ПРОГРАММИРОВАНИЕ, *2012*, *№ 1*, *с. 47–58*

ПРОГРАММНАЯ ИНЖЕНЕРИЯ

УДК 681.3.06

РАЗРАБОКА ТЕСТОВЫХ СИСТЕМ ДЛЯ МНОГОМОДУЛЬНЫХ МОДЕЛЕЙ АППАРАТУРЫ

© 2012 г. М.М. Чупилко
Институт Системного Программирования РАН
109004, Москва, ул. А. Солженицына, 25
E-mail: chupilko@ispras.ru
Поступила в редакцию 03.10.2011 г.

В данной статье предлагается подход к созданию тестовых систем для сложных моделей аппаратуры. Такие модели могут быть разделены на модули и верифицированы по отдельности. Предлагаемая архитектура построения одиночных тестовых систем и способ их объединения в полную тестовую систему основаны на имитационном моделировании. Метод соединения предполагает разработку компонентов тестовой системы с интерфейсами по своей идее близкими к TLM, то есть с использованием высокоуровневой модели коммуникации, основанной на сообщениях и, следовательно, упрощающей объединение нескольких тестовых систем в единую тестовую систему для всего тестируемого компонента.

1. ВВЕДЕНИЕ

Верификация аппаратуры остается очень актуальной в течение многих лет. Существует несколько методик проведения верификации, но до сих пор не создано единого решения. *Модели аппаратуры* создаются с помощью языков, *описывающих поведение аппаратуры (hardware description languages, HDLs,* например, Verilog [1]). Даже относительно простые модули (то есть части сложных моделей или очень простые модели) едва ли могут быть проверены с помощью ручного анализа исходного кода, не говоря уже о сложных моделях. Следовательно, автоматизация верификации, или проверка взаимного соответствия поведения модели и спецификаций этого поведения, является важной и требующей внимания. Существующие оценки говорят, что около 70% всего времени проектирования тратится на верификацию [2]. Реальная практика показывает, что обычно, по меньшей мере, половина от времени проектирования действительно тратится на достижение этой цели. Код, написанный на HDLs, называется *HDL-моделью* и может быть автоматически транслирован в *список соединений (net-list)* и затем на его основе создается реальное устройство. Если список соединений не будет изменен вручную, функциональность создаваемого устройства будет полностью повторять HDL-модель. Даже в случае некоторых изменений эквивалентность поведения может быть проверена с помощью специальных инструментов, например [3]. Следовательно, найти и исправить ошибки возможно уже на стадии создания HDL-модели. Заметим, что исправление функциональных ошибок на более поздних стадиях проектирования и, в частности, после производства чипов, требует больше усилий и времени из-за необходимости повторять все предыдущие стадии производства.

Сложные модели обычно разрабатываются с помощью *абстракции* и *декомпозиции.* Общим подходом является разработка сначала абстрактной системы в целом, а затем тщательная разработка ее компонентов – модулей. Обычно создаются тесты для всей системы (будем называть ее *DUV* или *реализацией),* но при этом они являются достаточно абстрактными. Поскольку некоторые части системы могут являться

критичными для ее поведения в целом, для этих частей создаются отдельные тесты. Если эти „маленькие“ тестовые системы смогли бы помочь улучшить тестовую систему для всего DUV, это привело бы к повышению уровня повторного использования разработанного кода и увеличению возможностей по нахождению ошибок общей тестовой системы, что являлось бы большим достижением.

Данная работа организована следующим образом. Во-первых, приводится обзор работ, связанных с верификацией моделей аппаратуры. Затем рассматривается архитектура тестовых систем, изначально предложенная в [4]. Далее вводится архитектура многомодульных тестовых систем. В разделе 5 рассматривается опыт применения подхода. Раздел 6 завершает статью.

2. ОБЗОР РАБОТ ПО ТЕМЕ ВЕРИФИКАЦИИ

Существует два основных способа верификации моделей аппаратуры. Во-первых, могут быть использованы *формальные методы,* чтобы, например, доказать выполнимость логических формул в формальной модели, построенной на основе модели аппаратуры в *проверке моделей (model checking)* [5]. В данном случае исследуется модель аппаратуры в статическом состоянии. Такие методы хорошо применимы в случае модульной верификации, но гибкости их применения недостаточно для случая действительно сложных DUV [6]. Для улучшения масштабируемости используется подход имитационного моделирования, приближенного к реальной работе DUV, с использованием *HDL-симулятора.* В таком случае может производиться полноценное функциональное тестирование работы DUV. Обычно подходы к верификации с использованием имитационного моделирования обладают высокой масштабируемостью, а тщательность проверок меняется в зависимости от доступных ресурсов и ограничений времени. Ниже под словом верификация мы будем иметь в виду только верификацию на основе имитационного моделирования.

Основными компонентами тестовых систем являются генератор тестовой последовательности (генератор стимулов), компонент проверки реакций (или тестовый оракул), компонент оценки полноты тестирования. Генератор тестовой последовательности может быть сделан вручную на основе явного указания списка тестовых воздействий. Также генераторы стимулов могут создавать воздействия полуавтоматически на основе ручного описания переменных с установленными ограничениями на их значения. В этом случае для генерации стимулов используется специальный механизм, который выбирает подмножество множества всех доступных стимулов, разрешает ограничения, наложенные на значения их переменных, и запускает стимулы. Этот подход называется *верификацией на основе ограничений (constraint-driven verification, CDV)* [7]. Другой хорошо известный подход к генерации стимулов – это генерация на *основе обхода конечных автоматов* [8], когда в качестве состояний конечных автоматов берутся состояния модели тестируемой системы, а переходы между ними задаются подаваемыми операциями. Тестовый оракул для осуществления своей деятельности должен знать корректное поведение DUV, например, с помощью *эталонной модели.* Компонент оценки полноты покрытия обычно работает на основе *покрытия кода реализации* или *кода функциональной модели.*

Основная тема данной статьи – возможность повторного использования разрабатываемых тестовых систем в многомодульном случае. Чтобы избежать дополнительных потенциальных проблем, мы предлагаем использование унифицированной архитектуры для всех таких тестовых систем. При объединении тестовых систем возникает вопрос об объединении каждого их компонента. Такая возможность должна предоставляться относительно простыми методами, изначально заложенными в архитектуру.

Среди наиболее широко распространенных подходов к верификации мы выбрали *Открытую методологию верификации (Open Verification Methodology, OVM* [9]) как наиболее яркого представителя, кажущегося наиболее подходящим для рассмотрения возможностей объединения тестовых систем. OVM задает

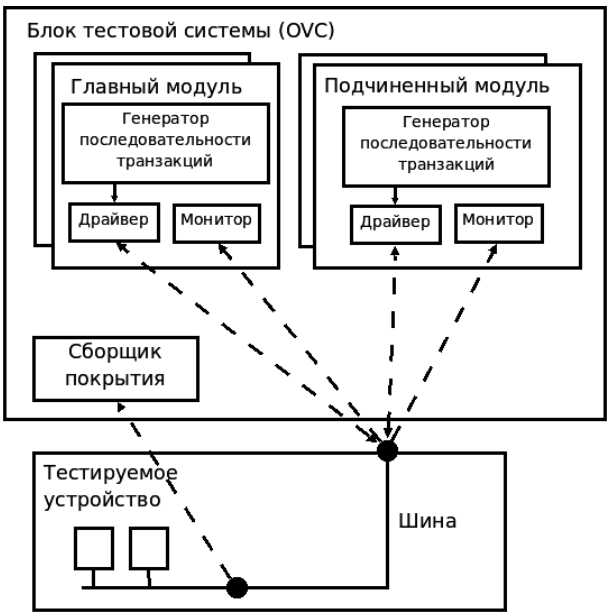

Рис. 1. OVC-блок тестовой системы.

архитектуру и разделение компонентов тестовых систем по нескольким слоям [10]. Тестовая система для каждого отдельного компонента представляет собой отдельный блок *(Open Verification Component, OVC)* (см. рис. 1). Каждый блок OVC содержит основные средства создания потока стимулов на основе механизма разрешения ограничений и средства передачи этого потока в DUV. Задача создания стимулов решается с помощью *генераторов последовательности транзакций,* где *транзакции – это абстрактные сообщения,* содержащие информацию о тестовой ситуации. Задача доставки решается так называемыми *транзакторами,* то есть компонентами, осуществляющими прямое и обратное преобразование данных между транзакциями и сигнальными линиями DUV. Компоненты OVC могут быть соединены друг с другом под управлением *объединенного тестового контроллера* или, иначе, *виртуального генератора* [7]. В этом случае каждый из OVC-блоков создает свой собственный поток стимулов, а разработчик тестовой системы может выключать избыточные генераторы, соединенные с более недоступными сигнальными линиями DUV. Компоненты выключенных OVC-блоков проверяют корректность реакций в соответствии с общим потоком стимулов, опосредованно влияющим на работу проверяемых модулей DUV. Обобщая вышесказанное, можно утверждать, что тестовые системы, сделанные в соответствии с OVM, удовлетворяют многим задачам, особенно возникающим при использовании для верификации соединения нескольких тестовых систем. Следует заметить, что методология OVM ориентирована на язык *SystemVerilog:* соединение таких тестовых систем с компонентами, написанными на других языках возможно, но связано с разработкой промежуточных компонентов.

Мы разработали новый подход и представили некоторые его аспекты в [4]. В той статье в основном затрагивались проблемы построения оракулов для отдельных тестовых систем. В этом разделе наш подход будет кратко рассмотрен, но более подробно это будет сделано в следующем разделе. Наш подход к верификации основан на имитационном моделировании и подразумевает два основных компонента тестовой системы: генератор стимулов и тестовый оракул (или компонент проверки реакций). Генератор должен создавать поток стимулов и передавать его оракулу, в основе которого обычно лежит эталонная модель, разработанная на выбранном уровне абстракции. Тест заканчивается, когда генератор сигнализирует об этом в соответствии со стратегией генерации.

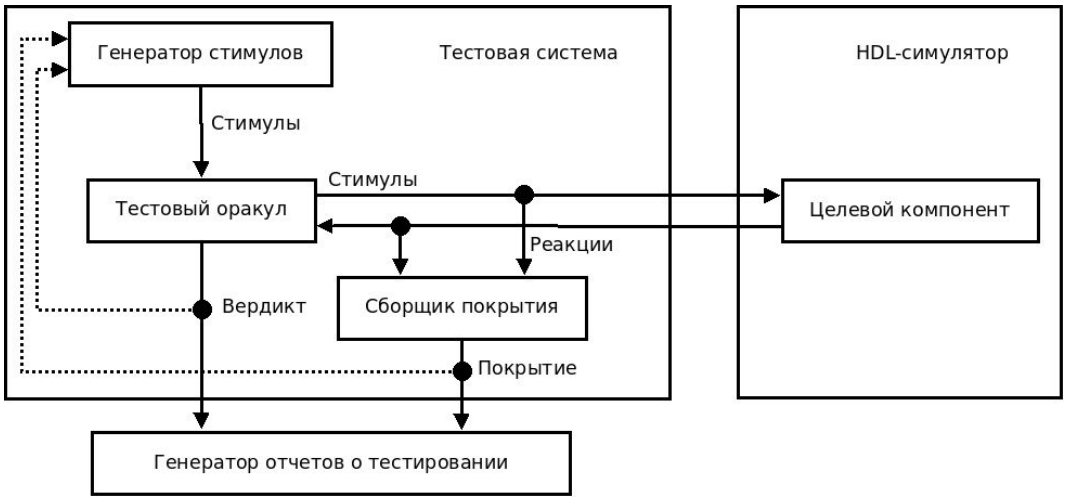

Рис. 2. Архитектура одиночной тестовой системы.

Одной из основных отличительных черт нашего подхода при сравнении его с OVM является то, что эталонные модели, используемые в компонентах проверки реакций, могут быть первоначально написаны на высоком уровне абстракции и потом они могут быть уточнены в соответствии с прогрессом разработки DUV. Чтобы сделать это возможным, в подходе используются специальные техники, такие как *механизм арбитража реакций, механизм детектирования реакций DUV,* а также некоторые другие. Кроме того, в процессе проектирования DUV верификаторы обычно разрабатывают *программный эмулятор* всего DUV. Поскольку обычно это делается с помощью C++, полезно придерживаться того же самого языка программирования в том числе для создания эталонных моделей для тестовых систем с помощью повторного использования частей эмуляторов. Так как подход, описанный в [4] использует C++, он имеет определенное преимущество перед OVM при решении задачи повторного использования частей эмулятора.

3. АРХИТЕКТУРА ОДИНОЧНОЙ ТЕСТОВОЙ СИСТЕМЫ

Архитектура, описанная в [4] была основана на технологии UniTESK [11], разработанной в Институте системного программирования РАН. Архитектура включает *генераторы стимулов* (включая обходчики *конечных автоматов,* в которых реализованы *неизбыточные алгоритмы обхода* [12]), *тестовый оракул (компонент проверки реакций),* а также *компонент отслеживания покрытия (сборщик покрытия)* и *генератор отчета о тестировании* (см. рис. 2). Генератор стимулов создает *последовательность сообщений* и посылает их как входные параметры при вызове методов запуска операций в эталонной модели. Эти вызовы называются *посылкой модельных стимулов:*

```
dut.start(&DUT::pop_stimulus, dut.iface1, msg);
```

Компонент проверки реакций обрабатывает сообщения и создает *модельные реакции.* Затем он посылает стимулы (которые теперь называются *реализационными стимулами)* в модель *тестируемого (целевого) компонента* и получает *реализационные реакции.* После этого тестовый оракул проверяет соответствие между реакциями модели и реализацией и возвращает *вердикт* касательно корректности тестируемого DUV на данном такте его работы. Компонент отслеживания покрытия сохраняет информацию об обновлении *функционального покрытия* на каждом такте. Генератор отчетов на основе трасс тестирования отображает важную информацию о процессе верификации, такую как вызываемые операции, достигнутое функциональное покрытие и результат верификации.

Наиболее сложным компонентом, требующим и позволяющим повторное использование,

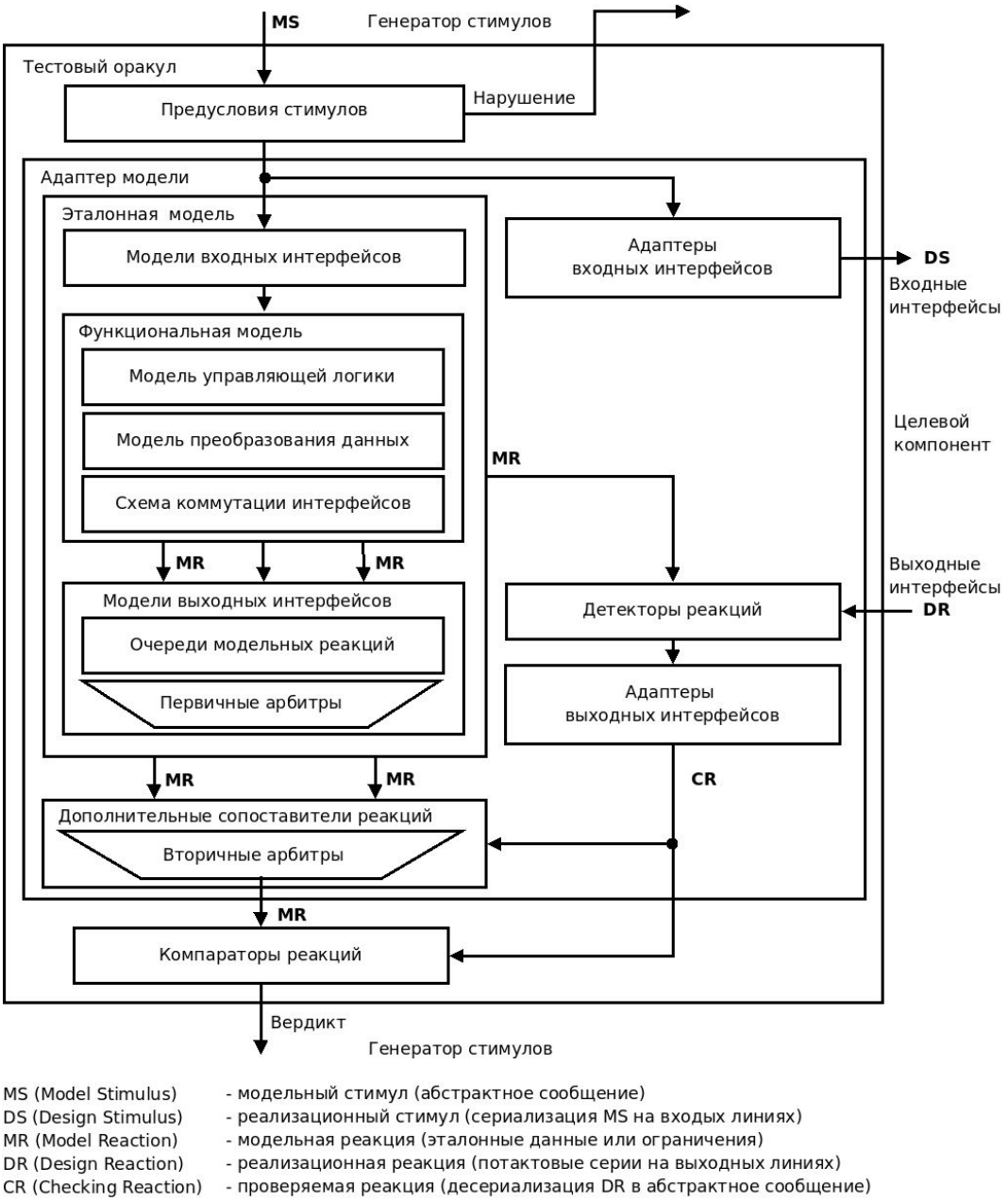

Рис. 3. Архитектура тестового оракула.

является тестовый оракул (см. рис. 3). Тестовый оракул дополняет свою эталонную модель всеми необходимыми функциями, которые позволяют генератору стимулов (или другому оракулу) использовать эталонную модель и адаптер эталонной модели.

Сообщения, посылаемые тестовому оракулу, называются *модельными стимулами* **(MS).** Генератор (или другой оракул) направляет поток **MS** в одну из *моделей входных интерфейсов.*

При получении **MS** проверяется *предусловие* того, что данный стимул **MS** может быть запущен в данном состоянии модели. **MS,** чьи условия запуска не выполнены, отвергается. В противном случае стимул подается на DUV через *адаптеры входных интерфейсов* (на этом шаге обрабатываемые MS называются реализационными *стимулами,* **DS)** и на *функциональную модель* через модели *входных*

интерфейсов, выбранные генератором.

Функциональная модель создает *модельные реакции* **(MR)** и размещает их в одну из моделей *выходных интерфейсов,* выбранную в соответствии с некоторыми критериями, включенными в функциональную модель.

Модели выходных интерфейсов содержат *очереди реакций,* хранящие **MR,** и *первичные арбитры,* выбирающие подмножество MR на данном такте симуляции. Арбитры работают в соответствии со стратегией, выбранной разработчиком тестов. Подмножество **MR** посылается в детекторы реакций для того, чтобы помочь распознать реализационные *реакции* **(DR).**

Компараторы реакций выбирают подмножество **MR** на моделях выходных интерфейсов и начинают ожидать соответствующих реакций тестируемого компонента. На данное подмножество могут быть наложены дополнительные ограничения с помощью вторичных арбитров, которые настраиваются разработчиком тестовой системы. На время ожидания реакций также накладывается определенное временное ограничение. Если оно нарушается, тестовая система сигнализирует об *ошибке ожидания* и останавливает свою работу.

Когда тестируемая реализация отвечает реакцией, она копируется в сообщение **DR,** которое будет находиться в одном из *адаптеров выходных интерфейсов* (соответствующая **MR,** если она найдена, помогает определить в каком именно). Если некоторые из реакций компонента **DR** не будут иметь соответствующих реакций модели **MR,** то тестовая система остановит свою работу с ошибкой прихода *неожиданной реакции.*

Адаптеры выходных интерфейсов посылают *проверенные реакции* **(CR)** в компаратор реакций для нахождения соответствующей **MR,** удовлетворяющей ограничениям, созданным вторичными арбитрами. После прохождения всех стадий, проверяется *постусловие:* отношение эквивалентности между **MR** и **CR.** Если **MR** и **CR** одинаковы, процесс тестирования продолжается. Если при сравнении будет обнаружена какая-нибудь проблема, тестовая система покажет данную ошибку и завершит свою работу.

Тест успешно заканчивается, когда генератор стимулов сделал все, что от него требовал разработчик тестовой системы (например, посещение всех достижимых состояний и др.).

4. АРХИТЕКТУРА ОБЪЕДИНЕННОЙ ТЕСТОВОЙ СИСТЕМЫ

Предлагаемый подход к разработке единичных тестовых систем является совместимым с идеями TLM создания интерфейсов компонентов тестовых систем и может быть использован при решении задачи создания тестовой системы для всего DUV на основе тестовых систем для отдельных модулей DUV. Повторно использовать тестовые системы для модулей удобно, когда части тестовых системы могут быть присоединены друг к другу через интерфейсы, которые они уже имеют. Следовательно, выбранный путь, придерживаться идей TLM при разработке интерфейсов, имеет определенные преимущества. TLM как раз и предназначен для решения задач повторного использования частей тестовой системы в том виде, в котором они уже есть, без обращения к копированию их частей и вставке кода в новые тестовые системы. Разработка сложных тестовых систем возможна с помощью следующих шагов.

Когда соединяются несколько тестовых систем между собой, некоторые из них перестают быть соединенными с DUV. Мы предлагаем создание общей тестовой системы с включением всех „маленьких“ тестовых оракулов, ранее находившихся в соединяемых тестовых системах (см. рис. 4). Следовательно, входные и выходные адаптеры и детекторы реакций должны быть модифицированы для создания соединения с другими тестовыми оракулами. К счастью, отделение оракула и эталонных моделей друг от друга позволяет нам сделать соединение без какой-либо существенной модификации эталонных моделей. При соединении компонентов проверки реакций, мы создаем общую тестовую систему и помещаем вложенные тестовые оракулы в нее. Общая тестовая система обладает своим генератором стимулов, тестовым оракулом и сборщиком покрытия. Повторное использование частей тестовых систем, разработанных ранее, при

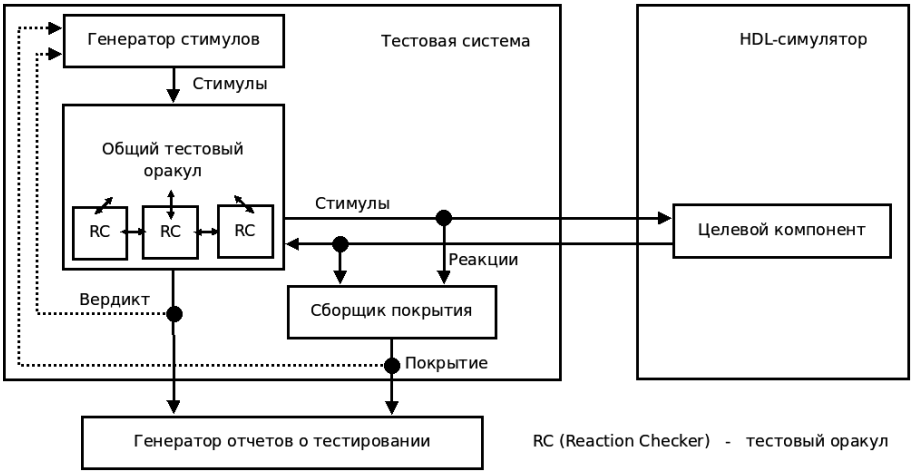

Рис. 4. Архитектура многомодульной тестовой системы.

разработке всех этих частей, – это само по себе хороший результат.

Генератор стимулов может унаследовать сценарные функции из ранее создававшихся генераторов, но только в том случае, если части создания модельных стимулов и вызова изначального тестового оракула были отделены друг от друга и находятся в разных функциях. В этом случае используемые тестовые оракулы могут быть легко заменены новым тестовым оракулом с помощью перегрузки соответствующих функций.

Сборщик покрытия – это общий компонент для всех тестовых систем. Для его настройки используется описание структур покрытия, которые необходимо в нем зарегистрировать. Используемый формат структур и способ их регистрации одинаков в одиночном и многомодульном случаях. Информация о покрытии обновляется на основе данных эталонной модели. Поскольку прежние модели включаются в общую тестовую систему, повторное использование покрытий получается само собой. Общая тестовая система просто вызывает необходимые функции на каждом такте симуляции, собирающие информацию о покрытии. Необходимо заметить, что структуры покрытия из одиночных тестовых систем в некоторых случаях не дают никакой интересной информации для случая объединенных DUV, и в этом случае необходимо задавать отслеживание пересечения предыдущих покрытия или создавать новые структуры покрытия, специально предназначенные для объединенного DUV.

Объединенные тестовые оракулы – это одна из наиболее сложных частей объединяемых тестовых систем. Общий тестовый оракул выглядит так же, как и одиночный оракул, но должен использовать функциональность всех вложенных в него тестовых систем. Только общему тестовому оракулу разрешено менять значения на сигнальных линиях DUV: прежние адаптеры входных и выходных интерфейсов выключаются. Их выключение возможно посредством их перегрузки методами, посылающими сообщения не в отсутствующий DUV, а в другие тестовые оракулы (см. рис. 5).

Для упрощения соединения компонентов проверки реакций мы предлагаем использовать каналы. Канал – это способ соединения адаптера выходного интерфейса и модели входного интерфейса между собой. Одним из необходимых условий соединения является трансляция сообщений из формата отправляющего интерфейса в формат, допустимый принимающим интерфейсом, что и возлагается на канал. Канал также может отправлять широковещательные сообщения в несколько принимающих интерфейсов. Использование каналов показано на рис. 6.

Перегруженные модели входных интерфейсов обычно содержат проверки предусловий, поэтому они могут проверять протокол взаимодействия между тестируемыми компонентами. Это может помочь вскрыть

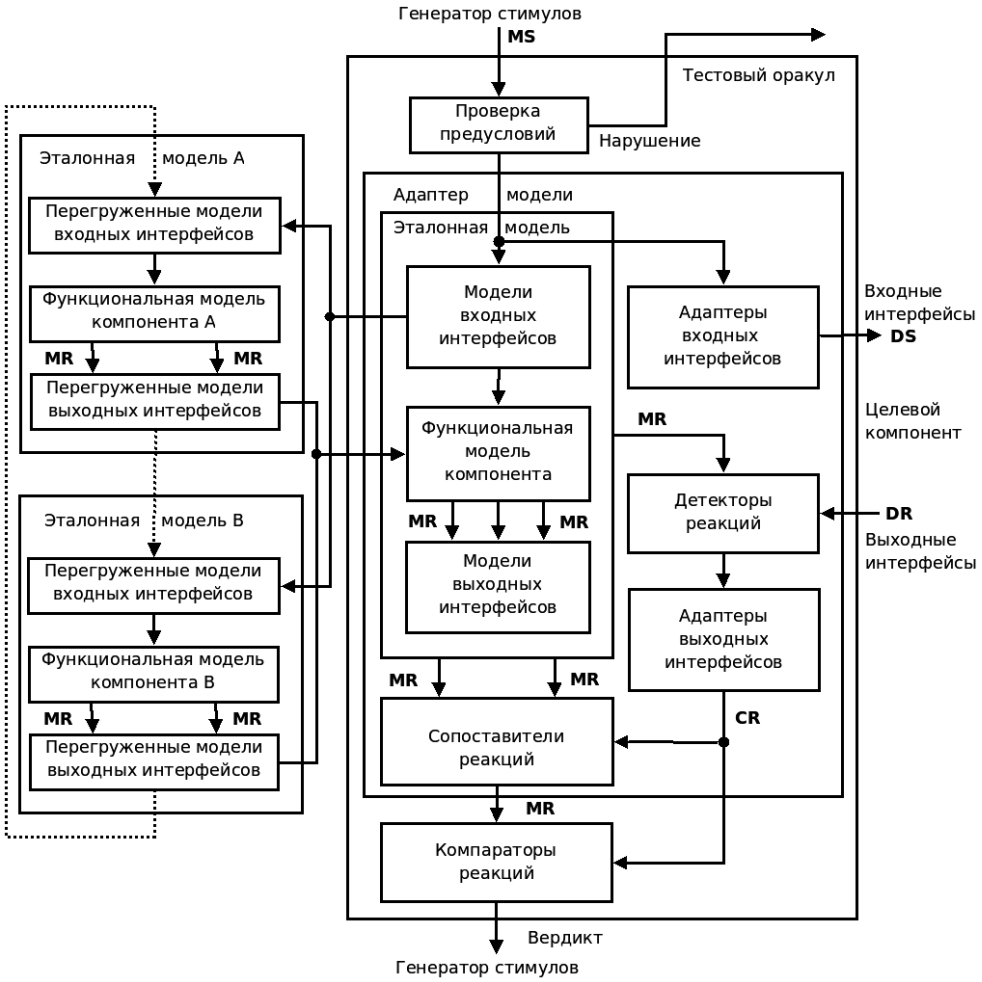

Рис. 5. Архитектура многомодульных компонентов проверки реакций.

проблемы, которые не могут быть обнаружены, если эталонная модель принимает входные воздействия только от генератора стимулов. Входные переменные имеют большой разброс значений, и в объединенном случае присваиваемые другими модулями значения из этого диапазона будут соответствовать ситуации сильно приближенной к реальной работе частей DUV и, что самое главное, могут быть проверены.

Данный подход имеет реализованную библиотеку поддержки на C++ и может использовать некоторые части системных эмуляторов, обычно разрабатываемых на C++. Более того, эталонные модели благодаря их архитектуре могут быть повторно использованы при разработке моделей аппаратуры на языке Verilog в качестве заглушек. В этом случае адаптеры входных и выходных интерфейсов работают необычным образом: адаптеры входных интерфейсов берут сообщения из DUV, обрабатывают их на функциональной модели и посылают сообщения в DUV с помощью адаптеров выходных интерфейсов (см. рис. 7).

Общая тестовая система контролирует тестовые оракулы, встроенные в DUV. Эти компоненты позволяют разработчику HDL-реализации ускорить процесс ее создания, так как код с той же самой функциональностью разрабатывается на языке C++ обычно быстрее, чем на Verilog.

Подводя некоторый итог, можно сказать, что данный подход позволяет разрабатывать тестовые системы, которые могут быть использованы в качестве частей общих тестовых систем. Эти тестовые системы проверяют не только выходные данные DUV, но и входные в них данные с помощью

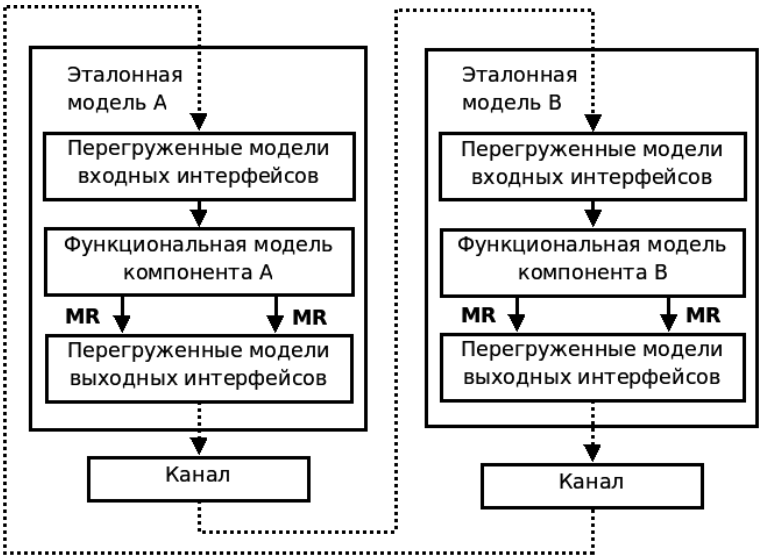

Рис. 6. Использование каналов в компонентах проверки реакций.

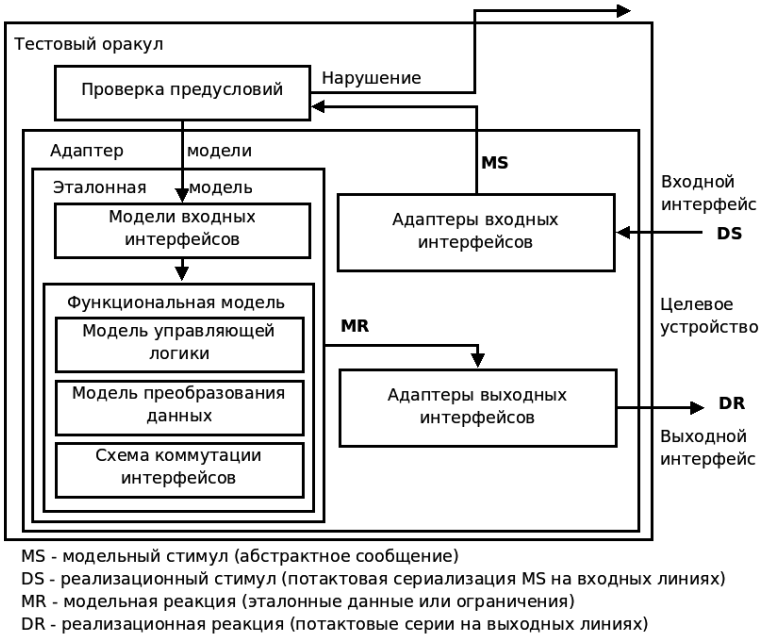

Рис. 7. Архитектура встроенного в DUV тестового оракула.

проверок предусловий. Когда тестовые системы подсоединены друг к другу, а не к DUV, они могут проверять поведение соседних систем, таким образом, они могут проверять протокол взаимодействия компонентов DUV. Каждая тестовая система поддерживает специальные средства, позволяющие создавать соединения, такие как интерфейсы и каналы. Кроме всего прочего, для уменьшения времени, затрачиваемого на создание первой версии компонентов DUV для системной верификации, тестовые системы могут быть включены в Verilog-код тестируемой системы, когда тестируемая система еще находится в стадии разработки. Подход поддерживается библиотекой на языке C++, поэтому тестовые системы, разрабатываемые в соответствии с подходом, также используют C++. Это дает возможность использовать богатый набор средств, предоставляемых данным языком для упрощения разработки. Необходимо заметить, что C++ обычно используется в

системных эмуляторах DUV, поэтому части эмуляторов могут быть повторно использованы в качестве эталонных моделей в тестовых системах, причем обратное утверждение также верно. И, наконец, библиотека является совместимой с подходом UniTESK, что означает возможность разработки высококачественных тестов на основе обхода конечных автоматов и использование возможностей распределенного тестирования на кластерах [13].

5. ОПЫТ ПРИМЕНЕНИЯ ПРЕДЛАГАЕМОГО ПОДХОДА

Подход к разработке одиночных тестовых систем показал свою эффективность в ряде проектов [4]. Наиболее интересные примеры приведены в таблице 1.

Затраты времени в таблице 1 включают написание плана верификации, разработку тестовой системы, а также сам процесс верификации и отладки разработанной тестовой системы. В случае неблокирующей кэш-памяти L2 затраты также включали учет постоянного изменения DUV. Таблица 1 показывает, что время, затраченное на верификацию, может быть грубо оценено в один человеко-месяц на одну тысячу строк первоначального кода. Случай L2 является исключением из этого правила, поскольку он потребовал дополнительных затрат на поддержку учета изменений.

Интересным бы являлось сравнение между полученными нами результатами и оценкой усилий, требующихся в случае альтернативного подхода (OVM). Недостаточное количество оценок применения OVM в доступных источниках не дали нам сделать это. Мы предполагаем, что время, затрачиваемое на разработку тестовых систем в данном случае близко к предлагаемому нами подходу. Это ощущение основано на том, что OVM также использует объектно-ориентированный язык программирования, кроме того, набор разрабатываемых компонентов выглядит похожим образом. Тем не менее, наш подход за то же самое время позволил бы использовать, в том числе, дополнительные средства создания потока стимулов, основанного на обходе конечных автоматов. По факту, данный вопрос не был нами детально исследован, и является вопросом дальнейших исследований.

Предлагаемый путь объединения – это новая возможность в ранее созданном подходе к созданию одиночных тестовых систем. Пока было проведено только несколько экспериментов для оценки возможности объединения. Сначала была разработана тестовая система для простого модуля FIFO. Это заняло примерно два дня, включая время на документирование проекта. Затем были взяты и соединены между собой три таких модуля. Два из них стали входными буферами в полученной схеме, третий – выходным. Между ними был помещен арбитр, выбиравший один из входных буферов для передачи данных в выходной. Арбитр всегда выбирал первый по счету входной буфер, если в нем были данные. Для тестирования этой ячейки мы соединили три тестовых системы для изначальных модулей FIFO и добавили функциональность арбитра довольно простым путем: функциональная модель всегда читала данные из первого FIFO, если там были какие-либо данные, в обратном случае читался второй FIFO. Генератор стимулов использовал изначальные сценарные функции, такие как „положить данные“ и „взять данные“, но с небольшими изменениями, так как теперь два FIFO могли только принимать данные, а третье – только выдавать. Адаптеры интерфейсов вложенных тестовых систем были перегружены. Ранее для осуществления записи и чтения использовались два интерфейса: один входной и один выходной. В случае входных FIFO, входные интерфейсы не изменились (код, устанавливающий значения на входные сигнальные линии из вложенных оракулов был удален ранее). Адаптеры выходных интерфейсов входных FIFO были перегружены для возможности осуществлять посылку сообщений по входным интерфейсам выходного FIFO. Так, изначально регистрация адаптеров интерфейсов выглядела так:

```
CPPTESK_SET_OUTPUT_ADAPTER(iface3,
    FIFOMediator::deserialize_iface3);
```

где deserializer (десериализатор) – это функция, транслирующая значения сигнальных линий DUV в реализационные сообщения. Как только выходной интерфейс iface3 перестал быть

Таблица 1. Опыт применения предлагаемого подхода.

Верифицируемый компонент	Глубина верификации	Исходный код, Тыс. строк кода	Затраты, Чел.-мес.
Буфер трансляции адресов (TLB)	До потактовой	2.5	2.5
Неблокирующая кэш-память L2	С детализированным временем	3	6
Коммутатор данных северного моста	До потактовой	3	3
Устройство доступа к памяти (MAU)	До потактовой	1	1

выходным, мы перегрузили его адаптер:

```
CPPTESK_SET_INNER_IFACE_ADAPTER
(FIFOMediator,
fifo0, fifo0.iface3, MM::deserialize_inner_iface0);
```

Новый десериализатор вызывает выходной FIFO fifo2:

```
CPPTESK_DEFINE_PROCESS(
  MM::deserialize_inner_iface0)
{
...
if(!fifo2.is_full()) {
...
 fifo2.start(&FIFO::push_msg,
  fifo2.iface1, data);
 }
}
```

Модель выходного интерфейса выходного FIFO просто передает сообщения в модель выходного интерфейса общего оракула. Для создания общей тестовой системы было затрачено около $\frac{1}{2}$ человеко-дня, включая время на исследование возможности соединения тестовых систем. Время, которое могло бы быть потрачено на разработку общей тестовой системы с нуля можно оценить в два дня, но это не очень важно. Время, затрачиваемое на соединение тестовых систем, имеет слабую зависимость от их сложности. В основном, оно зависит от числа входных и выходных интерфейсов и усилий, необходимых на их соединение. Можно привести следующую оценку: соединение одного исходящего и одного входящего интерфейса проводится примерно за один час. Это время тратится на разработку канала между интерфейсами, транслирующего их сообщения. Средний модуль аппаратуры по нашим оценкам включает порядка 10 входных и 10 выходных интерфейсов, поэтому соединение двух таких модулей было бы возможно за один–два человеко-дня, в соответствии с производительностью инженера.

6. ЗАКЛЮЧЕНИЕ

Данный подход предлагает удобный путь разработки тестовых систем для отдельных частей DUV, а затем объединение таких тестовых систем с высокой степенью повторного использования. Основными преимуществами подхода является проверка межмодульных соединений без написания дополнительного кода и специальные средства, упрощающие повторное использование. Подход поддерживается библиотекой, разработанной на языке C++, что позволяет использовать широкий набор средств данного языка при разработке тестовых систем. Тот факт, что системные эмуляторы обычно создаются на C++, указывает на возможность повторного использования частей этих эмуляторов в качестве эталонных моделей разрабатываемых тестовых систем, причем обратное – также верно. У подхода есть два бонуса. Во-первых, тестовые системы могут являться частями тестируемых компонентов и контролировать обмен данными с подсоединенными к ним модулями таких HDL-моделей. Это особенно актуально в том случае, когда DUV все еще разрабатывается для того, чтобы ускорить начало его верификации. Вторым бонусом является то, что библиотека, поддерживающая подход, разработана в соответствии с технологией UniTESK, которая позволяет создавать высококачественные тесты на основе обхода конечных автоматов и использовать кластеры для распараллеливания тестирования.

СПИСОК ЛИТЕРАТУРЫ

1. IEEE 1364–2005, Verilog Standard.

2. *Bergeron J.* Writing testbenches: functional verification of HDL models. Kluwer Academic Publishers, 2003.

3. Описание инструмента Formality от Synopsys (http://www.synopsys. com/Tools/Verification/ /FormalEquivalence/Pages/Formality.aspx)

4. *Chupilko M., Kamkin A.* A TLM-based approach to functional verification of hardware components at different abstraction levels. 12th Latin-American Test Workshop, 2011.

5. *Clarke E., Grumberg O., Peled D.* Model Checking. MIT Press, 1999.

6. *Kneuper R.* Limits of Formal Methods. Formal Aspects of Computing (1997) 3: 1–000, 1997.

7. *Iman S.* Step-by-step Functional Verification with SystemVerilog and OVM. Hansen Brown Publishing Company, 2008.

8. *Иванников В.П., Камкин А.С., Кулямин В.В., Петренко А.К.* Применение технологии UniTesK для функционального тестирования моделей аппаратного обеспечения. Препринты Института системного программирования РАН. Препринт 8. 2005.

9. Open Verification Methodology, http://www.ovmworld.org.

10. *Kamkin A.S., Chupilko M.M.* Survey of Modern Technologies of Simulation-Based Verification of Hardware. Programming and Computer Software, 37(3):147–152, 2011.

11. *Баранцев А.В.* Подход UniTesK к разработке тестов: достижения и перспективы. Труды Института системного программирования РАН. Т. 5. С. 121–156. 2004.

12. *Бурдонов И.Б., Косачев А.С., Кулямин В.В.* Неизбыточные алгоритмы обхода ориентированных графов: недетерминированный случай. Программирование. 30(1):2–17, 2004.

13. *Demakov A., Kamkin A., Sortov A.* High-Performance Testing: Parallelizing Functional Tests for Computer Systems Using Distributed Graph Exploration. Труды конференции „Облачные вычисления. Образование. Исследования. Разработки 2011“.