Применение технологии UniTesK для функционального тестирования моделей аппаратного обеспечения

В.П. Иванников, А.С. Камкин, В.В. Кулямин, А.К. Петренко
{ivan,kamkin,kuliamin,petrenko}@ispras.ru

Аннотация. В работе обсуждаются вопросы применимости технологии автоматизированного тестирования UniTesK к разработке функциональных тестов для моделей аппаратного обеспечения. Предлагаются способы расширения базовой архитектуры тестовой системы UniTesK для функционального тестирования моделей на языках Verilog HDL и SystemC. Для каждого из указанных классов моделей описывается процесс разработки теста с помощью инструмента CTesK и приводятся оценки возможности автоматизации шагов этого процесса.

1. Введение

Технология тестирования UniTesK [1-3] была разработана в Институте системного программирования РАН (ИСП РАН) [4]. Первоначальное и основное назначение технологии — разработка качественных функциональных тестов для программного обеспечения. В работе рассматривается новая и достаточно перспективная область применения технологии UniTesK — функциональное тестирование моделей аппаратного обеспечения.

Под моделями аппаратного обеспечения в данной работе понимаются, прежде всего, HDL-модели (*HDL models*), то есть модели, написанные на каком-нибудь языке описания аппаратуры (*HDL*, *hardware description language*), например, на VHDL или Verilog HDL [5], а также модели системного уровня, разрабатываемые на таких языках, как SystemC [6] или SystemVerilog [7]. Заметим, что в работе рассматриваются только модели цифрового аппаратного обеспечения.

Известно, что слабым звеном в технологической цепочке проектирования сложного аппаратного обеспечения является функциональная верификация. Согласно Бергерону (*Janick Bergeron*) [8] функциональная верификация занимает около 70% общего объема трудозатрат, число инженеров, занимающихся верификацией, примерно вдвое превосходит число проектировщиков, а размер исходного кода тестов (*testbenches*) достигает 80% общего размера кода проекта.

Технология UniTesK и реализующие ее инструменты были успешно применены для тестирования различных классов программных систем: ядер операционных систем, телекоммуникационных протоколов, систем реального

времени и т.д. Ключевым моментом в успешности применения технологии UniTesK является гибкая архитектура тестовой системы, которая позволяет легко адаптировать технологию к различным классам целевых систем [2].

В работе предлагаются способы расширения базовой архитектуры тестовой системы UniTesK для функционального тестирования моделей аппаратного обеспечения на языках Verilog HDL и SystemC. Для каждого из указанных классов моделей описывается процесс разработки теста с помощью инструмента CTesK [9] и приводятся оценки возможности автоматизации шагов этого процесса. При расширении архитектуры тестовой системы мы руководствовались тем соображением, что трудоемкость разработки дополнительных компонентов теста должна быть минимальной.

Языки Verilog HDL и SystemC были выбраны для исследования по следующим причинам. Язык Verilog HDL, на ряду с VHDL, является классическим языком описания аппаратуры, но, в отличие от последнего, он имеет близкий к языку программирования C синтаксис и стандартный интерфейс для вызова функций, написанных на C [5], что облегчает интеграцию с инструментом CTesK, кроме того, он гораздо чаще используется на практике для проектирования сложного аппаратного обеспечения. Что касается SystemC, он является современным и многообещающим языком системного уровня, позволяющим моделировать системы со смешанными аппаратными и программными частями (*HW/SW*) [10]. Таким образом, были выбраны два достаточно разных языка: один — классический язык описания аппаратуры, другой — современный язык системного уровня.

Структура статьи такова. Во втором, следующем за введением, разделе делается краткий обзор технологии UniTesK и инструмента CTesK. В третьем разделе рассматриваются особенности моделей аппаратного обеспечения и то, как эти особенности соотносятся с технологией UniTesK. Там же описывается пример небольшого устройства, для которого с помощью инструмента CTesK разрабатываются спецификация и сценарий тестирования. В четвертом и пятом разделах описывается процесс разработки теста для Verilog- и SystemC-моделей соответственно. В заключении рассматриваются направления дальнейшего развития предлагаемого подхода.

2. Краткий обзор технологии UniTesK

Технология UniTesK была разработана в ИСП РАН на основе опыта, полученного при разработке и применении технологии KVEST (*kernel verification and specification technology*) [11]. Общими чертами этих технологий являются использование формальных спецификаций в форме пред- и постусловий интерфейсных операций и инвариантов типов данных для автоматической генерации оракулов (компонентов тестовой системы, осуществляющих проверку правильности поведения целевой системы), а также применение конечно-автоматных моделей для построения последовательностей тестовых воздействий.

В отличие от технологии KVEST, в которой для спецификации требований использовался язык RSL (*RAISE specification language*) [12], технология UniTesK использует расширения широко известных языков программирования. На данный момент в ИСП РАН разработаны инструменты, поддерживающие работу с расширениями языков C, Java и C#: CTesK [9], J@T [13] и Ch@se [14] соответственно.

2.1. Архитектура тестовой системы UniTesK

Архитектура тестовой системы UniTesK [1-3] 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 [2]:

- Построение последовательности тестовых воздействий, нацеленной на достижение нужного покрытия.
- Создание единичного тестового воздействия в рамках 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: для построения последовательности тестовых воздействий и создания единичных воздействий — обходчик и итератор тестовых воздействий, для проверки правильности поведения целевой системы — оракул, для установления связи между тестовой системой и реализацией целевой системы — медиатор. Рассмотрим подробнее каждый из указанных компонентов.

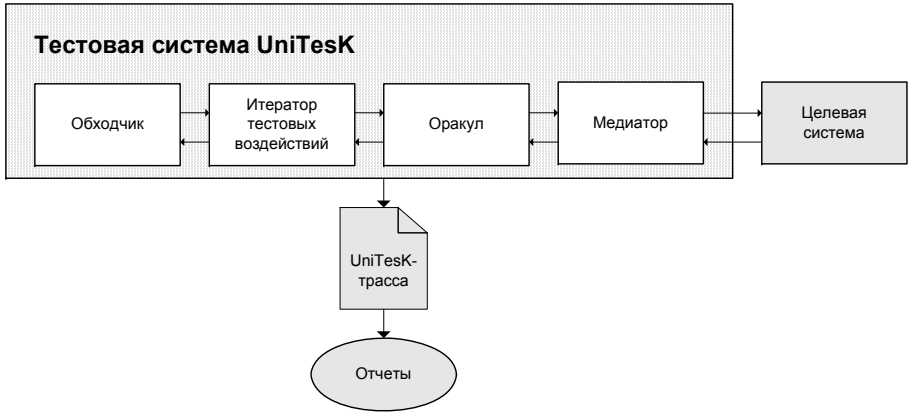

Рис. 1. Архитектура тестовой системы UniTesK.

Обходчик является библиотечным компонентом тестовой системы UniTesK и предназначен вместе с итератором тестовых воздействий для построения последовательности тестовых воздействий. В основе обходчика лежит алгоритм обхода графа состояний конечного автомата, моделирующего целевую систему на некотором уровне абстракции.

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 описание конечно-автоматной модели целевой системы.

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.

Медиатор связывает абстрактные формальные спецификации, описывающие требования к целевой системе, с конкретной реализацией целевой системы.

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

2.2. Инструмент разработки тестов CTesK

Инструмент CTesK [9], используемый в данной 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. Для пользователей Windows имеется модуль интеграции в среду разработки Microsoft Visual Studio 6.0.

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

- спецификационные функции — содержат спецификацию поведения целевой системы в ответ на единичное тестовое воздействие, а также определение структуры тестового покрытия;
- медиаторные функции — связывают спецификационные функции с тестовыми воздействиями на целевую систему;
- функция вычисления состояния теста — вычисляет состояние конечного автомата, моделирующего целевую систему.
- сценарные функции — описывают набор тестовых воздействий для каждого достижимого состояния теста;

Обходчик из библиотеки инструмента CTesK требует, чтобы конечный автомат, способ построения которого описан в тестовом сценарии (с помощью

функций двух последних видов), являлся детерминированным, а также имел сильно-связный граф состояний.

3. Особенности моделей аппаратного обеспечения

Модели аппаратного обеспечения на таких языках как Verilog HDL [5] и SystemC [6] 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. Каждый модуль имеет интерфейс — набор входов и выходов, через которые осуществляется соединение модуля с окружением, и реализацию, определяющую 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 понимают любые изменения уровней сигналов. Поскольку обычно рассматривают двоичные сигналы, выделяют 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[1].

Как правило, каждый модуль состоит из нескольких статически заданных параллельных процессов[2], каждый из которых реализует следующий цикл: сначала осуществляется ожидание одного или нескольких событий из заданного набора событий, затем 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: поведение таких моделей определяется не только последовательностью событий, но и длительностью временны*х* интервалов между ними. Время моделируется дискретной целочисленной величиной, физический смысл единицы времени можно задавать. 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 с целью анализа их поведения обычно используют симуляцию 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

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

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

времени, в этом способе модель рассматривается только в те моменты времени, когда наступают некоторые события.

Работа событийного симулятора (*event-driven simulator*) осуществляется следующим образом. В начале симуляции модельное время устанавливается в ноль. Далее в цикле, пока есть активные процессы[3], выбирается один из них и выполняется до тех пор, пока этот процесс не станет пассивным.

После того, как выполнены все активные процессы, симулятор проверяет, есть ли события, запланированные через дельта-задержку на текущий момент времени. Если такие события есть, симулятор реализует эти события и перевычисляет множество активных процессов, после чего цикл повторяется.

Если, после очередного выполнения цикла, событий запланированных на текущий момент времени нет, симулятор проверяет, есть ли события, запланированные на будущие моменты времени. Если таких событий нет, симуляция заканчивается, в противном случае, симулятор изменяет модельное время на время ближайшего события, реализует события, запланированные на этот момент времени и перевычисляет множество активных процессов, после чего цикл повторяется.

3.1. Модели аппаратного обеспечения и технология UniTesK

Теперь, после того как мы сделали краткий обзор технологии UniTesK и рассмотрели особенности моделей аппаратного обеспечения на языках высокого уровня, обсудим вопрос о применимости технологии UniTesK к функциональному тестированию таких моделей.

Традиционная архитектура тестовой системы для тестирования моделей аппаратного обеспечения выглядит следующим образом:

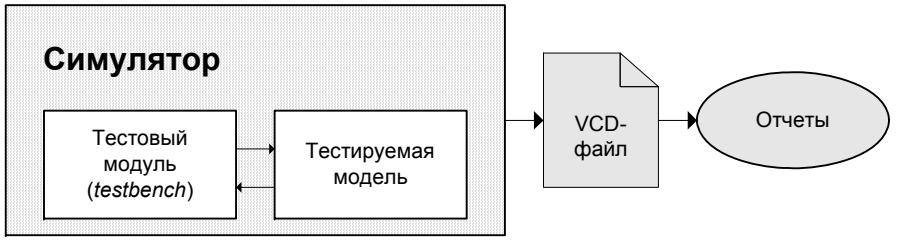

Рис. 2. Традиционная архитектура тестовой системы.

Тестовый модуль (*testbench*) последовательно подает на тестируемую модель тестовые воздействия и осуществляет проверку правильности реакций на них. Результатом теста является VCD[4]-файл, создаваемый симулятором и

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

[4] VCD (Value Change Dump) — формат для описания изменений значений сигналов во времени.

содержащий изменения значений сигналов на входах и выходах тестируемой модели во времени. Полученный файл обычно используется для визуализации волновой диаграммы теста — традиционного средства анализа результатов тестирования.

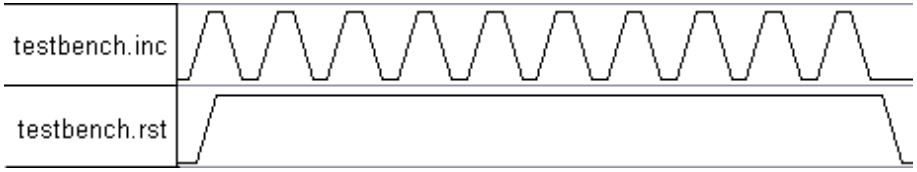

Рис. 3. Волновая диаграмма теста.

Наша цель — расширить тестовый модуль так, чтобы он включал в себя полномасштабную тестовую систему CTesK.

Для начала рассмотрим следующие общие вопросы:

- Как специфицировать модели аппаратного обеспечения на языке SeC.
- Как использовать понятие времени в рамках технологии UniTesK.
- Как адаптировать тестовую систему CTesK к выполнению в симуляторе.

Сначала рассмотрим вопрос о спецификации моделей аппаратного обеспечения. Для каждого процесса, определенного в модели, классифицируем используемые им входы на управляющие, информационные и несущественные. Будем называть вход управляющим, если процесс реагирует на возникающие на данном входе события, информационным, если значение сигнала на данном входе влияет на поведение процесса, все остальные входы будем называть несущественными.

Под единичным тестовым воздействием будем понимать непустое непротиворечивое множество событий из списка чувствительности одного из процессов, реализованных одновременно, а также набор значений сигналов на информационных входах этого процесса. Предположим, что списки чувствительности процессов, определенных в модели, попарно не пересекаются, а управляющие входы одного процесса не являются информационными входами другого. При таких ограничениях, тестовое воздействие может активизировать только один модельный процесс[5].

Для спецификации модели на языке SeC, множество всех тестовых воздействий должно быть разбито на группы, каждая из которых описывается одной спецификационной функцией. Один из возможных подходов разбиения состоит в следующем. Каждому процессу, определенному в модели, ставится в соответствие отдельная спецификационная функция, параметры которой

[5] В случае если одно тестовое воздействие может активизировать несколько модельных процессов, схема тестирования немного усложняется: для проверки правильности поведения модели нужно использовать возможные сериализации реакций активизированных процессов.

отражают множество событий из списка чувствительности, реализуемые на управляющих входах, а также значения подаваемых на информационные входы сигналов. В предусловии спецификационной функции, помимо разного рода смысловых проверок, должна осуществляться проверка того, что значения сигналов на управляющих входах процесса позволяют реализовать указанные в параметрах события, а в постусловии — то, что указанные события действительно были реализованы.

Проиллюстрируем этот подход на примере. Рассмотрим модуль на языке Verilog HDL.

```
module Module(x, y, z, r);
  input  x, y, z;
  output r; reg r;

  always @(posedge(x), negedge(y))
    begin: Process r = z; end

endmodule
```

Интерфейс модуля состоит из трех входов: `x`, `y`, `z` и одного выхода `r`. Для процесса, определенного в модуле, входы `x` и `y` являются управляющими, а вход `z` — информационным. При возникновении фронта сигнала на входе `x` или среза сигнала на входе `y` процесс присваивает `r` значение сигнала на входе `z`. Для спецификации возможности наступления событий из списка чувствительности процесса, состояние спецификационной модели данных должно включать текущие значения сигналов на входах `x` и `y`.

```
// спецификационная модель данных модуля
typedef struct
{
  // текущие значения сигналов на входах x и y
  bool x, y;

} Module;
```

Спецификационная функция, описывающая поведение процесса, будет иметь в качестве параметров указатель на состояние спецификационной модели данных, индикаторы событий из списка чувствительности и значение информационного входа `z`.

```
// спецификационная функция процесса
specification bool Process(Module *module,
  bool posedge_x, bool negedge_y, bool z)

  // процесс читает какие события произошли,
  // а также значение информационного входа z
  reads   posedge_x, negedge_y, z
  // тестовое воздействие изменяет значения
  // сигналов на управляющих входах
```

```
  updates x = module->x, y = module->y
{
  pre
  {
    // проверка возможности реализации событий
    return ((posedge x || negedge y) &&
      (posedge x => !x) && (negedge y => y));
  }

  post
  {
    // проверка реализации событий
    return ((posedge_x => x) && (negedge_y => !y))
&&
      Process == z;
  }
}
```

Заметим, что данный подход к выделению спецификационных функций является достаточно общим, а потому не всегда оптимальным. При его использовании можно автоматически генерировать шаблон спецификации по исходному коду модели. В приведенном примере проверки, допускающие автоматическую генерацию, подчеркнуты.

После того как определены спецификационные функции, тестовый сценарий разрабатывается обычным для инструмента CTesK образом: с помощью функции вычисления состояния и сценарных функций описывается конечный автомат теста, используемый для построения последовательности тестовых воздействий. Отметим следующие моменты. Для того, чтобы выполнимость предусловий спецификационных функций определялась только состоянием теста, без использования более детальной информации об истории тестовых воздействий, состояния теста должны включать текущие значения сигналов на управляющих входах. Для того, чтобы граф состояний конечного автомата был сильно-связен, необходимо чтобы для каждого события, используемого в тестовых воздействиях, существовало тестовое воздействие с обратным событием[6], так как в противном случае образуются тупиковые состояния теста.

В общем случае, поведение моделей аппаратного обеспечения определяется не только последовательностью тестовых воздействий, но и длительностью временны*х* интервалов между ними. Это характерно, например, для моделей устройств с таймерами. Для качественного тестирования таких моделей нужно уметь подбирать и комбинировать длительности временны*х* интервалов между тестовыми воздействиями.

[6] Если в тесте есть тестовое воздействие, включающее фронт (срез) некоторого сигнала, то в нем должно присутствовать тестовое воздействие, включающее срез (фронт) этого сигнала.

Поскольку в технологии UniTesK для построения последовательностей тестовых воздействий используются конечно-автоматные модели, естественно моделировать изменение времени с помощью специальных переходов. Это могут быть как просто переходы по времени, в которых изменяется только модельное время, а значения входных сигналов остаются неизменными, так и совмещенные переходы, в которых изменяются и модельное время, и значения входных сигналов. С точки зрения спецификации, изменение модельного времени является параметром соответствующей переходу спецификационной функции. Интересные для тестирования изменения модельного времени перебираются в тестовом сценарии.

Теперь несколько слов об адаптации тестовой системы к выполнению в симуляторе. Чтобы тестовая система могла быть выполнена в симуляторе, она должна быть оформлена как отдельный модельный процесс. При этом возможны две различные ситуации: когда тестовая система может напрямую управлять выполнением этого модельного процесса (см. раздел «Тестирование SystemC-моделей») и когда не может (см. раздел «Тестирование Verilog-моделей»).

В первом случае, для адаптации тестовой системы практически не требуется разработки дополнительных модулей. Главное, чтобы после подачи очередного тестового воздействия, модельный процесс, в котором выполняется тестовая система, приостанавливался, чтобы симулятор мог активизировать процесс тес-тируемой модели, занимающийся обработкой поданного воздействия. Приос-тановка процесса тестовой системы реализуется в медиаторных функциях.

Во втором случае, требуется разработать специальный модуль симулятора, называемый окружением тестируемой модели, который взаимодействует с тестовой системой, запущенной в виде отдельного потока. К функциям этого модуля относятся прием тестовых воздействий от тестовой системы, их подача на тестируемую модель, а также передача реакций тестируемой модели тестовой системе.

3.2. Пример счетчика

Рассмотрим пример небольшого устройства — счетчика, на Verilog- и SystemC-моделях которого мы будем иллюстрировать основные шаги разработки тестов. Интерфейс счетчика состоит из двух двоичных входов `inc` и `rst` и одного целочисленного выходного регистра `cnt`.

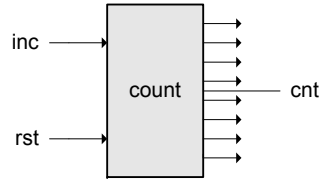

Рис. 4. Схема входов и выходов счетчика.

В ответ на фронт сигнала inc, устройство увеличивает содержимое регистра cnt на единицу, на фронт rst — обнуляет регистр cnt. Срезы сигналов не обрабатываются. Для упрощения спецификации и тестирования будем считать, что поведение устройства при одновременном возникновении фронтов сигналов на обоих входах не определено.

Рассмотрим спецификацию счетчика на языке SeC. Спецификационная модель данных включает в себя текущие значения входных сигналов inc и rst, а также выходного регистра cnt:

```
// спецификационная модель данных счетчика
typedef struct
{
  bool inc; // текущее значение сигнала
            // на входе inc
  bool rst; // текущее значение сигнала
            // на входе rst
  int  cnt; // текущее значение регистра cnt

} Model;
```

Спецификационные функции описывают поведение устройства в ответ на фронты и срезы сигналов inc и rst. Ниже приводится спецификационная функция inc_posedge_spec, описывающая реакцию на фронт сигнала на входе inc. Остальные функции определяются аналогичным образом.

```
// спецификация реакции на фронт сигнала на входе
inc
specification void inc_posedge_spec(Model *model)
  updates cnt = model->cnt,
          inc = model->inc
{
  pre { return model != NULL && inc == false
    && cnt < INT_MAX; }

  coverage C { return {single, "Single branch"}; }

  post { return inc == true && cnt == @cnt + 1; }
}
```

Рассмотрим сценарий тестирования счетчика. В качестве состояния конечного автомата для нашего примера будем просто использовать состояние спецификационной модели, то есть текущие значения входных сигналов inc и rst, а также выходного регистра cnt. В этом случае функция вычисления состояния конечного автомата будет выглядеть как показано ниже.

```
static List* scenario_state()
{
  List* list = create_List(&type_Integer);
```



```
  append_List(list, create_Integer(model.inc));
  append_List(list, create_Integer(model.rst));
  append_List(list, create_Integer(model.cnt));

  return list;
}
```

Чтобы ограничить число состояний конечного автомата, запретим подачу фронта inc в состояниях, в которых значение регистра cnt больше или равно десяти. Остальные стимулы (фронт rst, срезы inc и rst) сделаем допустимыми во всех достижимых состояниях.

В начальном состоянии теста устанавливаем низкие уровни сигналов inc и rst, а значению регистра cnt присваиваем ноль.

Ниже приводится сценарная функция для фронта inc. Остальные функции определяются аналогичным образом.

```
scenario bool inc_posedge_scen()
{
  if(model.cnt < 10)
  {
    if(pre_inc_posedge_spec(&model))
      inc_posedge_spec(&model);
  }

  return true;
}
```

4. Тестирование Verilog-моделей

В этом разделе предлагается способ расширения базовой архитектуры тестовой системы UniTesK для функционального тестирования Verilog-моделей аппаратного обеспечения. Описывается процесс разработки теста с помощью инструмента CTesK и приводятся оценки возможности автоматизации шагов этого процесса.

Помимо инструмента CTesK, нами использовались свободно распространяемый симулятор Icarus Verilog [15] и компилятор GCC из набора инструментов MinGW [16].

4.1. Архитектура тестовой системы

При расширении базовой архитектуры тестовой системы CTesK мы учитывали то обстоятельство, что используемый симулятор Icarus Verilog не позволяет напрямую управлять симуляцией через интерфейс VPI[7] [17], используемый

[7] VPI (Verilog Procedural Interface) или PLI (Programming Language Interface) 2.0 — стандартный интерфейс, предназначенный для вызова из Verilog-модулей функций, написанных на языке программирования C и других языках программирования.

нами для интеграции тестовой системой с симулятором. Предлагаемая архитектура тестовой системы показана на Рис. 5.

Тестовая система состоит из двух потоков:

- потока симулятора Verilog — основного потока;
- потока тестовой системы CTesK — подчиненного потока.

В начале симуляции при помощи вызова специальной функции из симулятора Verilog в отдельном потоке запускается тестовая система CTesK, которая в цикле предоставляет тестовые воздействия на тестируемую Verilog-модель, принимает реакции на них и оценивает правильность этих реакций.

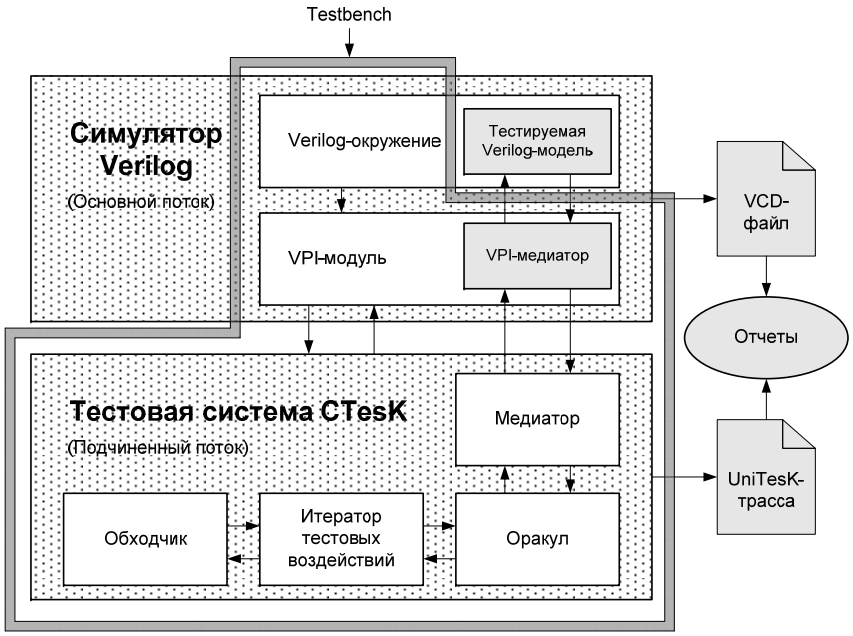

Рис. 5. Предлагаемая архитектура тестовой системы.

Verilog-окружение содержит экземпляр тестируемой Verilog-модели. Оно осуществляет инициализацию экземпляра тестируемой модели, запускает через VPI-модуль в отдельном потоке тестовую систему CTesK, от которой в цикле принимает тестовые воздействия и которой передает реакции тестируемой модели на них. Прием тестовых воздействий и передача реакций осуществляются через VPI-медиатор.

VPI-модуль связывает Verilog-окружение с тестовой системой CTesK. Он реализует функцию запуска тестовой системы CTesK и содержит в качестве составной части VPI-медиатор.

VPI-медиатор связывает экземпляр тестируемой Verilog-модели с медиатором тестовой системы CTesK. Он реализует прием тестовых воздействий от тестовой системы CTesK и посылку реакций тестируемой модели на них, а также осуществляет синхронизацию состояний экземпляра тестируемой модели и спецификационной модели данных тестовой системы CTesK.

4.1.1. Взаимодействие компонентов

Ниже показана последовательность взаимодействия основных компонентов тестовой системы. Чтобы не вдаваться в детали внутренних взаимодействий, такие компоненты, как обходчик и итератор тестовых воздействий, объединены на диаграмме в один компонент — генератор.

При вызове `$startScenario` из Verilog-окружения VPI-модуль запускает в отдельном потоке тестовую систему CTesK (генератор, оракул, медиатор). Далее в цикле осуществляются вызовы `$applyAction`, для приема очередного тестового воздействия, и `$checkAction`, для передачи реакции на него.

Генератор через оракул передает медиатору очередное тестовое воздействие `Action`, который преобразует его в посылку сообщения `ApplyAction` VPI-модулю и переходит в состояние ожидания реакции на него `WaitForCheck`.

VPI-модуль при вызове `$applyAction` переходит в состояние ожидания очередного тестового воздействия `WaitForAction` и выходит из него при получении сообщения `ApplyAction` от медиатора тестовой системы CTesK. После этого он вызовом `SetSignals` изменяет нужным образом входные сигналы экземпляра тестируемой Verilog-модели и передает управление Verilog-окружению, возвращая статус `OK`.

При вызове `$checkAction` VPI-модуль, используя `GetSignals`, получает значения выходных сигналов и синхронизирует состояния экземпляра тестируемой Verilog-модели и спецификации тестовой системы CTesK. После этого он посылает сообщение `ApplyCheck` медиатору.

При приеме сообщения `ApplyCheck` медиатор выходит из состояния `WaitForCheck` и передает управление оракулу, который проверяет правильность реакции экземпляра тестируемой Verilog-модели на тестовое воздействие.

Цикл завершается при получении VPI-модулем в состоянии ожидания тестового воздействия `WaitForAction` сообщения `Finish` о завершении теста от тестовой системы CTesK.

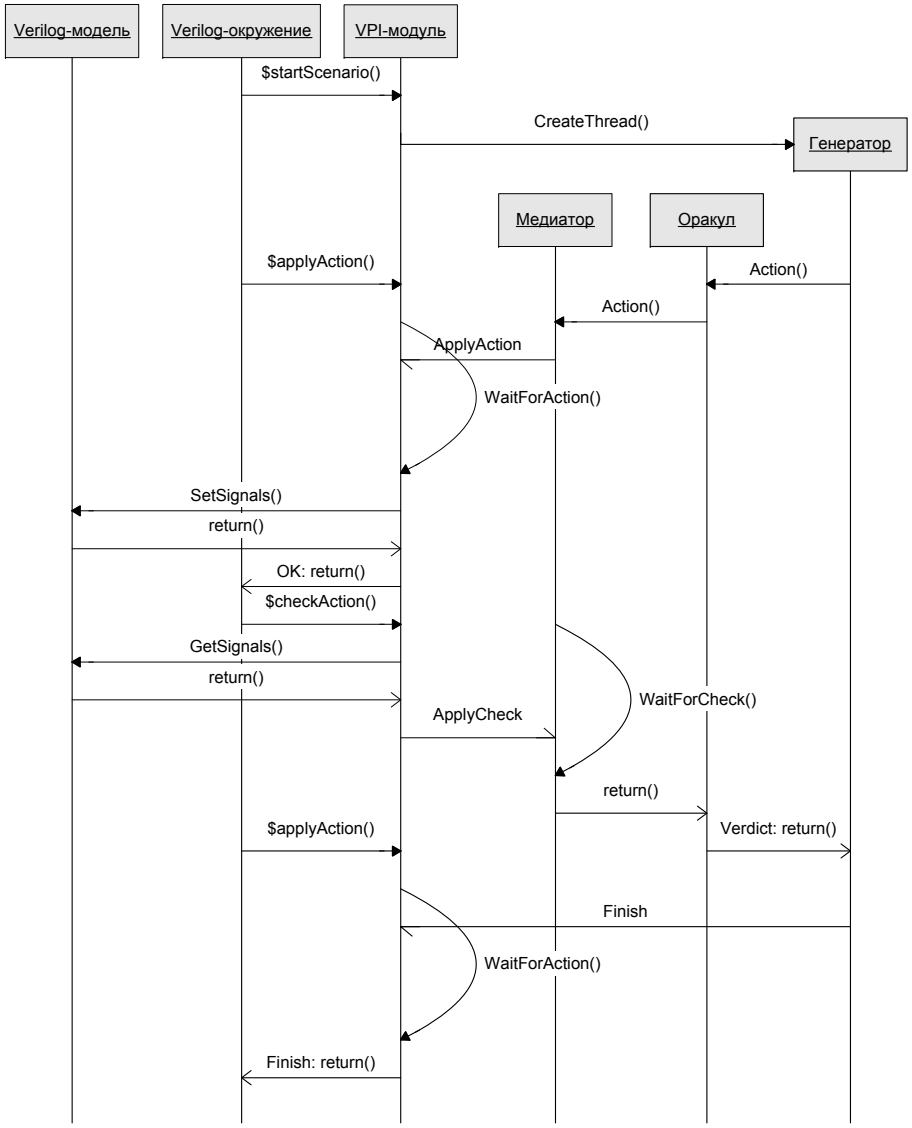

Рис. 6. Последовательность взаимодействия основных компонентов тестовой системы.

4.2. Разработка теста

В этом разделе подробно рассматривается процесс разработки теста для Verilog-моделей аппаратного обеспечения с помощью инструмента CTesK. Для

иллюстрации процесса будем использовать пример счетчика (см. раздел «Пример счетчика»), Verilog-модель которого приводится ниже.

```
module count(inc, rst);

  // входы inc и rst
  input inc, rst;

  // выходной регистр cnt
  integer cnt;

  // увеличивает счетчик
  task increment;
    begin cnt = cnt + 1; end
  endtask

  // сбрасывает счетчик
  task reset;
    begin cnt = 0; end
  endtask

  // в начальном состоянии счетчик сброшен
  initial
    begin reset; end

  // обработчик фронта сигнала на входе inc
  always @(posedge inc)
    begin increment; end

  // обработчик фронта сигнала на входе rst
  always @(posedge rst)
    begin reset; end

endmodule
```

Процесс разработки теста состоит из следующих шагов:

1. Разработка спецификации устройства.
2. Разработка модуля взаимодействия потоков.
3. Разработка медиатора.
4. Разработка тестового сценария.
5. Разработка Verilog-окружения.
6. Разработка VPI-модуля.

Рассмотрим подробно каждый из этих шагов за исключением шага разработки спецификации и шага разработки тестового сценария, которые были описаны выше (см. раздел «Пример счетчика»).

4.2.1. Разработка модуля взаимодействия потоков

Модуль взаимодействия потоков реализует функции синхронизации потока симулятора Verilog и потока тестовой системы CTesK. Эти функции используются медиатором и VPI-модулем, поэтому модуль взаимодействия потоков рекомендуется разрабатывать перед разработкой медиатора или VPI-модуля.

Модуль взаимодействия потоков должен реализовывать следующие функции:

- `wait_for_check` — функция ожидания реакции на тестовое воздействие. Вызывается в медиаторе;
- `wait_for_apply` — функция ожидания тестового воздействия. Вызывается в VPI-медиаторе, возвращает идентификатор тестового воздействия;
- `apply_check` — функция передачи реакции на тестовое воздействие. Вызывается в VPI-медиаторе;
- `apply_finish` — функция посылки сообщения о завершения теста. Вызывается тестовой системой CTesK;
- `apply_<воздействие>` — функции посылки тестовых воздействий. Вызываются в медиаторе.

Также в модуле взаимодействия потоков можно определить функции:

- `start_scenario` — функция запуска тестовой системы CTesK. Вызывается в VPI-медиаторе, создает необходимые для взаимодействия потоков ресурсы, устанавливает имя UniTesK-трассы, запускает в отдельном потоке тестовую систему CTesK;
- `end_scenario` — функция завершения работы тестовой системы CTesK. Вызывается в VPI-медиаторе при получении сообщения о завершении теста, освобождает созданные ресурсы.

Так как модуль использует средства создания и взаимодействия потоков, он является платформо-зависимым. Мы вели разработку на платформе Windows 2000 и использовали механизм событий Win32. Очевидно, что разработку модуля взаимодействия потоков для каждой конкретной платформы можно автоматизировать.

4.2.2. Разработка медиатора

В инструменте CTesK медиатор реализуется с помощью медиаторных функций, каждая из которых связывает спецификационную функцию с группой тестовых воздействий на целевую систему. Код медиаторной функции состоит из блока воздействия (блока **`call`**), в котором осуществляется тестовое воздействие, и блока синхронизации (блока **`state`**), в котором

осуществляется синхронизация состояния спецификационной модели данных с состоянием целевой системы.

Разработку медиатора для Verilog-модели можно осуществить автоматически по следующей схеме:

- для каждой спецификационной функции пишется медиаторная функция следующего вида:
  - блок **`call`** ≡ `{apply_<воздействие>(<параметры>);}`;
  - блок **`state`** ≡ `{wait_for_check();}`.

Медиаторная функция для спецификационной функции `inc_posedge_spec` будет выглядеть следующим образом:

```
// медиаторная функция для inc_posedge_spec
mediator inc_posedge_media for
  specification     void     inc_posedge_spec(Model
*model)
  updates cnt = model->cnt,
          inc = model->inc
{
  // посылаем тестовое воздействие
  call  { apply_inc_posedge(model); }

  // ожидаем реакции
  state { wait_for_check(); }
}
```

4.2.3. Разработка Verilog-окружения

Verilog-окружение представляет собой модуль верхнего уровня на языке Verilog HDL и разрабатывается по следующей схеме:

- для каждого входа тестируемой Verilog-модели внутри Verilog-окружения определяется однотипный регистр;
- определяется экземпляр тестируемой Verilog-модели `target`, в качестве аргументов которого выступают определенные ранее регистрами;
- определяется блок **`initial`**, внутри которого:
  - устанавливается имя VCD-файла и трассируемые в него сигналы;
  - вызывается системная задача `$startScenario`, запускающая тестовую систему CTesK;
  - в цикле с помощью системной функции `$applyAction` принимаются тестовые воздействия от тестовой системы CTesK и с помощью системной задачи `$checkAction` посылаются реакции на них.

Ниже приводится Verilog-окружение для Verilog-модели счетчика:

```
module testbench();

  parameter delay = 10;
```

```
    // входы экземпляра тестируемой Verilog-модели
    reg inc, rst;

    // экземпляр тестируемой Verilog-модели
    count target(inc, rst);

    initial
      begin
        // устанавливаем имя VCD-файла и
        // трассируемые сигналы
        $dumpfile("simulation.vcd");
        $dumpvars(1, testbench);

        // запускаем тестовую систему CTesK
        $startScenario();
        #(delay);

        // в цикле получаем тестовые воздействия и
        // передаем реакции на них
        while($applyAction() == 0)
          begin
            #(delay);
            $checkAction();
            #(delay);
          end
      end

  endmodule
```

Verilog-окружение использует системные задачи `$startScenario`, `$checkAction` и системную функцию `$applyAction`, которые должны быть реализованы в VPI-модуле. Видно, что разработку Verilog-окружения можно полностью автоматизировать.

4.2.4. Разработка VPI-модуля

VPI-модуль разрабатывается на языке программирования C с использованием интерфейса VPI. Он должен реализовывать следующие системные функции и задачи:

- `$startScenario` — вызывает функцию `start_scenario` модуля взаимодействия потоков;
- `$applyAction` — вызывает функцию `wait_for_action` модуля взаимодействия потоков и в зависимости от возвращаемого значения производит необходимые изменения входных сигналов тестируемой Verilog-модели;

- `$checkAction` — осуществляет синхронизацию состояний экземпляра Verilog-модели и спецификации тестовой системы CTesK, после этого вызывает функцию `apply_check` модуля взаимодействия потоков;

Системная функция `$applyAction` и системная задача `$checkAction` образуют VPI-медиатор.

Видно, что разработку части VPI-модуля, связанную с запуском тестовой системы CTesK, можно полностью автоматизировать.

4.3. Возможность автоматизации шагов разработки

Ниже приводится сводная таблица разрабатываемых модулей, в которой указаны поток выполнения модуля, язык разработки и возможность автоматизации разработки.

НАЗВАНИЕ МОДУЛЯ	ПОТОК ВЫПОЛНЕНИЯ	ЯЗЫК РАЗРАБОТКИ	ВОЗМОЖНОСТЬ АВТОМАТИЗАЦИИ
Спецификация устройства	CTesK	SeC	Нет
Модуль взаимо-действия потоков	CTesK / Verilog	SeC (POSIX, Win32)	Да
Медиатор	CTesK	SeC	Да
Тестовый сценарий	CTesK	SeC	Нет
Verilog-окружение	Verilog	Verilog	Да
VPI-модуль	Verilog	C (VPI)	Частично

В таблице показано, что разработку модуля взаимодействия потоков, медиатора и Verilog-окружения можно автоматизировать полностью, разработку VPI-модуля — частично.

5. Тестирование SystemC-моделей

В этом разделе предлагается способ расширения базовой архитектуры тестовой системы UniTesK для функционального тестирования SystemC-моделей аппаратного обеспечения. Описывается процесс разработки теста с помощью инструмента CTesK и приводятся оценки возможности автоматизации шагов этого процесса.

Помимо инструмента CTesK и свободно распространяемой библиотеки SystemC, нами использовалась среда разработки Microsoft Visual Studio 6.0.

5.1. Архитектура тестовой системы

Тестовая система CTesK выполняется в симуляторе SystemC, она подает на тестируемую SystemC-модель тестовые воздействия, принимает реакции на них и оценивает правильность этих реакций. Взаимодействие тестовой системы CTesK и тестируемой SystemC-модели осуществляется через C-медиатор.

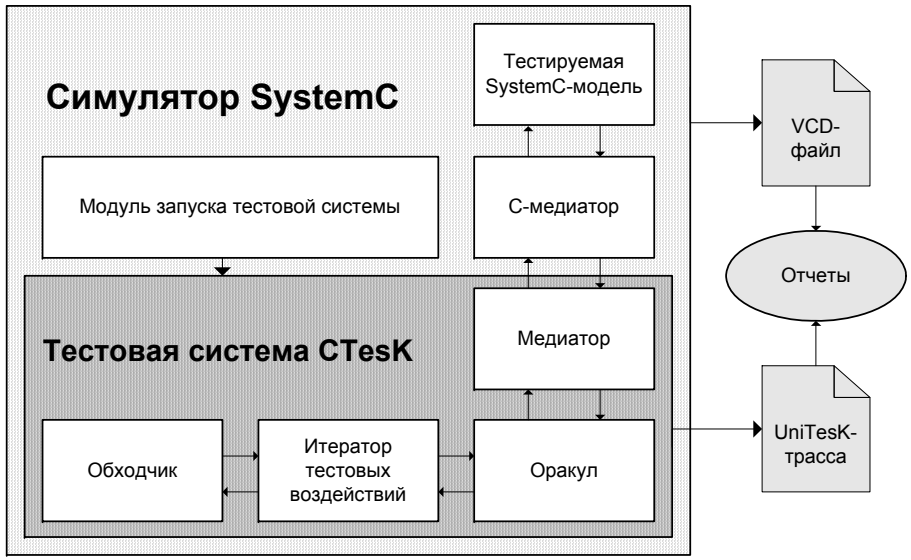

Рис. 7. Предлагаемая архитектура тестовой системы.

C-медиатор представляет собой реализацию программного интерфейса на языке программирования C, предназначенного для доступа к экземпляру тестируемой SystemC-модели. C-медиатор используется медиатором тестовой системы CTesK для подачи тестовых воздействий и приема реакций.

Модуль запуска тестовой системы предназначен для запуска тестовой системы CTesK в симуляторе SystemC.

5.2. Разработка теста

В этом разделе подробно рассматривается процесс разработки теста для SystemC-моделей аппаратного обеспечения с помощью инструмента CTesK. Для иллюстрации процесса будем использовать пример счетчика (см. раздел «Пример счетчика»), SystemC-модель которого приводится ниже.

```
SC_MODULE(count)
{
  // входы сигналы inc и rst
  sc_in<bool> inc;
  sc_in<bool> rst;

  // выходной регистр cnt
  int cnt;

  // обработчик изменения значения сигнала inc
  void increment(void) { if(inc.posedge()) cnt++;
}

  // обработчик изменения значения сигнала inc
  void reset(void) { if(rst.posedge()) cnt = 0; }

  SC_CTOR(count): cnt(0), inc(false), rst(false)
  {
    SC_METHOD(increment);
    sensitive(inc);

    SC_METHOD(reset);
    sensitive(rst);
  }
};
```

Процесс разработки теста состоит из следующих шагов:

1. Разработка C-медиатора.
2. Разработка спецификации системы.
3. Разработка медиатора.
4. Разработка тестового сценария.
5. Разработка модуля запуска тестовой системы.

Рассмотрим подробно каждый из этих шагов за исключением шага разработки спецификации и шага разработки тестового сценария, которые были описаны выше (см. раздел «Пример счетчика»).

5.2.1. Разработка C-медиатора

Поскольку инструмент CTesK предназначен для разработки тестов для программных интерфейсов на языке программирования C, то из компонентов тестовой системы CTesK нельзя напрямую обращаться к SystemC-модели. Все обращения к тестируемой модели должны осуществляться через специально разработанный C-медиатор, который предоставляет интерфейс для подачи тестовых воздействий и получения реакций тестируемой модели на них.

Ниже приводится интерфейс C-медиатора для SystemC-модели счетчика.

```
#ifdef __cplusplus
extern "C" {
#endif // #ifdef __cplusplus

  // интерфейсные функции, осуществляющие
  // тестовые воздействия
```

```
  // на экземпляр SystemC-модели счетчика
  void count_inc_posedge(void);
  void count_rst_posedge(void);
  void count_inc_negedge(void);
  void count_rst_negedge(void);

  // интерфейсные функции, получающие информацию
  //   о   состоянии   экземпляра   SystemC-модели
счетчика
  int count_inc(void);
  int count_rst(void);
  int count_cnt(void);

#ifdef __cplusplus
}
#endif // #ifdef __cplusplus
```

Заметим, что в функциях, реализующих тестовые воздействия, должна осуществляться приостановка модельного процесса тестовой системы для того, чтобы симулятор SystemC мог активизировать процесс тестируемой модели, занимающийся обработкой поданного воздействия.

5.2.2. Разработка медиатора

После того, как создан C-медиатор, разработка медиатора осуществляется по следующей схеме:

- разрабатывается функция синхронизации состояний `map_state_up`, осуществляющая синхронизацию состояния спецификационной модели данных тестовой системы CTesK с состоянием экземпляра тестируемой SystemC-модели;
- для каждой спецификационной функции пишется медиаторная функция следующего вида:
  - в блоке **`call`** осуществляется вызов соответствующей интерфейсной функции C-медиатора;
  - в блоке **`state`** осуществляется вызов функции `map_state_up`.

Ниже приводится медиаторная функция для спецификационной функции `inc_posedge_spec`.

```
// медиаторная функция для inc_posedge_spec
mediator inc_posedge_media for
  specification     void     inc_posedge_spec(Model
*model)
  updates cnt = model->cnt,
          inc = model->inc
{
```



```
  // вызываем соответствующую функцию C-медиатора
  call  { count_inc_posedge(); }

  // вызываем функцию синхронизации состояний
  state { map_state_up(model); }
}
```

Видно, что при совпадении интерфейсов C-медиатора и спецификации разработку медиатора можно автоматизировать.

5.2.3. Разработка модуля запуска тестовой системы

Разработка модуля запуска тестовой системы осуществляется по следующей схеме:

- разрабатывается функцию запуска тестовой системы CTesK;
- разрабатывается SystemC-модуль для вызова тестовой системы CTesK в отдельном модельном процессе.

Ниже приводится функция `count_start`, запускающая тестовый сценарий `count_scenario`.

```
// функция запуска тестового сценария
void count_start(const char *trace)
{
  addTraceToFile(trace);
  count_scenario(0, NULL);
}
```

Ниже приводится SystemC-модуль для вызова тестовой системы CTesK в отдельном модельном процессе.

```
// модуль запуска тестовой системы
SC_MODULE(count_testbench)
{
public:
  // определяем отдельный модельный процесс
  SC_CTOR(count_testbench) { SC_THREAD(main); }

  // метод запуска теста
  void start(void) { sc_start(); }

  // процесс тестовой системы CTesK
  void main(void)
  { count_start("simulation.unitrace"); }
};
```

Видно, что разработку модуля запуска тестовой системы можно полностью автоматизировать.

5.3. Возможность автоматизации шагов разработки

Ниже приводится сводная таблица разрабатываемых модулей, в которой указаны язык разработки модуля и возможность автоматизации его разработки.

НАЗВАНИЕ МОДУЛЯ	ЯЗЫК РАЗРАБОТКИ	ВОЗМОЖНОСТЬ АВТОМАТИЗАЦИИ
С-медиатор	C / SystemC	Нет
Спецификация системы	SeC	Нет
Медиатор	SeC	Да
Тестовый сценарий	SeC	Нет
Модуль запуска тестовой системы	SeC / SystemC	Да

В таблице показано, что разработку медиатора и модуля запуска тестовой системы можно автоматизировать полностью.

6. Заключение

В работе была рассмотрена новая и, на наш взгляд, достаточно перспективная область применения технологии UniTesK — функциональное тестирование моделей аппаратного обеспечения. Главным образом, в статье описывались подходы к организации взаимодействия между тестовой системой CTesK и симуляторами моделей на языках Verilog HDL и SystemC. Для каждого из указанных классов моделей был предложен способ расширения базовой архитектуры тестовой системы, показано, что разработку некоторых дополнительных компонентов удается полностью или частично автоматизировать.

На настоящий момент опыт применения технологии UniTesK и инструмента CTesK для тестирования моделей аппаратного обеспечения ограничивается небольшими примерами. Многие сложные вопросы такие, как декомпозиция спецификаций для упрощения описания сложных систем, тестирование систем с таймерами, а также тестирование систем со смешанными аппаратными и программными частями остались в работе нерассмотренными. То же относится к вопросам о возможности более тесной интеграция технологии UniTesK и инструмента CTesK с конкретными симуляторами и библиотеками. Данные вопросы являются темами будущих исследований.

Литература

1. http://www.unitesk.com — сайт, посвященный технологии тестирования UniTesK и реализующим ее инструментам.
2. 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/.
3. В.В. Кулямин, А.К. Петренко, А.С. Косачев, И.Б. Бурдонов. Подход UniTesK к разработке тестов. Программирование, 29(6): 25–43, 2003.
4. http://www.ispras.ru — сайт Института системного программирования РАН;
5. А.К. Поляков. Языки VHDL и VERILOG в проектировании цифровой аппаратуры. — М.: СОЛОН-Пресс, 2003.
6. http://www.systemc.org — сайт, посвященный языку SystemC.
7. http://www.systemverilog.org — сайт, посвященный языку SystemVerilog.
8. Bergeron, Janick. Writing testbenches: functional verification of HDL models. Kluwer Academic Publishers, 2000.
9. http://www.unitesk.com/products/ctesk/ — страница инструмента CTesK.
10. В. Немудров, Г. Мартин. Системы-на-кристалле. Проектирование и развитие. Москва: Техносфера, 2004.
11. I. Bourdonov, A. Kossatchev, A. Petrenko, and D. Galter. KVEST: Automated Generation of Test Suites from Formal Specifications. FM'99: Formal Methods. LNCS 1708, Springer-Verlag, 1999, pp. 608-621.
12. The RAISE Language Group. The RAISE Specification Language. Prentice-Hall BCS Practitioner Series. Prentice-Hall, Inc., 1993.
13. http://www.unitesk.com/products/jat/ — страница инструмента J@T.
14. http://www.unitesk.com/products/chase/ — страница инструмента Ch@se.
15. http://www.icarus.com/eda/verilog/ — страница симулятора Icarus Verilog.
16. http://www.mingw.org — сайт, посвященный набору инструментов MinGW.
17. Sutherland, Stuart. The Verilog PLI handbook: A User's Guide and Comprehensive Reference on the Verilog Programming Language Interface. Springer, 2002.