ПРОГРАММИРОВАНИЕ, 2014, № 1, с. 3–14

УДК 681.3.06

РАСШИРЯЕМАЯ СРЕДА ГЕНЕРАЦИИ ТЕСТОВЫХ ПРОГРАММ ДЛЯ МИКРОПРОЦЕССОРОВ *

© 2014 г. А.С. Камкин, Т.И. Сергеева, С.А. Смолов, А.Д. Татарников, М.М. Чупилко

Институт системного программирования РАН
109004 Москва, ул. А.Солженицына, 25
E-mail: {kamkin,leonsia,ssedai,andrewt,chupilko}@ispras.ru

Поступила в редакцию 10.08.2013

Создание тестовых программ и анализ результатов их выполнения – основной подход к функциональной верификации микропроцессоров на системном уровне. Имеется множество методов автоматизации разработки тестовых программ, начиная от генерации случайного кода и заканчивая нацеленным построением тестов на основе моделей, однако панацеи не существует: на практике применяются комбинации различных техник, дополняющих друг друга. К сожалению, в настоящее время нет решения, позволяющего интегрировать имеющиеся методы генерации тестовых программ в единую среду. Для верификации микропроцессоров инженеры вынуждены использовать большое число разнообразных генераторов тестов, что приводит к ряду трудностей: (1) необходимость обеспечения согласованности конфигураций инструментов (в каждом из них задействуется свое описание целевого микропроцессора, в результате чего часть информации дублируется); (2) необходимость разработки вспомогательных утилит для интеграции инструментов друг с другом (разные средства имеют разные интерфейсы и используют разные форматы данных). В статье описывается концепция расширяемой среды генерации тестовых программ для микропроцессоров: среда предоставляет единую методологию создания генераторов тестовых программ, поддерживает распространенные методы генерации тестов и допускает расширение новыми средствами тестирования. Предложенная концепция была частично реализована в системе MicroTESK (Microprocessor TEsting and Specification Kit).

1. ВВЕДЕНИЕ

Для обеспечения *корректности* и *надежности* функционирования микропроцессоров применяется целый комплекс мер. Такие меры известны как *верификация* и *тестирование* [1]. Верификация осуществляется на стадии разработки и предназначена для обнаружения логических ошибок в проектной модели микропроцессора. Тестирование проводится на стадии производства и нацелено на выявление физических дефектов в интегральных схемах. Для решения указанных задач применяются *тестовые программы* – специальные цепочки инструкций микропроцессора, приводящие к возникновению разнообразных ситуаций в его работе (внутренние события, взаимодействия между компонентами и другие) [2]. В дальнейшем для удобства мы будем использовать термин “тестирование” для обозначения как верификации, так и тестирования.

К настоящему моменту был предложен ряд методов автоматизированной генерации тестовых программ. Все эти методы можно разбить на следующие виды: (1) *случайная генерация* (построение псевдослучайных программ) [3]; (2) *комбинаторная генерация* (систематический перебор программ ограниченной длины) [2]; (3) *генерация на основе шаблонов* (построение программ по высокоуровневому описанию те-

*Работа частично поддержана Министерством Образования и Науки РФ (соглашение №8232 от 06/08/2012).

стовых сценариев) [4]; (4) *генерация на основе моделей* (построение программ, покрывающих определенный класс ситуаций в модели микропроцессора) [5]. Понятно, что универсального метода, применимого для решения всех задач тестирования, не существует: на практике разные подходы используются совместно, дополняя и усиливая друг друга. Как правило, общая функциональность микропроцессора проверяется на случайных тестах, а для проверки наиболее важных модулей и подсистем используются методы на основе моделей.

К сожалению, в настоящее время нет среды, которая смогла бы объединить различные методы генерации тестовых программ для микропроцессоров. Инженеры вынуждены использовать большое число инструментов с различными форматами входных и выходных данных: возникает проблема их интеграции и поддержки согласованности их конфигураций. Если для выполнения независимых работ (например, для создания нескольких тестовых программ разного вида) можно без проблем использовать разные инструменты, то для решения одной сложной задачи с помощью нескольких средств (например, для построения тестовой программы, состоящей из частей разного вида) может потребоваться нетривиальная схема их интеграции. Кроме того, разные инструменты используют разные, несовместимые друг с другом описания микропроцессора и тестовых сценариев, в результате чего одни и те же объекты определяются несколько раз, усложняя поддержку генераторов тестов.

В статье предлагается концепция расширяемой среды генерации тестовых программ для микропроцессоров. Среда построена вокруг *модели микропроцессора*, предоставляющей знания о целевом микропроцессоре (о его системе команд, регистрах и т.п.) в определенной форме. Модель доступна *генераторам тестов*, являющимся *расширениями* среды. Все генераторы реализуют одинаковые интерфейсы, позволяющие задавать параметры генерации (используемое подмножество системы команд, вероятности появления тех или иных инструкций и т.п.) и получать доступ к результатам генерации (внутреннему представлению построенных тестовых программ). Взаимодействие инженеров со средой осуществляется посредством *тестовых шаблонов*, в которых иерархическим образом задаются используемые генераторы тестов, параметры их работы и способы композиции результатов генерации в единую тестовую программу.

Оставшаяся часть статьи организована следующим образом. В разделе 2 делается обзор существующих методов и инструментов генерации тестовых программ. В разделе 3 проводится анализ описанных подходов и определяется концепция расширяемой среды генерации тестовых программ. В разделе 4 описывается архитектура среды и дается общее представление о ее главных компонентах: *среде моделирования* и *среде тестирования*. В разделе 5 рассматриваются среда моделирования и ее составляющие: *транслятор* и *библиотека моделирования*. В разделе 6 рассматривается среда тестирования и ее составляющие: *обработчик тестовых шаблонов*, *библиотека тестирования* и *механизм разрешения ограничений*. В разделе 7 резюмируются результаты работы и указываются возможные направления дальнейших исследований.

2. МЕТОДЫ ГЕНЕРАЦИИ ТЕСТОВЫХ ПРОГРАММ

Существует множество способов создания тестовых программ, которые можно разделить на два больших класса: (1) *ручная разработка* и (2) *автоматическая генерация*. В настоящее время ручная разработка тестов все применяется для тестирования сложно формализуемых и маловероятных ситуаций в поведении микропроцессоров, но для систематической верификации микропроцессоров она используется крайне редко. Что касается автоматизированных способов генерации, они могут быть разбиты на следующие виды: (1) *случайная генерация*, (2) *комбинаторная генерация*, (3) *генерация на основе шаблонов* и (4) *генерация на основе моделей*.

Случайная генерация – самый распространенный метод создания сложных, хотя и не нацеленных, тестовых программ для верификации микропроцессоров. Несмотря на легкость реализации, метод позволяет создавать интенсивный поток воздействий, приводящий к обнаружению нетривиальных ошибок. Одним из наиболее известных генераторов этого

типа является RAVEN (Random Architecture Verification Engine), разработанный компанией Obsidian Software (приобретенной корпорацией ARM). При построении тестовых программ инструмент не только использует случайную генерацию кода, но и принимает во внимание распространенные ошибки проектирования. Инструмент RAVEN построен на базе заранее разработанных генераторов тестов, но при этом допускает возможность расширения с помощью пользовательских компонентов [3]. К сожалению, ввиду недостатка общедоступной информации, детали реализации инструмента неясны.

Другой подход к созданию тестовых программ – *это комбинаторная генерация*. Анализ ошибок в микропроцессорах показывает, что многие из них могут быть обнаружены с помощью небольших тестовых примеров (состоящими из 2–5 инструкций). Таким образом, имеет смысл систематически перебирать короткие последовательности инструкций (включая *тестовые ситуации* для отдельных инструкций и *зависимости* между парами инструкций) [2]. Метод был реализован в первой версии среды MicroTESK (Microprocessor TEsting and Specification Kit), разрабатываемой в ИСП РАН. Инструмент поддерживает иерархическую декомпозицию генератора тестов на *итераторы* (каждый из которых отвечает за перебор своей части теста) и *композиторы* (объединяющие результаты работы внутренних итераторов в более сложные тестовые последовательности). Кроме того, MicroTESK позволяет генерировать тестовые программы, содержащие *инструкции перехода*: для этого строятся различные графы потока управления, для каждого из которых осуществляется перебор возможных путей выполнения ограниченной длины [7].

Следующий метод называется *генерацией на основе шаблонов*. *Тестовый шаблон* – это абстрактное представление тестовой программы, где вместо конкретных значений операндов инструкций (как это делается в обычной программе) указываются *ограничения (символические значения)*. Создавая тестовую программу, генератор пытается найти случайное решение соответствующей системы ограничений (такой подход обычно называется *случайной генерацией на основе ограничений* [8]). За счет автоматизации большого объема рутинной работы метод существенно увеличивает продуктивность работы инженеров. Наиболее известным генератором такого типа является Genesys-Pro (IBM Research) [4]. Инструмент использует два типа входных данных: (1) *модель*, описывающая архитектуру микропроцессора, и (2) *тестовые шаблоны*, задающие сценарии тестирования. Genesys-Pro генерирует тестовую программу последовательно, инструкцию за инструкцией: на каждом шаге выбирается инструкция, которую нужно вставить в программу, а затем формулируется и решается система ограничений на ее операнды.

В отличие от ранее рассмотренных подходов, *генерация на основе моделей* использует формальные модели микропроцессоров для построения тестовых программ (или тестовых шаблонов). Здесь необходимо внести ясность в терминологию. Есть два типа моделей микропроцессоров: (1) *модели уровня инструкций (поведенческие модели)* и (2) *микроархитектурные модели (структурные модели)*. Модели первого типа определяют системы команд микропроцессоров (это взгляд снаружи). Модели второго типа определяют структурную организацию микропроцессора (это взгляд изнутри). Все методы генерации тестовых программ явно или неявно используют модели уровня инструкций, но только некоторые из них используют модели микроархитектурного уровня (именно они называются *методами на основе моделей)*. Далее будут рассмотрены некоторые такие подходы.

В работе [5] предложен метод нацеленной генерации тестовых программ. Он использует подробные спецификации микропроцессоров на языке EXPRESSION [9] и переводит их в описание на языке SMV (Symbolic Model Verifier) [10]. Спецификации определяют структуру микропроцессора (компоненты и соединения между ними), его поведение (семантику инструкций), а также взаимосвязь между структурой и поведением. Ключевой частью работы является *модель ошибок*, описывающая типичные ошибки проектирования (ошибки в отдельных операциях, во взаимодействии параллельно выполняемых операций и др.). Для модели микропроцессора по модели ошибок генерируется множество формул, каждая из которых определяет условие

возникновения конкретной ошибки. Для каждой такой формулы с использованием инструмента SMV (и лежащего в его основе метода проверки моделей) строится тестовый пример (контрпример для отрицания формулы), который затем преобразуется в тестовую программу. По мнению авторов, метод не масштабируется на сложные микропроцессоры. В качестве дополнения к нему предлагается использовать подход на основе тестовых шаблонов. Шаблоны разрабатываются вручную и описывают цепочки инструкций, которые создают определенные ситуации в поведении микропроцессора (прежде всего, конвейерные конфликты). Генерация тестовых программ осуществляется с помощью графовой модели микропроцессора, извлеченной из спецификаций.

В работе [11] микроархитектура микропроцессора представляется в форме *операционной автоматной модели (OSM, Operation State Machine)*. OSM-модель включает в себя два уровня: (1) *уровень операций* и (2) *уровень аппаратуры*. На первом уровне описывается логика пошагового выполнения операций: каждая операция описывается отдельным расширенным автоматом (EFSM, Extended Finite State Machine). На втором уровне описываются аппаратные ресурсы микропроцессора: каждому ресурсу ставится в соответствие *маркер (token)*; управление маркерами осуществляется с помощью так называемых *менеджеров*. При выполнении переходов операционные автоматы могут захватывать и освобождать маркеры, обращаясь к соответствующим менеджерам. Модель микропроцессора определяется как композиция операционных и ресурсных автоматов. Тестовые программы генерируются путем обхода всех достижимых состояний и переходов объединенной OSM-модели.

3. КОНЦЕПЦИЯ РАСШИРЯЕМОЙ СРЕДЫ ГЕНЕРАЦИИ ТЕСТОВЫХ ПРОГРАММ

Проанализируем рассмотренные выше методы и инструменты генерации тестовых программ для микропроцессоров и определим понятие расширяемой среды генерации тестовых программ. В общих словах, *расширяемость* – это характеристика среды, показывающая, сколько усилий требуется для интеграции в нее нового или существующего компонента (в нашем случае – модели микропроцессора или генератора тестов). Чем меньше усилий требуется для этой работы, тем более расширяемой считается среда. Цель данной работы – предложить архитектуру среды генерации тестовых программ, которая бы минимизировала усилия, необходимые для создания новых компонентов и их интеграцию в среду.

Расширяемые среды обычно построены по единому принципу: в них есть *ядро (платформа)* и *расширения (дополнительные модули)*, связанные с ядром через заранее определенные *точки расширения*. Понятно, что интерфейсы между ядром и расширениями должны быть четко определены, как и способы установки расширений в среду и их вызова из среды для решения определенных задач. На наш взгляд, положительный эффект на расширяемость среды оказывает открытый исходный код: доступность кода может упрощать создание расширений.

Все подходы к генерации тестовых программ явно или неявно используют *модели уровня инструкций* (для создания тестовой программы необходимо знать *предусловия инструкций* и их *ассемблерный формат*), однако чем сложнее задачи тестирования, тем сложнее модели должны быть использованы для их решения. На наш взгляд, ядро среды генерации тестовых программ должно быть построено на базе моделей уровня инструкций и соответствующих методов генерации тестов (методов случайной и комбинаторной генерации, а также методов генерации на основе шаблонов), а более специализированные модели и основанные на них генераторы тестов должны устанавливаться в среду в форме расширений. Среду генерации тестовых программ удобно представить состоящей из двух частей: (1) *среды моделирования* и (2) *среды тестирования*.

Ядро среды моделирования позволяет описывать *регистры* (переменные, хранящие битовые векторы фиксированного размера), *память* (массив машинных слов) и *инструкции* (атомарные действия над регистрами и памятью). Более детальные спецификации предоставляются *расширениями модели*, подсоединяемыми к среде через *точки расширения* (обработчик доступа к памяти, обработчик запуска инструкции и

др.). Среда моделирования поддерживает стандартные расширения для описания типовых подсистем: *устройства управления памятью* (описание иерархия кэш-памяти, механизма трансляции адресов и др.) и *устройства управления конвейером* (описаний стадий конвейера, переключения потока управления и др.). Для стандартных расширений имеется возможность установки дополнительных расширений (например, для задания стратегии замещения строк в кэш-памяти или для описания поведения стадий конвейера).

Среда тестирования состоит из генераторов двух типов: (1) *генераторы тестовых последовательностей* и (2) *генераторы тестовых данных*.

Основными видами генераторов тестовых последовательностей являются *случайные* и *комбинаторные генераторы* [2], [3]. Это объясняется тем фактом, что среда моделирования основана на моделях уровня инструкций, которые не позволяют использовать более сложные методы генерации. Важным свойством среды тестирования является поддержка композиции тестовых программ [12]. Предположим, что существуют две тестовые программы (или два тестовых шаблона), нацеленные на создание разных и относительно независимых ситуаций в работе микропроцессора. Тогда программа, составленная путем "слияния" этих двух программ, может привести к одновременному (или близкому по времени) возникновению соответствующих ситуаций. Более сложные генераторы на основе моделей могут быть установлены в среду вместе с соответствующими средствами моделирования. Заметим, что расширение среды моделирования обычно сопровождается расширением среды тестирования (при добавлении нового типа моделей нужно определить, как генерировать тесты на их основе).

Перспективным подходом к построению тестовых данных является *генерация на основе разрешения ограничений* (как это сделано в Genesys-Pro [4]). Подразумевается, что тестовые ситуации выражаются в форме *ограничений* на значения операндов инструкций и состояние микропроцессора. Достоинством подхода является возможность комбинирования тестовых ситуаций посредством соединения их ограничений. В отличие от Genesys-Pro с его специализированным решателем ограничений (constraint solver) мы предлагаем использовать универсальные решатели, такие как Yices [13] и Z3 [14], поддерживающие язык SMT-LIB [15]. Кроме того, ядро генератора может быть расширено *пользовательскими генераторами тестовых данных*, что полезно в тех случаях, когда тестовые ситуации непросто выразить в форме ограничений. Среда тестирование включает библиотеку встроенных случайных и нацеленных генераторов тестов (например, для тестирования модулей арифметики с плавающей точкой [16]).

4. АРХИТЕКТУРА СРЕДЫ MICROTESK

Среда генерации тестовых программ MicroTESK включает две основные части: (1) *среду моделирования* и (2) *среду тестирования*. Назначение среды моделирования – описать целевой микропроцессор (определить *модель микропроцессора*), а также задать *модель тестового покрытия*. Модель (модель микропроцессора и модель тестового покрытия) извлекается из формальных спецификаций, написанных на *языке описания архитектуры (ADL, Architecture Description Language)*. Среда тестирования, в свою очередь, отвечает за генерацию тестовых программ для целевого микропроцессора на основе информации, предоставленной моделью. Цели тестирования определяются в тестовых *шаблонах*, написанных на *языке описания шаблонов (TDL, Template Description Language)*.

Среда моделирования MicroTESK состоит из следующих компонентов: (1) *транслятора*, анализирующего формальные спецификации на ADL-языке и создающего модель, и (2) *библиотеки моделирования*, содержащей интерфейсы, которые должны быть реализованы в модели, и стандартные блоки, из которых можно строить модель (см. рис. 1). Транслятор включает два компонента (back-ends): (1) *модуль генерации модели*, строящий исполнимую модель микропроцессора, и (2) *модуль извлечения покрытия*, извлекающий модель тестового покрытия для инструкций микропроцессора. Библиотека моделирования разделяется, соответственно, на (1) *библиотеку моделирования*

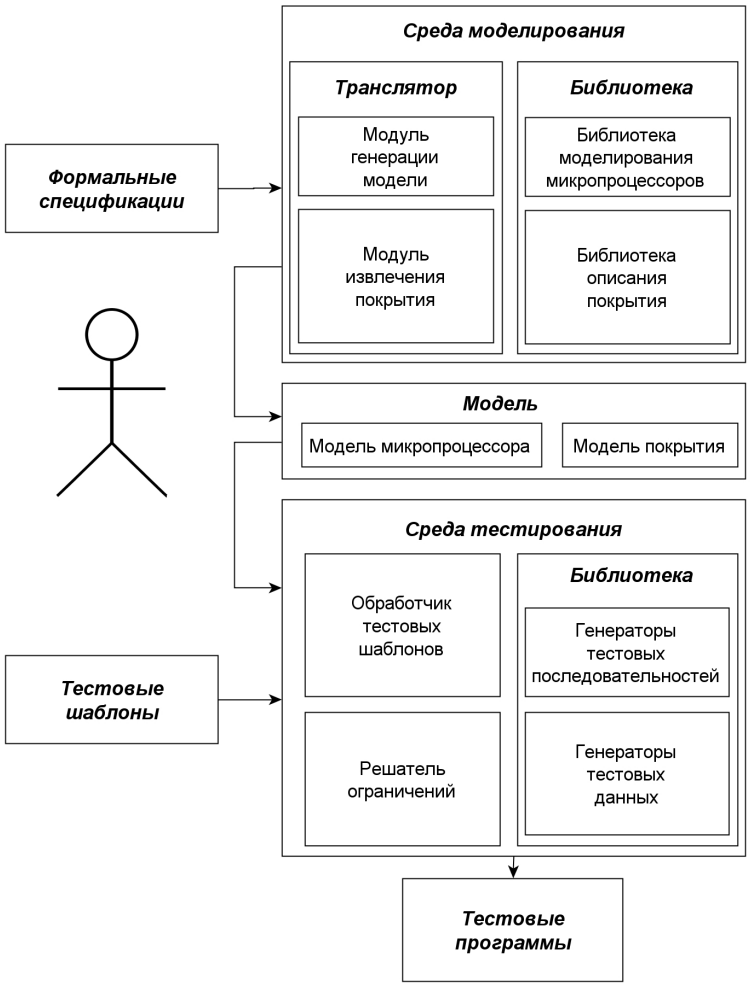

Рис. 1. Общая структура среды моделирования MicroTESK.

микропроцессоров и (2) *библиотеку описания покрытия*.

Среда тестирования MicroTESK состоит из следующих компонентов: (1) *обработчик тестовых шаблонов*, генерирующий тестовые программы по шаблонам, написанных на TDL-языке, (2) *библиотека тестирования*, содержащая многообразные *генераторы тестовых последовательностей* и *генераторы тестовых данных*, используемые обработчиком тестовых шаблонов, и (3) *решатель ограничений*, предоставляющий генераторам тестовых данных интерфейс для взаимодействия с внешними SMT-решателями.

Компоненты среды не являются монолитными и могут быть расширены средствами, ориентированными на решение специфических задач. Все расширения, связанные с одним и тем же аспектом проектирования микропроцессоров, обычно объединяются в *дополнительный модуль (plugin)* (см. рис. 2). Для того чтобы расширить среду средствами моделирования и тестирования некоторого аспекта проектирования, необходимо разработать расширения для всех компонентов среды, включая *библиотеку моделирования* (содержащую стандартные блоки для моделирования этого аспекта), *библиотеку тестирования* (включающую генераторы тестов, нацеленные на проверку этого аспекта), *язык спецификаций и транслятор* (позволяющие описать этот аспект на некотором языке).

Функциональность среды может быть разбита согласно следующим основным аспектам проектирования: (1) *система команд*, (2) *управление памятью* и (3) *управление конвейером*. Средства среды, соответствующие первому аспекту, обра-

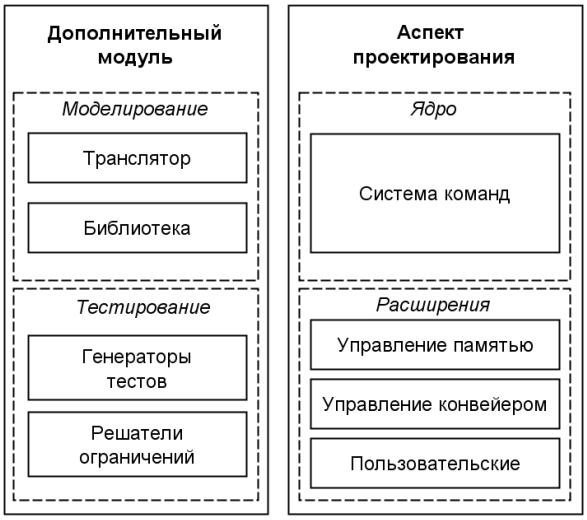

Рис. 2. Аспекты проектирования и организация дополнительных модулей в MicroTESK.

зуют ее ядро – они предназначены для моделирования инструкций микропроцессора и генерацию тестовых программ на основе моделей уровня инструкций (используются случайные, комбинаторные и основанные на шаблонах генераторы). Моделирование и тестирование механизмов управления памятью и конвейером поддерживается стандартными расширениями среды. Поддержка других аспектов проектирования осуществляется путем установки пользовательских расширений.

5. СРЕДА МОДЕЛИРОВАНИЯ MICROTESK

Среда моделирования MicroTESK предназначена для представления знаний о целевом микропроцессоре и передачи этого знания среде тестирования Общая схема работы среды моделирования следующая: (1) инженер разрабатывает *формальные спецификации* микропроцессора; (2) спецификации обрабатываются *транслятором*, который, используя *библиотеку моделирования*, создает модель микропроцессора. Рассмотрим компоненты среды моделирования подробнее.

5.1. Транслятор

Транслятор обрабатывает *формальные спецификации* микропроцессора и строит *модель микропроцессора* и *модель тестового покрытия* (применяя, соответственно, модуль генерации модели и модуль извлечения покрытия и используя встроенные библиотеки моделирования микропроцессоров и описания покрытия). Заметим, что спецификации микропроцессора могут быть написаны на нескольких языках, каждый из которых отвечает за свой аспект проектирования. Основная часть спецификаций связана с системой команд микропроцессора; другие части описывают управление памятью, конвейером и другие аспекты. Из-за неоднородности спецификаций транслятор фактически представляется в виде набора инструментов, обрабатывающих свои части спецификаций.

На данный момент MicroTESK поддерживает единственный ADL-язык для спецификации системы команд – Sim-nML [17, 18]. Ниже приведен код на этом языке, описывающий инструкцию сложения целых чисел (ADD) из системы команд MIPS [19]. Отметим следующее: (1) в спецификациях можно использовать функцию UNPREDICTABLE для определения ситуаций, в которых поведение микропроцессора не определено; (2) анализируя потоки управления в спецификациях инструкций, можно автоматически извлекать модель тестового покрытия; (3) инструкции могут быть объединены в группы, образуя абстрактные инструкции, на которые можно ссылаться в тестовых шаблонах; (4) спецификации являются исполнимыми и детерминированными, что позволяет среде предсказывать результаты выполнения программ [12].

```
op ADD(rd: GPR, rs: GPR, rt: GPR)
action = {
  if(NotWordValue(rs) || NotWordValue(rt))
  then
    UNPREDICTABLE();
  endif;
  tmp = rs<31..31>::rs<31..0> +
  rt<31..31>::rt<31..0>;
  if(tmp<32..32> != tmp<31..31>)
  then
    SignalException("IntegerOverflow");
  else
    rd = sign_extend(tmp<31..0>);
  endif;
}
syntax = format("add %s, %s, %s",
    rd.syntax, rs.syntax, rt.syntax)
op ALU = ADD | SUB | ...
```

5.2. *Библиотека моделирования микропроцессоров*

Библиотека моделирования микропроцессоров предназначена для создания исполнимых моделей микропроцессоров, используемых средой для интерпретации тестовых программ и отслеживания состояния модели в процессе генерации тестов. Отслеживание состояния необходимо для генерации *самопроверяющихся тестов* (программ со встроенными проверками корректности состояния микропроцессора). Таким образом, модель микропроцессора содержит *интерпретатор инструкций* и *функции доступа к состоянию модели*. Кроме того, модель предоставляет *мета-информацию*, описывающую программно видимые элементы микропроцессора: регистры, память, инструкции и др. Мета-информация является основным интерфейсом между средой моделирования и обработчиком тестовых шаблонов.

Библиотека проектирования имеет несколько *точек расширения* для подключения пользовательских компонентов. Набор точек расширения включает в себя: (1) *обработчик доступа к памяти* и (2) *обработчик запуска инструкций*. Обработчик первого типа вызывается каждый раз, как происходит обращение в память для чтения или записи. Он инкапсулирует логику управления памятью, включая трансляцию адресов и кэширование. Обработчик второго типа вызывается в момент запуска инструкции. С его помощью можно моделировать конвейер микропроцессора, раскладывая инструкции на микрооперации, и планирование их выполнения.

5.3. *Библиотека описания покрытия*

Библиотека описания покрытия позволяет определять ситуации, возможные в работе микропроцессоре (переполнение, кэш-попадание или кэш-промах, конвейерный конфликт и др.). Совокупность *тестовых ситуаций*, называемая *моделью тестового покрытия*, служит основой для генерации тестовых программ: прежде всего, для генерации создания тестовых данных для отдельных инструкций. Кроме *тестовых ситуаций* модель тестового покрытия содержит *правила группировки инструкций*, классифицирующие инструкции микропроцессора в соответствии с некоторыми критериями (по типам операндов, используемым ресурсам, структуре потока управления и т.д.). Как и модель микропроцессора, модель тестового покрытия предоставляет мета-информацию о своих элементах.

Каждая тестовая ситуация имеет уникальное имя, которое может быть использовано в тестовом шаблоне для указания на ситуацию. Существует привязка имен ситуаций к генераторам: таким образом, обработчик тестовых шаблонов знает, какой генератор следует использовать для создания той или иной тестовой ситуации. Описание ситуации включает всю необходимую информацию для ее создания генератором. Встроенный в среду генератор тестовых данных использует описание ситуаций в форме ограничений (генерация данных осуществляется на основе разрешения ограничений) [20].

6. СРЕДА ТЕСТИРОВАНИЯ MICROTESK

Среда тестирования MicroTESK отвечает за генерацию тестовых программ. Общая схема работы среды тестирования следующая: (1) инженер создает и подает на вход среде *тестовый шаблон*, описывающий сценарий тестирования микропроцессора; (2) *обработчик тестовых шаблонов* применяет для данного шаблона *генераторы тестовых последовательностей* и строит тестовую программу в *символической фор-*

ме (вместо конкретных значений операндов программа аннотирована именами тестовых ситуаций); (3) обработчик обращается к *генераторам тестовых данных* и конструирует конкретные значения операндов инструкций; (3) полученная тестовая программа дополняется *управляющим кодом*, инициализирующим регистры и память сгенерированными данными. Рассмотрим компоненты среды моделирования подробнее.

6.1. Обработчик тестовых шаблонов

Обработчик тестовых шаблонов преобразовывает *тестовые шаблоны* в *тестовые программы* используя зарегистрированные в среде генераторы тестовых последовательностей и тестовых данных. Поддерживаемый TDL-язык организован в форме библиотеки на языке Ruby [21]. Язык позволяет описывать последовательности инструкций в формате ассемблера, используя *мета-информацию*, предоставляемую *моделью микропроцессора*. Кроме того, язык поддерживает высокоуровневые конструкции описания тестовых сценариев, которые могут быть разделены на два типа: (1) *встроенные команды Ruby* (условные выражения, циклы и др.) и (2) *специальные команды* MicroTESK (блоки генерации тестовых последовательностей, ссылки на тестовые ситуации и др.). Ниже приведен простой пример тестового шаблона.

```
# Код в формате ассемблера
add r[1], r[2], r[3]
sub r[1], r[1], r[4]

# Управляющие конструкции Ruby
(1..3).each do |i|
  add r[i], r[i+1], r[i+2]
  sub r[i], r[i], r[i+3]
end

# Блок генерации
block (:engine ⇒ "random", :count ⇒ 2013)
{
  add r[1], r[2], r[3]
  sub r[1], r[2], r[3]
    # Ссылка на тестовую ситуацию
    do overflow end
}
```

Важной конструкцией, используемой в тестовых шаблонах, является *блок генерации тестовых последовательностей*. Тестовый шаблон является иерархической структурой, состоящей из блоков генерации тестовых последовательностей. Каждый из блоков содержит инструкции и вложенные блоки, а также задает используемый *генератор тестовых последовательностей* и его параметры. Обработчик тестовых шаблонов создает тестовые последовательности для вложенных блоков, применяя соответствующие генераторы, а затем комбинирует полученные тестовые последовательности и составляет из них тестовую программу (соответствующий пример дан в разделе *«Генераторы тестовых последовательностей»*).

Следует отметить, что обработчик тестовых шаблонов поддерживает создание самопроверяющихся тестов. При построении тестовой программы он может добавить специальный код (называемый *тестовым оракулом*), проверяющий корректность состояния микропроцессора в соответствующей точке программы. Тестовый оракул сравнивает данные, хранимые в регистрах и памяти, с эталонными значениями, вычисленными *интерпретатором инструкций*: если данные не совпадают, оракул сообщает об ошибке.

6.2. Генераторы тестовых последовательностей

Генератор тестовых последовательностей реализует некоторый метод перебора цепочек инструкций. Каждому блоку тестового шаблона сопоставляется свой генератор тестовых последовательностей. Поскольку блоки могут быть вложенными, генераторы могут быть скомбинированы рекурсивным образом. Для этого каждый нетерминальный блок должен определять две стратегии: (1) *комбинирование* результатов работы вложенных генераторов и (2) *композиция* (слияние) нескольких цепочек инструкций в единую тестовую последовательность.

Библиотека тестирования содержит некоторые предопределенные стратегии комбинирования и композиции. Поддерживаются следующие методы комбинирования: (1) *случайное комбинирование* (создаются случайные комбинации результатов работы генераторов), (2) *декартово произведение* (создаются все возмож-

ные комбинации результатов работы генераторов) и (3) *диагональ декартова произведения* (для построения комбинаций вложенные генераторы запрашиваются синхронно). Поддерживаемые методы композиции включают: (1) *случайную композицию* (тестовые последовательности смешиваются случайным образом), (2) *конкатенацию* (тестовые последовательности конкатенируются) и (3) *вложение* (тестовые последовательности вкладываются одна в другую). Пользователи могут добавлять в среду генераторы тестовых последовательностей, а также стратегии комбинирования и композиции. Рассмотрим простой пример.

```
# Блок генерации
block(:combine ⇒ “product”, :compose ⇒
⇒ “random”) {

  # Вложенный блок A
  block(:engine ⇒ “random”, :length ⇒ 3,
  :count => 2) {
    add r[a], r[b], r[c]
    sub r[d], r[e], r[f]
    mult r[g], r[h]
    div r[i], r[j]
  }
  # Вложенный блок B
  block(:engine ⇒ “permutate”) {
    ld r[k], r[l]
    st r[m], r[n]
  }
}
```

В приведенном примере блок верхнего уровня, содержит два вложенных блока, A и B. Блок A состоит из четырех инструкций: ADD, SUB, MULT и DIV. Блок B состоит из двух инструкций: LD и ST. Генератор тестовых последовательностей, связанный с A, создает две последовательности (:count ⇒ 2) длины три (:length ⇒ 3), составленные из перечисленных инструкций случайным образом (:engine ⇒ “random”). Генератор, связанный с B, создает все перестановки заданных инструкций (:engine ⇒ “permutate”). Генератор верхнего уровня создает все возможные комбинации результатов работы вложенных блоков (:combine ⇒ “product”) и случайным образом их смешивает (:compose ⇒ “random”). Результат обработки такого шаблона может быть следующим.

```
# Комбинация (1,1)
sub r[d], r[e], r[f]        # Блок A
ld r[k], r[l]               # Блок B
div r[i], r[j]              # Блок A
st r[m], r[n]               # Блок B
add r[a], r[b], r[c]        # Блок A

# Комбинация (1,2)
st r[m], r[n]               # Блок B
sub r[d], r[e], r[f]        # Блок A
ld r[k], r[l]               # Блок B
div r[i], r[j]              # Блок A
add r[a], r[b], r[c]        # Блок A

# Комбинация (2,1)
mult r[g], r[h]             # Блок A
mult r[g], r[h]             # Блок A
ld r[k], r[l]               # Блок B
add r[a], r[b], r[c]        # Блок A
st r[m], r[n]               # Блок B

# Комбинация (2,2)
mult r[g], r[h]             # Блок A
st r[m], r[n]               # Блок B
mult r[g], r[h]             # Блок A
ld r[k], r[l]               # Блок B
add r[a], r[b], r[c]        # Блок A
```

6.3. Генераторы тестовых данных

Задача *генераторов тестовых данных* заключается в построении значений операндов инструкций на основе заданных в шаблоне тестовых ситуаций. Среда MicroTESK поддерживает генерацию тестовых данных на основе *разрешения ограничений*. При построении значений операндов инструкций *обработчик тестовых шаблонов* выбирает генератор тестовых данных, соответствующий указанной ситуации, и запрашивает у *модели микропроцессора* состояние используемых в тестовой ситуации ресурсов (регистров, памяти и др.). После этого обработчик инициализирует соответствующие переменные и вызывает решатель ограничений.

Как только получены значения операндов инструкций, в тестовую программу добавляется *управляющий код*, инициализирующий соответствующие ресурсы микропроцессора. Например,

если операндом инструкции является регистр, управляющий код записывает в него необходимое значение. Следуя *концепции случайной генерации на основе ограничений*, разные вызовы генераторов тестовых данных могут приводить к разным наборам значений операндов (однако каждый набор должен соответствовать заданной системе ограничений).

6.4. Механизм разрешения ограничений

Механизм разрешения ограничений реализован с помощью набора решателей, доступ к которым осуществляется через унифицированный интерфейс. Решатели ограничений делятся на два основных типа: (1) *универсальные решатели*, применимые для широкого класса ограничений, и (2) *пользовательские решатели*, нацеленные на специфические задачи генерации тестовых данных.

К первому типу относятся SMT-решатели, основанные, например, на Yices [13] и Z3 [14]. Они поддерживают операции булевой алгебры, целочисленной арифметики, теории битовых векторов фиксированной длины и др. Взаимодействие с универсальными решателями ограничений реализовано с помощью библиотеки Java Constraint Solver API [20]. Библиотека позволяет создавать ограничения в виде набора Java-объектов, отображать их описания на языке SMT-LIB [15] и запускать внешний SMT-решатель.

Рассмотрим пример ограничения на языке SMT-LIB, описывающий ситуацию переполнения в инструкции целочисленного сложения 32-битных слов. В начале определяются типы (*define-sort*), функции (*define-fun*) и переменные (*declare-const*). Далее описываются предусловия на значения переменных и целевое ограничение (*assert*).

```
; Типы и функции
(define-sort Int_t () (_ BitVec 64))
(define-fun INT_ZERO () Int_t (_ bv0 64))
(define-fun INT_BASE_SIZE () Int_t (_ bv32
64))
(define-fun INT_SIGN_MASK () Int_t
  (bvshl (bvnot INT_ZERO) INT_BASE_SIZE))
(define-fun IsValidPos ((x!1 Int_t)) Bool
  (ite (= (bvand x!1 INT_SIGN_MASK)
  INT_ZERO) true false))
(define-fun IsValidNeg ((x!1 Int_t)) Bool
  (ite (= (bvand x!1 INT_SIGN_MASK)
  INT_SIGN_MASK) true false))
(define-fun IsValidSignedInt ((x!1 Int_t)) Bool
  (ite (or (IsValidPos x!1) (IsValidNeg x!1)) true
  false))

; Переменные
(declare-const a Int_t)
(declare-const b Int_t)

; Предусловия
(assert (IsValidSignedInt a))
(assert (IsValidSignedInt b))

; Ограничение
(assert (not (IsValidSignedInt (bvadd a b))))
```

Некоторые тестовые ситуации сложно выразить посредством ограничений (это справедливо для арифметики с плавающей точкой [22], управления памятью [23] и др.). Для описания и обработки таких ситуаций в среду могут быть добавлены пользовательские генераторы тестовых данных.

7. ЗАКЛЮЧЕНИЕ

В статье предложена архитектура расширяемой среды генерации тестовых программ для микропроцессоров. Описанный подход был реализован в среде MicroTESK (ИСП РАН). Разработанная платформа позволяет объединить множество разнообразных методов моделирования и тестирования микропроцессоров. Среда базируется на моделях уровня инструкций и поддерживает случайную, комбинаторную и основанную на шаблонах генерацию тестовых программ. Более сложные типы моделей и генераторов тестов могут быть добавлены в среду в качестве расширений. Формальная спецификация системы команд микропроцессора осуществляется на языке Sim-nML. На основе анализа спецификаций строится модель (модель микропроцессора и модель тестового покрытия). Описание шаблонов тестовых программ выполняется на языке Ruby с возможностью обращения к инструкциям, тестовым ситуациям и другим элементам построенной модели. Язык позволяет описывать сложные сценарии тестирования микропроцессоров и поддерживает композицию нескольких тестовых программ. В ближайшем будущем мы

планируем использовать среду MicroTESK для разработки генераторов тестовых программ для широко распространенных архитектур, включая ARM и MIPS. Мы также работаем над расширением среды средствами моделирования и тестирования типовых подсистем микропроцессоров (буферов трансляции адресов, модулей кэш-памяти, устройств управления и др.).

СПИСОК ЛИТЕРАТУРЫ

1. *Abadir M.S., Dasgupta S.* Guest Editors' Introduction: Microprocessor Test and Verification. IEEE Design & Test of Computers. 2000. V. 17. I. 4. 2000. P. 4–5.
2. *Камкин А.С.* Генерация тестовых программ для микропроцессоров. Сборник трудов ИСП РАН. 2008. Т. 14. Ч. 2. С. 23–63.
3. http://www.arm.com/community/partners/ /display product/rw/ProductId/5171/.
4. *Adir A., Almog E., Fournier L., Marcus E., Rimon M., Vinov M., Ziv A.* Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification. IEEE Design Test of Computers. 2004. V. 21. I. 2. P. 84–93.
5. *Mishra P., Dutt N.* Specification-Driven Directed Test Generation for Validation of Pipelined Processors. ACM Transactions on Design Automation of Electronic Systems (TODAES). 2008. V. 13. I. 3. P. 1–36.
6. http://forge.ispras.ru/projects/microtesk
7. *Камкин А.С.* Некоторые вопросы автоматизации построения тестовых программ для модулей обработки переходов микропроцессоров. Труды ИСП РАН. 2010. Т. 18. С. 129–150.
8. *Naveh Y., Rimon M., Jaeger I., Katz Y., Vinov M., Marcus E., Shurek G.* Constraint-Based Random Stimuli Generation for Hardware Verification. AI Magazine. 2007. V. 28. № 3. P. 13–30.
9. *Grun P., Halambi A., Khare A., Ganesh V., Dutt N., Nicolau A.* EXPRESSION: An ADL for System Level Design Exploration. Technical Report 1998-29, University of California, Irvine, 1998.
10. http://www.cs.cmu.edu/ ?modelcheck/smv.html.
11. *Dang T.N., Roychoudhury A., Mitra T., Mishra P.* Generating Test Programs to Cover Pipeline Interactions. Design Automation Conference (DAC), 2009. P. 142–147.
12. *Kamkin A., Kornykhin E., Vorobyev D.* Reconfigurable Model-Based Test Program Generator for Microprocessors. Software Testing, Verification and Validation Workshops (ICSTW), 2011. P. 47–54.
13. *Dutertre B., Moura L.* The YICES SMT Solver. 2006 http://yices.csl.sri.com/tool-paper.pdf
14. *Moura L., Bjorner N.* Z3: An Efficient SMT Solver. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2008. P. 337–340.
15. *Cok D.R.* The SMT-LIBv2 Language and Tools: A Tutorial. GrammaTech, Inc., Version 1.1, 2011.
16. *Aharoni M., Asaf S., Fournier L., Koifman A., Nagel R.* FPgen – A Test Generation Framework for Datapath Floating-Point Verification. High Level Design Validation and Test Workshop (HLDVT), 2003. P. 17–22.
17. *Freericks M.* The nML Machine Description Formalism. Techical Report, TU Berlin, FB20, Bericht 1991/15.
18. *Moona R.* Processor Models For Retargetable Tools. International Workshop on Rapid Systems Prototyping (RSP), 2000. P. 34–39.
19. MIPS64T M Architecture For Programmers. V. II: The MIPS64T M Instruction Set. Document Number: MD00087, Revision 2.00, June 9, 2003.
20. http://forge.ispras.ru/projects/solver-api.
21. http://www.ruby-lang.org.
22. *Камкин А.С., Чупилко М.М.* Тестирование модулей арифметики с плавающей точкой микропроцессоров на соответствие стандарту IEEE 754. Труды ИСП РАН. 2008. Ч. 2. С 7–22.
23. *Kornykhin E.V.* Generation of Test Data for Verification of Caching Mechanisms and Address Translation in Microprocessors. Programming and Computer Software. January 2010. V. 36. I. 1. P. 28–35.