УДК 004.415.538, 004.434

Особенности использования возможностей объектно-ориентированного программирования SystemVerilog для функциональной верификации многоядерных СнК

Ф.М. Путря

Открытое акционерное общество Научно-производственный центр "Электронные вычислительно-информационные системы", fputrya@elvees.com

***Аннотация* — Известно, что именно функциональная верификация становится самым сложным и затратным этапом разработки современных многоядерных систем на кристалле. По этой причине внедрение в маршрут разработки СнК новых инструментов и методов, ускоряющих процесс верификации, является приоритетной задачей. Возможности ООП, заложенные в SystemVerilog и созданные на основе этих возможностей верификационные библиотеки, такие, как UVM, позволяют значительно упростить процесс разработки верификацонных сред и тестов для СнК. В статье рассмотрены некоторые аспекты применения данных технологий, а также предложено решение, позволяющее унифицировать и использовать в платформенном подходе к разработке тестов также и эталонные C++ модели блоков СнК.**

***Ключевые слова* — функциональная верификация SystemVerilog, ООП, UVM, уровень транзакций, эталонная модель.**

I. Проблема верификации СНК

Функциональная верификация проекта системы на кристалле, направленная на как можно более раннее обнаружение максимального числа архитектурных и аппаратных ошибок в проекте - один из наиболее важных этапов разработки систем на кристалле (СнК). Колоссальная сложность современных СнК делает трудной и задачу их верификации, поскольку сложность тестирования обладает экспоненциальной зависимостью от сложности самой системы. Даже специалистам в области проектирования СнК подчас сложно выявить все архитектурные уязвимости, потенциально способные сказаться на работоспособности системы и таких её характеристиках как производительность, достигаемая на реальных приложениях. В системах такой сложности имеют место ошибки, которые обнаруживают себя только при одновременном сочетании множества условий, которые бывает весьма сложно создать при верификации, не говоря уже о выявлении полного набора таких критических сочетаний. Верификаторам в таких условиях важно не просто доказать, что проект системы на всех уровнях абстракции, начиная от блочного представления и уровня транзакций (TLM), заканчивая уровнем регистровых передач (RTL) и топологией, полностью соответствует заявленной спецификации, но и **доказать, что в текущей реализации проект позволяет успешно выполнять на нём целевые задачи во всех допустимых условиях применения системы**. В результате задача верификации теперь включает в себя не только проверку всех свойств в соответствии со спецификацией, но и целый пласт тестов, при разработке которых инженеры опираются не на спецификацию проектируемой СнК, а на алгоритмы конечных приложений. Более того, разработка СнК теперь неотделима от разработки программного обеспечения (ПО), которое будет исполняться на проектируемой СнК, и, таким образом, задача верификации распространяется уже на весь программно-аппаратный комплекс, а не только на модель аппаратуры.

Если касаться особенностей верификации именно многоядерных СнК, то среди них стоит выделить очень сложный характер трафика, приходящийся как на коммутационную логику, так и на отдельные блоки системы, такие как контроллеры памяти или контроллеры периферийных интерфейсов. Сложность трафика определяется перемежением потоков обращений от множества источников – нескольких ядер одного типа, но выполняющих задачи различного характера, специализированных ядер выполняющих сигнальную обработку, обработку видео, графики и звука, а также от встроенных интеллектуальных контроллеров обмена данными (DMA и спецпроцессоры). Другой характерной особенностью является сложность модели многоядерной системы, резко ограничивающей скорость моделирования. С одной стороны, данная особенность приводит к необходимости применения высокоуровневых моделей блоков, составляющих систему. В частности, в технологии виртуального прототипирования применяются TLM модели блоков, описанные на SystemC, и взаимодействующие между собой посредством передачи транзакций. Использование TLM представления системы позволяет ускорить процесс моделирования и даже выполнять отладку ПО

на модели всей системы. С другой стороны, данная проблема еще больше усиливает роль верификации отдельных блоков в общем процессе верификации системы.

Скорость моделирования отдельного блока на порядок больше, чем скорость моделирования всей системы. В этом случае каждый блок системы верифицируется в собственном тестовом окружении, имитирующим внутрикристальные и периферийные интерфейсы, к которым он подсоединен в системе. При декомпозиции проекта СнК на блоки снижается комбинаторная сложность проекта, что упрощает генерацию тестов. При локальном тестировании блока лучше наблюдаемость и контролируемость ошибок, проще создавать тестовые сценарии и всевозможные краевые тестовые случаи. При условии использования апробированных внутрикристальных интерфейсов такая локальная проверка блока позволяет повысить вероятность успешной интеграции блока в систему, хотя и не избавляет от необходимости тестов блока, выполняемых на уровне всей системы. Дополнительно при таком подходе появляется требование всеобъемлющей локальной верификации каждого отдельного блока, поскольку полноценная проверка блока в составе модели СнК становится практически невыполнимой задачей.

Огромный объем тестов и тестовых окружений, требующихся для верификации СнК, а также их сложность, становятся основной проблемой функциональной верификации. При этом ограничивающим фактором становится даже не столько требуемое для моделирования всех тестов машинное время, сколько время, требуемое для разработки полного набора тестов. На фоне постоянно сокращающихся сроков разработки современных СнК и нехватке верификаторов эта проблема только обостряется. Поэтому на ускорение процесса разработки тестов направлены все силы инженеров, занимающихся верификацией, в частности:

1) для современных СнК уже не может идти речи о написании всех тестовых сценариев вручную – необходимы механизмы генерации случайных тестов;

2) разработка тестов постепенно переносится на более высокий уровень абстракции (уровень транзакций, потоков, приложений);

3) большая часть тестов разрабатывается и отлаживается в локальных тестовых окружениях блоков;

4) как для разработки тестов, так и для разработки тестовых окружений, активно применяются методы объектно-ориентированного программирования (ООП), позволяющие ускорить процесс разработки, упросить повторное использование кода и сделать код теста более понятным и читаемым;

5) при верификации системы, состоящей из множества блоков (ядер различного типа и периферийных контроллеров), а особенно в случае разработки серии СнК, применяется **платформенный подход**. Данный подход предполагает наличие библиотеки унифицированных верификационных компонент (элементов тестового окружения) и тестовых сценариев, набор правил создания новых компонент и тестов, а также средств оценки качества верификационных компонент, включаемых в библиотеку, и их соответствия заданным правилам. Такой подход позволяет упростить повторное использование кода верификационных компонент и тестов как по горизонтали (для тестовых окружений разных блоков и разных СнК), так и по вертикали (повторное использование теста при локальной верификации блока, при верификации всей СнК и для представлений проекта СнК на различных уровнях абстракции).

II. Применение возможностей ООП в SystemVerilog для тестовых окружений. Библиотека UVM

Для верификации как СнК, так и блоков, из которых она состоит, необходимы тестовые окружения, имитирующие условия в которых данный блок или СнК будут функционировать. Для обеспечения возможности создания тестов на высоком уровне абстракции, динамической конфигурации тестового окружения, параллельного запуска нескольких тестовых сценариев, имитирующих воздействия на блок со стороны нескольких ядер и лучшей структуризации кода, в последнее время для создания тестовых окружений всё чаще используют SystemVerilog (далее SV)[1]. Как тестовые окружения, так и тесты в этом случае разрабатываются по принципам ООП, а сами тесты создаются как потоки транзакций различного уровня абстракции. Типичное тестовое окружение, созданное на SV, состоит из верификационных компонентов – объектов классов, выполняющих определенную функцию в тестовом окружении. Примерами задач, которые выполняют верификационные компоненты, могут быть: генерация тестовых последовательностей, управление потоком тестовых транзакций, сбор покрытия, проверка корректности работы тестируемого устройства (класс, выполняющий данную задачу, называют анализатором, он, как правило, связан с эталонной моделью устройства) [2]. Для перевода процесса разработки тестов на более высокий уровень абстракции в тестовое окружение вводятся специальные объекты — транзакторы. ***Транзактор (Transactor)*** – элемент тестового окружения, преобразующий транзакцию с одного уровня абстракции на другой. Частным случаем транзакторов являются драйверы и мониторы, переводящие тестовые последовательности с уровня транзакций на уровень сигналов (драйвер) и обратно (монитор). Для каждого типа интерфейса, будь то внутрикристальный интерфейс (AXI, Whishbone, OCP) или периферийный интерфейс (USB, Ethernet, PCI), нужен собственный транзактор. В методологии UVM (Universal Verification Methodology) конфигурируемые транзакторы, способные работать и как драйверы и как мониторы,

получили название агентов [3]. Разработка таких агентов задача сложная и кропотливая, включающая постоянную сверку со спецификацией интерфейса. Трудоёмкость разработки таких элементов влечет за собой естественное желание их повторного использования во всех текущих и будущих проектах. Это накладывает ряд ограничений на стиль кода агента, а также на интерфейс его взаимодействия с другими элементами тестового окружения, такими как генераторы тестовых последовательностей, анализаторы, TLM модели ведомых устройств. Поскольку каждый такой элемент, как правило, разрабатывается независимой командой верификаторов, для их успешной интеграции в рамках одного тестового окружения и дальнейшего повторного использования они должны использовать стандартизированные интерфейсы взаимодействия с другими элементами тестового окружения. Этого можно достичь путем выработки стандартов, которым должны следовать все разработчики фирмы, однако разработка таких стандартов и контроль за их исполнением - задача довольно сложная. Разработчиками методологии и библиотеки UVM предлагается решение данной проблемы. UVM предоставляет набор базовых классов для всех стандартных верификационных компонентов, необходимых для создания типичного тестового окружения. Наследование от данных классов не только ускоряет процесс разработки тестового окружения за счет повторного использования уже заложенной в базовые классы функциональности, но и упрощает процесс интеграции верификационных компонентов от разных разработчиков, поскольку они используют одинаковые интерфейсы, наследованные от общих базовых классов. Так универсальным средством связи элементов в UVM являются порты, по которым элемент может передавать или принимать транзакции.

Следование единой методологии упрощает взаимопонимание не только между разработчиками внутри фирмы, но и между разработчиками различных фирм. Использование UVM дополнительно позволяет добиться унификации и упростить повторное использование также и тестовых алгоритмов. Тесты в этом случае создаются на более высоком уровне транзакций, что, во-первых, ускоряет их разработку, а во-вторых, делает их более гибкими при переносе на другие проекты. В ряде случаев даже модификация системного интерфейса блока (например, замена интерфейса AXI на OCP) не приводит к необходимости переписывания всех ранее созданных тестовых сценариев для данного блока. Создание параметризированных тестов на высоком уровне абстракции, которые опираются именно на целевую функцию блока, ещё больше упростит задачу переноса тестовых алгоритмов между проектами СнК. Ярким примером могут служить последовательные периферийные порты, основным тестовым сценарием для которых, вне зависимости от аппаратной реализации, является передача пакетов данных с реализацией перебора всех характеристик пакетов в процессе тестирования.

Однако, как показала практика, само по себе применение UVM еще не является залогом успешной интеграции компонент от различных разработчиков. Так способ конфигурации UVM компонента, способы его взаимодействия с другими компонентами зависят от стиля разработчика этого компонента. Компоненты, созданные с применением различных стилей, сложно интегрировать в рамках одного окружения не только из-за путаницы в именованиях и способах конфигурации, но и из-за необходимости разработки дополнительного кода, необходимого для сопряжения различных компонент. Чтобы избежать подобных проблем, внутри фирмы необходим набор дополнительных правил разработки верификационных компонент. В частности, одна из проблем, касающаяся интеграции эталонной модели в тестовое окружение и практически не регламентированная в UVM рассмотрена далее.

III. Проблема эталонной модели. Взаимодействие SystemVerilog и C++

Суть любого теста заключается в сравнении результатов поведения тестируемой модели с некоторым эталоном. В случае тестов со встроенной самопроверкой требующаяся часть эталонной модели реализуется в самой тестовой программе, однако для метода верификации, использующего генерацию случайных тестов, более предпочтительна полноценная эталонная модель, которая способна формировать отклики на произвольные тестовые воздействия. В этом случае проверка осуществляется не тестовой программой, а путём сравнения откликов эталонной и тестируемой моделей, а также за счет дополнительного сравнения трасс активности моделей и сравнения полного состояния моделей в контрольных точках. Два последних метода предъявляют дополнительные требования к интерфейсу взаимодействия тестового окружения с эталонной и тестируемой моделями, однако при этом значительно увеличивается наблюдаемость ошибок и упрощается отладка тестов. В результате упрощается и ускоряется процедура генерации тестов, поскольку анализ корректности выполняется тестовым окружением, а от разработчика теста требуется лишь тестовая последовательность, обеспечивающая максимальное покрытие. Однако такой подход требует качественной модели тестируемого устройства.

В книгах и статьях, посвященных тестовым окружениям, часто упоминается эталонная модель в качестве одного из основных элементов тестового окружения, однако собственно разработке самой эталонной модели уделяется очень мало внимания. Реализация эталонной модели полностью определяется верификаторами конкретной СнК. Эталонная модель должна позволять использовать ее в виде верификационного компонента, интегрированного в UVM тестовое окружение для проверки корректности

прохождения тестов. Эталонная модель также может быть использована для подмены RTL модели определенного блока в составе модели всей СнК или группы блоков. Такая подмена может потребоваться, если RTL код конкретного блока пока еще не готов, а моделировать всю систему уже необходимо, либо если в целях верификации есть необходимость моделирования смешанной модели, содержащей и эталонные и RTL модели блоков. Дополнительно существует требование возможности динамического создания нескольких объектов эталонных моделей блоков одного или разных типов для построения сложных тестовых окружений для группы блоков или всей СнК. При этом для реализации платформенного подхода в верификации желательна определенная стандартизация и для эталонных моделей. Например, желательно наличие некоторого унифицированного интерфейса к эталонным моделям, упрощающего создание большого числа типовых тестовых окружений для разных блоков.

Эталонная модель может быть разработана с нуля полностью на SV, как класс, интегрируемый в тестовое окружение, однако это требует существенных трудозатрат. В тоже время практически для каждой СнК разрабатывается программная модель для задач отладки ПО. Такая модель, как правило, разрабатывается на C++ с использованием принципов ООП. Дополнительно на начальных этапах разработки СнК для более точной оценки производительности системы часто разрабатывается TLM модель СнК, которая чаще всего реализуется с использованием библиотеки SystemC [4]. В этом случае модель структурно составляется из блоков таким же образом, как и будущая RTL модель, за тем исключением, что блоки связываются посредством TLM интерфейсов. В стандарте TLM 2.0, разработанным OSCI, для этой цели введены классы соккетов, инкапсулирующих в себе каналы передачи и отклика транзакций. Цель TLM 2.0 заключается в стандартизации TLM моделей блоков, разрабатываемых разными людьми и облегчения их интеграции в рамках TLM модели системы. TLM модели блоков часто реализуются за счет использования кода из обычной C++ программной модели и SystemC обертки реализующей TLM интерфейс для блока. За счет совместимости C++ и SystemC такие обертки реализуются достаточно просто.

Проблемой такой TLM модели при интеграции в SV тестовое окружение является то, что на данный момент нет стандартизированного механизма взаимодействия SystemC TLM моделей и SV кода. Кроме того, SystemC обертка в случае связи с SV будет избыточной и негативно скажется на скорости моделирования. В SV для взаимодействия с C/C++ кодом предусмотрен достаточно простой интерфейс вызова функций — DPI [5]. Данный механизм позволяет вызывать функции как в коде C/C++ из SV, так и SV функции из C/C++. Однако нас интересует более сложная реализация такого взаимодействия — нам требуется использовать эталонную TLM модель как полноценный SV класс, удовлетворяющий всем описанным выше требованиям. Для удобства использования в SV компонент эталонной модели должен представлять собой SV класс, инкапсулирующий в себе процедуры взаимодействия с C++ моделью (класс SV обёртки). Для взаимодействия такого класса с элементами тестового окружения были выбраны библиотечные UVM классы TLM портов, которые, с одной стороны, схожи по функциональности с TLM соккетами SystemC моделей (что минимизирует затраты на адаптацию C++ модели к такой реализации), а с другой - позволяют подключать такую модель к стандартным UVM компонентам. Чтобы обеспечить возможность динамического создания нескольких таких классов необходима некоторая надстройка над DPI.

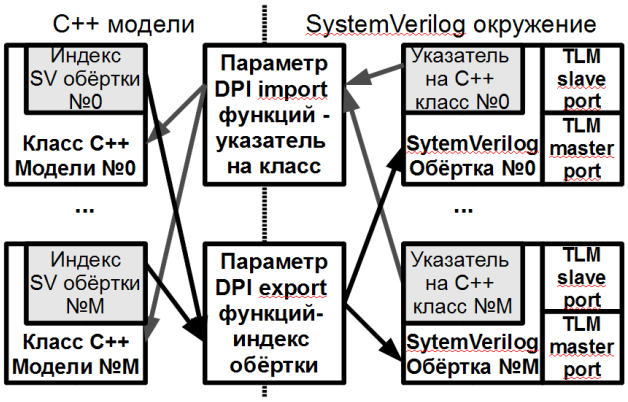

Рис. 1. Взаимодействие SV оберток и C++ моделей

DPI интерфейс не позволяет явно обращаться к методам классов. Штатными методами DPI обеспечивается только вызов глобальных функций. Чтобы обеспечить связь объекта SV обертки с соответствующим объектом C++ модели через глобальные функции необходимо передавать указатели на созданные объекты. С использованием SV типа chandle, эквивалентного C типу *void, можно сохранять и передавать указатели на объекты C++ моделей. SV не позволяет динамически преобразовывать тип chandle к указателям на классы, поэтому приходится для этих целей применять идентификаторы класса, позволяющие выбрать из динамического массива созданных объектов оберток, ту которая соответствует конкретному C++ классу. Таким образом, в каждой SV обертке хранится указатель на класс модели, а в каждой C++ модели идентификатор обертки. Передавая через глобальные интерфейсные функции один дополнительный параметр (указатель или индекс) в этом случае мы можем получать доступ к методам объекта C++ модели из соответствующего объекта SV обертки, или, наоборот, к методам нужного объекта SV обертки из объекта C++ модели, даже в случае, если создано несколько копий объектов модели (рис 1).

В общем случае устройство может иметь ведомый и ведущий интерфейсы. Дополнительно необходимо предусмотреть средства контроля за темпом исполнения TLM модели. Поскольку в C++ модели нет понятия о текущем времени моделирования, то в случае, если никакие специальные методы синхронизации не используются, может иметь место «разбегание» TLM модели и тестируемой модели, что усложнит отладку тестов. Условно назовем интерфейс, используемый для контроля за темпом моделирования, интерфейсом шага моделирования. Если набор DPI функций, используемых для реализации интерфейсов ведомого, ведущего и шага стандартизовать, то появится возможность создания базового класса SV обертки, на основе которого можно реализовывать обертки для моделей с произвольным числом ведомых и ведущих интерфейсов и создать единый шаблон тестового окружения для большинства блоков.

С точки зрения политики вызовов DPI функций есть два подхода. В первом случае все функции вызываются только из SV обертки. В этом случае код обертки максимально простой, однако усложняется реализация интерфейса со стороны C++ модели и появляется необходимость постоянной проверки наличия запросов транзакций со стороны C++ модели, что потенциально приводит к снижению скорости моделирования. Во втором варианте запрос на выполнение транзакции или отработку прерывания может быть передан в SV обертку со стороны C++ модели путем вызова DPI функций. При таком подходе C++ модель требует минимум модификаций, однако несколько усложняется уже код обертки – необходима реализация очередей, хранящих запросы от C++ модели, однако влияние на производительность минимизируется. Таким образом, второй вариант является более предпочтительным.

В случае ведомого интерфейса модель не является инициатором транзакций. Транзакции получает SV обертка по TLM порту, после чего, посредством DPI функций ведомого интерфейса, осуществляется доступ к C++ модели. DPI интерфейс ведомого может содержать всего одну функцию transponse(), вызываемую из окружения SV. Transponse - в UVM стандартное название функции, вызывающейся в TLM модели для получения эталонного ответа на тестовую транзакцию. Однако она может применяться только для моделей, дающих отклик «моментально». В остальных случаях необходимо разделение процессов

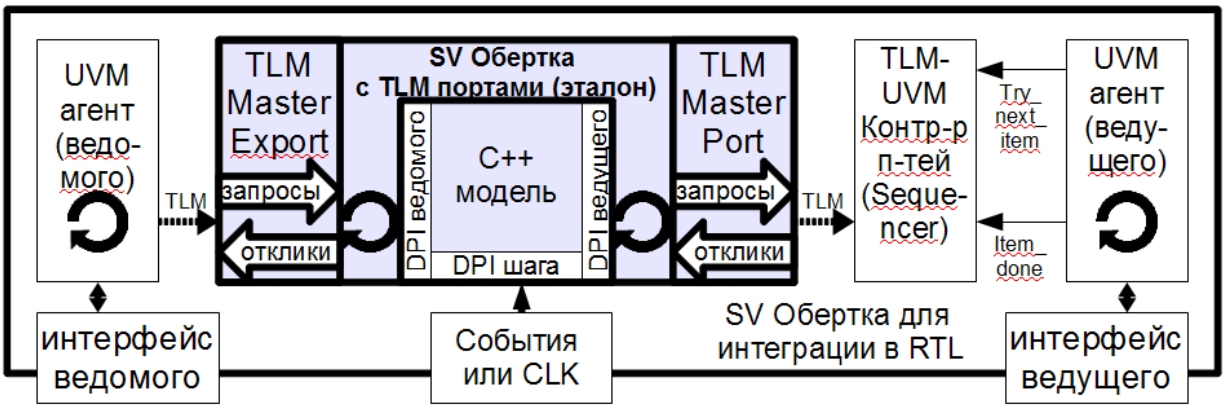

Рис 2. Структура SV обертки для подмены RTL модели блока, в её состав входит SV обертка C++ модели, используемая в качестве эталона и в тестовых окружениях (выделена серым)

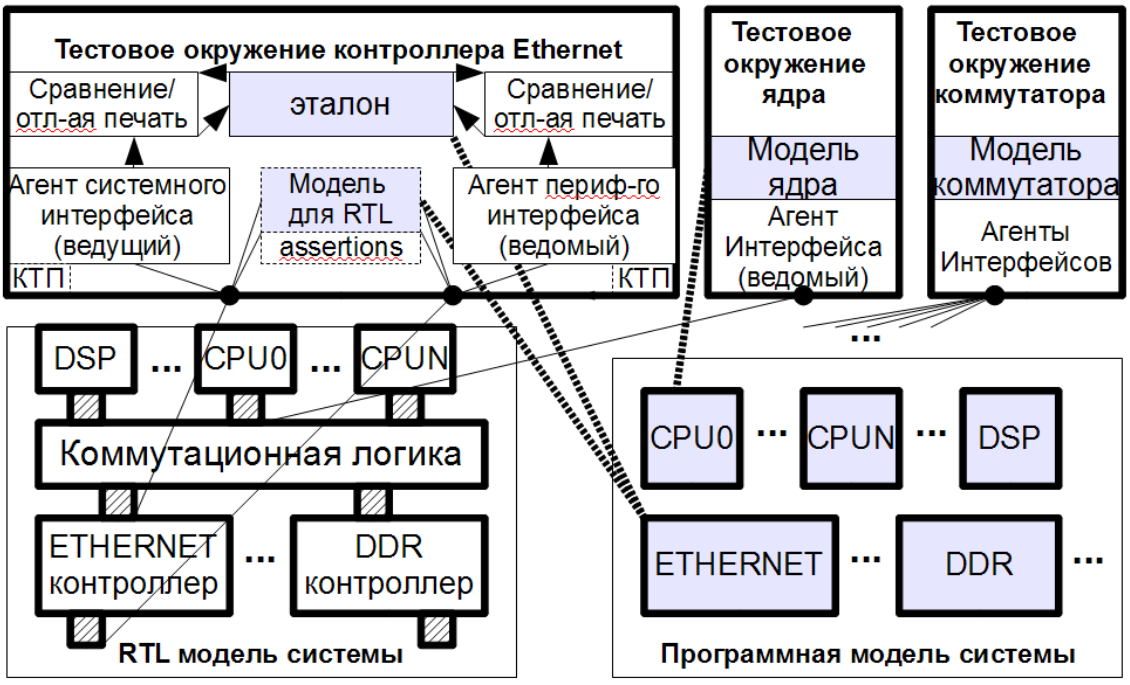

Рис 3. Повторное использование тестовых окружений блоков на уровне системы. Части кода программной модели используются как для подмены RTL модели при смешанном моделировании, так и для проверки корректности работы блока. КТП – контроллер тестовых последовательностей

подачи запросов и чтения откликов, для которых могут использоваться функции put_s_tran_req() (вызываемой из SV) и put_s_tran_rsp() (вызываемой из C++). В качестве параметров функциям необходимы указатель на C++ класс модели или индекс обертки и содержимое самой транзакции (адрес, тип обращения, размер данных, сами данные и прочая информация, характерная для транзакций, применительно к данной модели).

В случае ведущего интерфейса инициатором является C++ модель. В этом случае запрос на выполнение транзакции или обработку прерывания может быть передан в обертку со стороны модели вызовом функции put_m_tran_req(). Для минимизации вероятности возникновения зависаний, put_m_tran_req предлагается делать неблокирующей функцией, выполнение которой не требует симуляционного времени. Транзакция, полученная посредством вызова этой функции, кладется в очередь транзакций и при переполнении очереди просто возвращает ошибку в качестве результата функции, но не приводит к «зависанию» модели. Результат же транзакции возвращается посредством функции put_m_tran_rsp(), вызываемой из SV обёртки. Параметры данных функций такие же, как и в случае ведомого интерфейса, однако, для обеспечения возможности параллельного исполнения нескольких транзакций от одной модели (например, для случая многопоточного ядра), вводятся дополнительные параметры, используемые для идентификации потока, инициировавшего транзакцию.

Интерфейс шага моделирования позволяет запускать моделирование TLM эталона на определенное число шагов, контролируя, таким образом, темп моделирования блока. Число таких шагов в рамках одного вызова функции подбирается эмпирически для каждого тестового окружения так, чтобы минимизировать разбежку TLM и RTL моделей.

Для интеграции в RTL модель системы SV обертка расширяется агентами аппаратных интерфейсов, которые выполняют переход с уровня транзакций на уровень сигналов в соответствии со стандартами данных интерфейсов. Агент ведомого интерфейса подключается непосредственно к TLM порту. Агент ведущего интерфейса подключается к TLM порту через специальный преобразователь, реализующий стандартный интерфейс к UVM драйверу (seq_item_port, rsp_port). Агенты для такой обёртки могут быть взяты из стандартной библиотеки верификацонных UVM агентов интерфейсов, без необходимости их специально адаптировать (рис 2).

IV. Заключение

В маршруте разработки современных СнК активно применяется тестирование блоков с использованием локальных тестовых окружений и смешанное моделирование TLM и RTL моделей. Разработка каждого отдельного тестового окружения – трудоёмкая задача. Даже при наличии библиотеки верификационных компонентов при сборке окружения могут потребоваться дополнительные усилия на интеграцию разных компонент, требуется разработка и интеграция эталонной модели. Для подготовки первого варианта окружения в этом случае может потребоваться несколько недель.

В статье показано, что для реализации платформенного подхода в верификации необходим набор дополнительных правил создания верификационных компонент (дополняющих UVM), а также предложен метод, позволяющий повторно использовать код программной модели блоков в тестовых окружениях и стандартизовать способ интеграции таких моделей в тестовое окружение. За счет создания единообразной библиотеки компонент и шаблонного тестового окружения, разработка первого варианта окружения блока может быть выполнена за 1-2 дня.

Тестовое окружение блока, созданное по предложенному методу, включающее эталонную модель, компоненты сравнения и отладочной печати, может быть без изменений использовано на уровне модели СнК. Встроенная возможность подмены RTL модели программной моделью блока позволяет путём конфигурирования получать в зависимости от целей запуска тестов или готовности TLM или RTL моделей блоков либо полностью высокоуровневую модель системы (все блоки – TLM модели), либо полностью низкоуровневую модель (все блоки – RTL модели), либо смешанные модели (рис 3). В итоге сокращаются трудозатраты на разработку тестовых окружений и тестов, а также появляется возможность поиска баланса между точностью модели и скоростью моделирования для отладки тестов различных типов, начиная от тестов на проверку отдельных свойств блоков, заканчивая тяжеловесными прикладными программами. Использование программной модели в качестве эталона при верификации позволяет сделать программную модель достаточно точной уже на этапе разработки RTL.

Литература

[1] IEEE Standard for SystemVerilog — Unified Hardware Design, Specification, and Verification Language // IEEE 1800TM.

[2] Chris Spear, SystemVerilog for Verification: A Guide to Learning the Testbench Language Features: Springer. 2007. 301 c.

[3] Universal Verification Methodology (UVM) 1.0 EA Class Reference [Electronic resource] // http://www.uvmworld.org/ Date of access 10.01.2012.

[4] OSCI TLM-2.0 Language Reference Manual.

[5] Aynsley J. SystemVerilog Meets C++: Re-use of Existing C/C++ Models Just Got Easier // [Electronic resource] www.doulos.com Date of access 10.01.2012.