Средства функциональной верификации микропроцессоров

А.С. Камкин, А.М. Коцыняк, С.А. Смолов, А.А. Сортов, А.Д. Татарников, М.М. Чупилко
{kamkin,kotsynyak,ssedai,sortov,andrewt,chupilko}@ispras.ru

Аннотация. Обеспечение корректности микропроцессоров и другой микроэлектронной аппаратуры является фундаментальной проблемой, для решения которой применяют разнообразные средства функциональной верификации. В отличие от программ, ошибки в которых исправляются сравнительно просто, дефекты в интегральных схемах (конструктивные и производственные) не могут быть устранены. Несмотря на то, что постоянно совершенствуются системы автоматизированного проектирования (САПР), инструменты генерации тестов и методы анализа схем, верификация остается самым узким местом процесса разработки (на нее тратится около 70% всех ресурсов проектирования). В работе делается краткий обзор средств верификации микропроцессоров, рассматриваются проблемы, возникающие в промышленной практике, анализируются возможные пути их решения. Значительная часть статьи посвящена исследованиям по верификации аппаратуры, проводимым в ИСП РАН: подводятся итоги выполненных работ, описываются текущие разработки, формулируются направления дальнейших исследований.

Ключевые слова: микропроцессоры; цифровая аппаратура; верификация; валидация; тестирование; генерация тестов; моделирование; языки описания архитектуры; распараллеливание.

1. Введение

Верификацией называется проверка соответствия результатов, полученных на отдельных этапах проектирования (разработки) программных и аппаратных систем, требованиям и ограничениям, установленным для них на предыдущих этапах (на начальном этапе проверяется соответствие исходным требованиям — техническому заданию) [1]. Основной задачей верификации является контроль качества проектирования, включая такие его аспекты, как корректность, надежность, производительность, энергопотребление, эргономика и многие другие. В рамках настоящей статьи рассматривается лишь один из них — *функциональная корректность*. Комплекс мер, нацеленный на обеспечение корректности разрабатываемой системы (прежде всего, на обнаружение и исправление ошибок проектирования), называется

функциональной верификацией (в дальнейшем под верификацией будет пониматься именно функциональная верификация).

Год	***Микропроцессор***	***Транзисторы***
1971	4004	2 300
1974	8080	5 000
1978	8086	29 000
1982	186	55 000
1982	286	134 000
1985	386	275 000
1989	486	1 180 235
1993	Pentium	3 100 000
1997	Pentium II	7 500 000
1999	Pentium III	24 000 000
2000	Pentium 4	42 000 000
2001	Itanium	25 000 000
2002	Itanium 2	220 000 000
2004	Itanium 2 9M	592 000 000
2006	Core 2 Duo (2 ядра)	291 000 000
2008	Core i7 (4 ядер)	731 000 000
2011	Core i7 (6 ядер)	2 270 000 000
2012	Xeon Westmere-EX (8 ядер)	2 600 000 000
2012	Xeon Phi (62)	5 000 000 000

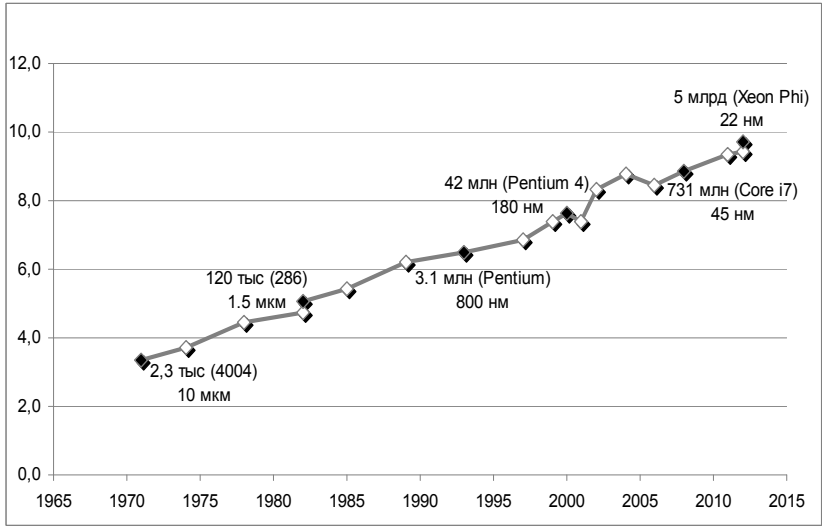

Рис. 1. Рост числа транзисторов (на графике — десятичный логарифм) в микропроцессорах фирмы Intel [2]

Тема статьи ограничена не только видом верификации, но и типом рассматриваемых систем — *микропроцессорами*, программно управляемыми устройствами, предназначенными для цифровой обработки данных. Поскольку функциональность микропроцессора определяется реализуемой им системой команд, то, в самых общих словах, задача верификации состоит в проверке того, что микропроцессор (точнее, его проектная модель или схема) корректно реализует все указанные в техническом задании команды (инструкции): результат выполнения каждой команды во всех возможных ситуациях соответствует ее спецификации [3]. Верификация микропроцессора — чрезвычайно трудоемкая задача. По некоторым оценкам, затраты на нее составляют порядка 70% от всех затрат на проектирование, число инженеров-верификаторов примерно вдвое превосходит число инженеров-разработчиков, а исходный код тестовых систем составляет до 80% от общего объема кода [4]. С ростом сложности микропроцессоров (закон Мура [5] работает до сих пор — см. рис. 1) ситуация только ухудшается — возможности методов верификации отстают от развития микропроцессоров; соответственно, проверка корректности (и без того являющаяся самым узким местом процесса проектирования) вовлекает в себя все бóльшие объемы ресурсов. К примеру, над верификацией микропроцессора Pentium 4 (2000 г.) работала команда, состоявшая приблизительно из 70 человек, а для прогона тестов использовалось около 6 тысяч компьютеров, работавших в круглосуточном режиме [6, 7].

Почему верификация микропроцессоров так важна? Потому что никто не хочет доверять свою жизнь, здоровье и благополучие системам, содержащим ошибки, которые при некоторых обстоятельствах могут повести себя непредсказуемым образом, а микропроцессоры — это основа основ всех компьютерных систем. Неслучайно пользователи компьютеров (которых с каждым годом становится все больше и больше) очень бурно реагируют на ошибки, обнаруживаемые в микропроцессорах. Показательна в этом плане ошибка в реализации команды деления в микропроцессоре Pentium компании Intel, обнаруженная в 1994 г. [8]. Несмотря на то, что большинства пользователей эта проблема не касалась, и вероятность ее возникновения крайне мала, для сохранения имиджа компании Intel пришлось организовать замену микросхем, что обошлось ей в 475 миллионов долларов. В 2007 г. много шума наделала ошибка в реализации механизмов кэширования 4-ядерного микропроцессора AMD Phenom (ошибка #298 [9]), которая может приводить к зависанию системы или порче данных. В 2008 г. в Сети обсуждалась близкая проблема в Intel Core i7 (Nehalem), но, как выяснилось, тревога оказалась ложной [10]. С другой стороны, в Core i7 хватает и других проблем — в спецификации [11] (2011 г.) перечислены 153 ошибки, из которых только 16 имеют статус «исправлена» и 2 «исправление запланировано». Следует понимать, что это лишь известные проблемы — общее число ошибок может быть существенно больше.

Трудоемкость и актуальность верификации стимулируют научные исследования в этой области. Основная тенденция в развитии средств верификации состоит в усилении роли *формальных методов* — методов, основанных на анализе математических (формальных) моделей систем, модулей и протоколов их взаимодействия [12]. В рамках формальной верификации используются специфические техники, такие как *проверка моделей*, *дедуктивный анализ*, *проверка эквивалентности* и другие [1]. Использование формальных методов требует значительных усилий на построение моделей, однако если модели построены, то их анализ в значительной мере может быть выполнен автоматически. Лидером в области формальной верификации микропроцессоров является компания Intel, которая формально проверяет широкий класс устройств: от модулей арифметики с плавающей точкой до протоколов обеспечения когерентности памяти [13]. К 2015 г. Intel планирует увеличить использование формальных методов в модульной верификации микропроцессоров до 50% [14, 15]. Методы тестирования по-прежнему доминируют, но следует отметить, что за последние 20 лет они претерпели значительную трансформацию: современные подходы сочетают в себе как эвристические, так и формальные техники [16].

Свой вклад в развитие средств функциональной верификации микропроцессоров вносит и ИСП РАН, занимающийся этой тематикой с 2005 г. Сфера интересов Института включает технологии промышленной верификации микропроцессоров на модульном и системном уровнях, а также формальные методы проектирования и верификации микропроцессоров. Помимо выполнения теоретических исследований и разработки инструментальных средств Институт сотрудничает с ведущими отечественными производителями микропроцессоров, НИИСИ РАН и ЗАО «МЦСТ», выполняя для них проекты по верификации с использованием разработанных средств. В статье описываются результаты, полученные Институтом в области верификации микропроцессоров, рассматриваются текущие разработки, обрисовываются направления дальнейших исследований.

Статья организована следующим образом. Раздел 2 представляет собой краткое введение в предметную область. В разделе 3 описываются выполненные исследования и разработки. Раздел разбит на два подраздела, посвященных модульной и системной верификации соответственно. Каждый подраздел содержит несколько частей, отражающих основные вехи проделанных работ. Сравнение предложенных методов и реализованных инструментов с существующими подходами осуществляется непосредственно в той части, которая описывает предлагаемое решение. Раздел 4 рассматривает текущие и перспективные разработки Института. Как и раздел 3, он разбит на два подраздела, посвященных модульной и системной верификации. Раздел 5 завершает статью.

2. Средства верификации микропроцессоров

В общих словах, процесс проектирования микропроцессора состоит из четырех основных этапов, на каждом из которых создается его модель определенного уровня абстракции: (1) *архитектурное проектирование*, (2) *детальное проектирование*, (3) *логическое проектирование*, (4) *физическое проектирование* [3]. Рассмотрим первые два этапа более подробно (этапы логического и физического синтеза автоматизированы средствами современных САПР и в данной статье не рассматриваются).

На этапе архитектурного проектирования разрабатывается система команд микропроцессора (*макроархитектура*) и уточняется его внутренняя структура (*микроархитектура*). Основными средствами, используемыми на данном этапе, являются (1) *языки программирования общего назначения*, (2) *языки системного проектирования* (*SLDL, System-Level Design Languages*) и (3) *языки описания архитектуры* (*ADL*, *Architecture Description Languages*) [17, 18, 19]. Примеры языков указанных типов представлены в табл. 1. Результатом архитектурного проектирования является *симулятор микропроцессора* — программная модель, позволяющая интерпретировать программы, написанные в соответствующей системе команд. Симуляторы микропроцессоров используются для кросс-разработки ПО, а также для верификации, где они выступают в качестве эталонных моделей (см. раздел 2.3).

Табл. 1. Языки, используемые при проектировании микропроцессоров

Тип языка	***Примеры***
Языки программирования общего назначения	C, C++, Perl, Python
Языки описания архитектуры (ADL)	LISA, EXPRESSION, ISDL, nML
Языки системного проектирования (SLDL)	SystemC, SystemVerilog, Bluespec
Языки описания аппаратуры (HDL)	Verilog, VHDL

На этапе детального проектирования применяются *языки описания аппаратуры* (*HDL*, *Hardware Description Languages*), такие как VHDL и Verilog, позволяющие предельно точно описывать структуру и поведение микропроцессора [17]. Результатом этапа является *модель уровня регистровых передач* (*RTL*, *Register Transfer Level*), которая с потактовой точностью определяет пересылки данных, возникающие при работе устройства. RTL-модель (называемая также *HDL-моделью* или *HDL-описанием*) преобразуется (посредством логического и физического синтеза) в представление (как правило, основанное на *фотошаблонах*), используемое при производстве интегральных схем. Хотя верификация присутствует на всех этапах разработки, особенно она актуальна при создании HDL-модели, поскольку функциональность, описанная на этом этапе, впоследствии не изменяется [3].

2.1. Методы верификации микропроцессоров

Существующие методы верификации микропроцессоров можно разбить на три основных класса: (1) *экспертиза*, (2) *имитационная верификация* (*simulation-based verification*), также называемая *динамической верификацией* или *тестированием*, и (3) *формальная верификация*. Кроме того, существуют так называемые *гибридные методы* [16] (другие названия — *синтетические методы* [20] и *полуформальные методы* [21]), которые используют комбинации указанных подходов (прежде всего, комбинации имитационных и формальных методов).

К *экспертизе* относятся методы верификации, в которых оценка результатов проектирования выполняется людьми путем умозрительного анализа (инспекция кода, визуальный анализ схем и т.п.). Отличительной чертой экспертизы является возможность ее выполнения с использованием только результатов проектирования, а не их формальных моделей (как в формальной верификации) или результатов работы (как в имитационной верификации). Гипотетически, экспертиза позволяет выявлять практически любые виды ошибок, причем на самых ранних стадиях. В то же время она не может быть автоматизирована и ее эффективность существенно зависит от опыта и мотивации ее участников [1].

Под *имитационной верификацией* обычно понимается тестирование HDL-моделей аппаратуры, выполняемое в специальной среде имитационного моделирования — *HDL-симуляторе* [4]. Для применения этого подхода необходимо иметь модель микропроцессора, что невозможно на ранних этапах проектирования. Создание набора тестов, позволяющих адекватно проверить такое сложное устройство, как микропроцессор, является чрезвычайно трудоемкой задачей. Однако наличие простых методик (например, *случайная генерация тестов*), а также возможность проверить реальное поведение на реальных примерах являются основными причинами, по которым имитационная верификация широко распространена (более подробно этот класс методов рассмотрен в разделах 2.2 и 2.3).

Формальная верификация основана на построении математической (формальной) модели системы и ее анализе на предмет выполнимости свойств, также выраженных формально (например, свойств *безопасности* (*safety*) — недостижимости ошибочных состояний — и *живости* (*liveness*) — отсутствия зависаний и зацикливаний) [22]. Модель может разрабатываться вручную или извлекаться из HDL-описания устройства [23]. Преимущество формальной верификации состоит в том, что проверка осуществляется для всех возможных вариантов поведения модели, что позволяет считать верификацию исчерпывающей (для заданных модели и свойств) [24]. Недостатком является высокая трудоемкость, связанная с необходимостью разработки формальной модели (если модель не извлекается из исходного кода) и обоснованием ее адекватности (эквивалентности исходному HDL-описанию) [25].

В табл. 2 представлено распределение найденных ошибок в зависимости от используемого метода верификации в проекте Pentium 4 компании Intel (2000 г.) [6]. Согласно представленным данным, 74% ошибок находится с помощью имитационной верификации, причем бóльшая их часть приходится на модульную верификацию (автономную проверку модулей микропроцессора). Формальные методы применялись ограниченно, поэтому процент ошибок, найденных таким способом, невелик (около 6%), однако это такие ошибки, которые практически невозможно обнаружить с помощью других средств. Несмотря на то, что статистике, приведенной в табл. 2, более 10 лет, она все еще сохраняет свою актуальность (особенно для отечественных компаний, которые уступают ведущим мировым компаниям по уровню развития технологий верификации[1]).

Табл. 2. Метод верификации — число найденных ошибок (Pentium 4, 2000 г.) [6]

Метод верификации	***Число ошибок***	***Процент ошибок***
Имитационная верификация (модульный уровень)	3411	43.5%
Имитационная верификация (системный уровень)	2398	30.5%
Экспертиза (инспекция кода)	1554	20.0%
Формальная верификация	492	6.0%
Итого	**7855**	**100%**

Доминирующим подходом (даже в таких компаниях, как Intel, AMD и IBM) по-прежнему является имитационная верификация. Чтобы повысить ее эффективность микропроцессор декомпозируется на множество относительно простых модулей (устройств, блоков, подсистем, кластеров), каждый из которых проверяется автономно. Таким образом, помимо обязательной *системной верификации*, оценивающей работоспособность микропроцессора в целом, применяют еще и *модульную верификацию* для более тщательной проверки отдельных устройств. Ниже кратко рассматривается, как имитационная верификация реализуется на модульном и системном уровнях.

2.2. Средства модульной верификации

Имитационная верификация на *модульном уровне* осуществляется с помощью *тестовых систем* (*testbenches*) — специализированных программ, которые в автоматическом режиме подают на верифицируемую HDL-модель *тестовые воздействия* (*стимулы*) и проверяют корректность выдаваемых ею *ответов*

[1] В условиях ограниченного финансирования обычно экономят на верификации.

(*реакций*) [4]. Типичная архитектура тестовой системы представлена на рис. 2. Тестовая система выполняется в HDL-симуляторе и эмулирует *окружение* тестируемой HDL-модели. В общих словах, она решает три задачи: (1) генерация тестовой последовательности, (2) проверка корректности поведения HDL-модели и (3) оценка полноты тестирования. За решение каждой задачи отвечает свой компонент тестовой системы: соответственно *генератор стимулов*, *тестовый оракул* и *сборщик тестового покрытия*.

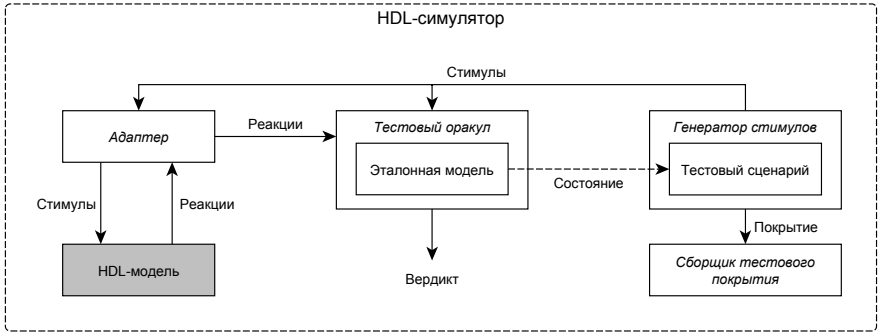

Рис. 2. Архитектура тестовой системы (системы имитационной верификации)

Генератор стимулов на основе *тестовых сценариев* или иных *параметров генерации* создает поток стимулов на тестируемую HDL-модель. Цель генератора — реализовать максимально возможное число ситуаций в работе устройства. В зависимости от используемой технологии сценарии (или непосредственно генераторы) описываются разными способами: используя техники *случайной генерации*, в том числе *случайной генерации на основе ограничений* [26] или на основе *конечно-автоматных моделей* (*графов состояний*). Первый подход используется в технологии UVM (Universal Verification Methodology) [27], продвигаемой организацией Accellera Systems Initiative; второй — в технологии UniTESK (Unified TEsting and Specification Kit) [28], разрабатываемой в ИСП РАН (см. раздел 3.1). Перспективным направлением является генерация тестов на основе статического анализа HDL-описания (см. раздел 4.1). Дополнительное представление о методах построения тестов для микропроцессоров дает раздел 2.3, посвященный системной верификации.

Тестовый оракул проверяет корректность поведения HDL-модели и выносит *вердикт* о его соответствии (или несоответствии) требованиям. Как правило, требования задаются посредством *эталонной модели* — упрощенной реализации устройства на языке программирования (C или C++) или SLDL-языке (SystemC или SystemVerilog). Также используются спецификации в декларативной форме: *расширенные регулярные выражения* [29],

контрактные спецификации [30], *системы правил* [31] и *темпоральные утверждения* [32]. Самым популярным формализмом, используемым для описания свойств аппаратуры, является *темпоральная логика линейного времени* (*LTL*, *Linear Temporal Logic*) [33], расширения которой используются в *языках верификации аппаратуры* (*HVL*, *Hardware Verification Languages*): ForSpec [34], OpenVera [35], Property Specification Language (PSL) [36], *e* [37], SystemVerilog[2] [38].

Сборщик тестового покрытия предназначен для оценки полноты тестирования и принятия решения о его завершении. Для этого используются количественные показатели, называемые *метриками* [39]. Метрики могут определяться на основе разных артефактов проектирования (HDL-описания, эталонной модели, формальной спецификации или документации). Основная идея состоит в следующем. Некоторым систематическим образом определяется набор *тестовых ситуаций*, составляющих в совокупности *тестовое покрытие*. Для заданного выполнения тестовой системы (и HDL-модели) метрика возвращает число реализованных ситуаций. Цель тестирования — покрыть все ситуации в рамках выбранной модели покрытия. Метрики на основе реализации называются *структурными,* а метрики на основе спецификации — *функциональными*. Широко используются метрики на основе кода (HDL-описания и эталонной модели): покрытие строк кода, ветвей и путей в графе потока управления, условий. Для аппаратуры большую роль играют также метрики на основе автоматных моделей: покрытие состояний и переходов.

Как правило, тестовые системы разрабатываются не на уровне RTL, а на более высоком уровне абстракции — *уровне транзакций* (*TLM*, *Transaction Level Modeling*) [40, 25]. TLM-модели основаны на парадигме *передачи сообщений*: компоненты модели соединяются *каналами* и взаимодействуют друг с другом посредством *транзакций* (посылки и приема сообщений). Соответственно, стимулы и реакции внутри тестовой системы имеют форму сообщений, а их передача на *входные интерфейсы* HDL-модели (логически связанные группы сигналов) и прием с *выходных интерфейсов* осуществляется с помощью специализированных каналов, *адаптеров* (*транзакторов*), инкапсулирующих детали преобразования данных между разными уровнями. Для разработки TLM-моделей используются SLDL-языки, а также языки программирования общего назначения (см. табл. 1).

[2] Язык SystemVerilog может использоваться как для проектирования аппаратуры, так и для верификации. Такие языки называются *языками описания и верификации аппаратуры* (*HDVL*, *Hardware Description and Verification Languages*).

2.3. Средства системной верификации

Верификация на *системном уровне*[3] осуществляется путем создания *тестовых программ* и анализа результатов их выполнения на HDL-модели микропроцессора [41, 42]. Как правило, тестовые программы разрабатываются на *языке ассемблера*, а их целью является создание разнообразных ситуаций в работе микропроцессора (особые случаи выполнения арифметических операций, возникновение прерываний, заполнение буферов, вытеснение данных из кэш-памяти, сложные взаимодействия между модулями и т.п.). Ввиду высокой трудоемкости верификации тестовые программы обычно генерируются автоматически. Тесты, разработанные вручную, используются для проверки сложно формализуемых и маловероятных ситуаций (такие тесты создаются с привлечением экспертного знания об особенностях реализации той или иной подсистемы микропроцессора). Для верификации также применяется код, полученный с помощью компиляции существующих программ на языках высокого уровня: известных библиотек (вроде glibc), архитектурно-независимых тестов (типа арифметического теста PARANOIA [43]), тестов на оптимизирующий компилятор (создаваемый для разрабатываемого микропроцессора) и других.

Для автоматического построения тестовых программ применяются следующие способы [44]: (1) *случайная генерация*, (2) *комбинаторная генерация*, (3) *генерация на основе шаблонов* и (4) *генерация на основе моделей*. *Случайная генерация* является самым распространенным методом создания тестовых программ. Примерами генераторов случайных тестов являются RAVEN (Random Architecture Verification Engine) [45], разработанный в Obsidian Software и в настоящее время используемый в ARM, и INTEG [46], созданный в НИИСИ РАН. Идея *комбинаторной генерации* состоит в систематическом переборе тестовых программ небольшого размера [42]. Различные варианты этого подхода реализованы в инструменте MicroTESK (Microprocessor TEsting and Specification Kit) [47], разрабатываемом в ИСП РАН (см. раздел 3.2.1). Более общий подход к генерации тестовых программ основан на использовании *шаблонов* — абстрактных символических представлений тестовых программ. Построение программ по шаблонам базируется на *случайной генерации на основе ограничений* [26]. Примерами инструментов такого типа являются Genesys-Pro [41, 48] от IBM Research и MicroTESK (см. раздел 3.2.2). Для генерации высококачественных тестовых программ могут использоваться *модели уровня микроархитектуры* [49, 50]. Коммерческие инструменты такого типа нам не известны, однако исследования на эту тему ведутся (см. раздел 4.2.1).

[3] Для многоядерных микропроцессоров также применяется *верификация уровня ядра*. Сказанное в этом разделе относится как к системной верификации, так и к верификации уровня ядра.

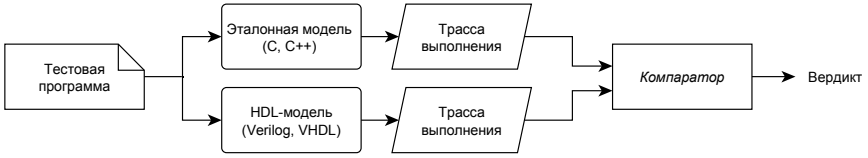

Рис. 3. Системная верификация посредством сравнения трасс

Для проверки корректности поведения HDL-модели микропроцессора применяются два основных метода: (1) сравнение *трасс выполнения* тестовых программ с эталонными трассами и (2) использование тестовых программ со *встроенными проверками* (*self-checking test programs*). В первом подходе (см. рис. 3) при выполнении программы на HDL-модели создается трасса выполнения, отражающая события, возникающие в микропроцессоре. Полученная трасса сравнивается с эталонной трассой — трассой, полученной при выполнении той же программы на *симуляторе микропроцессора*. Сравнение трасс осуществляется с помощью специальной программы, называемой *компаратором*. Во втором методе в код тестовой программы (вручную или автоматически) включаются проверки, которые необходимо осуществить в процессе выполнения программы. Программы со встроенными проверками можно использовать не только для тестирования проекта в HDL-симуляторе, но и для его верификации с помощью аппаратных ускорителей и опытных образцов интегральных схем (*post-silicon verification*). Точность проверок при этом снижается (программно можно наблюдать лишь малую часть событий, возникающих в микропроцессоре: запись данных в регистр или память, прерывание и другие), однако производительность значительно возрастает (год круглосуточного тестирования с использованием тысяч компьютеров соответствует нескольким минутам реальной работы микропроцессора [7]).

Для принятия решения о завершении верификации, как и на модульном уровне, используются метрики тестового покрытия, однако они задают необходимое, но не достаточное условие сдачи проекта в производство (на системном уровне практически невозможно создать адекватную модель тестового покрытия). Большинство производителей микропроцессоров используют подход, основанный на измерении *частоты обнаружения ошибок*. Этот метод предполагает постоянный прогон случайно генерируемых тестов: когда частота обнаружения ошибок становится устойчиво низкой, и в течение длительного времени ошибки не проявляются, тестирование прекращается (при условии, что достигнуты показатели, заданные метриками). В начале 2000-х гг. в компании Motorola одним из необходимых условий сдачи проекта в производство была безошибочная работа HDL-модели в течение 40 миллиардов тактов на случайных тестах [51].

3. Выполненные исследования и разработки

Исследования по верификации аппаратуры ведутся в ИСП РАН с 2005 г., когда была показана применимость технологии UniTESK [28] к модульному тестированию HDL-моделей, и были разработаны средства интеграции тестовых систем, создаваемых с помощью инструмента CTESK [52], с HDL-симуляторами [53]. Немного позже, в 2006 г., начались совместные работы с НИИСИ РАН — организацией, занимающейся промышленной разработкой микропроцессоров. Темой первой из них стала генерация тестовых программ для подсистемы управления памятью (MMU, Memory Management Unit) MIPS-совместимого микропроцессора [54]. Вторая работа (которая тоже проводилась в 2006 г.) была посвящена модульной верификации буфера преобразования адресов (TLB, Translation Lookaside Buffer) [55]. Эти работы стали отправными точками дальнейших исследований и разработок в области верификации микропроцессоров, о которых рассказывается в этом разделе (следуя структуре статьи, сначала описываются средства модульной верификации, а потом — системной).

3.1. Разработки в области модульной верификации

При создании средств модульной верификации микропроцессоров и другой аппаратуры (на модульном уровне микропроцессорная специфика не заметна) нами использовался имеющийся задел в области тестирования программного обеспечения (ПО) — прежде всего, технология UniTESK [28], развиваемая в ИСП РАН с момента его основания в 1994 г. (технология является преемницей подхода KVEST [56]). Технология базируется на *контрактных спецификациях* в форме *пред-* и *постусловий* интерфейсных операций и на уникальных средствах генерации тестовых последовательностей, основанных на факторизации (обобщении) состояний тестируемой системы и обходе графа состояний с помощью *неизбыточных алгоритмов* [57, 58].

Контрактные спецификации и основанный на них процесс проектирования (Design by Contract) [59] были предложены Мейером в 1986 г. в результате прикладного развития идей Флойда, Хоара и Дейкстры 1960-х – 1970-х гг. о формализации программирования [60, 61, 62]. Компонент, предоставляя окружению некоторую операцию, указывает требования, которые должны быть выполнены окружением перед ее вызовом (*предусловие*); при выполнении заданных требований компонент, в свою очередь, гарантирует достижение определенного результата (*постусловие*). Спецификации такого типа широко используются в практике тестирования ПО, поскольку они привычны для разработчиков и позволяют автоматически строить тестовые оракулы.

3.1.1 Контрактные спецификации конвейера

Используя опыт тестирования ПО, прежде всего, реализаций коммуникационных протоколов (для которых, как и для аппаратуры,

характерны параллелизм и событийный характер поведения) [63], подход UniTESK был адаптирован для верификации HDL-моделей аппаратуры. Это потребовало уточнения понятия контракта. Классический контракт — это пара логических формул $\Phi = \langle \varphi, \psi \rangle$, определенных над множеством видимых переменных (параметров вызова операции, возвращаемого ей результата, глобальных переменных программы). Первая формула, φ, называется *предусловием*, а вторая, ψ, — *постусловием*. Контракт интерпретируется следующим образом: если непосредственно перед вызовом операции было выполнено предусловие φ, то непосредственно после выполнения операции должно быть выполнено постусловие ψ. Нарушение контракта трактуется как *ошибка*: нарушение предусловия — ошибка окружения, нарушение постусловия при выполненном предусловии — ошибка компонента.

В общем случае реализуемые аппаратурой операции не являются *атомарными* — это процессы, функционирующие в дискретном времени, которые в определенные моменты могут вступать во взаимодействие с окружением, выдавая наружу результаты выполнения элементарных действий — *микроопераций*. Следует обратить внимание на то, что процессы выполнения разных операций могут перекрываться по времени: для аппаратуры, ввиду физических особенностей ее реализации, характерен высокий уровень параллелизма, что на логическом уровне выражается в конвейерно-параллельной организации (см. рис. 4). Последнее обстоятельство определило выбор названия для предлагаемого метода формальной спецификации — *контрактные спецификации конвейера* [64] (другое название — *контрактные спецификации потактовой точности* [65]).

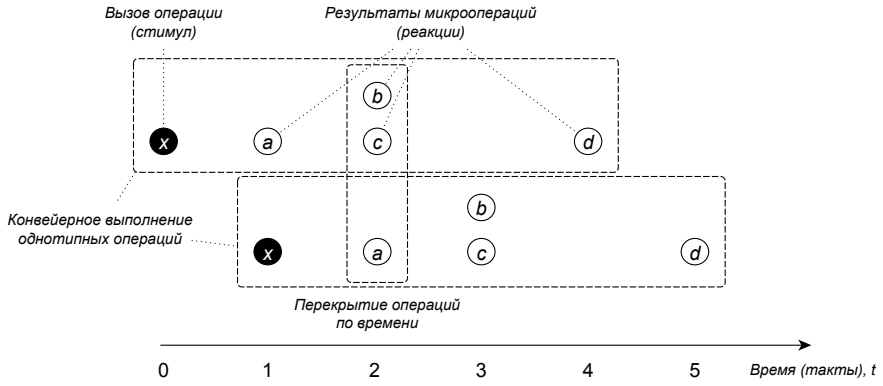

Рис. 4. Конвейерное выполнение двух однотипных операций

Для аппаратуры без внутренних блокировок контракт операции имеет вид $\Phi = \langle \varphi, \{(\psi_i, t_i)\}_{i=1,n} \rangle$, где φ — предусловие операции, ψ_i — постусловие i-ой микрооперации, t_i — время завершения выполнения i-ой микрооперации

(относительно времени вызова операции), n — общее число микроопераций [30]. Например, контракт операции x, представленной на рис. 4, имеет следующую структуру $\Phi_x = \langle \varphi_x, \{(\psi_a, 1), (\psi_b, 2), (\psi_c, 2), (\psi_d, 4)\} \rangle$. Семантику подобных контрактов можно определить в терминах логики LTL [33]. Так, контракт, приведенный выше, эквивалентен следующей формуле[4]: $\mathbf{G}(\phi_x \rightarrow \varphi_x) \wedge \mathbf{G}(\phi_x \rightarrow (\mathbf{X}\psi_a \wedge \mathbf{XX}(\psi_b \wedge \psi_c) \wedge \mathbf{XXXX}\psi_d))$, где ϕ_x обозначает факт вызова операции x. Другой способ определить семантику спецификаций — описать на их основе тестовый оракул. Пусть X — алфавит операций. Для $x \in X$ введем следующие обозначения: $\psi_x(t)$ — конъюнкция постусловий микроопераций, помеченных временно́й меткой t, T_x — время выполнения операции (время завершения последней микрооперации). Тестовый оракул работает синхронно с проверяемой системой и вызывается на каждом такте. Возможная организация выполняемой процедуры приведена в примере 1 (в начале работы $S = \varnothing$ и $v = true$; в случае ошибки в v заносится $false$).

Пример 1. Организация тестового оракула для операций без блокировок

```
01:  for x ∈ X do                                  проверка предусловий операций
02:    if φ_x ∧ ¬φ_x then v ← false
03:  for x ∈ X do                                  эмуляция вызова операций
04:    if φ_x then S ← S ∪ {(x, 0)}
05:  for (x, t) ∈ S do                             проверка постусловий микроопераций
06:    if ¬ψ_x(t) then v ← false
07:  S ← {(x, t+1) | (x, t) ∈ S ∧ t < T_x}         эмуляция такта работы системы
```

Если в аппаратуре возможны внутренние блокировки, формализация контракта операции немного усложняется. Предположим, что в рассматриваемом примере одноименные микрооперации, а также микрооперации a и b не могут выполняться одновременно (например, они используют общий ресурс), причем микрооперация a имеет больший приоритет по сравнению с b, а при других конфликтах приоритет получает та микрооперация, родительская операция которой была вызвана раньше. В такой ситуации конвейерное выполнение операций могло бы выглядеть, как показано на рис. 5. Формально блокировки конвейера могут быть описаны с помощью предусловий микроопераций, называемых также *охранными условиями* (*guards*) [64, 66]. Соответственно, контракт операции имеет вид

[4] Темпоральный оператор **G** (**G**lobally) требует, чтобы формула выполнялась всегда (начиная с текущего момента времени), **X** (ne**X**t time) — чтобы формула выполнялась в следующий момент времени.

$\Phi = \langle \varphi, \{(\gamma_i, \psi_i, t_i)\}_{i=1,n} \rangle$, где γ_i — охранное условие i-ой микрооперации. Охранные условия интерпретируются следующим образом: если для некоторой микрооперации завершились все предшествующие ей действия и выполнено охранное условие, в этот момент должны быть видимы результаты ее работы[5].

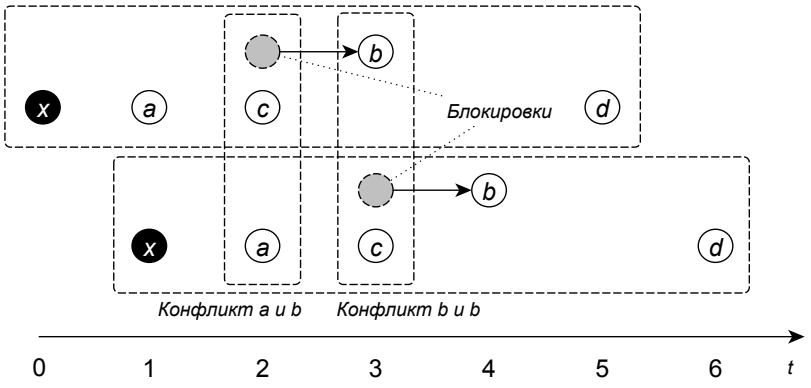

Рис. 5. Конфликты и блокировки при конвейерном выполнении операций

Выразить такие свойства с помощью LTL проблематично. Возможным вариантом видится замена подформул вида $\mathbf{X}\ldots\mathbf{X}\psi_i$ (используемых при описании операций без блокировок) на формулы[6] $\neg\gamma_i \mathbf{U} (\gamma_i \wedge \psi_i)$, однако и этот вариант не вполне адекватен — в нем не принимается во внимание то обстоятельство, что охранное условие микрооперации должно учитываться только после того, как завершились все предшествующие действия (вследствие этого возможны ложные обнаружения ошибок). В предположении, что у каждой выполняемой операции в любой момент времени в обработке находится не более одной микрооперации, семантику спецификаций можно выразить следующим алгоритмом (см. пример 2) работы тестового оракула (первые четыре строки полностью совпадают с описанием алгоритма для случая операций без блокировок).

[5] Возможна и другая интерпретация — результаты работы микрооперации становятся видимыми через такт после того, как охранное условие стало истинным.

[6] Бинарный темпоральный оператор **U** (**U**ntil) требует, чтобы первая формула была выполнена до тех пор, пока не станет истинной вторая формула (такой момент обязательно должен наступить).

Пример 2. Организация тестового оракула для операций с блокировками

…	…	…
05:	$E \leftarrow \{(x, t) \mid (x, t) \in S \wedge \gamma_x(t)\}$	вычисление множества активных микроопераций
06:	**for** $(x, t) \in E$ **do**	проверка постусловий активных микроопераций
07:	**if** $\neg\psi_x(t)$ **then** $v \leftarrow$ *false*	
08:	$S \leftarrow \{(x, t{+}1) \mid (x, t) \in E \wedge t < T_x\} \cup (S \backslash E)$	эмуляция такта работы системы

Контрактные спецификации могут быть расширены для операций со сложной структурой потока управления [67]. Поток управления описывается ориентированным графом, в котором возможны следующие типы вершин: начало и конец (*start* и *stop*), микрооперация (*stage*), ветвление и соединение потока управления (*switch* и *merge*), создание параллельных процессов и их соединение (*fork* и *join*). Семантика подобных блок-схем может быть естественным образом формализована. Не останавливаясь на этом вопросе подробно, отметим следующие моменты: (1) как и прежде, каждая микрооперация описывается парой $\langle\gamma, \psi\rangle$; (2) обработка всех типов вершин, за исключением *stage* и *join* (когда есть незавершившиеся процессы), осуществляется мгновенно (сразу после обработки микрооперации вычисляется множество следующих микроопераций); (3) для моделирования задержек можно использовать вершины типа *delay*, в которых указана величина задержки (вершина *delay* Δt раскрывается в цепочку из Δt микроопераций с тривиальным контрактом $\langle true, true\rangle$). Описание графа потока управления операции из рассматриваемого примера (см. рис. 4) приведено на рис. 6.

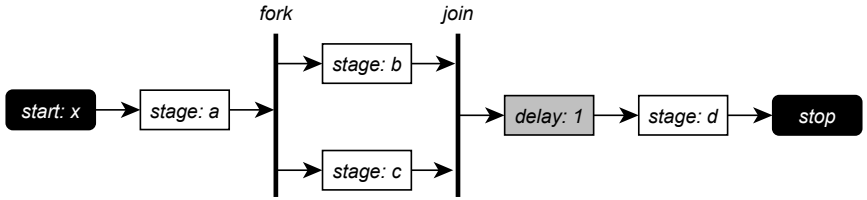

Рис. 6. Описание графа потока управления операции

Остановимся на важном вопросе — проверке выполнимости условий спецификации в процессе верификации. Вообще говоря, формулы, используемые в спецификациях, задают ограничения не только на интерфейсные сигналы, но и на состояние устройства. HDL-симуляторы предоставляют средства для доступа к внутренним переменным HDL-модели,

что позволяет определять условия на состояние и проверять их в процессе верификации. Однако сильная привязка спецификаций (и тестовой системы в целом) к реализации нежелательна с технологической точки зрения — изменения реализации (по крайней мере, незначительные) не должны приводить к изменению спецификации. Как правило, спецификация имеет свой набор переменных, значения которых либо синхронизируются с состоянием реализации (с помощью адаптера — см. раздел 2.1), либо изменяются в самой спецификации (для этого в нее добавляется исполнимая часть).

Чаще используется второй подход — каждая микрооперация описывается тройкой $\langle \gamma, \alpha, \psi \rangle$, где α — это *действие* (элементарная программа) [68]. Отметим также, что точное определение охранных условий γ (по существу, управляющей логики устройства) не всегда требуется — часто неважно, как устроен арбитр, управляющий доступом к некоторому ресурсу, главное, чтобы он был *безопасным* (*safe*) и *справедливым* (*fair*) — обеспечивал взаимное исключение процессов и рано или поздно давал доступ каждому желающему процессу[7]: $\mathbf{G}(|\{\gamma \in \Gamma_R \mid \gamma\}| \le 1)$ и $\mathbf{F}\gamma$ для всех $\gamma \in \Gamma_R$, где Γ_R — множество охранных условий микроопераций, желающих получить доступ к ресурсу R. Когда управляющая логика не специфицируется детально, охранные условия определяются через переменные HDL-модели (выходные или внутренние сигналы) — такие спецификации называются *адаптивными*, поскольку, используя обратную связь от реализации, они подстраиваются под ее поведение [69].

Контрактные спецификации конвейера можно использовать не только для проверки корректности поведения HDL-модели, но и для построения тестовой последовательности [65]. В подходе, описанном в [66], тестовая последовательность генерируется *на лету* (*on the fly*) на основе обхода состояний управления (множеств параллельно выполняемых микроопераций). Для этого используются *неизбыточные алгоритмы обхода графов*, позволяющие пошагово обходить сильно связные графы, располагая в каждый момент времени только информацией о текущей вершине (состоянии) и множестве исходящих из нее дуг (допустимых стимулов) [57, 58]. В большинстве проектов по верификации аппаратуры использовалась оптимизированная версия алгоритма α_{ndfsm}, ориентированного на графы, в которых имеется детерминированный (по стимулам) сильно-связный полный остовный подграф [70].

Заметим, что граф, состояниями которого являются множества параллельно выполняемых микроопераций, не обязан быть детерминированным (даже в смысле наличия детерминированного остовного подграфа). Для

[7] Темпоральный оператор **F** (in the **F**uture) требует, чтобы формула когда-нибудь выполнилась (начиная с текущего момента времени): $\mathbf{F}\gamma \equiv \mathit{true}\ \mathbf{U}\ \gamma$.

детерминизации обходимого графа был предложен следующий подход: в состояние обработки операции (часть состояния управления) добавляется информация о *ситуации*, возникающей при выполнении операции, и *зависимостях* операции от других обрабатываемых операций [65]. Ситуация задает путь в графе потока управления операции (между вершинами *start* и *stop*) и детерминизирует выбор очередной микрооперации, зависимости определяют конфликты между операциями и детерминизируют установку блокировок. Понятно, что для сложного устройства граф может быть огромным, а его обобщение может снизить качество верификации. Для решения этой проблемы был предложен метод параллельного обхода графа на компьютерном кластере (см. раздел 3.1.3).

Инструментальная поддержка разработки контрактных спецификаций конвейера и тестов на их основе реализована в виде библиотечного расширения PIPE (от англ. pipeline — конвейер) инструмента CTESK [52]. Библиотека разработана на языке SeC (Specification extension of C), используемом в инструменте CTESK, и не привязана к конкретному языку описания аппаратуры. Основные возможности библиотеки PIPE можно разбить на две группы: (1) описание структуры конвейера — композиция операций из множества микроопераций с использованием описанных выше примитивов (*switch*, *merge*, *fork*, *join* и других); (2) эмуляция поведения конвейера — потактовое моделирование конвейера, основанное на отслеживании множества параллельно выполняемых микроопераций, и выполнение проверок, определенных в постусловиях микроопераций. Для разработки тестовых сценариев и генерации тестов использовались стандартные средства инструмента CTESK. В примере 3 приведено описание графа потока управления операции, рассмотренной ранее (см. рис. 6), средствами библиотеки PIPE.

Пример 3. Описание операции с использованием примитивов PIPE

```
01:  specification void x(...) {                  описание операции x:
02:    pre { ... }                                описание предусловия операции
03:    coverage { ... }                           описание тестового покрытия
04:  }
05:  reaction Process* a(void) {                  описание микрооперации a:
06:    pre { ... }                                описание охранного условия микрооперации
07:    post { ... }                               описание постусловия микрооперации
08:  }
09:  Operation *op = create_Operation(x);         описание графа потока управления операции x:
10:  register_Stage (op, 0, a);                   вершина stage (метка 0, спецификация a)
```

11:	register_Fork	(op, 1, 3);	вершина *fork* (метка 1, следующие вершины 2 и 3)
12:	register_Stage	(op, 2, *b*);	вершина *stage* (метка 2, спецификация *b*)
13:	register_Edge	(op, 2, 4);	дуга между вершинами 2 и 4
14:	register_Stage	(op, 3, *c*);	вершина *stage* (метка 3, спецификация *c*)
15:	register_Join	(op, 4);	вершина *join* (метка 4)
16:	register_Delay	(op, 5, 1);	вершина *delay* (метка 5, задержка 1)
17:	register_Stage	(op, 6, *d*);	вершина *stage* (метка 6, спецификация *d*)
18:	register_Stop	(op, 7);	вершина *stop* (метка 7)

Предложенный метод и поддерживающий его инструмент (CTESK вместе с библиотекой PIPE) в 2007-2010 гг. использовались в нескольких проектах по верификации модулей микропроцессоров: буфера преобразования адресов TLB [55], модуля арифметики с плавающей точкой (FPU, Floating Point Unit)[8] [71], кэш-памяти второго уровня (L2) [72] и коммутатора северного моста (Databox) [67]. Во всех модулях были обнаружены ошибки (включая критические), которые были пропущены другими методами верификации (это не удивительно, поскольку детальность спецификаций и систематичность перебора состояний делают подход сравнимым по полноте верификации с формальными методами). Наиболее сложный модуль, проверенный описанным методом, — коммутатор северного моста (описание модуля составляет около 6 тысяч строк на языке Verilog; модуль реализует около 60 различных типов операций[9]; выполнение некоторых операций занимает около 40 тактов). Метод хорошо себя зарекомендовал для сравнительно небольших и детально задокументированных устройств; для сложных проектов со скудной документацией затраты на разработку и отладку потактовых спецификаций трудно оценить [73]. Трудоемкость верификации устройств средней сложности (3-6 тысяч строк кода) составляют 0.7-1.3 человеко-месяца на 1 тысячу строк кода [72].

Сравнивая контрактные спецификации конвейера с другими формализмами, используемыми для описания поведения аппаратуры, следует еще раз упомянуть темпоральную логику LTL и ее варианты. Как было показано ранее, логика LTL не способна адекватно описать некоторые свойства, которые простым образом описываются с помощью контрактных

[8] При верификации модуля арифметики с плавающей точкой основное внимание уделялось не описанию структуры конвейера, а генерации тестовых данных.

[9] Под типами операций здесь понимаются не только коды операций, но и различные способы выполнения (ветви функциональности).

спецификаций. В языках верификации аппаратуры, поддерживающих темпоральные утверждения (см. раздел 2.2), выразительные возможности LTL существенно расширены: можно обращаться к значениям переменных в прошлом и будущем, задавать временны́е интервалы наступления событий, описывать регулярные события, создавать параметризованные шаблоны утверждений, использовать в утверждениях присваивания [34]. Темпоральная логика такого вида позволяет описывать широкий диапазон свойств и активно используется в имитационной верификации, однако, как показывает опыт, инженеры не используют всех ее возможностей, а ограничиваются описанием простых причинно-следственных связей, которые, на наш взгляд, более естественно описываются в терминах контрактов и действий: $\langle \gamma, \alpha, \psi \rangle$ — «если γ, выполнить α и проверить ψ».

Заметим, что добавление в контракты действий приводит нас к классическому формализму — *охраняемым действиям* (*guarded actions*), введенному Дейкстрой в 1975 г. [74], который в последнее время активно используется для высокоуровнего описания аппаратуры с последующим логическим синтезом [75] (в этой связи следует упомянуть язык Bluespec [76]). Охраняемыми действиями называются пары $\langle \gamma, \alpha \rangle$ (в другой форме — $\gamma \rightarrow \alpha$), интерпретируемые следующим образом: когда истинно условие γ, выполняется действие α. Учитывая то, что в контрактных спецификациях конвейера условия и действия структурированы в графы (см. рис. 6), рассматриваемая форма спецификаций имеет много общего с *алгоритмическими машинами состояний* (*ASM*, *Algorithmic State Machines*) — диаграммами, применяемыми для описания и синтеза аппаратуры [77, 78]. В ASM, однако, нет проверок (постусловий микроопераций) и не поддерживается параллельное выполнение микроопераций (вершины типа *fork* и *join*).

3.1.2 Событийные спецификации аппаратуры

При применении метода верификации на основе потактово точных контрактных спецификаций в промышленной практике были выявлены его ограничения и слабые места [40]: (1) *высокие требования к качеству документации* — как правило, описание модулей не отличается полнотой (акцент делается на их структуре, а не на требованиях к поведению); тонкости работы устройства, необходимые для создания потактовых спецификаций, описываются разработчиками с трудом (часто происходит апелляция к исходному коду); (2) *медленный старт верификации* — сроки проведения верификации ограничены, а результаты (единственным значимым результатом на практике являются найденные ошибки) требуется получить как можно скорее; детальные спецификации позволяют провести тщательную проверку, но результаты могут быть получены слишком поздно; (3) *сложность интеграции в процессы проектирования* — разработка модулей микропроцессоров часто сопровождается созданием их обобщенных моделей

на языках высокого уровня, таких как C++ и SystemC; вместо разработки спецификаций с нуля целесообразно использовать имеющиеся наработки.

В результате анализа опыта промышленной верификации используемый подход был расширен (2011 г.). В теоретическом плане, расширение подхода связано с переходом от потактовой парадигмы к *событийной*, что позволяет использовать спецификации с неточной моделью времени (с упрощенной или отсутствующей управляющей логикой). Используемая техника проверки корректности поведения HDL-моделей основана на динамическом сопоставлении трасс (частично упорядоченных множеств событий) [79]: одна трасса порождается реализацией, вторая — спецификацией (эталонной моделью). Практическая сторона работ связана с переходом от языка SeC (используемом в CTESK расширения ANSI C) к языку программирования C++, что дает возможность использовать в составе тестовых систем модели устройств (компоненты симулятора), создаваемые в процессе архитектурного проектирования (см. раздел 2).

Наблюдаемое поведение HDL-модели описывается *временны́м словом* — последовательностью вида $\{(a_i, t_i)\}_{i \geq 0}$, где a_i — событие, а t_i — время его наступления. События, происходящие в одно и то же время, допустимы, но должны возникать на разных интерфейсах устройства. Поведение эталонной модели задается *временно́й трассой* (или просто *трассой*) — частично упорядоченным множеством пар (a, t). Частичный порядок описывает причинно-следственные связи между событиями и другие ограничения, связанные с упорядоченностью. Заметим, что временны́е слова являются частной разновидностью трасс — трассами с тривиальным порядком (по сути, это означает отсутствие информации о взаимосвязях между событиями). При использовании неточных эталонных моделей трассы обобщаются — вместо временны́х меток указываются *временны́е интервалы*, задающие ограничения на времена возникновения событий в реализации [80]. Пример показан на рис. 7: наблюдаемое поведение верифицируемой HDL-модели описывается трассой $\{(b, 1), (a, 2), (c, 3), (d, 5)\}$, поведение эталонной модели — $\{(a, 1), (b, 2), (c, 2), (d, 3)\}$; при обобщении эталонной трассы временны́е метки расширяются до интервалов, но, вместе с тем, в трассу добавляется информация о зависимостях между событиями.

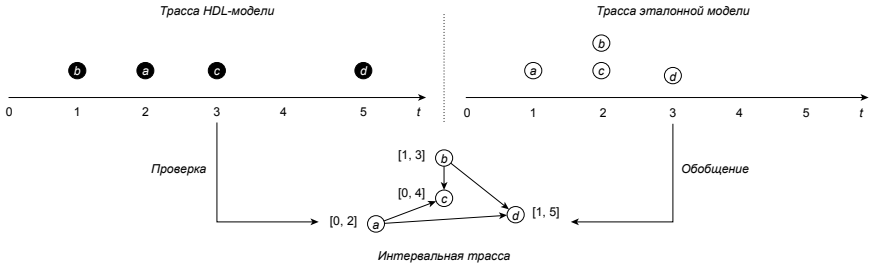

Рис. 7. Схема проверки соответствия между HDL-моделью и эталонной моделью

В общих словах, суть выполняемой проверки поведения следующая: трасса реализации (HDL-модели) должна быть линеаризацией трассы спецификации (эталонной модели), а временны́е метки событий реализации не должны выходить за рамки интервалов, заданных в соответствующих событиях спецификации (формальное определение можно найти в работе [79]). Важно отметить, что проверка соответствия трасс выполняется в динамике, а не после завершения верификации. Рис. 8 иллюстрирует проверку корректности поведения на примере трасс, показанных на рис. 7 (рассматривается момент времени $t = 4$). В верхней части изображены реакции HDL-модели, в нижней — реакции эталонной модели; отношение порядка между реакциями эталонной модели показано стрелками; пунктирные линии соединяют сопоставленные реакции.

Тестовый оракул, осуществляющий сопоставление трасс, имеет два типа входов: (1) реакции HDL-модели и (2) реакции эталонной модели. На каждом такте тестовый оракул выполняет следующие действия: (1) *прием реакций* — полученные реакции спецификации и реализации, X' и Y', заносятся в соответствующие частично упорядоченные множества событий, X и Y: $X \leftarrow X \cup X'$, $Y \leftarrow Y \cup Y'$; из обоих множеств удаляется пересечение множеств минимальных (по отношению порядка) элементов[10]:

$$X \leftarrow X \setminus (min(X) \cap min(Y)),\ Y \leftarrow Y \setminus (min(X) \cap min(Y))$$

(2) *проверка превышения лимита времени* — если время нахождения реакции в соответствующем множестве превышает заданную величину, фиксируется ошибка: *пропущенная реакция* (для реакции спецификации) или *неожиданная реакция* (для реакции реализации).

[10] Как отмечалось выше, поведение реализации (как правило) моделируется трассой с тривиальным порядком (порядком, задаваемым отношением равенства), для которого все события являются минимальными.

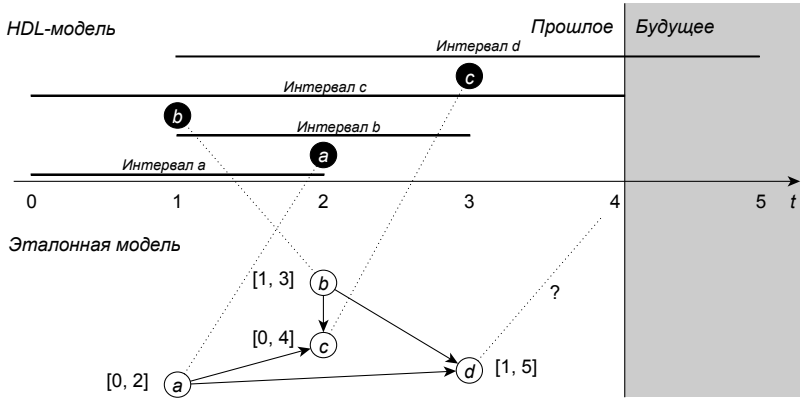

Рис. 8. Соответствие между HDL-моделью и эталонной моделью

Упрощенное описание алгоритма работы оракула на каждом такте верификации приведено в примере 4 (в начале работы $X = Y = \varnothing$ и $v = true$; в случае ошибки в v заносится *false*). Более детальное и формализованное описание можно найти в работе [79].

Пример 4. Примерный алгоритм работы тестового оракула

```
01:  receive X', Y'                          прием множеств реакций на текущем такте
02:  for (x, t) ∈ X' do                      сопоставление реакции эталонной модели
03:     if (x, t) ∉ Y then X ← (x, t)        (несопоставленные реакции откладываются)
04:  for (y, t) ∈ Y' do                      сопоставление реакции HDL-модели
05:     if (y, t) ∉ X then Y ← (y, t)        (несопоставленные реакции откладываются)
06:  for (x, t) ∈ X do                       если для реакции эталонной модели превышен лимит времени,
07:     if timeout(x, t) then v ← false      фиксируется ошибка: пропущенная реакция
08:  for (y, t) ∈ Y do                       если для реакции HDL-модели превышен лимит времени,
09:     if timeout(y, t) then v ← false      фиксируется ошибка: неожиданная реакция
```


Рассмотренный подход к проверке корректности поведения HDL-моделей был реализован в инструменте C++TESK [81]. Отношение порядка между реакциями эталонной модели может быть задано двумя способами: (1) непосредственно в эталонной модели и (2) с помощью *арбитров реакций* [40]. Арбитр реакций задается для каждого выходного интерфейса и по запросу тестовой системы выдает очередную реакцию эталонной модели. Арбитры делятся на три типа: (1) *модельные арбитры* (которые при выборе реакции ориентируются только на порядок их выдачи эталонной моделью), (2) *адаптивные арбитры* (которые осуществляют выбор реакции эталонной модели путем сопоставления с заданной реакцией реализации) и (3) *двухуровневые арбитры* (которые работают в два этапа: сначала строят множество возможных реакций эталонной модели, используя модельный арбитр (множество минимальных по отношению порядка реакций), а затем применяют к полученному множеству адаптивный арбитр).

В основе C++TESK лежит библиотека классов, обеспечивающая разработку тестовых систем для HDL-моделей аппаратуры на уровне TLM (см. раздел 2.2). Средства инструмента позволяют создавать такие компоненты, как эталонные модели, адаптеры, тестовые сценарии, сборщики тестового покрытия и другие (см. рис. 2). Использование для создания тестовых систем языка программирования C++ позволяет без труда встраивать в них сторонние эталонные модели, разработанные на этом языке. В примере 5 показано описание операции, изображенной на рис. 6, средствами C++TESK. Обратим внимание, что процессы могут быть запущены как параллельным (*PARALLEL*), так и последовательным образом (*SEQUENTIAL*). При запуске процесса указывается интерфейс ($iface_a$, $iface_b$, $iface_c$, $iface_d$) и входное сообщение для обработки (msg_a, msg_b, msg_c, msg_d).

Пример 5. Описание операции с использованием примитивов C++TESK

```
01:  DEFINE_STIMULUS (x) {                   описание операции x:
02:     START_STIMULUS();                    начало операции
03:     CALL_PROCESS(SEQUENTIAL, a, iface_a, msg_a);   запуск микрооперации a
04:     CALL_PROCESS(SEQUENTIAL, b, iface_b, msg_b);   запуск микрооперации b
05:     CALL_PROCESS(PARALLEL, c, iface_c, msg_c);     запуск микрооперации c (параллельно b)
06:     CYCLE();                             задержка в один такт
07:     CALL_PROCESS(SEQUENTIAL, d, iface_d, msg_d);   запуск микрооперации d
08:     STOP_STIMULUS();                     завершение операции
09:  }                                       описание операции x:
```

Инструмент C++TESK в 2011-2013 гг. использовался в ряде проектов по верификации модулей микропроцессоров [40, 79, 73]: межпроцессорного коммутатора данных (MAU Hub, Memory Access Unit's Hub), системного контроллера прерываний (SAPIC, System Advanced Programmable Interrupt Controller), модуля поиска по таблице страниц (TLU, Table Lookup Unit), коммутатора северного моста (Databox), буфера команд (IB, Instruction Buffer), подсистем кэширования 2-ого и 3-ого уровней (L2 и L3). Предложенный метод позволил создать эталонные модели для модулей средней сложности без наличия детальной документации и найти большое количество сложных ошибок. Трудоемкость разработки тестовой системы в среднем оценивается в 1 человеко-месяц на 1 тысячу строк кода HDL-описания (причем работа по созданию эталонной модели трудно распараллеливается). Метод применим для верификации модулей средней сложности (до 10 тысяч строк кода) и модулей большой сложности (10-50 тысяч строк кода), но допускающих абстракцию управляющей логики. Для более сложных подсистем целесообразнее использовать верификацию с помощью тестовых программ (см. раздел 3.2).

Рассмотрим подходы, близкие к предлагаемому методу. В работе [82] используется модель *автомата с частично упорядоченными входными/выходными событиями* (*POIOA*, *Partial Order Input/Output Automaton*) для представления поведения эталонной модели и реализации. Если (1) реализация сообщает о приеме неподдерживаемых стимулов, (2) можно установить порядок выдачи реакций реализацией, (3) время ответа реализации ограничено, и (4) каждый единичный переход в эталонной модели соответствует единичному переходу в реализации, можно установить соответствие между реализацией и эталонной моделью: они соответствуют друг другу, если реализация принимает допустимые эталонной моделью стимулы и выдает описываемые ею реакции в допустимом порядке. Данный подход к определению соответствия схож с предложенным нами, но отличается отсутствием контроля временны́х интервалов. В работе [63] рассматривается подход к тестированию параллельных систем с помощью неявно заданных *асинхронных конечных автоматов* (*AFSM*, *Asynchronous Finite State Machines*). Поведение реализации проверяется только в стационарных состояниях, в которых не ожидается выдача реакций: (1) собираются все события, и определяется их частичный порядок; (2) строятся и проверяются все возможные линеаризации событий. Считается, что реализация соответствует эталонной модели, если допускается хотя бы одна линеаризация. Данный подход, в отличие от предложенного метода, не может использоваться с произвольными генераторами стимулов, поскольку в нем требуется, чтобы в процессе тестирования регулярно возникали стационарные состояния, но, с другой стороны, он применим в том случае, когда не известен точный порядок выдачи реакций реализацией.

3.1.3 Распараллеливание процесса верификации

Способ генерации тестовых последовательностей, используемый в технологии UniTESK [28], основан на обходе графа состояний тестируемой системы. Отличительной чертой технологии является то, что граф состояний задается неявно (с помощью функции построения текущего состояния и множества допустимых стимулов) и строится по мере его обхода [57, 58]. При проходе по дуге графа осуществляется подача стимула на тестируемую систему. Обход графа завершается после прохождения всех дуг (когда обнаружено, что ни из одной известной вершины не выходит непройденная дуга) или обнаружения ошибки (при невыполнении одного из постусловий). Для тестирования HDL-моделей аппаратуры пространство состояний задавалось множествами параллельно выполняемых микроопераций (см. раздел 3.1.1). Такой способ позволяет добиться высоких показателей тестового покрытия (см. раздел 2.2), однако получаемые графы могут быть очень большими. Так, для системного контроллера прерываний SAPIC (простого устройства, HDL-описание которого содержит около 2 тысяч строк кода) граф состояний (для одного из тестовых сценариев[11]) содержал около 100 тысяч вершин и 500 тысяч дуг, а для модели коммутатора северного моста Databox (около 6 тысяч строк кода) — около 500 тысяч состояний и 2 миллионов дуг. Размер графов, используемых для генерации тестовых последовательностей, может быть уменьшен с помощью техник сокращения [83], однако это может снизить качество верификации.

Чтобы ускорить процесс тестирования (обход больших графов состояний) были использованы возможности, предоставляемые распределенными вычислительными системами (компьютерными кластерами). Был разработан метод, позволяющий динамически распараллеливать обход графа, не располагая при этом никакой информацией о его структуре до начала тестирования [84]. Реализация метода потребовала расширения классической архитектуры тестовой системы. Это было сделано следующим образом. На каждом компьютере кластера выполняется свой процесс тестовой системы. Процесс осуществляет обход графа, а полученной информацией о пройденной части графа обменивается с процессами на других компьютерах кластера (заметим, что разные процессы могут использовать разные алгоритмы обхода). Архитектура процесса распределенной тестовой системы представлена на рис. 9 (для наглядности такие компоненты, как тестовый оракул и адаптер, опущены).

Процесс включает в себя следующие компоненты: (1) *реализация алгоритма обхода* (*обходчик*), (2) *представление графа состояний* (*хранилище*),

[11] Каждый тестовый сценарий предназначен для проверки определенного аспекта тестируемой системы; получаемый в результате граф состояний обычно существенно меньше совокупного графа, моделирующего систему.

(3) *синхронизатор* и (3) *генератор стимулов*. Обходчик — это библиотечный компонент, который по запросу строит маршрут из текущей вершины графа в вершину, из которой выходят еще не пройденные дуги, и выбирает одну из них. Хранилище отвечает за представление пройденной части графа (посещенных вершин, допустимых в них стимулов, пройденных дуг). Синхронизатор получает от обходчика информацию о новых пройденных дугах и рассылает ее синхронизаторам других процессов; он также получает информацию о дугах, пройденных другими процессами, и передает ее своему обходчику (который добавляет информацию в хранилище). Генератор стимулов является активной частью тестовой системы, использующей обходчик для построения путей в графе и тестовый сценарий для вычисления текущего состояния и определения допустимых в нем стимулов. Более подробно архитектура тестовой системы и протокол синхронизации процессов описаны в работе [84].

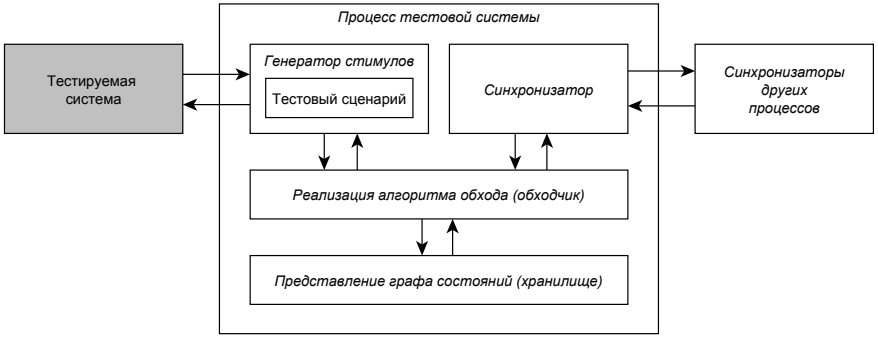

Рис. 9. Архитектура процесса распределенной тестовой системы

Предложенный метод реализован в рамках проекта CTESK [52] (в настоящее время разработанные средства также доступны в инструменте C++TESK [81]). Для управления процессом тестирования разработан *координатор*, позволяющий через Web-интерфейс задавать такие параметры, как используемые для тестирования компьютеры, топологию связей между ними (граф обмена информацией) и другие. Координатор запускает процессы тестовой системы и собирает информацию о состоянии тестирования (размере пройденного графа, числе найденных ошибках и достигнутом уровне тестового покрытия). Так как обход графа может занимать продолжительное время, система поддерживает возможность сохранения и последующего восстановления пройденной части графа. Это позволяет при необходимости прервать тестирование или же продолжить его после сбоя без потери полученных результатов.

В 2010-2012 гг. подход применялся при верификации нескольких модулей микропроцессоров: коммутатора северного моста Databox, системного

контроллера прерываний SAPIC и межпроцессорного коммутатора данных MAU Hub. Размеры графов доходили до нескольких миллионов вершин. Использование описанного подхода позволило существенно снизить время выполнения тестов, коэффициент эффективности распараллеливания превосходил 0.8 (то есть при использовании, например, 100 компьютеров время выполнения тестов уменьшалось более чем в 80 раз) [85]. Существующая реализация имеет ограничение — каждый процесс тестовой системы хранит совокупный граф состояний, между тем, графы могут быть настолько велики, что не помещаются в оперативной памяти компьютера (а при использовании внешней памяти сильно возрастает время построения путей). Одно из возможных решений этой проблемы предлагается в статье [86] — способ параллельного обхода, позволяющий избежать дублирования информации о пройденной части графа в процессах, распределяя ее между ними.

3.2. Разработки в области системной верификации

Если при создании методологии модульной верификации микропроцессоров мы отталкивались от опыта разработки и применения технологии тестирования UniTESK [28], то, столкнувшись в 2006 г. с новой для нас задачей системной верификации (с задачей генерации тестовых программ на языке ассемблера для одноядерного MIPS-совместимого микропроцессора), выбор метода решения был неочевиден. Наиболее близкими подходами, разработанными в ИСП РАН, представлялись методы автоматического построения тестов (программ на языках высокого уровня) для проверки синтаксических анализаторов [87] и оптимизирующих компиляторов [88]. Эти методы основаны на синтаксических моделях: в первом из них используется грамматика языка, во втором — редуцированная грамматика, построенная с учетом алгоритма оптимизации.

В отличие от языков высокого уровня язык ассемблера имеет бедный, нерекурсивный синтаксис, что снижает значимость синтаксически-ориентированных методов (хотя в некоторой форме они используются для генерации ассемблерных программ [89] и могут быть полезны для построения графов потока управления программ [90] и композиции сложных программ из простых [91]). Следует также отметить, что методы, описанные в [87, 88] не учитывают семантику грамматических конструкций — между тем, это важно для генерации *тестовых программ со встроенными проверками* (см. раздел 2.3). На основе анализа большого числа методов и инструментов генерации тестовых программ для микропроцессоров (см., например, [42]) в качестве спецификаций были выбраны *модели уровня системы команд*.

3.2.1 Комбинаторная генерация тестовых программ

Модели уровня системы команд описывают микропроцессор в форме переменных (регистров и памяти) и команд — атомарных действий над

переменными (для которых также указываются предусловия). В отличие от детальных моделей, используемых для модульной верификации микропроцессоров, такие модели абстрагируются от структуры конвейера и совместного выполнения команд на нем. Основное назначение моделей уровня системы команд — формализация семантики команд и формирование программистского взгляда на микропроцессор (программист, составляя программу, считает команды атомарными — выполнение последовательности команд равносильно их последовательному выполнению одна за другой). Описание семантики команд на псевдокоде можно встретить в большинстве руководств по архитектуре микропроцессоров. В примере 6 приведено описание команды *ADD* (целочисленное сложение) из руководства по системе команд MIPS [92]. Команда имеет формат *ADD rd*, *rs*, *rt*, где *rs*, *rt* и *rd* — это номера регистров общего назначения (GPR, General-Purpose Registers).

Пример 6. Описание команды ADD архитектуры MIPS [92]

01:	**if** NotWordValue(GPR[rs]) **or** NotWordValue(GPR[rt]) **then**	проверка предусловия команды:
02:	**UNPREDICTABLE**	если предусловие не выполнено,
03:	**endif**	результат команды непредсказуем
04:	temp ← ($GPR[rs]_{31}$‖$GPR[rs]_{31..0}$) + ($GPR[rt]_{31}$‖$GPR[rt]_{31..0}$)	описание производимых действий:
05:	**if** $temp_{32} \neq temp_{31}$ **then**	если возникает переполнение,
06:	SignalException(IntegerOverflow)	Генерируется исключение,
07:	**else**	в противном случае
08:	GPR[rd] ← sign_extend($temp_{31..0}$)	записывается в регистр *r*
09:	**endif**	проверка предусловия команды:

На основе анализа спецификации команды можно выделить набор *тестовых ситуаций* — ограничений на значения операндов команды и состояние микропроцессора, характеризующих различные способы выполнения этой команды. Как правило, тестовая ситуация соответствует одному из возможных путей в графе потока управления спецификации. В приведенном выше примере можно идентифицировать две тестовые ситуации: *целочисленное переполнение* ($temp_{32} \neq temp_{31}$) и *нормальное выполнение* ($temp_{32} = temp_{31}$), где $temp = (GPR[rs]_{31} \parallel GPR[rs]_{31..0}) + (GPR[rt]_{31} \parallel GPR[rt]_{31..0})$ (символ ‖ обозначает конкатенацию битовых векторов, а нижние индексы используются для выделения подмассива). Покрытие всех тестовых ситуаций реализуемых микропроцессором команд является необходимым, но не достаточным условием качественной системной верификации.

Другим типом информации, полезным для верификации микропроцессора, является информация о *зависимостях* между командами, характеризующая совместное выполнение команд на конвейере и, прежде всего, конфликты использования ресурсов (см. рис. 5). Обычно подобные сведения (за исключением очевидных зависимостей по регистрам) не отражаются в моделях уровня системы команд (и руководствах по системам команд) — их источником выступают спецификации конкретных микропроцессоров. В качестве примера рассмотрим следующую зависимость между командами обращения в память — попадание в разные строки кэш-памяти L1, расположенные по одному индексу (в одном множестве). Зависимость определяется организацией кэширования. В микропроцессорах семейства RM7000 кэширование осуществляется по физическим 36-битным адресам: биты 2..0 адреса определяют позицию байта внутри двойного слова, биты 4..3 — позицию двойного слова в строке кэш-памяти, биты 11..5 — индекс строки и, наконец, биты 35..12 — тэг, используемый для проверки попадания данных в кэш-память [93] (см. пример 7).

Пример 7. Формат физического адреса в микропроцессоре RM7000 [93]

Тэг данных		***Индекс строки L1***		***Номер двойного слова***		***Номер байта***	
35	12	11	5	4	3	2	0

Таким образом, две команды связаны зависимостью данного типа, если для физических адресов, по которым происходит реальное обращение в память (обозначим их *x* и *y*), выполнено ограничение $(x_{11..5} = y_{11..5}) \wedge (x_{35..12} \neq y_{35..12})$. Виртуальные 40-битные адреса (обозначим их *a* и *b*) должны быть подобраны соответствующим образом — если размер страницы виртуальной памяти составляет 4 килобайта (в этом случае биты 11..0 виртуального адреса задают смещение, а биты 39..12 — номер страницы), должно выполняться условие $(a_{11..5} = b_{11..5}) \wedge (a_{39..12} \neq b_{39..12})$, причем $map(a_{39..12}) \neq map(b_{39..12})$, где *map* — отображение страниц виртуальной памяти в страницы физической памяти, осуществляемое с помощью буфера трансляции адресов. Более детально различные типы зависимостей по физическим и виртуальным адресам рассмотрены в статье [54].

Идея предложенного подхода к построению тестовых программ, названного *комбинаторной генерацией на основе моделей* [94], основана на двух предположениях: (1) поведение микропроцессора определяется множеством параллельно выполняемых команд (загрузкой конвейера), зависимостями между командами (характером обращений к ресурсам микропроцессора), а также ситуациями, возникающими при выполнении команд (путями выполнения команд в конвейере); (2) ошибки (в RISC-процессорах) могут быть обнаружены с помощью небольших тестовых примеров (состоящих из 2-5 команд) [42, 94]. В простейшем случае генерация тестовых программ

осуществляется путем систематического перебора коротких последовательностей команд (включая перебор тестовых ситуаций для отдельных команд и зависимостей между парами команд). Для сокращения перебора могут использоваться эвристики (объединение команд в классы эквивалентности, ограничение числа зависимостей и другие).

Подход также позволяет комбинировать простые тестовые шаблоны (последовательности команд с заданными тестовыми ситуациями и зависимостями без указания конкретных значений операндов) в сложные [91]. Целью комбинирования является создание нетривиальных ситуаций в работе микропроцессора (одновременных исключений, параллельных конфликтов и других). Составление сложных тестовых шаблонов из простых может осуществляться с помощью следующих операций: *наложение* ($T_1 \parallel T_2$ — объединение множеств тестовых ситуаций и зависимостей у однотипных шаблонов), *сдвиг* ($T_1 >> T_2$ — соединение шаблонов без отождествления команд со сдвигом одного относительно другого), *конкатенация* ($T_1 \cdot T_2$ — последовательное расположение шаблонов друг за другом) и *вложение* ($T_1[T_2]$ — помещение одного шаблона внутрь другого). Генерация тестовых программ для фиксированного набора базовых шаблонов осуществляется путем перебора синтаксических деревьев ограниченного размера, описывающих составные шаблоны. В примере 8 приведен результат комбинирования простых тестовых шаблонов с помощью указанных операций ($T_1[T_2 >> T_3] \cdot T_4$) [95].

Пример 8. Результат комбинирования тестовых шаблонов

01:	LD ?, ?(?) @ hit<TLB>(PFN1)	шаблон T_1 (зависимость по адресам)
02:	ADD r, ?, ?	шаблон T_2 (зависимость по регистрам)
03:	DIV.S ?, ?, ?	шаблон T_3 (конфликт между командами)
04:	SUB ?, r, ?	шаблон T_2
05:	DIV.D ?, ?, ?	шаблон T_3
06:	SD ?, ?(?) @ hit<TLB>(PFN2) && PFN1 = PFN2	шаблон T_1
07:	FPU ?, ?, ? @ Overflow	шаблон T_4 (исключение

Отдельного упоминания заслуживает подход к генерации тестовых программ, содержащих команды ветвления. Как минимум, такие программы не должны зацикливаться. Построение тестовых программ осуществляется перебором разнообразных графов потока управления (*структур переходов*) и маршрутов в них (*трасс выполнения*) [90]. Перебор трасс выполнения для заданной структуры переходов основан на поиске в глубину в графе потока управления. Чтобы программа выполнялась согласно заданной построенной трассе, в нее

встраиваются вспомогательные команды, называемые *управляющими*. Представленный ниже пример 9 содержит команду перехода *BEQ* (переход по равенству), которая выполняется два раза (когда она выполняется первый раз, условие перехода истинно; затем условие делается ложным, и выполнение теста завершается). Управляющая команда (*ADDI*) инкрементирует один из регистров, используемых в команде перехода (вначале значения регистров равны).

Пример 9. Тестовый шаблон, содержащий команды ветвления

01:	INIT:	инициализация регистров
02:	ORI r1, r0, 2014	r1 = 2014
03:	ORI r2, r0, 2014	r2 = 2014
04:	...	
05:	J START	переход на начальную метку
06:	...	
07:	BACK:	промежуточная метка
08:	ADDI r1, r1, 1	управляющая команда
09:	FPU ?, ?, ?	команда тестового шаблона
10:	START:	начальная метка
11:	BEQ r1, r2, BACK @ trace = {true, false}	условный переход по равенству
12:	LD ?, ?(?)	команда тестового шаблона
13:	STOP:	инициализация регистров

Предложенный метод был реализован в генераторе тестовых программ для микропроцессоров MicroTESK[12] [47] (первоначально инструмент был назван TestFusion 4M [94]). Генератор реализован на языке программирования Java и состоит из трех основных компонентов: (1) *библиотека моделирования* — средства высокоуровневого описания микропроцессоров и реализуемых ими команд; (2) *библиотека тестирования* — средства разработки генераторов тестовых данных и тестовых последовательностей; (3) *графическая оболочка* — пользовательский интерфейс, позволяющий задавать параметры генерации тестовых программ. Следует отметить, что в MicroTESK тестовые ситуации и

[12] В этом разделе речь идет о генераторе MicroTESK версии 1.0 (2006-2008 гг.); в текущей версии инструмента (версии 2.0) реализованы дополнительные возможности, которые рассматриваются в разделе 3.2.2.

зависимости описываются конструктивно — для каждого типа тестовой ситуации (и зависимости) разрабатывается генератор, конструирующий случайные тестовые данные, удовлетворяющие заданной ситуации. В примере 10 приведен фрагмент описания команды *ADD* архитектуры MIPS (см. пример 6).

Пример 10. Описание команды ADD с использованием примитивов MicroTESK

```
01:  public class ADD extends Instruction {                  класс, описывающий команду ADD
02:    public ADD() {                                        конструктор класса команды:
03:      rd = add(this, OUT, GPR, WORD);                     добавление операндов
04:      …
05:    }
06:    public void execute(Processor processor) {            метод, эмулирующий выполнение команды:
07:      Data result = add(rs.getWord(), rt.getWord());      код соответствует спецификации команды (см. пример 6)
08:      if(isOverflow32(result))
09:        { setException(new IntegerOverflow()); }
10:      else
11:        { rd.setWord(result); }
12:    }
13:    public String toString() {                            метод вывода команды в формате
14:      return format("ADD %s, %s, %s", rd, rs, rt);        языка ассемблера
15:    }}
```

Предложенный метод и генератор тестовых программ MicroTESK в 2006-2010 гг. использовались в нескольких проектах по верификации микропроцессоров и их компонентов: подсистема управления памятью (MMU, Memory Management Unit) [42, 54], MIPS-совместимый микропроцессор [42], два арифметических сопроцессора (CP1, CP2) [91]. Во всех указанных проектах были обнаружены ошибки, которые были пропущены другими

методами верификации. Наиболее сложное устройство, проверенное описанным методом, — MIPS-совместимый микропроцессор (одноядерный микропроцессор, реализующий более 200 команд; имеет в своем составе буфер трансляции адресов и двухуровневую кэш-память). При использовании метода не было выявлено ограничений на сложность верифицируемых систем (в классе одноядерных микропроцессоров). Трудоемкость разработки генератора тестовых программ определяется числом команд микропроцессора и составляет приблизительно 0.8 человеко-дня на одну команду (судя по всему, этот показатель может быть снижен за счет использования специализированных ADL-языков). Другие подходы к генерации тестовых программ рассмотрены в разделе 2.3.

3.2.2 Генерация тестовых программ на основе шаблонов

Одна из проблем, которая возникает при разработке нового микропроцессора, состоит в том, что имеющиеся генераторы тестовых программ сложно адаптировать к его архитектуре. Для этого приходится вручную модифицировать логику генерации, что приводит к дополнительным затратам ресурсов и чревато внесением ошибок. Генераторы, основанные на моделях, такие как Genesys-Pro [41], решают эту проблему, выделяя функции генерации в архитектурно-независимые компоненты. Адаптация генератора осуществляется путем изменения модели, инкапсулирующей архитектурно-зависимую информацию. Для того чтобы облегчить создание модели нами был предложен подход [96], состоящий в использовании ADL-языков [19].

По характеру описываемых свойств ADL-языки принято разделять на две группы: *структурные* и *поведенческие*. Языки первой группы ориентированы на спецификацию микроархитектуры (примером такого языка является MIMOLA). Вторая группа концентрируется на описании команд микропроцессора (к этой категории можно отнести ISDL и nML). Кроме того, существуют *смешанные языки*, сочетающие возможности языков обеих групп (например, LISA и EXPRESSION). Для спецификации макроархитектуры наиболее подходят поведенческие ADL-языки, позволяющие явно описывать синтаксис и семантику команд. На основе этой информации может быть построена обобщенная модель микропроцессора, которую можно использовать для настройки генератора под конкретную архитектуру. Из нескольких поведенческих и смешанных ADL-языков, таких как LISA, EXPRESSION, ISDL и nML [19], был выбран последний. Данный язык обладает интуитивно понятным синтаксисом, имеет доступную документацию и используется в нескольких зарубежных университетах и коммерческих компаниях.

Язык nML [97] был разработан в начале 1990-х гг. в Берлинском техническом университете (Technische Universität Berlin) и изначально предназначался для создания симуляторов микропроцессоров и настраиваемых компиляторов. С конца 1990-х гг. nML активно используется Target Compiler Technologies [98]

для решения таких задач, как генерация симуляторов, компиляторов и HDL-моделей. Эта компания расширила язык nML средствами моделирования конвейера [98, 19]. Другая версия языка nML была разработана в Индийском технологическом институте Каппур (Indian Institute of Technology Kanpur) и получила название Sim-nML [99]. В язык были добавлены средства описания ресурсов микропроцессора. В Исследовательском институте информатики в Тулузе (Institut de Recherche en Informatique de Toulouse) для создания симуляторов микропроцессоров применяется язык nMP, основанный на Sim-nML [100, 101]. Среда генерации тестовых программ MicroTESK (версии 2.0) [47], разрабатываемая в ИСП РАН с 2010 г., использует nML для спецификации архитектуры верифицируемого микропроцессора.

Пример 11. Описание команды ADD на языке nML

```
01:  op ADD (rd : index, rs :  REG, rt : REG)                    команда ADD:
02:     syntax = format("ADD %d, %s, %s"                         ассемблерный
     rd, rs.syntax, rt.syntax)                                   формат
03:     image = format("000000%s%s%5b00000100000",               бинарный
     rs.image, rt.image, rd)                                     формат
04:     action = {                                               описание
                                                                 семантики
05:        temp = rs<31..31>::rs<31..0> + rt<31..31>::rt<31..0>;
06:        if (temp<32..32> != temp<31..31>) then
07:           SignalException("Integer Overflow Exception");
08:        else
09:           GPR[rd] = temp<31..0>;
10:        endif;
11:     }
12:  mode REG (r : index) = if (r == 0) then 0 else              режим
     GPR[r] endif                                                адресации
                                                                 REG:
13:     syntax = format("%d", r)                                 ассемблерный
                                                                 формат
14:     image = format("%5b", r                                  бинарный
                                                                 формат
```

Спецификация на языке nML включает описание следующих сущностей: (1) *элементов хранения данных* (памяти, регистров, внутренних переменных), определяющих состояние микропроцессора; (2) *команд*, выполнение которых изменяет это состояние; (3) *режимов адресации*, реализующих абстракцию

доступа к элементам хранения данных. Пример 11 содержит спецификацию команды *ADD* архитектуры MIPS (см. пример 6) и спецификацию режима адресации *REG*, который она использует для чтения данных из регистров общего назначения.

Архитектура среды генерации тестовых программ MicroTESK показана на рис. 10. Среда включает в себя *транслятор*, который преобразует спецификацию на языке nML в исполнимую *модель микропроцессора* и *модель покрытия*. Модель микропроцессора строится на основе *библиотеки моделирования* и реализует следующие функции: (1) предоставление *мета-информации,* описывающей элементы микропроцессора (регистры, память, команды и т.д.); (2) *симуляция* команд микропроцессора; (3) *мониторинг* состояния элементов памяти модели. Модель покрытия содержит информацию о ситуациях, которые могут возникнуть в процессе выполнения команд [102].

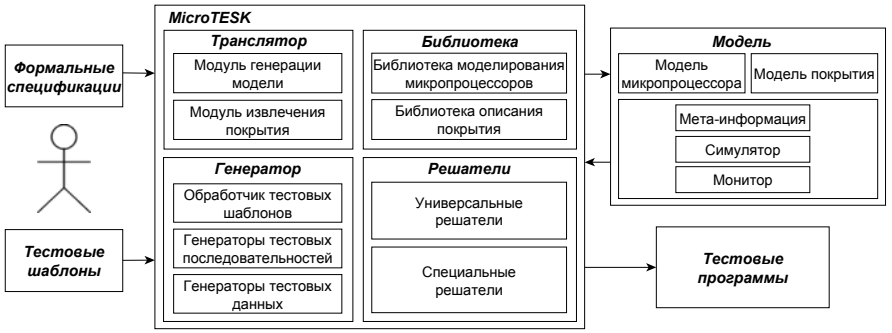

Рис. 10. Архитектура среды генерации тестовых программ MicroTESK

MicroTESK генерирует тесты на основе *тестовых шаблонов*, представляющих собой абстрактные описания тестовых программ. Для их разработки используется *язык описания тестовых шаблонов*, реализованный в виде набора библиотек на языке Ruby [103], что отличает его от узкоспециализированных языков, используемых в инструментах Genesys-Pro [41] и RAVEN [45]. Средства языка описания тестовых шаблонов можно разделить на две группы: (1) встроенные конструкции Ruby (условные операторы, операторы циклов и т.д.) и (2) средства библиотек MicroTESK (базовые классы тестовых шаблонов, конструкции описания блоков построения тестовых последовательностей и другие). Пример 12 демонстрирует формат тестового шаблона.

Пример 12. Структура тестового шаблона на языке Ruby

```
01:  class Template2014 < Template     класс тестового примера
02:    def pre ... end                 инициализирующий код
03:    def run ... end                 основной код
04:    def post ... end                завершающий код
05:  end
```

Основная секция тестового шаблона представляет собой иерархическую систему блоков, каждый из которых описывает способ генерации тестовой последовательности. Объединяющие блоки заданным образом комбинируют тестовые последовательности, описанные во вложенных блоках (см. раздел 3.2.1). Такой подход позволяет применять различные методы построения тестов в одном тестовом шаблоне (см. пример 13).

Пример 13. Фрагмент тестового шаблона с вложенными блоками

```
01:  block (:combine => "product",          объединяющий блок
                                            случайно смешивает все
                                            возможные комбинации
                                            результатов вложенных
                                            блоков
02:          :compose => "random") {
03:    block (:engine => "random",          вложенный блок
                                            генерирует 2 случайные
                                            последовательности
                                            длины 3
04:           :length => 3, :count => 2) {
05:      ADD reg(8), reg(16), reg(17)
06:      SUB reg(9), reg(18), reg(19)
07:      AND reg(10), reg(16), reg(17)
08:      OR reg(11), reg(18), reg(19)
09:    }
10:    block (:engine => "permutate") {     вложенный блок гене-
                                            рирует последова-
                                            тельности из всех воз-
                                            можных перестановок
11:      LW reg(12), reg(20), imm16(0)      команд
12:      SW reg(19), reg(21), imm16(4)
13:    }
14:  }
```

За генерацию тестовых программ по шаблонам в среде MicroTESK отвечает *обработчик тестовых шаблонов*, который формирует тестовые

последовательности, вызывая *генераторы*, указанные в блоках. Результатом являются *символические тестовые программы*, в которых операнды команд задаются с помощью *тестовых ситуаций* (ограничений, описывающих условия возникновения определенных событий в работе микропроцессора). Для конструирования значений операндов используются *генераторы тестовых данных*, основанные на *решателях ограничений*. После генерации значений операндов в тестовую программу добавляется *инициализирующий код*, в котором полученные значения записываются в соответствующие элементы памяти. Среда MicroTESK предоставляет набор *универсальных* и *специальных* решателей. Первые применимы для ограничений в широком классе теорий (булевой алгебры, целочисленной арифметики, теории битовых векторов и многих других). Вторые же предназначены для решения узкоспециализированных задач (например, для воспроизведения событий, связанных с подсистемой управления памятью [104, 105]).

4. Текущие работы и перспективы

При выборе тем для своих исследований ИСП РАН отталкивается от потребностей индустрии. Партнерские отношения с ведущими отечественными разработчиками микропроцессоров — НИИСИ РАН и ЗАО «МЦСТ» — позволяют изнутри увидеть проблемы, возникающие при промышленной верификации, и поставить актуальные задачи перед научными сотрудниками, аспирантами и студентами. Описанные выше подходы к проверке микропроцессоров были испытаны в реальных проектах, и, надо сказать, не всегда результаты их использования были положительными [73]. Мы тщательно анализируем негативный опыт, глубже проникаем в проблематику и стараемся найти решения возникающих задач. В этом разделе делается краткий обзор тем, над которыми мы работаем в настоящее время, и которые, так или иначе, имеют индустриальные корни.

4.1. Развитие средств модульной верификации

В промышленных технологиях верификации микропроцессоров возрастает роль формальных методов (проверки моделей, дедуктивного анализа, проверки эквивалентности). Одной из приоритетных задач модульной верификации является автоматическое построение формальной модели по HDL-описанию и генерация тестов на ее основе. Естественным формализмом для описания таких моделей является *расширенный конечный автомат* (*EFSM, Extended Finite State Machine*). В отличие от классического автомата, EFSM обладает переменными, а его переходы имеют особый вид. Каждый переход содержит охранное условие и действие над переменными: переход может сработать, только когда истинно охранное условие; при этом выполняется действие. Таким образом, EFSM — это множество *охраняемых действий* (см. раздел 3.1.1) над общими переменными, структурированное в систему переходов. Формализм охраняемых действий применяется для

описания сложных аппаратных систем, и для него разработаны эффективные алгоритмы синтеза [106].

Для извлечения EFSM-модели (автомата) из HDL-описания предлагается следующий подход [23]. Сначала из исходного кода извлекаются охраняемые действия. Затем из множества входных сигналов HDL-описания выделяется сигнал, по изменению которого меняется внутреннее состояние описываемого устройства (синхросигнал). После чего идентифицируются внутренние переменные HDL-описания, значения которых кодируют состояния устройства. Множество состояний автомата строится путем декомпозиции охранных условий, содержащих выделенные переменные состояния, на набор попарно несовместных ограничений (каждое из полученных ограничений соответствует одному состоянию EFSM-модели). Отношение переходов конструируется следующим образом. Для каждого действия проверяется совместимость его охранного условия с ограничениями, задающими состояния автомата. Таким образом определяется, из каких состояний есть переходы, помеченные данным охраняемым действием. Если имеется несколько состояний автомата, с которыми совместимо охраняемое условие, осуществляется «расщепление» перехода. Для того чтобы определить, в какое состояние ведет переход, проверяется совместимость результата символического выполнения действия с ограничениями состояний.

В качестве направлений дальнейших исследований рассматриваются следующие задачи: (1) генерация тестовых последовательностей на основе EFSM-моделей [107] и (2) формальный анализ EFSM-моделей на предмет достижимости ошибочных ситуаций (гонок, зависаний, конфликтов использования ресурсов) [108].

4.2. Развитие средств системной верификации

В рамках системной верификации микропроцессоров имеются две важные и взаимосвязанные темы, практически не затронутые нашими разработками: (1) верификация микропроцессоров на основе *моделей уровня микроархитектуры* и (2) верификация *многоядерных* микропроцессоров и *многопроцессорных* вычислительных комплексов. В настоящее время в ИСП РАН ведутся исследования по обоим направлениям. Работы по первой теме проводятся в контексте создания среды генерации тестовых программ MicroTESK (см. раздел 3.2). Исследования по второй теме выполняются совместно с компанией ЗАО «МЦСТ», занимающейся разработкой многоядерных микропроцессоров и систем на их основе: МЦСТ-R500S (2 ядра, 2008 г.), Эльбрус-2С+ (2+4 ядра, 2011 г.), МЦСТ-R1000 (4 ядра, 2012 г.) и других [109]. Рассмотрим кратко содержание обозначенных работ.

4.2.1 Моделирование на уровне микроархитектуры

Модели, используемые в настоящее время в MicroTESK, игнорируют устройство конвейера и динамику совместного выполнения команд. С одной

стороны, это делает инструмент проще и облегчает его изучение и использование. С другой стороны, это сказывается на качестве генерируемых тестовых программ — на некотором этапе увеличение уровня тестового покрытия микропроцессора становится возможным только при существенном увеличении объема тестов. Обычно эта проблема решается с помощью ручного создания *направленных тестов*, точечно покрывающих ситуации, оставшиеся неохваченными, однако такой подход имеет два очевидных недостатка: (1) структурные метрики тестового покрытия (см. раздел 2.2) не позволяют выразить ситуации, связанные с взаимодействиями параллельных процессов; (2) ручная разработка тестов требует много ресурсов.

Модели уровня микроархитектуры позволяют задавать более адекватные метрики тестового покрытия, а также автоматизировать генерацию тестов, направленных на возникновение определенных событий в работе микропроцессора (особых случаев выполнения операций, прерываний, ситуаций заполнения буферов и т.п.). Кроме того, они делают возможной формальную верификацию проекта и могут использоваться для синтеза HDL-описания и генерации таких инструментов, как симулятор и компилятор (как правило, чем детальнее модель, тем точнее симулятор и эффективнее компилятор). В свою очередь, наличие средств кросс-разработки позволяет исследовать альтернативы проектных решений (выбор той или иной микроархитектуры) с использованием ПО (типовых задач), предназначенного для целевого микропроцессора [19].

Спецификация микропроцессора — это очень сложная задача, и выбор адекватного формализма имеет первостепенное значение. Важными требованиями к формализму, используемому для описания микроархитектуры, являются: (1) возможность частичной спецификации проекта (возможность инкрементального описания подсистем и связей между ними), (2) возможность спецификации на разных уровнях абстракции (от обобщенных протоколов взаимодействия подсистем до детально описанных коммуникационных сетей), (3) возможность повторного использования элементов моделей (возможность составления моделей из имеющихся частей). Скорее всего, решение представляет собой комбинацию различных подходов: сетей Петри, конечных автоматов, алгебр и исчислений процессов.

Среди возможных вариантов рассматриваются следующие нотации: (1) графические языки типа UML (Unified Modeling Language) [110]; (2) Sim-nML [101] — расширение ADL-языка nML, используемого в генераторе тестовых программ MicroTESK (см. раздел 3.2.2); (3) EXPRESSION [19] — ADL-язык смешанного типа (позволяющий описывать не только систему команд микропроцессора, но и его структуру); (4) xMAS (Executable MicroArchitecture Specifications) [111] — язык описания коммуникационных сетей; (5) Promela (Process meta language) [112] — язык описания процессов и протоколов, используемый в инструменте проверки моделей SPIN;

(6) Bluespec [76] — язык, основанный на парадигме атомарных охраняемых действий.

4.2.2 Верификация реализаций протоколов когерентности

Вторая задача (верификация многоядерных микропроцессоров и многопроцессорных комплексов) связана с проверкой механизмов обеспечения *когерентности памяти*. Каждый узел системы (вычислительное ядро или микропроцессор) имеет в своем составе кэш-память, из-за чего в системе могут существовать несколько копий одних данных – одна копия в основной памяти и несколько копий в кэш-памяти узлов. При изменении копии данных в одном из узлов другие совладельцы этих данных должны согласованным образом изменить свои копии (или удалить их). Взаимодействие между узлами системы осуществляется согласно *протоколу когерентности*, за реализацию которого отвечают соответствующие механизмы подсистемы управления памятью. Их разработка осуществляется в два этапа: (1) проектирование протокола когерентности (и создание его формальной модели); (2) реализация протокола в аппаратуре. На первом этапе осуществляется верификация *модели* протокола; для этого используются формальные методы (прежде всего, проверка моделей и дедуктивный анализ). Для проверки *реализации* протокола (HDL-модели подсистемы управления памятью) обычно используется случайная генерация тестов, что не гарантирует выявления всех ошибок (пример – ошибка #298 в AMD Phenom [9]).

Представляется целесообразным использовать модели, применяемые для верификации протоколов когерентности, для тестирования HDL-моделей подсистем управления памятью. При этом возникают две задачи: (1) генерация тестов по модели и (2) детерминированное (сохраняющее порядок событий) воспроизведение построенных тестов на реализации. Первая задача решается методами тестирования на основе моделей (в том числе методами обхода графов состояний). Исследования, проводимые совместно с ЗАО «МЦСТ», сфокусированы на второй задаче и нацелены на создание средств, позволяющих тестировать реализации протоколов когерентности на основе *трасс* выполнения моделей. Разрабатываемые средства позволяют воспроизводить трассы на реализации, проверять корректность поведения реализации и оценивать уровень тестового покрытия. Кроме того, средства являются универсальными — применимыми к системам, основанным на разных архитектурах и протоколах когерентности [113]. Прототип инструмента позволил обнаружить ошибку в эталонной модели подсистемы управления памятью, которая не была обнаружена другими способами верификации.

4.3. Унификация и интеграция средств верификации

Среди задач, решаемых разными средствами верификации микропроцессоров (как модульного, так и системного уровней), есть много похожих. Унификация интерфейсов *механизмов* (*engines*) решения общих подзадач и создание библиотек переиспользуемых компонентов снижает трудоемкость разработки инструментов верификации, упрощает их интеграцию друг с другом и создает предпосылки для создания расширяемых сред [20, 44, 114]. Так, во многих средствах верификации (формальной и имитационной) применяются техники разрешения ограничений и реализующие их инструменты — *решатели* (*solvers*), например, Yices [115], Z3 [116] и другие. Сфера применения решателей широка: генерация тестовых данных [26], статический анализ HDL-описаний [23], символическая проверка моделей [117]. Ввиду многочисленности таких инструментов (каждый решатель лучше справляется с ограничениями того или иного вида) имеется потребность в унификации интерфейсов работы с ними.

В ИСП РАН была разработана библиотека JCS API (Java Constraint Solver API) [118], позволяющая описывать ограничения в форме объектов (синтаксических деревьев) на языке Java (на котором разработаны многие наши инструменты) и вызывать внешние решатели. В качестве основного средства взаимодействия JCS API с решателями используется язык SMT-LIB (Satisfiability Modulo Theories Library) [119], поддерживаемый многими инструментами. Библиотека, изначально разработанная в рамках проекта MicroTESK [47], где она применялась для генерации тестовых данных (см. раздел 3.2.2), в настоящее время активно используется в наших разработках в области анализа HDL-описаний (см. раздел 4.1). Развитие JCS API осуществляется по трем направлениям: (1) расширение множества поддерживаемых операций и типов данных; (2) разработка средств трансформации ограничений (и других формул); (3) создание эффективных решателей, ориентированных на типичные виды ограничений, встречающиеся при верификации аппаратуры.

Другими работами общего характера, выполняемыми в ИСП РАН и связанными с верификацией микропроцессоров, являются: (1) эффективная реализация операций над битовыми векторами (такие операции могут быть полезны при построении симуляторов HDL- и ADL-моделей, а также при создании генераторов тестовых данных); (2) разработка библиотеки для представления моделей аппаратуры (автоматов, сетей и т.п.) и их анализа (библиотека может применяться в составе разных инструментов верификации, включая генераторы тестов и инструменты проверки моделей); (3) разработка средств интеграции разных инструментов верификации (преобразователей моделей, представленных в формате одного инструмента, в формат другого). В качестве примера интеграции разных инструментов верификации можно привести следующий. Для статического анализа HDL-описаний (см. раздел 4.1) мы используем открытую платформу ZamiaCAD [120], которая по

описанию аппаратуры на языке VHDL строит *граф экземпляров модулей* (*IG*, *Instantiation Graph*), преобразуемый с помощью средств интеграции во внутреннее представление наших инструментов.

5. Заключение

Несмотря на развитие методов функциональной верификации, их возможности не справляются со все возрастающей сложностью микропроцессоров [3]. Если первые устройства мог проверить буквально один человек путем непосредственного анализа (экспертизы) схемы из логических вентилей или даже транзисторов (микросхема 4004, выпущенная в 1971 г., состояла менее чем из 2.5 тысяч транзисторов, соединения между которыми можно изобразить на листе ватмана), то сейчас штат инженеров-верификаторов достигает внушительных размеров (последние образцы микропроцессоров содержат в 2 миллиона раз большее число транзисторов по сравнению с микросхемой 4004 — см. рис. 1 [2]). Костяк команды, работающей над проверкой Pentium Pro (5.5 миллиона транзисторов), состоял из 10 человек, в проекте Pentium 4 (42 миллиона транзисторов) к ним присоединилось еще 60 [6]. Скорее всего, сейчас штат верификаторов в таких компаниях, как Intel, насчитывает несколько сотен специалистов (с тех пор прошло более 10 лет, а число транзисторов выросло более чем в 100 раз). Трудоемкость верификации огромна — сотни человеко-лет и миллионы часов компьютерного времени (только на формальную проверку Pentium 4 было затрачено 60 человеко-лет [51]).

Мечтой всех производителей микропроцессоров является автоматическое доказательство корректности проектирования. Некоторые компании, университеты и исследовательские институты проводят большую работу, чтобы приблизиться к этой мечте. По заявлению Боба Бентли (Bob Bentley), к 2015 г. Intel планирует увеличить долю использования формальных методов в модульной верификации микропроцессоров до 50% — это серьезный вызов, который потребует кардинальной перестройки процессов проектирования [14]. Если говорить о ситуации в настоящий момент времени, то основным подходом по-прежнему остается тестирование; но, нужно отметить, методы тестирования не стоят на месте и многое заимствуют из формальных методов. Скорее всего, взаимное проникновение разных методов верификации будет продолжаться, что приведет к появлению новых гибридных методов. Именно таким подходам, находящимся на стыке разных методов, посвящены исследования, выполняемые в ИСП РАН: формальная спецификация (на уровнях микро- и макроархитектуры), генерация тестов по моделям (потактовым, событийным, обобщенным), статический анализ HDL-описаний и другие. Результаты этих исследований нашли воплощение в инструментах CTESK [52], C++TESK [81], MicroTESK [47] и JCS API [118], которые успешно применялись в промышленных проектах.

Отдельно следует упомянуть проблему кадров. Кен Элбин (Ken Albin) из компании Motorola справедливо утверждает, что инженеры-верификаторы должны иметь совершенно другой набор знаний и навыков, нежели инженеры-проектировщики [51]. Среди качеств, желательных для верификатора, можно отметить следующие: умение изменять уровень абстракции модели, опыт программирования, знание основ проектирования аппаратуры, умение работать с огромными массивами данных, хорошие коммуникационные навыки и внутренняя мотивация. Кроме того, для работы в области формальной верификации требуется серьезная математическая подготовка (дискретная математика, математическая логика, теория трансляции и преобразования программ). Многие коммерческие компании осознают необходимость в подготовке высококлассных верификаторов и сотрудничают с ведущими университетами. Для развития отрасли крайне важно, чтобы университеты и исследовательские институты вели образовательную деятельность в области верификации и готовили квалифицированные кадры. На протяжении нескольких лет в ИСП РАН читаются лекции по методам верификации для студентов старших курсов МФТИ. Мы рассчитываем, что работа Института будет способствовать росту надежности микропроцессоров.

Список литературы

[1] В.В. Кулямин. *Методы верификации программного обеспечения*, 2008. 111 с. (http://www.ispras.ru/~kuliamin/docs/VerMethods-2008-ru.pdf)

[2] *Статистика числа транзисторов в микропроцессорах* — http://en.wikipedia.org/wiki/Transistor_count

[3] А.С. Камкин. *Верификация микропроцессоров: борьба с ошибками и управление качеством*. Электроника: НТБ, №3, 2010. С. 98-104.

[4] J. Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, 2000. 354 p.

[5] G.E. Moore. *Cramming More Components onto Integrated Circuits*. Electronics Magazine, 86(1), 1965. P. 82-85. (http://www.cs.utexas.edu/~fussell/courses/cs352h/papers/moore.pdf)

[6] B. Bentley. *Validating the Intel® Pentium 4® Microprocessor*. Design Automation Conference (DAC), 2001. P. 244-248.

[7] B. Bentley. *Validating a Modern Microprocessor*. International Conference on Computer Aided Verification (CAV), 2005. P. 2-4. (http://www.cav2005.inf.ed.ac.uk/bentley_CAV_07_08_2005.ppt)

[8] *FDIV Replacement Program: Description of the Flaw* — http://www.intel.com/support/processors/pentium/sb/CS-013007.htm

[9] *Revision Guide for AMD Family 10h Processors*. Revision 3.84, August 2011. (http://developer.amd.com/wordpress/media/2012/10/41322.pdf)

[10] S. Barak. *Intel Denies Core i7 TLB Bug*. The Inquirer, December 02 2008. (http://www.theinquirer.net/inquirer/news/1049427/intel-denies-core-i7-tlb-bug)

[11] *Intel® Core™ i7-900 Desktop Processor Extreme Edition Series and Intel® Core™ i7-900 Desktop Processor Series*. Specification Update, May 2011. (http://download.intel.com/design/processor/specupdt/320836.pdf)

[12] E.M. Clarke, J.M. Wing. *Formal Methods: State of the Art and Future Directions.* ACM Computing Surveys, 28(4), 1996. P. 626-643.

[13] J. Harrison. *Formal Methods at Intel — An Overview*. Second NASA Formal Methods Symposium (NFM), 2010. (http://www.cl.cam.ac.uk/~jrh13/slides/nasa-14apr10/slides.pdf)

[14] P. McLellan. *History of Formal Verification at Intel*. DAC Blog, December 05 2012. (http://blog.dac.com/post/2012/12/05/History-of-Formal-Verification-at-Intel.aspx)

[15] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, A. Naik. *Replacing Testing with Formal Verification in Intel® Core™ i7 Processor Execution Engine Validation*. International Conference on Computer Aided Verification (CAV), 2009. P. 414-429.

[16] J. Bhadra, M. Abadir, S. Ray, L. Wang. *A Survey of Hybrid Techniques for Functional Verification*. IEEE Design & Test of Computers, 24(22), 2007. P. 112-122.

[17] Z. Navabi. *Languages for Design and Implementation of Hardware*. W.-K. Chen (Ed.). The VLSI Handbook. CRC Press, 2007. 2320 p.

[18] S. Mikhani, Z. Navabi. *System Level Design Languages*. W.-K. Chen (Ed.). The VLSI Handbook. CRC Press, 2007. 2320 p.

[19] P. Mishra, N. Dutt (Eds.). *Processor Description Languages. Systems on Silicon*. Morgan Kaufmann, 2008. 432 p.

[20] В. Кулямин. *Перспективы интеграции методов верификации программного обеспечения*. Труды ИСП РАН, 16, 2009. С. 73-88.

[21] S. Agbaria, D. Carmi, O. Cohen, D. Korchemny, M. Lifshits, A. Nadel. *SAT-Based Semiformal Verification of Hardware*. Formal Methods in Computer-Aided Design (FMCAD), 2010. P. 25-32.

[22] H. Jain, D. Kroening, N. Sharygina, E. Clarke. *VCEGAR: Verilog Counterexample Guided Abstraction Refinement* Tools and Algorithms for Construction and Analysis of Systems (TACAS), 2007. P. 583-586.

[23] A. Kamkin, S. Smolov, I. Melnichenko. *Static Analysis of HDL Descriptions: Extracting Models for Verification.* East-West Design and Test Symposium (EWDTS), 2013. P. 184-187.

[24] W.K. Lam. *Hardware Design Verification: Simulation and Formal Method-Based Approaches*. Prentice Hall, 2005. 624 p.

[25] N. Bombieri, F. Fummi, G. Pravadelli, J. Marques-Silva. *Towards Equivalence Checking between TLM and RTL Models.* IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE), 2007. P. 113-122.

[26] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. Marcus, G. Shurek. *Constraint-Based Random Stimuli Generation for Hardware Verification*. AI Magazine, 28(3), 2007. P. 13-30.

[27] *Технология верификации UVM* — http://www.accellera.org/community/uvm

[28] А.В. Баранцев, И.Б. Бурдонов, А.В. Демаков, С.В. Зеленов, А.С. Косачев, В.В. Кулямин, В.А. Омельченко, Н.В. Пакулин, А.К. Петренко, А.В. Хорошилов. *Подход UniTesK к разработке тестов: достижения и перспективы*. Труды ИСП РАН, 5, 2004. С. 151-156.

[29] K. Sen, G. Rosu. *Generating Optimal Monitors for Extended Regular Expressions*. Electronic Notes in Theoretical Computer Science, 89(2), 2003. P. 162-181.

[30] В.П. Иванников, А.С. Камкин, А.С. Косачев, В.В. Кулямин, А.К. Петренко. *Использование контрактных спецификаций для представления требований и функционального тестирования моделей аппаратуры*. Программирование, № 5, 2007. С. 47-61.

[31] H. Barringer, D. Rydeheard, K. Havelund. *Rule Systems for Run-Time Monitoring: From Eagle to RuleR*. International Workshop on Runtime Verification, 2007. P. 111-125.

[32] A. Bauer, M. Leucker, C. Schallhart. *Runtime Verification for LTL and TLTL*. ACM Transactions on Software Engineering and Methodology, 20(4), 2011. P. 14:1-14:64.

[33] A. Pnueli. *Temporal Logic of Programs*. Symposium on Foundation of Computer Science (SFCS), 1977. P. 46-57.

[34] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M. Vardi, Y. Zbar. *The ForSpec Temporal Logic: A New Temporal Property-Specification Language*. Tools and Algorithms for Construction and Analysis of Systems (TACAS), 2002. P. 296-311.

[35] *OpenVera® Language Reference Manual: Assertions*. Version 1.4.1, November 2004.

[36] *1850-2010 — IEEE Standard for Property Specification Language (PSL)*, 2010.

[37] *1647-2011 — IEEE Standard for the Functional Verification Language e*, 2011.

[38] *1800-2012 — IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language*, 2013.

[39] A. Pizialli. *Functional Verification Coverage Measurement and Analysis*. Kluwer Academic Publishers, 2004. 216 p.

[40] M. Chupilko, A. Kamkin. *A TLM-Based Approach to Functional Verification of Hardware Components at Different Abstraction Levels.* Latin American Test Workshop (LATW), 2011. P. 1-6.

[41] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, A. Ziv. *Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification.* IEEE Design & Test of Computers, 21(2), 2004. P. 84-93.

[42] А.С. Камкин. *Генерация тестовых программ для микропроцессоров.* Труды ИСП РАН, 14(2), 2008. С. 23-63.

[43] *Тестовая программа PARANOIA* — http://people.sc.fsu.edu/~%20jburkardt/c_src/paranoia/paranoia.html

[44] А.С. Камкин, Т.И. Сергеева, С.А. Смолов, А.Д. Татарников, М.М. Чупилко. *Расширяемая среда генерации тестовых программ для микропроцессоров.* Программирование, № 1, 2014. (*в печати*)

[45] *Слайды по генератору тестовых программ RAVEN* — http://www.slideshare.net/DVClub/introducing-obsidian-software-and-ravengcs-for-powerpc

[46] I.V. Gribkov, A.V. Zakharov, P.P. Koltsov, N.V. Kotovich, A.A. Kravchenko, A.S. Koutsaev, A.S. Osipov, I.S Khisambeev. *INTEG: A Stochastic Testing System for Microprocessor Verification*. WSEAS International Conference on Circuits, Systems, Signal and Telecommunications (CSST), 2007. P. 55-59.

[47] *Страница инструмента MicroTESK* — http://forge.ispras.ru/projects/microtesk

[48] *Страница инструмента Genesys-Pro* — https://www.research.ibm.com/haifa/projects/verification/genesys_pro/index.shtml

[49] P. Mishra, N. Dutt. *Specification-Driven Directed Test Generation for Validation of Pipelined Processors.* ACM Transactions on Design Automation of Electronic Systems, 13(3), 2008. P. 1-36.

[50] T.N. Dang, A. Roychoudhury, T. Mitra, P. Mishra. *Generating Test Programs to Cover Pipeline Interactions.* Design Automation Conference (DAC), 2009. P. 142-147.

[51] N. Mokhoff. *Intel, Motorola Report Formal Verification Gains.* EE Times, June 21 2001. (http://www.eetimes.com/document.asp?doc_id=1215957)

[52] *Страница инструмента CTESK* — http://forge.ispras.ru/projects/ctesk

[53] В.П. Иванников, А.С. Камкин, В.В. Кулямин, А.К. Петренко. *Применение технологии UniTesK для функционального тестирования моделей аппаратного обеспечения*. Препринт ИСП РАН, 2005. 16 с.

[54] Д.Н. Воробьев, А.С. Камкин. *Генерация тестовых программ для подсистемы управления памятью микропроцессора*. Труды ИСП РАН, 17, 2009. С. 119-132.

[55] С.И. Аряшев, А.С. Камкин, Б.Ю. Рогаткин. *Тестирование RTL-моделей аппаратуры с помощью технологии UniTESK на примере блока преобразования адресов микропроцессора*. Электроника, микро- и наноэлектроника, 2007. С. 183-187.

[56] I. Bourdonov, A. Kossatchev, A. Petrenko, D. Galter. *KVEST: Automated Generation of Test Suites from Formal Specifications. Formal Methods.* World Congress on Formal Methods in the Development of Computing Systems (FM), 1, 1999. P. 608-621.

[57] И.Б. Бурдонов, А.С. Косачев, В.В. Кулямин. *Неизбыточные алгоритмы обхода ориентированных графов. Детерминированный случай*. Программирование, № 5, 2003. С. 11-30.

[58] И.Б. Бурдонов, А.С. Косачев, В.В. Кулямин. *Неизбыточные алгоритмы обхода ориентированных графов. Недетерминированный случай*. Программирование, № 1, 2004. С. 4-24.

[59] B. Meyer. *Design by Contract*. Technical Report TR-EI-12/CO, Interactive Software Engineering Inc, 1986.

[60] R.W. Floyd. *Assigning Meaning to Programs*. Symposium on Applied Mathematics, 1967. P. 19-32.

[61] C.A.R. Hoare. *An Axiomatic Basis for Computer Programming*. Communications of the ACM, 12(10), 1969. P. 576-585.

[62] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. 217 p.

[63] V. Kuliamin, A. Petrenko, N. Pakoulin, A. Kossatchev, I. Bourdonov. *Integration of Functional and Timed Testing of Real-Time and Concurrent Systems*. Perspectives of System Informatics, 2003. P. 450-461.

[64] A. Kamkin. *Contract Specification of Pipelined Designs: Application to Testbench Automation*. Spring Young Researchers' Colloquium on Software Engineering (SYRCoSE), 2007. P. 7-13.

[65] A. Kamkin. *Coverage-Directed Verification of Microprocessor Units Based on Cycle-Accurate Contract Specifications*. East-West Design & Test Symposium (EWDTS), 2008. P. 84-87.

[66] А.С. Камкин. *Метод формальной спецификации аппаратуры с конвейерной организацией и его приложение к задачам функционального тестирования*. Труды ИСП РАН, 16, 2009. С. 107-128.

[67] M. Chupilko, A. Kamkin. *Developing Cycle-Accurate Contract Specifications for Synchronous Parallel-Pipeline Hardware: Application to Verification*. Baltic Electronics Conference (BEC), 2010. P. 185-188.

[68] M. Chupilko, A. Kamkin. *Specification-Driven Testbench Development for Synchronous Parallel-Pipeline Designs*. NORCHIP, 2009. P. 1-4.

[69] M. Chupilko, A. Kamkin. *Contract Specification of Hardware Designs at Different Abstraction Levels: Application to Functional Verification*. Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE), 2010. P. 125-129.

[70] А.В. Хорошилов. *Спецификация и тестирование систем с асинхронным интерфейсом*. Препринт ИСП РАН, 2006. 139 с.

[71] А.С. Камкин, М.М. Чупилко. *Тестирование модулей арифметики с плавающей точкой микропроцессоров на соответствие стандарту IEEE 754*. Труды ИСП РАН, 14(2), 2008. С. 7-22.

[72] M. Chupilko, A. Kamkin, D. Vorobyev. *Methodology and Experience of Simulation-Based Verification of Microprocessor Units Based on Cycle-Accurate Contract Specifications.* Spring Young Researchers' Colloquium on Software Engineering (SYRCoSE), 2008. P. 25-31.

[73] Р.А. Баратов, А.С. Камкин, В.М. Майорова, А.Н. Мешков, А.А. Сортов, М.А. Якушева. *Трудности модульной верификации аппаратуры на примере буфера команд микропроцессора «Эльбрус-2S».* Вопросы радиоэлектроники, № 3, 2013. С. 84-96.

[74] E. Dijkstra. *Guarded Commands, Non-determinacy and Formal Derivation of Programs*. Communications of the ACM, 18(8), 1975. P. 453-457.

[75] D.L. Rosenband, Arvind. *Hardware Synthesis from Guarded Atomic Actions with Performance Specifications*. International Conference on Computer-Aided Design (ICCAD), 2005. P. 784-791.

[76] *Bluespec^{TM} SystemVerilog Reference Guide*, 2012.

[77] C. Clare. *Designing Logic Systems Using State Machines*. McGraw-Hill, 1973. 150 p.

[78] S. Baranov. *Logic and System Design of Digital Systems*. TUT Press, 2008. 266 p.

[79] В.П. Иванников, А.С. Камкин, М.М. Чупилко. *Проверка корректности поведения HDL-моделей цифровой аппаратуры на основе динамического сопоставления трасс.* Tools & Methods of Program Analysis (TMPA), 2013. С. 71-82.

[80] M. Chupilko, A. Kamkin. *Runtime Verification Based on Executable Models: On-the-Fly Matching of Timed Traces.* Model-Based Testing Workshop (MBT), 2013. P. 67-81.

[81] *Страница инструмента C++TESK* — http://forge.ispras.ru/cpptesk

[82] G. von Bochmann, S. Haar, C. Jard, G.V. Jourdan. *Testing Systems Specified as Partial Order Input/Output Automata*. International Conference on Testing of Software and Communicating Systems (TestCom), 2008. P. 169-183.

[83] И.Б. Бурдонов, А.С. Косачев, В.В. Кулямин. *Использование конечных автоматов для тестирования программ.* Программирование, 26(2), 2000. С. 61-73.

[84] И.Б. Бурдонов, С.Г. Грошев, А.В. Демаков, А.С. Камкин, А.С. Косачев, А.А. Сортов. *Параллельное тестирование больших автоматных моделей*. Вестник ННГУ, № 3, 2011. С. 187-193.

[85] A. Demakov, A. Kamkin, A. Sortov. *High-Performance Testing: Parallelizing Functional Tests for Computer Systems Using Distributed Graph Exploration*. Open Cirrus Summit, 2011.

[86] И.Б. Бурдонов, А.С. Косачев. *Обход неизвестного графа коллективом автоматов.* Научный сервис в сети Интернет: все грани параллелизма, 2013. С. 228-232.

[87] А.В. Демаков, С.А. Зеленова, С.В. Зеленов. *Тестирование парсеров текстов на формальных языках.* Программные системы и инструменты: Тематический сборник факультета ВМиК МГУ, № 2, 2001. С. 150-156.

[88] С.В. Зеленов, С.А. Зеленова, А.С. Косачев, А.К. Петренко. *Применение модельного подхода для автоматического тестирования оптимизирующих компиляторов*, 2003.(http://citforum.ru/SE/testing/compilers)

[89] L.-M. Wu, K. Wang, C.-Y. Chiu. *A BNF-Based Automatic Test Program Generator for Compatible Microprocessor Verification.* ACM Transactions on Design Automation of Electronic Systems, 9(1), 2004. P. 105-132.

[90] А.С. Камкин. *Некоторые вопросы автоматизации построения тестовых программ для модулей обработки переходов микропроцессоров*. Труды ИСП РАН, 18, 2010. С. 129-149.

[91] Д.Н. Воробьев, А.С. Камкин. *Генерация тестовых программ для микропроцессоров на основе шаблонов конвейерных конфликтов*. Труды ИСП РАН, 18, 2010. С. 91-113.

[92] *MIPS64™ Architecture For Programmers, Revision 2.0.* MIPS Technologies Inc, June 9 2003.

[93] *RM7000 Family User Manual*. Issue 1, May 2001.

[94] А.С. Камкин. *Комбинаторная генерация тестовых программ для микропроцессоров на основе моделей.* Препринт ИСП РАН, 2008. 18 с.

[95] A. Kamkin, E. Kornykhin, D. Vorobyev. *Reconfigurable Model-Based Test Program Generator for Microprocessors*. International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2011. P. 47-54.

[96] A. Kamkin, A. Tatarnikov. *MicroTESK: An ADL-Based Reconfigurable Test Program Generator for Microprocessors.* Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE), 2012. P. 64-69.

[97] M. Freericks. *The nML Machine Description Formalism. Techical Report*. TU Berlin, FB20, Bericht, 1991/15. 47 p.

[98] *Страница компании Target Compiler Technologies* — http://www.retarget.com

[99] S. Chandra, R. Moona. *Retargetable Functional Simulator using High Level Processor Models*. VLSI Design, 2000. P. 424-429.

[100] *Страница инструмента GLISS* — http://www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id_rubrique=54

[101] H. Cassé, J. Barre, R. Vaillant, P. Sainrat. *Fast Instruction-Accurate Simulation with SimNML*. Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO), 2011. P. 8-12.

[102] A. Kamkin, T. Sergeeva, A. Tatarnikov, A. Utekhin. *MicroTESK: An Extendable Framework for Test Program Generation*. Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE), 2013. P. 51-57.

[103] *Язык программирования Ruby* — http://www.ruby-lang.org

[104] E. Kornykhin. *SMT-Based Test Program Generation for Cache-Memory Testing.* East-West Design & Test Symposium (EWDTS), 2009. P. 124-127.

[105] E. Kornykhin. *Generation of Test Data for Verification of Caching Mechanisms and Address Translation in Microprocessors.* Programming and Computing Software, 36(1), 2010. P. 28-35.

[106] J. Brandt, M. Gemünde, K. Schneider, S. Shukla, J.-P. Talpin. *Integrating System Descriptions by Clocked Guarded Actions.* Forum on Design Languages, 2011. P. 1-8.

[107] G. Guglielmo, L. Guglielmo, F. Fummi, G. Pravadelli. *Efficient Generation of Stimuli for Functional Verification by Backjumping Across Extended FSMs.* Journal of Electronic Testing, 27(2), 2011. P. 37-162.

[108] B. Karaçali, K.-C. Tai, M.A. Vouk. *Deadlock Detection of EFSMs using Simultaneous Reachability Analysis*. Dependable Systems and Networks (DSN), 2000. P. 315-324.

[109] А.К. Ким, В.И. Перекатов, С.Г. Ермаков. *Микропроцессоры и вычислительные комплексы семейства «Эльбрус»*. СПб.: Питер, 2013. 272 с.

[110] *Сайт ресурсов UML* — http://www.uml.org

[111] S. Chatterjee, M. Kishinevsky, U. Ogras. *xMAS: Quick Formal Modeling of Communication Fabrics to Enable Verification*. IEEE Design & Test of Computers, 2011. P. 80-88.

[112] G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley, 2003. 608 p.

[113] А.С. Камкин, М.В. Петроченков. *Система поддержки верификации реализаций протоколов когерентности с использованием формальных методов*. Вопросы радиоэлектроники, серия ЭВТ, 2014. (*в печати*)

[114] K. Schneider, T. Kropf. *A Unified Approach for Combining Different Formalisms for Hardware Verification.* International Conference on Formal Methods in Computer Aided Design (FMCAD), 1996. P. 202-217.

[115] B. Dutertre, L. Moura. *The YICES SMT Solver*, 2006. (http://yices.csl.sri.com/tool-paper.pdf).

[116] L. Moura, N. Bjørner. *Z3: An Efficient SMT Solver.* Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2008. P. 337-340.

[117] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993. 194 p.

[118] *Библиотека Java Constraint Solver API* — http://forge.ispras.ru/projects/solver-api

[119] D.R. Cok. *The SMT-LIBv2 Language and Tools: A Tutorial.* GrammaTech, Inc., Version 1.1, 2011.

[120] *Страница инструмента ZamiaCAD* — http://zamiacad.sourceforge.net

Tools for Functional Verification of Microprocessors

A. Kamkin, A. Kotsynyak, S. Smolov, A. Sortov, A. Tatarnikov, M. Chupilko
ISP RAS, Moscow, Russia
{kamkin,kotsynyak,ssedai,sortov,andrewt,chupilko}@ispras.ru

Abstract. Ensuring the correctness of microprocessors and other microelectronic equipment is a fundamental problem. To deal with it, various tools for functional verification are used. Unlike bugs in software programs which are relatively easy to fix (it does not apply to their consequences), defects in integrated circuits (both design and manufacturing ones) cannot be removed. In spite of continuous development of computer-aided design (CAD) systems, test generation tools and approaches to analysis of circuits, verification remains the bottleneck of the microprocessor design cycle (it accounts for approximately 70 percent of total design resources). The article gives a brief overview of microprocessor verification tools, describes issues that commonly occur in industrial practice and analyzes possible ways to solve them. The main part of the article is dedicated to research in the field of unit- and system-level hardware verification conducted at ISPRAS. It describes such approaches as contract specification of pipeline, event-driven hardware specification, parallel/distributed testing, combinatorial test program generation and template-based test program generation. The article also summarizes the outcomes of accomplished projects, describes the present works and formulates the directions of further research.

Keywords: microprocessors, hardware, verification, validation, testing, test generation, modeling, architecture description languages, parallelization.

References

[1]. Kuliamin V.V. Metody verifikatsii programmnogo obespecheniya [Methods for Software Verification]. 2008, 111 p. (in Russian). (http://www.ispras.ru/~kuliamin/docs/VerMethods-2008-ru.pdf)
[2]. Transistor count in microprocessors — http://en.wikipedia.org/wiki/Transistor_count
[3]. Kamkin A.S. Verifikatsiya mikroprotsessorov: bor'ba s oshibkami i upravlenie kachestvom [Microprocessor Verification. Combating Errors and Quality Control]. EHlektronika: NTB [Electronics: Science, Technology, Busincess], no. 3, 2010. pp. 98-104 (in Russian).
[4]. Bergeron J. Writing Testbenches: Functional Verification of HDL Models. Kluwer Academic Publishers, 2000. 354 p. doi:10.1007/978-1-4615-0302-6
[5]. Moore G.E. Cramming More Components onto Integrated Circuits. Electronics Magazine, vol. 86, no. 1, 1965. P. 82-85. (http://www.cs.utexas.edu/~fussell/courses/cs352h/papers/moore.pdf) doi:10.1109/JPROC.1998.658762
[6]. Bentley B. Validating the Intel® Pentium 4® Microprocessor. Proc. Design Automation Conference (DAC), 2001. pp. 244-248. doi:10.1145/378239.378473
[7]. Bentley B. Validating a Modern Microprocessor. Proc. International Conference on Computer Aided Verification (CAV), 2005. pp. 2-4. (http://www.cav2005.inf.ed.ac.uk/bentley_CAV_07_08_2005.ppt) (DOI 10.1007/11513988_2)
[8]. FDIV Replacement Program: Description of the Flaw — http://www.intel.com/support/processors/pentium/sb/CS-013007.htm
[9]. Revision Guide for AMD Family 10h Processors. Revision 3.84, August 2011. (http://developer.amd.com/wordpress/media/2012/10/41322.pdf)
[10]. Barak S. Intel Denies Core i7 TLB Bug. The Inquirer, December 02 2008. (http://www.theinquirer.net/inquirer/news/1049427/intel-denies-core-i7-tlb-bug)
[11]. Intel® Core™ i7-900 Desktop Processor Extreme Edition Series and Intel® Core™ i7-900 Desktop Processor Series. Specification Update, May 2011. (http://download.intel.com/design/processor/specupdt/320836.pdf)
[12]. Clarke E.M., Wing J.M. Formal Methods: State of the Art and Future Directions. ACM Computing Surveys, vol. 28, no. 4, 1996. pp. 626-643.
[13]. Harrison J. Formal Methods at Intel — An Overview. Second NASA Formal Methods Symposium (NFM), 2010. (http://www.cl.cam.ac.uk/~jrh13/slides/nasa-14apr10/slides.pdf)
[14]. McLellan P. History of Formal Verification at Intel. DAC Blog, December 05 2012. (http://blog.dac.com/post/2012/12/05/History-of-Formal-Verification-at-Intel.aspx)
[15]. Kaivola R, Ghughal R, Narasimhan N, Telfer A, Whittemore J, Pandav S, Slobodová A, Taylor C, Frolov V, Reeber E, Naik A. Replacing Testing with Formal Verification in Intel® Core™ i7 Processor Execution Engine Validation. Proc. International Conference on Computer Aided Verification (CAV), 2009. pp. 414-429. doi:10.1007/978-3-642-02658-4_32
[16]. Bhadra J, Abadir M, Ray S, Wang L. A Survey of Hybrid Techniques for Functional Verification. IEEE Design & Test of Computers, vol. 24, no. 22, 2007. pp. 112-122. doi:10.1109/MDT.2007.30
[17]. Navabi Z. Languages for Design and Implementation of Hardware. W.-K. Chen (Ed.). The VLSI Handbook. CRC Press, 2007. 2320 p.
[18]. Mikhani S, Navabi Z. System Level Design Languages. W.-K. Chen (Ed.). The VLSI Handbook. CRC Press, 2007. 2320 p.
[19]. Mishra P, Dutt N (Eds.). Processor Description Languages. Systems on Silicon. Morgan Kaufmann, 2008. 432 p.
[20]. Kuliamin V.V. Perspektivy integratsii metodov verifikatsii programmnogo obespecheniya [Prospects for Integration of Software Verification Methods]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 16, 2009. pp. 73-88 (in Russian).
[21]. Agbaria S, Carmi D, Cohen O, Korchemny D, Lifshits M, Nadel A. SAT-Based Semiformal Verification of Hardware. Proc. Formal Methods in Computer-Aided Design (FMCAD), 2010. pp. 25-32.
[22]. Jain H, Kroening D, Sharygina N, Clarke E. VCEGAR: Verilog Counterexample Guided Abstraction Refinement. Proc. Tools and Algorithms for Construction and Analysis of Systems (TACAS), 2007. pp. 583-586. doi:10.1007/978-3-540-71209-1_45
[23]. Kamkin A, Smolov S, Melnichenko I. Static Analysis of HDL Descriptions: Extracting Models for Verification. Proc. East-West Design and Test Symposium (EWDTS), 2013. pp. 184-187. doi:10.1109/EWDTS.2013.6673126
[24]. Lam W.K. Hardware Design Verification: Simulation and Formal Method-Based Approaches. Prentice Hall, 2005. 624 p.

[25]. Bombieri N, Fummi F, Pravadelli G, Marques-Silva J. Towards Equivalence Checking between TLM and RTL Models. Proc. IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE), 2007. pp. 113-122. doi:10.1109/MEMCOD.2007.371236
[26]. Naveh Y, Rimon M, Jaeger I, Katz Y, Vinov M, Marcus E, Shurek G. Constraint-Based Random Stimuli Generation for Hardware Verification. AI Magazine, vol. 28, no. 3, 2007. pp. 13-30. doi:10.1609/aimag.v28i3.2052
[27]. UVM verification methodology — http://www.accellera.org/community/uvm
[28]. Barantsev A.V., Bourdonov I.B., Demakov A.V., Zelenov S.V., Kossatchev A.S., Kuliamin V.V., Omeltchenko V.A., Pakoulin N.V., Petrenko A.K., Khoroshilov A.V. Podkhod UniTesK k razrabotke testov: dostizheniya i perspektivy [UniTesK Approach to Test Development: Achievements and Prospects]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 5, 2004. pp. 151-156 (in Russian).
[29]. Sen K, Rosu G. Generating Optimal Monitors for Extended Regular Expressions. Electronic Notes in Theoretical Computer Science, vol. 89, no. 2, 2003. pp. 162-181. doi:10.1016/S1571-0661(04)81051-X
[30]. Ivannikov V.P., Kamkin A.S., Kossatchev A.S., Kuliamin V.V., Petrenko A.K. The Use of Contract Specifications for Representing Requirements and for Functional Testing of Hardware Models. Programming and Computer Software, vol. 33, no. 5, 2007. pp. 272-282. doi:10.1134/S0361768807050039
[31]. Barringer H, Rydeheard D, Havelund K. Rule Systems for Run-Time Monitoring: From Eagle to RuleR. Proc. International Workshop on Runtime Verification, 2007. pp. 111-125. doi:10.1007/978-3-540-77395-5_10
[32]. Bauer A, Leucker M, Schallhart C. Runtime Verification for LTL and TLTL. ACM Transactions on Software Engineering and Methodology, vol. 20, no. 4, 2011. pp. 14:1-14:64. doi:10.1145/2000799.2000800
[33]. Pnueli A. Temporal Logic of Programs. Proc. Symposium on Foundation of Computer Science (SFCS), 1977. pp. 46-57. doi:10.1109/SFCS.1977.32
[34]. Armoni R, Fix L, Flaisher A, Gerth R, Ginsburg B, Kanza T, Landver A, Mador-Haim S, Singerman E, Tiemeyer E, Vardi M, Zbar Y. The ForSpec Temporal Logic: A New Temporal Property-Specification Language. Proc. Tools and Algorithms for Construction and Analysis of Systems (TACAS), 2002. pp. 296-311. doi:10.1007/3-540-46002-0_21
[35]. OpenVera® Language Reference Manual: Assertions. Version 1.4.1, November 2004.
[36]. 1850-2010 — IEEE Standard for Property Specification Language (PSL), 2010. doi:10.1109/IEEESTD.2005.97780
[37]. 1647-2011 — IEEE Standard for the Functional Verification Language e, 2011. doi:10.1109/IEEESTD.2011.6006495
[38]. 1800-2012 — IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language, 2013. doi:10.1109/IEEESTD.2013.6469140
[39]. Pizialli A. Functional Verification Coverage Measurement and Analysis. Kluwer Academic Publishers, 2004. 216 p. doi:10.1007/b117979
[40]. Chupilko M, Kamkin A. A TLM-Based Approach to Functional Verification of Hardware Components at Different Abstraction Levels. Proc. Latin American Test Workshop (LATW), 2011. pp. 1-6. doi:10.1109/LATW.2011.5985902
[41]. 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, vol. 21, no. 2, 2004. pp. 84-93. doi:10.1109/MDT.2004.1277900
[42]. Kamkin A.S. Generatsiya testovykh programm dlya mikroprotsessorov [Test Program Generation for Microprocessors]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 14, no. 2, 2008. pp. 23-63 (in Russian).
[43]. Test program PARANOIA — http://people.sc.fsu.edu/~%20jburkardt/c_src/paranoia/paranoia.html
[44]. Kamkin A.S., Sergeeva T.I., Smolov S.A., Tatarnikov A.D., Chupilko M.M. Extensible Environment for Test Program Generation for Microprocessors. Programming and Computer Software, vol. 40, no. 1, 2014. pp. 1-9. doi:10.1134/S0361768814010046
[45]. RAVEN test program generator — http://www.slideshare.net/DVClub/introducing-obsidian-software-and-ravengcs-for-powerpc
[46]. Gribkov I.V., Zakharov A.V., Koltsov P.P., Kotovich N.V., Kravchenko A.A., Koutsaev A.S., Osipov A.S., Khisambeev I.S. INTEG: A Stochastic Testing System for Microprocessor Verification. Proc. WSEAS International Conference on Circuits, Systems, Signal and Telecommunications (CSST), 2007. pp. 55-59.
[47]. MicroTESK tool — http://forge.ispras.ru/projects/microtesk
[48]. Genesys-Pro tool — https://www.research.ibm.com/haifa/projects/verification/genesys_pro/index.shtml
[49]. Mishra P., Dutt N. Specification-Driven Directed Test Generation for Validation of Pipelined Processors. ACM Transactions on Design Automation of Electronic Systems, vol. 13, no. 3, 2008. pp. 1-36. doi:10.1145/1367045.1367051
[50]. Dang T.N., Roychoudhury A., Mitra T., Mishra P. Generating Test Programs to Cover Pipeline Interactions. Proc. Design Automation Conference (DAC), 2009. pp. 142-147. doi:10.1145/1629911.1629953
[51]. Mokhoff N. Intel, Motorola Report Formal Verification Gains. EE Times, June 21 2001. (http://www.eetimes.com/document.asp?doc_id=1215957)
[52]. CTESK tool — http://forge.ispras.ru/projects/ctesk
[53]. Ivannikov V.P., Kamkin A.S., Kuliamin V.V., Petrenko A.K. Primenenie tekhnologii UniTesK dlya funktsional'nogo testirovaniya modelej apparatnogo obespecheniya [Application of the UniTESK Technology for Functional Testing of Hardware Models]. Preprint ISP RAN [Preprint of ISP RAS], 2005. 16 p. (in Russian).
[54]. Vorobyev D.N., Kamkin A.S. Generatsiya testovykh programm dlya podsistemy upravleniya pamyat'yu mikroprotsessora [Test Program Generation for Memory Management Units of Microprocessors]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 17, 2009. pp. 119-132 (in Russian).
[55]. Aryashev S.I., Kamkin A.S., Rogatkin B.Yu. Testirovanie RTL-modelej apparatury s pomoshh'yu tekhnologii UniTESK na primere bloka preobrazovaniya adresov mikroprotsessora [Using the UniTESK Technology for Testing RTL Hardware Models by the Example of the Microprocessor Translation Lookaside Buffer]. Proc. EHlektronika, mikro- i nanoehlektronika [Electronics, micro- and nanoelectronics], 2007. pp. 183-187 (in Russian).
[56]. Bourdonov I., Kossatchev A., Petrenko A., Galter D. KVEST: Automated Generation of Test Suites from Formal Specifications. Proc. World Congress on Formal Methods in the Development of Computing Systems (FM), vol. 1, 1999. pp. 608-621. doi:10.1007/3-540-48119-2_34
[57]. Bourdonov I.B., Kossatchev A.S., Kuliamin V.V. Irredundant Algorithms for Traversing Directed Graphs: The Deterministic Case. Programming and Computer Software, vol. 29, no. 5, 2003. pp. 245-258. doi:10.1023/A:1025733107700

[58]. Bourdonov I.B., Kossatchev A.S., Kuliamin V.V. Irredundant Algorithms for Traversing Directed Graphs: The Nondeterministic Case. Programming and Computer Software, vol. 30, no. 1, 2004. pp. 2-17. doi:10.1023/B:PACS.0000013436.72070.95
[59]. Meyer B. Design by Contract. Technical Report TR-EI-12/CO, Interactive Software Engineering Inc, 1986. doi:10.1109/2.161279
[60]. Floyd R.W. Assigning Meaning to Programs. Proc. Symposium on Applied Mathematics, 1967. pp. 19-32.
[61]. Hoare C.A.R. An Axiomatic Basis for Computer Programming. Communications of the ACM, vol. 12, no. 10, 1969. pp. 576-585. doi:10.1145/363235.363259
[62]. Dijkstra E.W. A Discipline of Programming. Prentice-Hall, 1976. 217 p.
[63]. Kuliamin V, Petrenko A, Pakoulin N, Kossatchev A, Bourdonov I. Integration of Functional and Timed Testing of Real-Time and Concurrent Systems. Proc. Perspectives of System Informatics, 2003. pp. 450-461. doi:10.1007/978-3-540-39866-0_45
[64]. Kamkin A. Contract Specification of Pipelined Designs: Application to Testbench Automation. Proc. Spring Young Researchers' Colloquium on Software Engineering (SYRCoSE), 2007. pp. 7-13.
[65]. Kamkin A. Coverage-Directed Verification of Microprocessor Units Based on Cycle-Accurate Contract Specifications. Proc. East-West Design & Test Symposium (EWDTS), 2008. pp. 84-87. doi:10.1109/EWDTS.2008.5580153
[66]. Kamkin A.S. Metod formal'noj spetsifikatsii apparatury s konvejernoj organizatsiej i ego prilozhenie k zadacham funktsional'nogo testirovaniya [Method for Formal Specification of Pipelined Designs and Its Application to Functional Testing]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 16, 2009. pp. 107-128 (in Russian).
[67]. Chupilko M, Kamkin A. Developing Cycle-Accurate Contract Specifications for Synchronous Parallel-Pipeline Hardware: Application to Verification. Proc. Baltic Electronics Conference (BEC), 2010. pp. 185-188. doi:10.1109/BEC.2010.5631143
[68]. Chupilko M, Kamkin A. Specification-Driven Testbench Development for Synchronous Parallel-Pipeline Designs. Proc. NORCHIP, 2009. pp. 1-4. doi:10.1109/NORCHP.2009.5397808
[69]. Chupilko M, Kamkin A. Contract Specification of Hardware Designs at Different Abstraction Levels: Application to Functional Verification. Proc. Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE), 2010. pp. 125-129.
[70]. Khoroshilov A.V. Spetsifikatsiya i testirovanie sistem s asinkhronnym interfejsom [Specification and Testing of Systems with Asynchronous Interfaces]. Preprint ISP RAN [Preprint of ISP RAS], 2006. 139 p (in Russian).
[71]. Kamkin A.S., Chupilko M.M. Testirovanie modulej arifmetiki s plavayushhej tochkoj mikroprotsessorov na sootvetstvie standartu IEEE 754 [Testing Microprocessor Floating Point Arithmetic Modules for Conformity to the IEEE 754 Standard]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 14, no. 2, 2008. pp. 7-22 (in Russian).
[72]. Chupilko M, Kamkin A, Vorobyev D. Methodology and Experience of Simulation-Based Verification of Microprocessor Units Based on Cycle-Accurate Contract Specifications. Proc. Spring Young Researchers' Colloquium on Software Engineering (SYRCoSE), 2008. pp. 25-31.
[73]. Baratov R.A., Kamkin A.S., Mayorova V.M., Meshkov A.N., Sortov A.A., Yakusheva M.A. Trudnosti modul'noj verifikatsii apparatury na primere bufera komand mikroprotsessora Elbrus-2S [Difficulties of the Unit-Level Hardware Verification on the Example of the Instruction Buffer of the Elbrus-2S Microprocessor]. Voprosy radioehlektroniki [Issues of Radio Electronics], no. 3, 2013. pp. 84-96 (in Russian).

[74]. Dijkstra E. Guarded Commands, Non-determinacy and Formal Derivation of Programs. Communications of the ACM, vol. 18, no. 8, 1975. pp. 453-457. doi:10.1145/360933.360975
[75]. Rosenband D.L., Arvind. Hardware Synthesis from Guarded Atomic Actions with Performance Specifications. Proc. International Conference on Computer-Aided Design (ICCAD), 2005. pp. 784-791. doi:10.1109/ICCAD.2005.1560170
[76]. BluespecTM SystemVerilog Reference Guide, 2012.
[77]. Clare C. Designing Logic Systems Using State Machines. McGraw-Hill, 1973. 150 p.
[78]. Baranov S. Logic and System Design of Digital Systems. TUT Press, 2008. 266 p.
[79]. Ivannikov V.P., Kamkin A.S., Chupilko M.M. Proverka korrektnosti povedeniya HDL-modelej tsifrovoj apparatury na osnove dinamicheskogo sopostavleniya trass [Checking Behavior Correctness of HDL Models of Digital Hardware By On-the-Fly Trace Matching]. Proc. Tools & Methods of Program Analysis (TMPA), 2013. pp. 71-82 (in Russian).
[80]. Chupilko M, Kamkin A. Runtime Verification Based on Executable Models: On-the-Fly Matching of Timed Traces. Proc. Model-Based Testing Workshop (MBT), 2013. pp. 67-81. doi: 10.4204/EPTCS.111.6
[81]. C++TESK tool — http://forge.ispras.ru/cpptesk
[82]. von Bochmann G, Haar S, Jard C, Jourdan G.V. Testing Systems Specified as Partial Order Input/Output Automata. Proc. International Conference on Testing of Software and Communicating Systems (TestCom), 2008. pp. 169-183. doi:10.1007/978-3-540-68524-1_13
[83]. Bourdonov I.B., Kossatchev A.S., Kuliamin V.V. Using Finite State Machines in Program Testing. Programming and Computer Software, vol. 26, no. 2, 2000, pp. 61–73.
[84]. Bourdonov I.B., Groshev S.G., Demakov A.V., Kamkin A.S., Kossatchev A.S., Sortov A.A. Parallel'noe testirovanie bol'shikh avtomatnykh modelej [Parallel Testing of Large Automata Models]. Vestnik Nizhegorodskogo universiteta im. N.I. Lobachevskogo [Vestnik of Lobachevsky State University of Nizhny Novgorod], no. 3, 2011. pp. 187-193 (in Russian).
[85]. Demakov A, Kamkin A, Sortov A. High-Performance Testing: Parallelizing Functional Tests for Computer Systems Using Distributed Graph Exploration. Proc. Open Cirrus Summit, 2011.
[86]. Bourdonov I.B., Kossatchev A.S. Obkhod neizvestnogo grafa kollektivom avtomatov [Traversal of an Unknown Graph by a Collective of Automata]. Proc. Nauchnyj servis v seti Internet: vse grani parallelizma [Scientific Service in Internet: All Facets of Parallelism], 2013. pp. 228-232 (in Russian).
[87]. Demakov A.V., Zelenova S.A., Zelenov S.V. Testirovanie parserov tekstov na formal'nykh yazykakh [Testing Parsers of Formal Languages]. Programmnye sistemy i instrumenty: Tematicheskij sbornik fakul'teta VMiK MGU [Software Systems and Tools: Proceedings of CMC Department of MSU], no. 2, 2001. pp. 150-156 (in Russian).
[88]. Zelenov S.V., Zelenova S.A., Kossatchev A.S., Petrenko A.K. Primenenie model'nogo podkhoda dlya avtomaticheskogo testirovaniya optimiziruyushhikh kompilyatorov [Application of the Model-Based Approach to Automated Testing of Optimized Compilers], 2003 (in Russian). (http://citforum.ru/SE/testing/compilers)
[89]. Wu L.-M., Wang K, Chiu C.-Y. A BNF-Based Automatic Test Program Generator for Compatible Microprocessor Verification. ACM Transactions on Design Automation of Electronic Systems, vol. 9, no. 1, 2004. pp. 105-132. doi:10.1145/966137.966142

[90]. Kamkin A.S. Nekotorye voprosy avtomatizatsii postroeniya testovykh programm dlya modulej obrabotki perekhodov mikroprotsessorov [Some Issues of Automation of Test Program Generation for Branch Units of Microprocessors]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 18, 2010. pp. 129-149 (in Russian).
[91]. Vorobyev D.N., Kamkin A.S. Generatsiya testovykh programm dlya mikroprotsessorov na osnove shablonov konvejernykh konfliktov [Test Program Generation for Microprocessors Based on Pipeline Hazards Templates]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 18, 2010. pp. 91-113 (in Russian).
[92]. MIPS64™ Architecture For Programmers, Revision 2.0. MIPS Technologies Inc, June 9 2003.
[93]. RM7000 Family User Manual. Issue 1, May 2001.
[94]. Kamkin A.S. Kombinatornaya generatsiya testovykh programm dlya mikroprotsessorov na osnove modelej [Combinatorial Model-Based Test Program Generation for Microprocessors]. Preprint ISP RAN [Preprint of ISP RAS], 2008. 18 p. (in Russian).
[95]. Kamkin A, Kornykhin E, Vorobyev D. Reconfigurable Model-Based Test Program Generator for Microprocessors. Proc. International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2011. pp. 47-54. doi:10.1109/ICSTW.2011.35
[96]. Kamkin A, Tatarnikov A. MicroTESK: An ADL-Based Reconfigurable Test Program Generator for Microprocessors. Proc. Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE), 2012. pp. 64-69.
[97]. Freericks M. The nML Machine Description Formalism. Techical Report. TU Berlin, FB20, Bericht, 1991/15. 47 p.
[98]. Target Compiler Technologies — http://www.retarget.com
[99]. Chandra S, Moona R. Retargetable Functional Simulator using High Level Processor Models. VLSI Design, 2000. pp. 424-429. doi:10.1109/ICVD.2000.812644
[100]. GLISS tool — http://www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id_rubrique=54
[101]. Cassé H, Barre J, Vaillant R, Sainrat P. Fast Instruction-Accurate Simulation with SimNML. Proc. Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO), 2011. pp. 8-12.
[102]. Kamkin A, Sergeeva T, Tatarnikov A, Utekhin A. MicroTESK: An Extendable Framework for Test Program Generation. Proc. Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE), 2013. pp. 51-57.
[103]. Ruby programming language — http://www.ruby-lang.org
[104]. Kornykhin E. SMT-Based Test Program Generation for Cache-Memory Testing. Proc. East-West Design & Test Symposium (EWDTS), 2009. pp. 124-127.
[105]. Kornykhin E. Generation of Test Data for Verification of Caching Mechanisms and Address Translation in Microprocessors. Programming and Computing Software, vol. 36, no. 1, 2010. pp. 28-35. doi:10.1134/S0361768810010056
[106]. Brandt J, Gemünde M, Schneider K, Shukla S, Talpin J.-P. Integrating System Descriptions by Clocked Guarded Actions. Proc. Forum on Design Languages (FDL), 2011. pp. 1-8.
[107]. Guglielmo G, Guglielmo L, Fummi F, Pravadelli G. Efficient Generation of Stimuli for Functional Verification by Backjumping Across Extended FSMs. Journal of Electronic Testing, vol. 27, no. 2, 2011. pp. 37-162. doi:10.1007/s10836-011-5209-8
[108]. Karaçali B, Tai K.-C., Vouk M.A. Deadlock Detection of EFSMs using Simultaneous Reachability Analysis. Proc. Dependable Systems and Networks (DSN), 2000. pp. 315-324. doi:10.1109/ICDSN.2000.857555

[109]. Kim A.K., Perekatov V.I., Ermakov S.G. Mikroprotsessory i vychislitel'nye kompleksy semejstva Elbrus [Microprocessors and Computer Systems of the Elbrus Family]. Saint-Petersburg, Piter, 2013. 272 p. (in Russian).
[110]. UML language — http://www.uml.org
[111]. Chatterjee S, Kishinevsky M, Ogras U. xMAS: Quick Formal Modeling of Communication Fabrics to Enable Verification. IEEE Design & Test of Computers, vol. 29, no. 3, 2011.pp. 80-88. doi:10.1109/MDT.2011.72
[112]. Holzmann G.J. The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, 2003. 608 p.
[113]. Kamkin A.S., Petrochenkov M.V. Sistema podderzhki verifikatsii realizatsij protokolov kogerentnosti s ispol'zovaniem formal'nykh metodov [A System to Support Formal Methods-Based Verification of Coherence Protocol Implementations]. Voprosy radioehlektroniki [Issues of Radio Electronics], no. 3, 2014. pp. 27-38 (in Russian).
[114]. Schneider K, Kropf T. A Unified Approach for Combining Different Formalisms for Hardware Verification. Proc. International Conference on Formal Methods in Computer Aided Design (FMCAD), 1996. pp. 202-217. doi:10.1007/BFb0031809
[115]. Dutertre B, Moura L. The YICES SMT Solver, 2006. (http://yices.csl.sri.com/tool-paper.pdf).
[116]. Moura L, Bjørner N. Z3: An Efficient SMT Solver. Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2008. pp. 337-340. doi:10.1007/978-3-540-78800-3_24
[117]. McMillan K.L. Symbolic Model Checking. Kluwer Academic, 1993. 194 p.
[118]. Java Constraint Solver API — http://forge.ispras.ru/projects/solver-api
[119]. Cok D.R. The SMT-LIBv2 Language and Tools: A Tutorial. GrammaTech, Inc., Version 1.1, 2011.
[120]. ZamiaCAD tool — http://zamiacad.sourceforge.net