ISSN 1816-0301 (Print)
ISSN 2617-6963 (Online)

СТАТЬИ ПО МАТЕРИАЛАМ ВОСЬМОЙ МЕЖДУНАРОДНОЙ НАУЧНОЙ КОНФЕРЕНЦИИ «ТАНАЕВСКИЕ ЧТЕНИЯ»

ARTICLES ON THE MATERIALS OF EIGHTH INTERNATIONAL SCIENTIFIC CONFERENCE "TANAYEVSKY READING"

УДК 50.51.17

Поступила в редакцию 05.06.2018
Received 05.06.2018

Д. И. Черемисинов

Объединенный институт проблем информатики
Национальной академии наук Беларуси, Минск, Беларусь

ИСПОЛЬЗОВАНИЕ ЯЗЫКА ПРАЛУ ДЛЯ ВЕРИФИКАЦИИ ЦИФРОВЫХ УСТРОЙСТВ

Аннотация. Рассматривается задача создания испытательного стенда для функциональной верификации. В процессе верификации устанавливается сводимость (эквивалентность) спецификации устройства и модели уровня регистровых передач (register-transfer level, RTL) – логической сети, построенной в процессе синтеза. В универсальной методологии верификации (universal verification methodology, UVM), наиболее часто используемой в современном проектировании цифровых устройств для функциональной верификации, стратегией тестирования, определяющей способ построения тестового примера, является случайный выбор в пространстве входных воздействий (coverage-driven constrained-random transaction-level self-checking testbenches). Правила и рекомендации UVM содержат стандартизованную структуру испытательного стенда, которая ориентирована на разработку трансформационных устройств. В случае если моделью разрабатываемого устройства является алгоритм поведения, предлагается строить испытательный стенд как модель окружающей среды проектируемого устройства, представленную на языке ПРАЛУ. Модель среды разрабатываемого устройства позволяет избегать ситуаций, когда испытуемое устройство верифицируется с достаточным покрытием схемы тестами, но в неполном окружении. Для разработки испытательного стенда в среде симулятора языка описания аппаратуры модель окружающей среды на ПРАЛУ может быть автоматически преобразована в модель уровня транзакций.

Ключевые слова: верификация цифровых устройств, моделирование на уровне транзакций, испытательный стенд реактивных устройств, язык ПРАЛУ, барьерный механизм синхронизации

Для цитирования. Черемисинов, Д. И. Использование языка пралу для верификации цифровых устройств / Д. И. Черемисинов // Информатика. – 2018. – Т. 15, № 4. – С. 86–98.

D. I. Cheremisinov

The United Institute of Informatics Problems of the National Academy of Sciences of Belarus, Minsk, Belarus

PRALU LANGUAGE – THE TOOL FOR VERIFYING DIGITAL DEVICES

Abstract. The task of creating a testbench for functional verification is considered. This verification process establishes the reconvergence (equivalence) of the device specification and the register-transfer level (RTL) model – a logical network which was built in the synthesis process. In the UVM methodology, usually used in the modern design of digital devices for functional verification, a testing strategy, that determines the way in which a test case is constructed, is the random selection of space-driven constrained-random transaction-level self-checking testbenches. The rules and recommendations of UVM contain a standardized structure of the test bench, which is oriented towards the development of transformational devices. For the case where the model of the design is a behavior algorithm, it is proposed to build

a testbench as a model of the environment of the design presented in the language of PRALU. The environment model of the developed device allows to avoid situations when the device under test is verified with sufficient coverage, but in an incomplete environment. The environment model on PRALU can be automatically converted into a transaction level model to develop a testbench in the simulator environment of the hardware description language.

Keywords: hardware verification, transaction-level model, reactive system testbench, PRALU language, barrier synchronization method

For citation. Cheremisinov D. I. PRALU language – the tool for verifying digital devices. *Informatics*, 2018, vol. 15, no. 4, pp. 86–98 (in Russian).

Введение. Термин «испытательный стенд для верификации цифровых устройств» (testbench) обозначает программу для симулятора языка описания аппаратуры, которая используется для задания входной последовательности, подаваемой на испытуемое устройство, и, может быть, для наблюдения за его реакциями. Testbench разрабатывается на языке описания аппаратуры (специализированном языке), таком как SystemVerilog, или на обычном языке программирования, например C [1, 2]. В этом случае испытательный стенд является средством автоматизации процесса верификации.

Testbench представляет собой весьма специфичный тип программы, для разработки которой применяются специальные системы программирования. В настоящее время для создания испытательных стендов имеются методологии и программные средства, разработанные крупными компаниями [3]. Целью разработки testbench является решение задачи верификации определенного класса. Операция верификации связана с определенной операцией проектирования, они вместе создают модель процесса верификации, или модель сведения (от англ. reconvergence model) [3] (рис. 1). Модель сведения вместе с парными операциями определяет сводимые друг к другу модели проектируемого устройства. Далее будет рассматриваться задача создания испытательного стенда для функциональной верификации. В этом процессе верификации устанавливается сводимость (эквивалентность) спецификации устройства и модели уровня регистровых передач – логической сети, построенной в процессе синтеза. Важно отметить, что можно установить соответствие результата синтеза и спецификации, только если спецификация написана на формальном языке с точной семантикой. Процесс функциональной верификации представляет собой отладку RTL-модели.

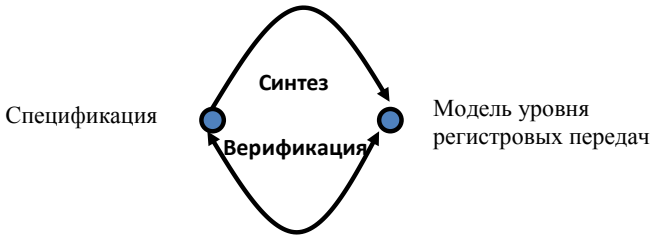


Рис. 1. Модель процесса функциональной верификации

Чтобы автоматизировать сравнение испытуемого устройства со спецификацией, они должны быть представлены в форме, которую можно выполнить на компьютере, используя некоторую программу. Симулятор является самым нетрудоемким способом выполнения обеих моделей. Отладка на симуляторе по методике применения полностью повторяет процесс отладки в программировании. Отладка RTL-моделей состоит в устранении ошибок, факт существования которых уже установлен [4]. Обнаружение ошибки устанавливается в ходе специально спроектированного эксперимента. Автоматическое выполнение такого эксперимента над моделью проектируемого устройства и обеспечивает testbench. Этот эксперимент называется тестом верификации. Дальнейшим этапом проектирования будет создание набора тестов, запускаемых на модели в автоматическом режиме. Другие технологии верификации, такие как статический анализ, проверка на модели и доказательства, обладают огромным потенциалом, но ни одна из них не является столь совершенной, чтобы заменить тесты как доминирующую технологию [4]. Цель проектирования на уровне RTL является продуктом тестирования и отладки в процессе верификации.

Testbench фактически представляет собой модель внешней среды проектируемого цифрового устройства. На рис. 2 показано, как испытательный стенд взаимодействует с верифицируемым устройством (design under verification, DUV). Вместе с верифицируемым устройством testbench образует полностью закрытую систему, не имеющую входов или выходов. Назначение теста верификации – выявление ошибки путем вызова сбоя. Сбоем является несовпадение ожидаемого результата для правильно работающего устройства с тем, который получен симулятором при подаче входных последовательностей, допустимых для разрабатываемого устройства. Тестирование, в отличие от отладки, не касается исправления ошибок, а служит только для их поиска.

Тестирование и отладка программ и схем в настоящее время являются одной из отраслей информатики (software engineering). Как научное знание эта отрасль использует набор терминов и понятий – онтологию тестирования и отладки. Термины и понятия англоязычной онтологии тестирования и отладки зафиксированы как профессиональные стандарты. Русскоязычная онтология тестирования и отладки еще не разработана. В настоящей статье используются переводы (может быть, не самые удачные) терминов и понятий англоязычной онтологии тестирования и отладки из работы [5]. Термины «ошибка», «отказ», «устранение отказа» понимаются в смысле работы [5].

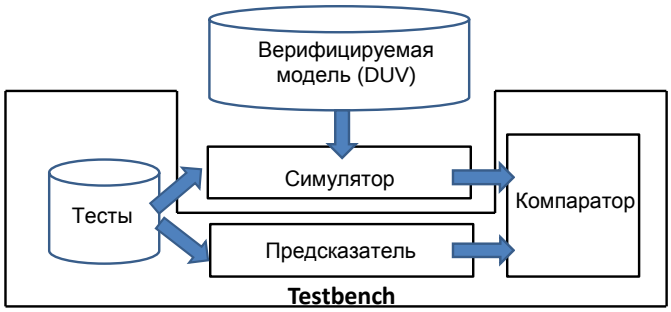


Рис. 2. Испытательный стенд на основе симулятора языка модели проектируемого устройства

Характерной особенностью неавтоматизированного проектирования является склонность исправленных в ходе отладки ошибок к реанимации. Проверка того, что после устранения очередной ошибки предыдущие исправления все еще работают, называется регрессионным тестированием. Каждый обнаруженный сбой дает тестовый пример, включаемый в набор тестов проекта для регрессионного тестирования. Успешное выполнение набора тестов косвенно свидетельствует о возможной эквивалентности моделей устройства, используемых в процессе верификации. Успешный тест может служить таким свидетельством только в том случае, если он ранее обнаруживал ошибку (тест не проходил). Прохождение теста свидетельствует об устранении отказа и исправлении ошибки.

В настоящее время имеются методология и программные средства для разработки testbench, ориентированные на случайные тесты с учетом ограничений для достижения покрытия путей распространения сигналов (coverage-driven constrained-random self-checking testbenches) [3]. Ключевой характеристикой этой методологии являются временные затраты на обнаружение ошибок. Характеристики окружающей среды учитываются косвенно через ограничения на пространство входных воздействий. Предполагается, что эти ограничения могут быть заданы явно в компактной форме. Например, если пространство входных воздействий – это пространство булевых векторов, то тестовый пример представляет собой определенный вектор, а ограничения могут быть заданы в виде булевой функции [3].

Существует большой класс устройств, условия использования которых формулируются с такой же или даже большей сложностью, чем функция устройства. В этом случае для построения testbench необходимо использовать модель внешней среды. Предлагается в качестве такой модели применять алгоритм управления на языке ПРАЛУ [6].

Методологии функциональной верификации. Первоначально методология разработки testbench рассматривалась как проблема создания компьютерной модели реального испытательного оборудования. Программы или аппаратура, создающие впечатление действительности и отображающие часть реальных явлений и свойств в виртуальной среде, называются симуляторами. Симуляторы имеют неустранимый недостаток – скорость моделирования. Скорость моделирования оборудования, в котором сигналы передаются со скоростью света и миллионы транзисторов переключаются более одного миллиарда раз в секунду, на компьютере общего назначения, способном выполнить порядка миллиарда последовательных инструкций в секунду, ниже скорости физического выполнения оборудования на много порядков. Один из способов оптимизировать производительность симулятора – это не моделировать то, что не нужно имитировать.

Современная методология разработки testbench появилась около 20 лет назад. В компании Verisity предложили рассматривать эту разработку (рис. 3) как проблему программирования. Был создан специальный язык программирования, называемый «e», и связанная с ним методология повторного использования – «e RM» [7]. Сейчас стандартная методология разработки testbench называется UVM [8]. Она представляет собой библиотеку языка программирования SystemVerilog с открытым исходным кодом, позволяющую создавать гибкие компоненты испытательного стенда.

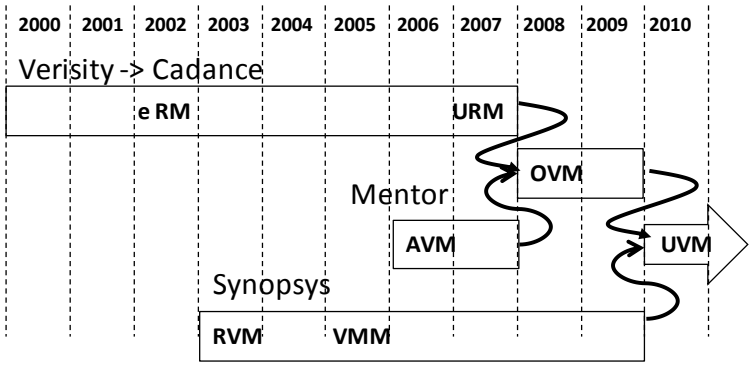


Рис. 3. История разработки методологий верификации

Основная цель UVM – сделать программу testbench более переносимой и создать рынок программных компонентов VIP (от англ. verification intellectual property, верификационные IP-ядра) для testbench. IP-ядра (IP cores), или IP-блоки, – это готовые блоки для проектирования микросхем. IP-блок представляет собой многократно используемый блок интегральной схемы, который является интеллектуальной собственностью некоторого лица.

Первой библиотекой верификации на основе SystemVerilog была OVM (open verification methodology, открытая методология верификации). Она состояла из объектов и процедур для генерации тестов, сбора данных и управления процессом верификации. Методология верификации VMM (verification methodology manual) оказалась успешной и широко применялась при создании испытательных стендов для верификации современных СБИС. VMM была разработана в фирме Synopsys, в методиках VMM предлагалось использовать такие методы программирования, как объектно-ориентированный подход, рандомизация и ограничения при генерации тестов, покрытие схемы тестами, которые позволяют повысить эффективность testbench.

Благодаря современным методам программирования возросла эффективность программирования вспомогательных компонентов testbench на основе общих шаблонов проектирования программ. Одним из достижений современной методологии разработки testbench является создание классов для подключения симулятора, обеспечивающего выполнение моделей верифицируемого устройства. Программирование testbench выполняется в специальной среде разработки и исполнения – Synopcys VCS. Эта среда позволяет разрабатывать и выполнять программы на смеси языков SystemVerilog, VHDL и SystemC. Компоненты на разных языках

скомпилированы в машинный код, а имитация моделей испытуемого устройства выполняется с использованием одного механизма моделирования.

Ключевой проблемой функциональной верификации является создание набора тестов (test suite), который достаточно ясно демонстрирует правильность совместной работы окружающей среды и проектируемого устройства. Создание теста – это искусство исследования проектируемого устройства, хотя по спецификации возможна и автоматическая генерация тестов. Недостаток данного способа состоит в том, что такие тесты не исследуют двусмысленности или пробелы спецификации. Стратегия тестирования с использованием знаний о проектируемом устройстве является проблемно-ориентированой (directed tests approach) [9].

Специфической для предлагаемого устройства задачей верификации является разработка генератора тестов, обеспечивающего покрытие схемы тестами. Покрытие тестами служит мерой, используемой для описания степени, в которой заданный набор тестов охватывает элементы схемы. Схема с высоким охватом тестированием имеет больше областей исходного кода, выполняющихся во время тестирования. Мера покрытия кода заимствована из методологии тестирования программ. Охват тестированием характеризует вероятность наличия необнаруженных ошибок проектирования программы. При высоком охвате тестированием такая вероятность меньше по сравнению с низким охватом. Для расчета покрытия тестами кода программы используются метрики программ [9]. Покрытие кода измеряет степень тестирования описания схемы в процентах. Информация о покрытии кода представляется в виде отчета о покрытии строк исходного кода при выполнении заданного набора тестов. Формирование этого отчета при верификации схем автоматизировано. Покрытие кода можно рассматривать как количественную меру продвижения процесса создания описания устройства.

В методологии UVM стратегией тестирования, определяющей способ построения тестового примера, является случайный выбор в пространстве входных воздействий (coverage-driven constrained-random transaction-level self-checking testbenches). Интуиция подсказывает, что любая стратегия тестирования, использующая знания о программе, должна быть лучше случайного выбора. Однако сравнение стратегий на основе «скорости» обнаружения отказов показывает, что случайное тестирование часто превосходит проблемно-ориентированную стратегию [9]. В современных симуляционных средах, основанных на UVM, используется настраиваемая генерация псевдослучайных тестов (constrained random) для автоматического исследования поведения проектируемого устройства. Полнота исследования контролируется на основе различных показателей покрытия тестами (coverage metric). На рис. 4 показана типичная среда UVM [10].

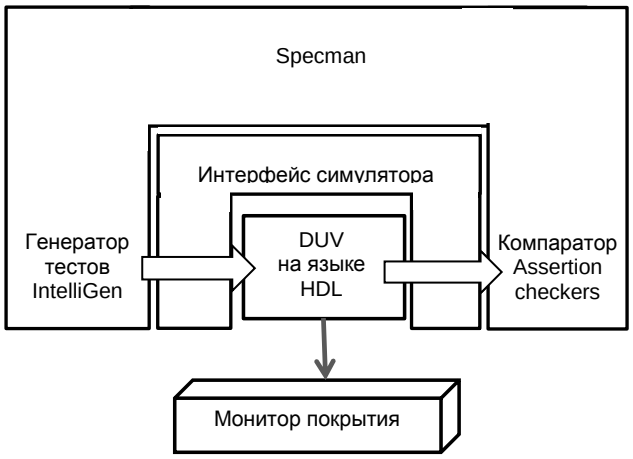


Рис. 4. Среда для выполнения компиляции и отладки testbench от компании Cadence Design Systems

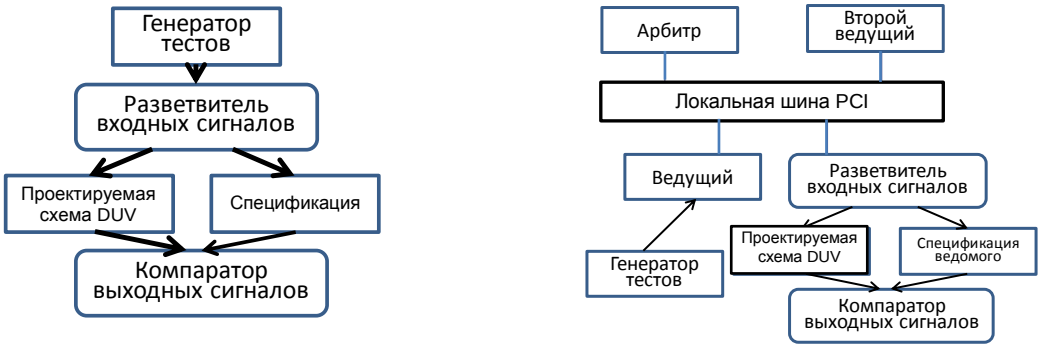
Степень покрытия была принята в качестве критерия полноты исследования тестами из-за низкой стоимости ее вычисления. Вычисление данной метрики обеспечивается автоматически симулятором и не требует много дополнительных действий от пользователя. Обычно это задание дополнительной опции компиляции программы testbench. Стопроцентное покрытие не

является свидетельством такой же правильности или полноты верификации. Разумеется, любой метод, который выявляет правильность или полноту верификации, представляет большой интерес для всех заинтересованных сторон – от менеджеров до разработчиков и клиентов.

Трансформационные и реактивные устройства. В самой общей классификации цифровые устройства делятся на последовательностные (с памятью) и комбинационные (без памяти, stateless). В работе [6] было предложено характеризовать цифровые устройства по типу алгоритмического описания. Устройства, моделью которых являются классические алгоритмы (алгоритмы планирования), относятся к трансформационному типу. Цель трансформационной системы – вычисление некоторого результата по заданным исходным данным посредством конечной последовательности шагов. С функциональной точки зрения устройства трансформационного типа – это устройства без памяти, так как результат работы алгоритма зависит только от известных в момент начала работы входных данных. Примерами таких устройств (систем) являются процессоры, компиляторы языков программирования, веб-серверы. В системах stateless элементы памяти могут иметься, но их состояние не сохраняется между операциями.

Цель реактивной системы состоит не в том, чтобы получать некий результат, а в том, чтобы осуществлять взаимодействие с окружающей средой. В англоязычной литературе для систем с трансформационным поведением используется слово parallel, для реактивных систем – слово concurrent. В книге [6] алгоритмическое описание реактивной системы называется алгоритмом управления. Функционирование реактивных систем в идеале никогда не заканчивается. Отсюда следует, что алгоритм реактивной системы не является алгоритмом в смысле классической теории алгоритмов (отсутствует признак конечного числа шагов). Примерами таких устройств являются контроллеры периферийных устройств компьютера, подключаемые к общей шине, встроенные системы, устройства управления оборудованием. Следует отметить, что в последнее время термин «реактивная система» употребляется часто и обозначает [11] программные системы, в которых асинхронно обрабатываются потоки данных с непредопределенным объемом. Тем не менее для формализации этих алгоритмов можно использовать тот же подход, что и для трансформационных систем, – описание путем задания формального языка и абстрактного механизма вычислений. Далее в качестве признака реактивной системы будет применяться именно способ задания требуемого поведения – алгоритм поведения.

Большая часть начального цикла верификации (отладки) цифровых устройств проводится в среде симулятора языка описания аппаратуры (см. рис. 2), так как на ранних стадиях проектирования схема устройства подвержена частым и крупным изменениям. В этих условиях испытательный стенд содержит кроме схемы проектируемого устройства и его образцовое описание (спецификацию), которое вычисляет ожидаемые ответы (предсказатель на рис. 2), сопоставляемые с результатами моделирования проектируемой схемы. Если сравнение неудачное, схема модифицируется, чтобы привести результат ее моделирования в соответствие с формальной спецификацией. Специфической частью такого испытательного стенда является генератор тестовых последовательностей (рис. 5, *а*).



а) *б)*

Рис. 5. Испытательные стенды: *а)* трансформационного устройства; *б)* ведомого устройства шины PCI

Испытательный стенд реактивных устройств. Одной из целей разработчиков инструментов проверки является снижение сложности программы испытательного стенда. Известны правила и рекомендации создания стандартизованной структуры испытательного стенда [12], которые ориентированы на разработку трансформационных устройств. Например, в методических материалах технологии верификации VMM [12] типовым примером использования технологии является разработка сумматора с плавающей запятой Single Precision. Рекомендации сводятся к использованию методов объектного программирования для создания генераторов тестовых последовательностей, табло результатов сравнения и среды интеграции компонентов испытательного стенда.

В качестве примера проблем, возникающих при верификации реактивных цифровых устройств, рассмотрим схему [13], осуществляющую индикацию принятого по шине PCI слова. Шина PCI (peripheral component interconnect, взаимосвязь периферийных компонентов) используется для организации взаимодействия устройств, составляющих компьютер (процессора, памяти, контроллеров жестких дисков и др.). Проектируемое устройство всегда является ведомым и выполняет единственную команду записи (в это устройство) одного слова. Для того чтобы выполнить передачу данных по шине, необходимы два устройства (ведущее и ведомое), поведение которых нужно скоординировать, причем ведущим может быть любое устройство, совместимое со стандартом PCI и способное выполнять эту роль. Закрытая система, являющаяся моделью испытательного стенда для верификации такого устройства [14], изображена на рис. 5, *б*.

На рис. 5 видна разница в сложности окружающей среды проектируемых устройств. Рассматриваемая закрытая система состоит из автономных частей (агентов взаимодействия), которые работают вместе [15]. Совместное поведение агентов описывается моделью, которая называется протоколом шины PCI. Именно этот протокол служит основой структуры и функций частей испытательного стенда на рис. 5, *б*. В стандарте PCI поведение демонстрируется неформально с помощью временных диаграмм. Формальные модели демонстрируют поведение автономных частей закрытой системы. Для встроенных систем работу по декомпозиции протокола на модели отдельных агентов нужно выполнять при проектировании.

Моделирование на уровне транзакций. В методологии UVM предполагается, что спецификации проектирования имеют уровень транзакций (уровень системы). Чтобы повысить производительность проектирования на этом более высоком уровне, потребовалась разработка методов моделирования и синтеза. Модели уровня транзакций (transaction-level model, TLM) [16] позволяют решить проблемы проектирования, интеграции и верификации для современных сложных цифровых устройств. TLM дает возможность, используя модели цифровых устройств на более высоком уровне абстракции, обеспечить быстрое моделирование, упрощая процесс комплексной отладки, а также исследовать несколько альтернативных проектных решений, так как при этом применяются менее объемные модели.

Модели уровня транзакций представляют цифровое устройство в виде компонентов с поведением, заданным как набор параллельных взаимодействующих процессов. Взаимодействие в этих моделях представляется как коммуникация, причем актом коммуникации служит «транзакция» через абстрактный канал. Моделирование на уровне транзакций в первую очередь является средством повышения эффективности, оно до 1000 раз быстрее, чем моделирование на уровне RTL [16] (рис. 6). Ускорение достигается за счет абстрагирования деталей взаимодействия, что неизбежно приводит к потере точности моделирования.

В проектировании на основе UVM предполагается, что VIP-ядра для создания testbench имеются для каждого используемого в устройстве IP-ядра. Это позволяет быстро перейти от набора алгоритмов к набору проверенных блоков RTL, которые необходимо интегрировать. Фактически стандартом для создания моделей системного уровня служит язык SystemC. Среди различных языков для системного уровня он является наиболее популярным и широко используемым ведущими компаниями в этой области. Кроме того, SystemC 2.1 является стандартом IEEE 1666 [16].

Среда моделирования	**Тип модели**	**Относительная скорость моделирования**	**Длительность разработки модели**
	Функциональная	10 000*x* (1 мин)	20*x* (2 дня)
	Структурная	1000*x*	
	Транзакционная	100*x*	
	Параллельная	10*x*	2*x* (2 недели)
	RTL	1*x* (7 дней)	1*x* (2 месяца)

Рис. 6. Скорость моделирования на уровне транзакций

Язык SystemC является расширением языка C++. Это расширение представлено в виде свободно доступной библиотеки классов и макросов, реализованных на C++ и описывающих классы файлами заголовков, и библиотеки функций, которая содержит ядро моделирования. Модель на языке SystemC может быть скомпилирована любым ANSI-совместимым компилятором C++. Результирующая исполняемая программа реализует симулятор с интегрированными средствами управления имитацией.

Поскольку язык SystemC является расширением базового языка, все возможности C++ доступны при моделировании, но разработчик должен придерживаться стиля программирования языка SystemC. Набор типов данных в SystemC содержит необходимые элементы модели системного уровня, такие как биты, битовые векторы, векторы и скаляры четырехзначной логики (низкий, высокий, неопределенный и высокий импеданс), и типы данных для арифметики с фиксированной точкой произвольного размера. Параллелизм в SystemC имеет семантику чередования операций последовательных процессов. Процесс представляет собой понятие языка SystemC, отличающееся от понятия, используемого в теории операционных систем. Параллельные процессы языка SystemC – это потоки (thread), которые планируются для последовательного выполнения собственным планировщиком SystemC на основе невытесняющего (non-preemptive) мультипрограммирования (cooperative multitasking) [17] по принципу «мгновенного» обслуживания [18].

Потоки являются методом адаптации последовательной модели вычислений языка C ++ для параллельных систем. Для поддержки потоков не требуется сильных синтаксических изменений в языке программирования, а операционные системы и аппаратура компьютера обеспечивают их эффективную реализацию. В практике разработки программного обеспечения общего назначения этот подход к параллельному программированию доминирует над всеми другими. Хотя потоки кажутся небольшой модификацией последовательных вычислений, на самом деле они представляют собой огромный сдвиг в семантике. В многопоточной программе отсутствуют наиболее важные и привлекательные свойства последовательных вычислений: понятность, предсказуемость и детерминизм. Модель вычислений многопоточности обладает существенным недетерминизмом, и процессы языка SystemC имеют специальную организацию для устранения этого недетерминизма. Потоки – это последовательные процессы, которые имеют общую память. Процессы теории вычислений (составные действия [22]) называются на языке SystemC поведением (behavior). Таким образом, процессы SystemC – это синтаксическая конструкция, смыслом которой является поведение. Они имеют вид функций языка C++.

В RTL поведение элементов обычно синхронизируется по сигналу синхронизации. Сигналы RTL-модели изменяют свои значения по «тику часов» (генератора синхросигналов), что приводит к синхронизации поведения элементов. «Тик часов» – это условный интервал времени, за который выполняется одна простая команда. В TLM синхронизация происходит, когда между двумя блоками передается сообщение, т. е. выполняется транзакция, и отдельный сигнал синхронизации больше не нужен для моделирования.

Неблокирующая синхронизация. В SystemC различают два основных вида процессов: SC_THREAD и SC_METHOD. Процессы вида SC_THREAD можно приостановить, вызвав wait(.). Функция wait(.) вызывает переключение контекста процесса планировщиком SystemC [19]. В процессах вида SC_METHOD переключение контекста происходит при возврате из функции процесса. В целом testbench представляет собой многопоточную программу с неблокирующей синхронизацией, так как совместный доступ к данным потоки синхронизируют без использования блокирующих механизмов синхронизации.

Проще говоря, модель вычислений на уровне транзакций состоит из набора последовательных потоков управления, называемых процессами, которые взаимодействуют через общие структуры данных, называемые каналами. Каждый канальный объект имеет тип, определяющий набор возможных состояний и набор примитивных операций, которые предоставляют единственное средство для манипулирования этим объектом. Каждый процесс применяет последовательность операций к объектам, выдавая вызов и получая соответствующий ответ. Эти последовательности операций с канальным объектом называются транзакциями. Запросы транзакций выполняются путем вызова функций интерфейса каналов, которые в SystemC представлены библиотечными классами. Транзакции являются атомарными операциями.

Операция атомарна, если она выполняется целиком либо не выполняется вовсе, т. е. она не может быть частично выполнена и частично не выполнена. Атомарная операция выполняется только одним процессом. В параллельной системе процессы могут одновременно обращаться к общему объекту. Поскольку несколько процессов обращаются к одному объекту, может возникнуть ситуация, когда один процесс обращается к объекту, который изменяет другой процесс. Этот пример демонстрирует необходимость линеаризуемости. Линеаризуемость представляет собой свойство программы, в которой результат любого параллельного выполнения действий (операций) эквивалентен некоторому последовательному выполнению [20]. Для любого другого потока выполнение линеаризуемой операции является мгновенным: операция либо не начата, либо завершена. Линеаризуемость представляет собой сильное условие правильности, которое обеспечивает детерминированность результата при одновременном доступе к объекту несколькими процессами. Это свойство безопасности, которое гарантирует, что атомарные операции не будут завершены неожиданным или непредсказуемым образом.

Синхронизация процессов модели на уровне транзакций осуществляется барьерным механизмом [21]. Барьер – это точки исходного кода в группе процессов, в которых каждый процесс должен приостановиться и подождать достижения барьера процессами группы. В SystemC точки барьера задаются функцией wait (.). До достижения барьера изменений в общих структурах данных не происходит. После достижения барьера запланированные изменения в этих структурах происходят мгновенно.

Модели уровня транзакций могут использоваться для упрощения интеграции и тестирования, но в настоящее время такие модели не существуют. Попытки вручную создать TLM в SystemC, описав оборудование на ANSI C++, подвержены ошибкам и требуют много времени. Проектирование многопоточной программы с неблокирующей синхронизацией трудоемко потому, что неблокирующая синхронизация – это используемое для описания программы свойство, которое не связано с ее реализацией и которое трудно доказать. Критерий «неблокирующая синхронизация» состоит в том, что в бесконечном исполнении многопоточной программы вызов каждого метода бесконечно часто заканчивается [20]. С практической точки зрения невыполнение этого свойства означает ошибочную реализацию.

Моделирование встроенных систем с использованием языка ПРАЛУ. Алгоритмы на ПРАЛУ конструируются из операций ожидания и действия, содержательный смысл которых может быть довольно произвольным. Формально в описании языка ПРАЛУ [6] функции таких операций не определены, вместо этого регламентируются условия завершения ожиданий и действий. С завершением операции связывается наступление некоторого события в пространстве переменных алгоритма, причем для операции ожидания событие в пространстве переменных служит причиной ее завершения, а для операции действия, наоборот, завершение операции вызывает определенное событие в пространстве переменных.

Все переменные алгоритма на ПРАЛУ являются булевыми. События, связанные с завершением операций, должны быть представимы конъюнкциями переменных алгоритма. Эти конъюнкции составляют основу выражения языка ПРАЛУ, задающего операцию. Для описания порядка выполнения операций в алгоритмах на ПРАЛУ используется язык сетей Петри. Алгоритм на ПРАЛУ является совокупностью описаний переходов сети Петри. Описание перехода называется цепочкой и включает перечень меток позиций, из которых запускается переход, перечень операций ожидания и действия, задающих его интерпретацию, и перечень меток позиций, в которые происходит переход.

Алгоритм на ПРАЛУ описывает замкнутую систему, если все события в пространстве переменных вызываются реализацией операций действия алгоритма. В случае незамкнутых систем в множестве переменных алгоритма можно выделить подмножество, для которого значения переменных задаются внешней средой. Транзакции в алгоритмах на ПРАЛУ представлены операциями ожидания и действия, имеющими общую переменную и описывающими событие взаимодействия [22].

В модели уровня транзакций цепочки ПРАЛУ интерпретируются как процессы, что требует уточнения семантики операций ожидания и действия, так как в этом случае они оказываются неэлементарными. Суть уточнения состоит в организации вычислений таким образом, чтобы линейный порядок реализации операций алгоритма являлся доопределением частичного порядка, задаваемого исходным параллельным алгоритмом. При этом параллелизм понимается как возможность упорядочивать операции произвольным образом. В такой интерпретации алгоритмы на ПРАЛУ обладают свойством линеаризуемости, т. е. результат параллельного выполнения операций ПРАЛУ эквивалентен некоторому последовательному выполнению. Доопределяя операции ожидания и действия в модели уровня транзакций, опишем их в виде композиций более простых операций, выполняемых строго последовательно. Набор операций, образующих композиции, назовем базисом алгоритмического разложения параллельных алгоритмов.

В предлагаемом базисе барьерный механизм синхронизации составляют операции образования, приостановки и прекращения реализации ветви. Структура данных барьера синхронизации представлена в памяти очередью готовых ветвей (ОГ) и очередью ждущих ветвей (ОЖ). Ветвью является совокупность последовательных подпроцессов, начинающихся с некоторой заданной операции. Последовательным подпроцессом обычно называют максимальную цепь операций процесса, находящихся в отношении непосредственного следования. Операция образования ветви заключается в занесении первой операции ветви в ОГ. При приостановке ветвь, начинающаяся с операции, которая выполняется в текущей ветви следующей, заносится в ОЖ, затем из ОГ извлекается другая ветвь и выполняется ее первая операция. Смысл операции прекращения ветви ясен из ее названия: ветвь удаляется из ОГ. Алгоритм останавливает работу операцией прекращения реализации. Дисциплина обслуживания очередей может быть любой. Она оказывает влияние на упорядоченность событий, взаимосвязь которых алгоритмом на ПРАЛУ не регламентируется (они параллельны).

Структурой данных в алгоритмах на ПРАЛУ является вектор переменных алгоритма. Чтобы до достижения барьера не происходило изменений значений переменных, компонентами этого вектора выступают пары, представляющие текущее значение переменной и ее планируемое значение. Доступ к компонентам вектора переменных осуществляется операциями установки значений переменных алгоритма, задающими планируемые значения, и операцией проверки условных переменных, читающей текущее значение.

Барьер в алгоритмах на ПРАЛУ состоит из операций приостановки. При выполнении процессом операции простановки этот процесс ожидает достижения барьера другими процессами. Барьер достигнут, когда ОГ пуста. При достижении барьера запускается единственный процесс, задачами которого являются пересылка элементов из ОЖ в ОГ (ОЖ становится пустой), ввод очередных значений в планируемые значения, если система незамкнута, и пересылка планируемых значений в текущие для каждой компоненты вектора переменных. Этот процесс заканчивается запуском первой операции из ОГ.

Опуская детали, получим, что реализация операции ожидания состоит из последовательного выполнения операций приостановки и проверки переменных, операция действия – из операций установки переменных. Операции образования ветвей выполняются при обработке пе-

речня позиций перехода. Каждая из операций базиса может быть реализована в современных микропроцессорах одной командой, в том числе и приостановка – аналог функции wait(.) в SystemC. Отдельный планировщик не требуется. Возможна трансляция алгоритмов на ПРАЛУ на язык С (и С++), хотя в большинстве современных языков отсутствуют конструкции, аналогичные машинной операции «возврат после перехода с возвратом».

В языках описания системного уровня связь между компонентами реализуется через каналы. Неправильное использование типа канала или установка неправильного размера буфера в канале могут привести к ситуации взаимоблокировки. Для автоматического обнаружения взаимоблокировок в проекте на SpecC нужно выполнять статический анализ модели, извлекать временные отношения событий, а затем анализировать эти соотношения. Такие задачи в настоящее время не имеют удовлетворительного решения. Модели, первоначально представленные на ПРАЛУ, после автоматической проверки корректности и трансляции гарантированно свободны от недетерминизма поведения и взаимоблокировок.

Главное достоинство модели уровня транзакций на языке ПРАЛУ состоит в возможности автоматической проверки корректности модели. Для языка ПРАЛУ разработана теория корректности алгоритмов, содержащая ряд методов формальной верификации. Наиболее существенными свойствами «правильных» алгоритмов, наличие которых можно проверить формально, являются безызбыточность, восстанавливаемость, непротиворечивость, устойчивость и самосогласованность [5].

Заключение. Для описания поведения встроенных устройств требуются предположения об источнике входных воздействий, потому что для их работы критически важной является последовательность входных состояний, зависящая от среды, в которой работает устройство. Использование ПРАЛУ дает возможность описать временную упорядоченность событий, возникающих при работе системы целиком (встроенной системы и ее окружения), абстрагируясь от всех деталей, кроме тех, которые выражаются причинно-следственными и временными отношениями. Язык ПРАЛУ дает возможность описать поведение иерархически, отражая структуру частей системы и организацию их взаимодействия. Модель окружения встроенной системы в этом случае служит основой логической структуры испытательного стенда. Для разработки testbench в среде симулятора языка описания аппаратуры модель окружающей среды на ПРАЛУ может быть автоматически преобразована в модель на соответствующем языке. В настоящее время существуют синтезаторы языка ПРАЛУ в модели аппаратуры на языках Verilog и С [22].

Работа выполнена при финансовой поддержке БРФФИ (проект Ф17АРМ-008).

Список использованных источников

1. Bergeron, J. Writing Testbenches using SystemVerilog / J. Bergeron. – N. Y. : Springer Science + Business Media, 2006. – 411 p.

2. Бибило, П. Н. Моделирование и верификация цифровых систем на языке VHDL / П. Н. Бибило, Н. А. Авдеев. – М. : URSS, 2017. – 342 с.

3. An Accellera Organization. Universal Verification Methodology (UVM) 1.1 Class Reference [Electronic resource]. – 2011. – Mode of access: http://www.accellera.org/images/downloads/standards/uvm/UVM_1.1_Class_Reference_Final_06062011.pdf. – Date of access: 10.02.2018.

4. Meyer, B. Seven principles of software testing / B. Meyer // Computer. – 2008. – Vol. 41, no. 8. – P. 99–101.

5. Glasser, M. Open Verification Methodology Cookbook / M. Glasser. – N. Y. : Springer, 2010. – 236 p.

6. Закревский, А. Д. Параллельные алгоритмы логического управления / А. Д. Закревский. – Минск : Ин-т техн. кибернетики НАН Беларуси, 1999. – 202 с.

7. Anderson, T. L. Verifying SoCs from the Inside Out / T. L. Anderson [Electronic resource]. – Mode of access: chipdesignmag.com/display.php?articleId=5153. – Date of access: 10.02.2018.

8. Open source VHDL verification methodology. User's Guide Rev. 1.2 [Electronic resource] / ed. J. Lewis. – Mode of access: http://osvvm.org/downloads. – Date of access: 02.09.2013.

9. Kaner, C. What is a good test case? / C. Kaner // Software Testing Analysis & Review Conference (STAR) East [Electronic resource]. – 2003. – Mode of access: http://kaner.com/pdfs/GoodTest.pdf. – Date of access: 10.02.2018.

10. Naveh, R. Cadence and Specman / R. Naveh // ACP summer school [Electronic resource]. – 2011. – Mode of access: http://www.gecode.org/events/acp-summerschool-2011/slides/Cadence%20CSP.pdf. – Date of access: 10.02.2018.

11. Halbwachs, N. Synchronous Programming of Reactive Systems / N. Halbwachs. – Springer-Verlag, 2010. – 192 p.

12. The development of advanced verification environments using System Verilog / M. Keaveney [et al.] // IET Irish Signals and Systems Conf., 2008. – Galway, 2008. – P. 303–308.

13. Черемисинов, Д. И. Моделирование локальной шины PCI с использованием языка ПРАЛУ / Д. И. Черемисинов // Танаевские чтения : докл. Четвертой Междунар. конф., 29–30 марта 2010 г., Минск. – Минск : ОИПИ НАН Беларуси, 2010. – С. 124–128.

14. Wang, D. Formal Verification of the PCI Local Bus: A Step Towards IP Core Based System-On-Chip Design Verification / D. Wang [Electronic resource]. – Mode of access: https://www.cs.cmu.edu/~dongw/master-thesis.pdf. – Date of access: 10.02.2018.

15. Cheremisinov, D. I. The specification of agent interaction in multi-agent systems / D. I. Cheremisinov // Intelligent Information Management. – 2009. – No. 1. – P. 65–72.

16. Grotker, T. System Design with SystemC / T. Grotker. – Kluwer Academic Publishers, 2002. – 219 p.

17. Liu, C. L. Scheduling algorithms for multiprogramming in a hard real time environment / C. L. Liu, J. W. Layland // J. of the ACM. –1973. – Vol. 20, no. 1. – P. 44–61.

18. Закревский, А. Д. Система программирования ЛЯПАС-М / А. Д. Закревский, Н. Р. Торопов. – Минск : Наука и техника, 1978. – 220 с.

19. Muller, W. An ASM Based SystemC Simulation Semantics / W. Muller, J. Ruf, W. Rosenstiel // SystemC – Methodologies and Applications. – Kluwer Academic Publishers, 2003. – P. 97–126.

20. Herlihy, M. P. Linearizability: A correctness condition for concurrent objects / M. P. Herlihy, J. M. Wing // ACM Trans. Program. Lang. Syst. – 1990. – Vol. 12, no. 3. – P. 463–492.

21. Solihin, Y. Fundamentals of Parallel Multicore Architecture / Y. Solihin. – CRC Press, 2015. – 494 p.

22. Черемисинов, Д. И. Проектирование и анализ параллелизма в процессах и программах / Д. И. Черемисинов. – Минск : Беларус. навука, 2011. – 300 с.

References

1. Bergeron J. *Writing Testbenches using SystemVerilog*. New York, Springer, Science + Business Media, 2006, 411 p.

2. Bibilo P. N. Modelirovanie i verifikacija cifrovyh sistem na jazyke VHDL. *Modeling and Verification of Digital Systems in the VHDL Language.* Moscow, URSS Publ., 2017, 342 p. (in Russian).

3. *An Accellera Organization. Universal Verification Methodology (UVM) 1.1 Class Reference*. Available at: http://www.accellera.org/images/downloads/standards/uvm/UVM_1.1_Class_Reference_Final_06062011.pdf. (accessed 10.02.2018).

4. Meyer B. Seven principles of software testing. *Computer*, 2008, vol. 41, no. 8, pp. 99–101.

5. Glasser M. *Open Verification Methodology Cookbook*. New York, Springer, 2010, 236 p.

6. Zakrevskij A. D. Parallel'nye algoritmy logicheskogo upravlenija. *Parallel Logic Control Algorithms*, Minsk, Institut tehnicheskoj kibernetiki Nacional'noj akademii nauk Belarusi, 1999, 202 p. (in Russian).

7. Anderson T. L. *Verifying SoCs from the Inside Out*. Available at: chipdesignmag.com/display.php?articleId=5153 (accessed 10.02.2018).

8. Lewis J. (ed.) *Open source VHDL verification methodology. User's Guide Rev. 1.2*. Available at: http://osvvm.org/downloads (accessed 02.09.2013).

9. Kaner C. What is a good test case? *Software Testing Analysis & Review Conference (STAR) East.* Available at: http://kaner.com/pdfs/GoodTest.pdf (accessed 10.02.2018).

10. Naveh R. Cadence and Specman. *ACP Summer School*. Available at: http://www.gecode.org/events/acp-summerschool-2011/slides/Cadence%20CSP.pdf (accessed 10.02.2018).

11. Halbwachs N. *Synchronous Programming of Reactive Systems*. Springer-Verlag, 2010, 192 p.

12. Keaveney M., McMahon A., O'Keeffe N., Keane K., O'Reilly J. The development of advanced verification environments using System Verilog. *IET Irish Signals and Systems Conference, 2008*. Galway, 2008, pp. 303–308.

13. Cheremisinov D. I. Modelirovanie lokal'noj shiny PCI c ispol'zovaniem jazyka PRALU [A local PCI bus modeling by the PRALU language]. Tanaevskie chtenija: doklady Chetvertoj Mezhdunarodnoj konferencii [*Tanaev Readings: Reports of the Fourth International Conference, 29–30 Mar. 2010, Minsk*]. Minsk, The United Institute of Informatics Problems of the National Academy of Sciences of Belarus, 2010, pp. 124–128 (in Russian).

14. Wang D. *Formal Verification of the PCI Local Bus: A Step Towards IP Core Based System-On-Chip Design Verification.* Available at: https://www.cs.cmu.edu/~dongw/master-thesis.pdf (accessed 10.02.2018).

15. Cheremisinov D. I. The specification of agent interaction in multi-agent systems. *Intelligent Information Management*, 2009, no. 1, pp. 65–72.

16. Grotker T. *System Design with SystemC*. Kluwer Academic Publishers, 2002, 219 p.

17. Liu C. L., Layland J. W. Scheduling algorithms for multiprogramming in a hard real time environment. *Journal of the ACM*, 1973, vol. 20, no. 1, pp. 44–61.

18. Zakrevskij A. D., Toropov N. R. Sistema programmirovanija LJaPAS-M. *Programming System LYaPAS-M*. Minsk, Nauka i tehnika Publ., 1978, 220 p. (in Russian).

19. Muller W., Ruf J., Rosenstiel W. An ASM Based SystemC Simulation Semantics. *SystemC – Methodologies and Applications.* Kluwer Academic Publishers, 2003, pp. 97–126.

20. Herlihy M. P., Wing J. M. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 1990, vol. 12, no. 3, pp. 463–492.

21. Solihin Y. *Fundamentals of Parallel Multicore Architecture*. CRC Press, 2015, 494 p.

22. Cheremisinov D. I. Proektirovanie i analiz parallelizma v processah i programmah. *Design and the Analysis of Parallelism in Processes and Programs.* Minsk, Belaruskaja Navuka Publ., 2011, 300 p. (in Russian).

Информация об авторе

Черемисинов Дмитрий Иванович – кандидат технических наук, ведущий научный сотрудник, Объединенный институт проблем информатики Национальной академии наук Беларуси (ул. Сурганова, 6, 220012, Минск, Республика Беларусь).

E-mail: cher@newman.bas-net.by

Information about the author

Dmitriy I. Cheremisinov – Cand. Sci. (Eng.), Leading Researcher, The United Institute of Informatics Problems of the National Academy of Sciences of Belarus, (6, Surganova Str., 220012, Minsk, Republic of Belarus).

E-mail: cher@newman.bas-net.by