется спецификационная функция, в которой определяется предусловие операции и структура тестового покрытия. Постусловие спецификационной функции обычно возвращает **true**, поскольку все проверки, как правило, определяются в постусловиях микроопераций:

```
// спецификация операции
specification void operation_spec(...)
{
    // предусловие операции
    pre { ... }
    // определение структуры тестового покрытия
    coverage C { ... }
    // постусловие операции обычно возвращает true
    post { return true; }
}
```

Для каждой микрооперации, входящей в состав специфицируемой операции, пишется функция отложенной реакции, в которой определяется ее постусловие:

```
// спецификация микрооперации
reaction Operation* micro_return(void)
{
    // постусловие микрооперации
    post { ... }
}
```

Далее определяется *функция временно́й композиции микроопераций*, которая, во-первых, добавляет стимул (операцию вместе с набором аргументов) в *очередь стимулов* с указанием времени, необходимого для обработки стимула (*time*), во-вторых, для каждой микрооперации добавляет соответствующую реакцию в *очередь реакций* с указанием номера такта (относительно текущего времени), в конце которого следует осуществить проверку реакции ($tick_i$):

```
// временная композиция микроопераций
void operation_time_comp(...)
{
    Operation *descriptor = create_operation(...);

    // добавление стимула в очередь стимулов
    register_stimulus(create_stimulus(time,
descriptor)));

    // добавление реакций в очередь реакций
```

```
    register_reaction(micro1_return, tick1,
descriptor);
    ...
    register_reaction(micron_return, tickn,
descriptor);
}
```

Очередь стимулов содержит стимулы, обрабатываемые модулем в текущее время. Для каждого стимула в очереди хранится время, которое он уже обрабатывается. Очередь реакций содержит еще не завершенные микрооперации. Для каждой микрооперации хранится время, через которое микрооперация завершится и можно будет осуществить проверку реакции. Изменение времени осуществляется *функцией сдвига времени*. После изменения времени вызываются *функции обработки очередей стимулов* и *реакций*. Функция обработки очереди стимулов удаляет из очереди полностью обработанные стимулы. Функция обработки очереди реакций регистрирует отложенные реакции для всех завершившихся микроопераций, которые после этого удаляются из очереди.

Для иллюстрации синтаксиса языка SeC приведем очень простой пример. Рассмотрим устройство, называемое *8-ми битным счетчиком* (Рис. 4).

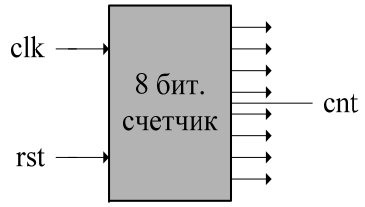

Рис. 4. Схема входов и выходов 8-ми битного счетчика.

Интерфейс счетчика состоит из двух двоичных входов `clk` и `rst` и одного 8-ми битного выходного регистра `cnt`. Если уровень сигнала `rst` низкий, фронт сигнала тактового импульса `clk` увеличивает счетчик `cnt` по модулю 256; иначе счетчику присваивается значение 0.

Ниже приводится спецификационная функция, описывающая операцию увеличения счетчика, то есть поведение счетчика в ответ на фронт `clk` при низком уровне сигнала `rst`. Поскольку операция является простой, спецификация выполнена без привлечения функций отложенных реакций.


```
// спецификация операции увеличения счетчика
specification void increment_spec(counter_8bit *counter)
  updates cnt = counter->cnt,
          rst = counter->rst
{
  // предусловие операции
  pre { return rst == false; }
    // определение структуры тестового покрытия
  coverage C { return { SingleBranch, "Single branch" }; }
    // постусловие операции
  post { return cnt == (@cnt + 1) % 0xff; }
}
```

5. Сравнение с существующими подходами

В данном разделе приводится сравнение предлагаемого подхода с существующими методами спецификации аппаратуры, поддерживаемыми современными *языками верификации аппаратуры* (*HVL*, *hardware verification languages*). Языки верификации аппаратуры, к которым относятся PSL, OpenVera, SystemVerilog и др. [17], включают в себя конструкции языков описания аппаратуры, языков программирования, а также специальные средства, ориентированные на разработку спецификаций и тестов. К последним относятся средства спецификации поведения, определения структуры тестового покрытия и генерации тестовых данных. Мы сравниваем только способы спецификации.

Средства спецификации поведения, используемые в современных языках верификации аппаратуры, базируются на *темпоральной логике линейного времени* (*LTL*, *linear temporal logic*) и/или *темпоральной логике ветвящегося времени* (*CTL*, *computation tree logic*) [17]. Языки, по крайней мере, по части спецификации, имеют следующие корни: ForSpec (Intel) [18] (для языков, использующих логику LTL[9]) и Sugar (IBM) [19] (для языков, использующих логику CTL). Ниже приведена диаграмма, показывающая влияние некоторых языков верификации аппаратуры друг на друга.

Логика CTL используется преимущественно для формальной верификации систем. Для целей симуляции и тестирования больший интерес представляет логика линейного времени LTL. Поскольку все языки верификации аппаратуры, в которых используется LTL, имеют схожие средства спецификации, для сравнения с предлагаемым подходом мы будем использовать только один из них — OpenVera [20]. Данный язык поддерживается многими инструментами; кроме того, он является открытым.

[9] Вариант логики LTL, используемой в ForSpec, называется FTL (ForSpec temporal logic) [18].

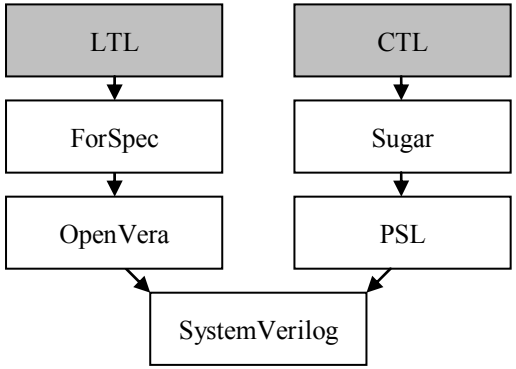

Рис. 5. Влияние языков верификации аппаратуры друг на друга.

Язык OpenVera был разработан в 1995 году компанией Systems Science. Первоначальное название языка — Vera. В 1998 году System Science была поглощена компанией Synopsys. В 2001 году Synopsys сделала язык открытым и переименовала его в OpenVera. Для спецификации поведения OpenVera предоставляет специальный язык формулирования *темпоральных утверждений* (*temporal assertions*), который называется OVA (OpenVera assertions) [21, 22].

OVA оперирует с ограниченными по времени последовательностями событий, в которых можно обращаться к прошлому и будущему. Из простых последовательностей можно строить более сложные с помощью логических связок, таких как AND и OR, или используя регулярные выражения. В языке имеются средства объединения темпоральных утверждений в параметризованные библиотеки спецификаций. Проиллюстрируем синтаксис OVA на простом примере.

```
// тактовый сигнал
clock negedge(clk)
{
    // граничные значения счетчика
    bool cnt_00: (cnt == 8'h00);
    bool cnt_ff: (cnt == 8'hff);

    // событие переполнения счетчика
    event e_overflow: cnt_ff #1 cnt_00;
}

// утверждение, запрещающее переполнение счетчика
assert a_overflow: forbid(e_overflow);
```


В примере определяется событие переполнения счетчика `e_overflow`, а утверждение `a_overflow` запрещает возникновение такого события.

В подходе OpenVera, как и в других подходах на основе темпоральных логик, упор делается на *временну́ю декомпозицию* операций. Для каждой операции сначала выделяется ее временна́я структура — допустимые последовательности событий и задержки между ними; затем определяются предикаты, описывающие отдельные события; после этого предикаты, относящиеся к одному моменту времени, некоторым образом группируются (Рис. 6).

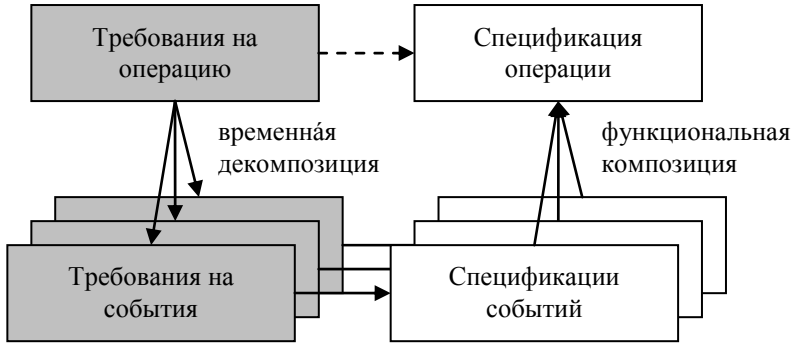

Рис. 6. Построения спецификации в подходах на основе темпоральных логик.

В подходе, предлагаемом нами, основной акцент ставится на *функциональную декомпозицию* операций. Первым делом выделяется функциональная структура операции — набор микроопераций; каждая микрооперация специфицируется; после этого производится временна́я композиция спецификаций (Рис. 2).

Мы полагаем, что функциональная структура операции более устойчива по сравнению с временно́й. Тем самым, подходы, основанные на функциональной декомпозиции операций, позволяют разрабатывать спецификации более устойчивые к изменениям реализации по сравнению с подходами на основе временно́й декомпозиции.

К достоинствам предлагаемого подхода также можно отнести наглядность и простоту. Пред- и постусловия обычно понятнее формул темпоральной логики и не требуют от разработчика тестов каких-нибудь специальных знаний.

6. Опыт практического применения подхода

Предлагаемый подход был применен на практике при тестировании *буфера трансляции адресов* (*TLB*, *translation lookaside buffer*) микропроцессора с MIPS64-совместимой архитектурой [23, 24].

Буфер трансляции адресов, входящий в состав большинства современных микропроцессоров, предназначен для кэширования *таблицы страниц* — таблицы операционной системы, хранящей соответствие между номерами виртуальных и физических страниц памяти. Использование такого буфера позволяет значительно увеличить скорость трансляции адресов. Буфер представляет собой ассоциативную память с фиксированным числом записей. Помимо интерфейса для трансляции адресов, он предоставляет интерфейс для чтения и изменения содержимого этой памяти.

Трансляция виртуального адреса осуществляется следующим образом. Если буфер содержит запись с нужным номером виртуальной страницы, в выходном регистре модуля формируется соответствующий физический адрес; в противном случае, на одном из выходов модуля устанавливается сигнал, говорящий об отсутствии в буфере требуемой записи.

6.1. Структура и функциональность модуля

Рассмотрим устройство тестируемого модуля. Память TLB состоит из 64 ячеек, которые составляют *объединенный TLB* (*JTLB*, *joint TLB*). Кроме того, для повышения производительности модуль содержит дополнительные буферы: *TLB данных* (*DTLB*, *data TLB*) и *TLB инструкций* (*ITLB*, *instruction TLB*). DTLB используется при трансляции адресов данных, ITLB — при трансляции адресов инструкций. Оба буфера содержат по 4 ячейки, содержимое буферов является подмножеством JTLB, обновление происходит по алгоритму LRU (last recently used).

Каждая ячейка TLB условно делится на две секции: секция-ключ и секция-значение. Секция-ключ включает в себя спецификатор сегмента памяти (*R*), номер виртуальной страницы, деленный на два (*VPN2*), идентификатор процесса (*ASID*), бит глобальной трансляции адресов (*G*) и маску страницы (*MASK*). Секция-значение состоит из двух подсекций, каждая из которых содержит номер физической страницы (PFN_i), бит разрешения чтения (V_i), бит разрешения записи (D_i) и политику кэширования страницы (C_i). Какая именно подсекция будет использована при трансляции адреса, определяется младшим битом номера виртуальной страницы.

Интерфейс тестируемого модуля TLB состоит из 30 входов (16 входов общего назначения, 3 входа DTLB, 4 входа ITLB, 7 входов JTLB) и 31 выходов (6 выходов общего назначения, 9 выходов DTLB, 8 выходов ITLB, 8 выходов JTLB)[10]. Функциональность модуля включает операции записи, чтения и проверки наличия ячейки в памяти, а также операции трансляции адресов данных и инструкций. RTL-модель модуля разработана на языке Verilog и составляет ≈8 000 строк кода.

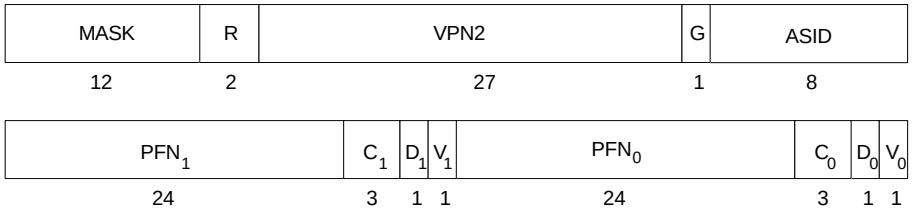

Рис. 7. Структура ячейки буфера трансляции адресов.

6.2. Разработка спецификаций модуля

Основные требования к буферу трансляции адресов были получены в письменной форме от разработчиков модуля. В процессе формализации требования уточнялись в результате общения с разработчиками и чтения технической документации. Следует отметить, что все сформулированные разработчиками требования были легко представлены в форме пред- и постусловий.

Проект продемонстрировал удобство и сравнительно небольшую трудоемкость разработки контрактных спецификаций для моделей аппаратного обеспечения. Спецификации были разработаны одним человеком за ≈2 недели, а их объем составил ≈2 500 строк кода на SeC. Отметим также, что в результате проекта было найдено несколько ошибок в реализации модуля.

7. Заключение

Изначально контрактные спецификации были предложены для описания интерфейсов программных компонентов, но при определенной доработке их вполне можно использовать для описания модулей аппаратного обеспечения. Такие спецификации, с одной стороны, удобны для разработчиков, поскольку хорошо привязываются к архитектуре системы, с другой стороны, на их основе можно автоматически генерировать тестовые оракулы, проверяющие соответствие поведения целевой системы требованиям, описанным в

[10] При подсчете числа входов и выходов не учитывался интерфейс JTAG (joint test action group) — стандартный интерфейс, используемый для тестирования аппаратуры с помощью метода граничного сканирования.

спецификациях. Практическая апробация подхода в проекте по тестированию буфера трансляции адресов микропроцессора показала удобство представления требований к аппаратуре в форме пред- и постусловий и продемонстрировала сравнительно небольшую трудоемкость разработки спецификаций.

На настоящий момент нами получен определенный опыт использования технологии тестирования UniTESK и инструмента CTESK для спецификации и тестирования моделей аппаратного обеспечения. Опыт показывает, что некоторые шаги разработки тестов могут быть полностью или частично автоматизированы. Детальное исследование этого вопроса и разработка инструментальной поддержки для автоматизации шагов разработки тестов является основным направлением дальнейшей работы.

Литература

[1] Statistical Analysis of Floating Point Flaw in the Pentium Processor. Intel Corporation, November 1994.
[2] B. Beizer. The Pentium Bug – An Industry Watershed. Testing Techniques Newsletter (TTN), TTN Online Edition, September 1995.
[3] A. Wolfe. For Intel, It's a Case of FPU All Over Again. EE Times, May 1997.
[4] B. Meyer. Design by Contract. Technical Report TR-EI-12/CO, Interactive Software Engineering Inc., 1986.
[5] B. Meyer. Applying `Design by Contract'. IEEE Computer, vol. 25, No. 10, October 1992.
[6] А.В. Баранцев, И.Б. Бурдонов, А.В. Демаков, С.В. Зеленов, А.С. Косачев, В.В. Кулямин, В.А. Омельченко, Н.В. Пакулин, А.К. Петренко, А.В. Хорошилов. Подход UniTesK к разработке тестов: достижения и перспективы. (Опубликовано на http://www.citforum.ru/SE/testing/unitesk/)
[7] IEEE Standard VHDL Language Reference Manual. IEEE Std 1076-1987.
[8] IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language. IEEE Std 1364-1995.
[9] http://www.systemc.org
[10] http://www.systemverilog.org
[11] http://www.unitesk.com
[12] http://www.ispras.ru
[13] I. Bourdonov, A. Kossatchev, V. Kuliamin, and A. Petrenko. UniTesK Test Suite Architecture. FME'2002. LNCS 2391, Springer-Verlag, 2002.
[14] В.П. Иванников, А.С. Камкин, В.В. Кулямин, А.П. Петренко. Применение технологии UniTESK для функционального тестирования моделей аппаратного обеспечения. Препринт 8, Институт системного программирования РАН, Москва, 2005. (Опубликовано на http://citforum.ru/SE/testing/unitesk_hard/)
[15] A. Kamkin. The UniTESK Approach to Specification-Based Validation of Hardware Designs. IEEE-ISoLA'2006: The 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, November 2006.
[16] А.В. Хорошилов. Спецификация и тестирование систем с асинхронным интерфейсом. Препринт 12, Институт системного программирования РАН, Москва, 2006. (Опубликовано на http://www.citforum.ru/SE/testing/asynchronous_interface/)

[17] S.A. Edwards. Design and Verification Languages. Technical Report, Columbia University, New York, USA, November 2004.
[18] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M. Vardi, and Y. Zbar. The ForSpec Temporal Logic: A New Temporal Property-Specification Language. Tools and Algorithms for Construction and Analysis of Systems, 2002.
[19] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The Temporal Logic Sugar. Lecture Notes in Computer Science, 2001.
[20] http://www.open-vera.com
[21] OpenVera® Language Reference Manual: Assertions. Version 1.4.1, November 2004.
[22] OpenVera® Assertions. Blueprint for Productivity and Product Quality. March 2003. (Опубликовано на http://www.synopsys.com/products/simulation/ova_wp.html)
[23] http://www.mips.com/content/Products/Architecture/MIPS64
[24] MIPS64™ Architecture For Programmers. Revision 2.0. MIPS Tecnologies Inc., June 9, 2003.