**ИСП** Институт Системного Программирования
Российской Академии наук

# Труды Института Системного Программирования

**Том 20**

Москва 2011

**ИСП** **Учреждение Российской академии наук**
**Институт системного программирования РАН**

# Труды Института Системного Программирования

**Том 20**

Под редакцией
академика РАН В.П. Иванникова

Москва 2011

УДК51

Труды Института системного программирования: Том 20.
/Под ред. В.П. Иванникова/ – М.: ИСП РАН, 2011. – 296 с.

В двадцатом томе Трудов Института системного программирования публикуются статьи, содержащие результаты работ, проводимых в Институте в 2010-2011 годах.

# Содержание

# П р е д и с л о в и е

Очередной, 20-й том Трудов Института системного программирования РАН содержит 14 статей, посвященных различным аспектам системного программирования.

В статье А.Ю.Тихонова и А.И. Аветисяна «Развитие taint-анализа для решения задачи поиска некоторых типов закладок» задача анализа программного обеспечения (ПО) рассматривается как задача получения некоторых свойств исследуемого ПО. Этими свойствами могут быть и описания реализуемых алгоритмов, и информация о структурах файлов, данных в памяти или пакетов сетевых протоколов, и информация об имеющихся ошибках. В статье рассматривается подход к анализу программного обеспечения, представленного в виде исполняемого кода при отсутствии исходных текстов, с целью выявления некоторых видов программных закладок, которые, в соответствии с ГОСТ Р 51275-2006 являются преднамеренно внесенными в ПО функциональными объектами, при определенных условиях инициирующими реализацию недекларированных возможностей ПО.

Статья Д. В. Силакова «Использование аппаратной виртуализации в контексте информационной безопасности» посвящена возможностям использования аппаратной виртуализации для решения различных задач информационной безопасности. Предлагается обзор подходов к повышению безопасности программных систем, основанных на использовании виртуализации. Приводится обзор возможных сценариев использования виртуализации злоумышленниками. Указываются области применения и ограничения существующих решений и дальнейшие перспективы развития рассматриваемой области.

Кирилл Батузов и Алексей Меркулов представили статью«Оптимизация динамической двоичной трансляции». Двоичная трансляция – это процесс получения по заданной программе P программы Q, удовлетворяющей заданным требованиям, если обе программы записаны в виде машинных кодов. Если двоичная трансляция производится во время выполнения программы, то она называется динамической двоичной трансляцией. В статье рассматриваются возможности применения различных оптимизаций во время динамической двоичной трансляции.

В статье «Декларативный интерфейс поиска дефектов по синтаксическим деревьям: язык KAST» ее автор С. В. Сыромятников отмечает, что во многих случаях дефекты программного кода могут быть выявлены путём анализа соответствующего синтаксического дерева. Рассматриваются преимущества и недостатки данного подхода в сравнении с более сложными видами статического анализа, и обосновывается необходимость предоставления пользователю интерфейса для написания собственных обнаружителей дефектов. Обсуждаются различные подходы к реализации подобного

интерфейса. Описывается новый декларативный язык, позволяющий пользователю описывать дефекты кода в виде шаблонов для синтаксических деревьев, и рассматриваются некоторые аспекты работы анализатора этого языка.

В статье Л. В. Дворянского «О моделировании счётчиков с бесконечным числом значений в обыкновенных сетях Петри» проводится анализ моделирования в обыкновенных сетях Петри счетчиков с бесконечным числом состояний. Обосновывается выбор отношения эквивалентности симуляции готовности в качестве отношения реализации для моделирования счётчиков. Демонстрируется, что в сетях Петри невозможно промоделировать счётчики с бесконечным числом состояний. Представлена минимальная модель счётчика с конечным числом значений.

В.В. Липаев в своей статье «Кросс-система программирования ЯУЗА-6 для специализированных ЭВМ реального времени» рассматривает особенности и проблемы эффективного создания в 70-е годы сложных комплексов программ оборонного применения, базирующихся на специализированных ЭВМ. В статье представлены методы, требования и реализация адаптируемых кросс-систем автоматизации программирования и тестирования для различных типов сложных комплексов программ объектных ЭВМ. Изложены результаты многолетнего применения на БЭСМ-6 в ряде предприятий кросс-системы автоматизации программирования и отладки ЯУЗА-6.

Тематике систем реального времени посвящена и статья С.В. Зеленова «Планирование строго периодических задач в системах реального времени». Как указывает автор, одним из важнейших аспектов функционирования систем реального времени является планирование задач. Классические алгоритмы планирования периодических задач работают лишь в случае, когда время запуска каждой задачи может варьироваться внутри разных периодов ее выполнения. Однако в настоящее время имеется потребность в составлении таких расписаний, в которых время между соседними запусками одной периодической задачи было бы фиксировано и равнялось бы длине периода. Предлагается алгоритм построения расписаний, близких к оптимальным в смысле минимизации общего количество прерываний задач, с учетом этого дополнительного требования.

В статье Н. В. Пакулина и А. Н. Тугаенко «Тестирование протоколов электронной почты Интернета с использованием моделей» рассматриваются вопросы тестирования почтовых протоколов с использованием формальных моделей протоколов: предложен метод моделирования почтовых протоколов, рассмотрены особенности почтовых протоколов в контексте тестирования, представлены результаты тестирования популярных почтовых серверов с открытым кодом. В качестве примера представлена разработка тестовых наборов для протоколов SMTP и POP3 на языке JavaTESK – расширении языка Java, реализующем тестирование с применением формальных методов. Тестовые наборы состоят из двух частей: независимого тестирования

соответствия протоколов спецификациям и совместного теста, имитирующего работу почтовых протоколов в сети.

Тема тестирования на основе применения формальных методов продолжается в статье А.С. Камкина и М.М. Чупилко «Механизмы поддержки функционального тестирования моделей аппаратуры на разных уровнях абстракции». На разных этапах проектирования аппаратуры используются разные представления целевой системы. Соответственно, в зависимости от зрелости проекта применяются разные методы верификации, в частности, разные методы построения эталонных моделей, используемых для оценки корректности проектируемой аппаратуры. Различие в уровнях абстракции усложняет повторное использование тестовых систем, созданных в начале проектирования, для верификации аналогичных компонентов, но на более поздних этапах. В статье предлагается подход к построению эталонных моделей аппаратуры и тестовых оракулов на их основе, упрощающий повторное использование тестовых систем и снижающий затраты на верификацию.

В статье В.С. Мутилина, Е.М. Новикова, А.В. Страха, А.В. Хорошилова и П.Е. Шведа «Архитектура Linux Driver Verification» статье исследуются требования к построению архитектуры открытой системы верификации, которая могла бы предоставить площадку для экспериментов с различными методами статического анализа кода на реальном программном обеспечении и в то же время являлась полноценной системой верификации, готовой к индустриальному применению. По результатам обсуждения требований предлагается архитектура такой системы верификации и детально рассматриваются ее компоненты. В заключении описывается имеющийся практический опыт работы с предложенной архитектурой и предлагаются пути дальнейшего развития.

В статье С.Д. Кузнецова «Транзакционные параллельные СУБД: новая волна» отмечается, что в области систем управления данными без совместного использования ресурсов образовались два фронта: "NoSQL", где отрицаются основные принципы, свойственные СУБД, и "один размер непригоден для всех", где упор делается на специализацию систем при сохранении важнейших свойств СУБД. Особенно интересным является противостояние этих фронтов в области "транзакционных" систем управления данными. Опираясь на "теорему" CAP Эрика Брювера, представители лагеря NoSQL отказываются от обеспечения в своих системах традиционных свойств ACID. В статье обсуждается суть "теоремы" Брювера, и обосновывается, что она не имеет отношения к свойствам ACID. Рассматриваются современные исследовательские работы, обеспечивающие классические ACID-транзакции в параллельных средах без общих ресурсов, а также наиболее здравые подходы, в которых из чисто прагматических соображений свойства ACID частично ослабляются.

Е. А. Костычев, В. А. Омельченко и С.В. Зеленов в статье «Нацеленная генерация данных для тестирования приложений над базами данных» справедливо утверждают, что широко распространенным типом программного обеспечения являются приложения, обрабатывающие большие массивы данных. В частности, такие приложения решают задачи интеграции данных в области интеграции корпоративных приложений. Специфика функционального тестирования подобных приложений заключается в большом количестве возможных комбинаций входных данных. При использовании существующих подходов гарантированное покрытие функциональности тестируемого приложения может быть достигнуто только полным перебором возможных комбинаций исходных данных. При помощи подхода генерации данных, предлагаемого в статье, возможно достижение покрытия функциональности приложения с близким к оптимальному объемом данных.

В статье А.В. Коршунова «Извлечение ключевых терминов из сообщений микроблогов с помощью Википедии» описывается способ извлечения ключевых терминов из сообщений микроблогов с использованием информации, полученной путём анализа структуры и содержимого интернет-энциклопедии Википедия. Работа алгоритма основана на расчёте для каждого термина его «информативности», т.е. оценки вероятности того, что он может быть выбран ключевым в тексте. В ходе тестирования разработанный алгоритм показал удовлетворительные результаты в условиях поставленной задачи, существенно опережая аналоги. В качестве демонстрации возможного применения разработанного алгоритма был реализован прототип системы контекстной рекламы. Сформулированы также варианты использования информации, полученной путём анализа сообщений Twitter, для реализации различных вспомогательных сервисов.

Завершает том статья И. С. Таранова «Использование префиксного дерева для хранения и поиска строк во внешней памяти». В этой статье представлена структура данных для поиска и эффективного хранения во внешней памяти массивов текстовых строк, реализованная для поддержки индексов в XML СУБД Sedna. Описываются алгоритмы для вставки, удаления и поиска строк переменной длинны в префиксных деревьях, хранимых на дисках. Приводятся результаты сравнения реализации новой структуры с существующей реализацией B-дерева. Показано, что в некоторых случаях предложенная структура данных занимает в несколько раз меньше места во внешней памяти при той же скорости поиска.

Академик РАН В.П. Иванников

# Развитие taint-анализа для решения задачи поиска программных закладок

*А.Ю.Тихонов, А.И. Аветисян*
*{fireboo@ispras.ru}, {arut@ispras.ru}*

**Аннотация.** Задачу анализа программного обеспечения (ПО) будем рассматривать как задачу получения некоторых свойств исследуемого ПО. Это могут быть и описания реализуемых алгоритмов, и информация о структурах файлов, данных в памяти или пакетов сетевых протоколов, и информация об имеющихся ошибках. В данной статье будет рассмотрен подход к анализу программного обеспечения, представленного в виде исполняемого кода при отсутствии исходных текстов, с целью выявления некоторых видов программных закладок. Согласно ГОСТ Р 51275-2006 программная закладка – это преднамеренно внесенный в ПО функциональный объект, который при определенных условиях инициирует реализацию недекларированных возможностей ПО. Программная закладка может быть реализована в виде вредоносной программы или программного кода.
Предлагаемый подход к анализу описывается в пятом разделе статьи. Первые четыре раздела кратко описывают принципы работы системы анализа программного обеспечения по трассе исполнения программы TREX, в рамках которой ведется практическая реализация данного подхода.

**Ключевые слова:** безопасность ПО, недекларированные возможности, вредоносный код, анализ уязвимостей

## *1. Введение*

Система TREX предназначена для анализа ПО, представленного в виде исполняемого кода, при отсутствии исходных текстов [1], [2], [3]. В качестве анализируемого программного обеспечения выступает не какая-то отдельно взятая программа или исполняемый модуль, а вся программная система, код которой выполняется процессором. Предполагается, что анализируемая система может противодействовать анализу путем использования различных методов защиты от статического и динамического анализа — такое предположение верно как по отношению к вредоносному ПО, так и по отношению к обычным программам, разработчики которых хотели защититься, например, от несанкционированной модификации кода. В основе алгоритмов анализа трассы лежат принципы анализа потока управления и потока данных.

Рассматриваемый в работе подход основан на известных методах анализа потока управления и данных. Особое внимание уделено уточнению понятия закладки. Существующие определения даны в нормативной базе, которая создавалась в первую очередь под решение задач сертификации на отсутствие недекларированных возможностей (НДВ). При этом разработчик обязан предоставить исходные тексты, а в некоторых случаях — и проектную документацию, которую можно использовать как основу для построения формальной модели исследуемой системы. В реальной жизни исследуемая программа функционирует в недоверенном программном окружении, запрашивая сервисы со стороны такого окружения и подвергаясь его воздействию. В этом случае анализируемой системой является вся программная система в целом. Перед началом анализа в распоряжении аналитика нет ни исходных текстов, ни проектной документации. Нормативные документы анализ в таких условиях не предусматривают. Кроме того, зарубежные исследователи отмечают принципиальную невозможность проведения анализа программ только на уровне исходных текстов, без анализа исполняемого кода — так называемая проблема WYSINWYX (What You See Is Not What You eXecute) [4]. Важным вопросом является построение формального описания модели исследуемой системы, на основе которого можно было бы различать допустимые и недопустимые пути исполнения или состояния программы. Такое описание можно назвать политикой безопасности, вот только для возможности использования при анализе исполняемого кода объекты и субъекты такой политики должны в итоге сводиться к уровню машинных команд, ячеек памяти и состояний программы. Существующие нормативные документы не описывают как это сделать.

В работе сделана попытка заполнить пробел между существующей нормативной базой и практикой при решении задачи поиска вредоносного кода, реализующего некоторые виды программных закладок. В основе подхода лежат методы слайсинга и динамического taint-анализа. В текущем виде описанный подход является вариантом динамического анализа кода, при котором отслеживаются зависимости между инструкциями по управлению и по данным, фактически возникающие при выполнении программы на конкретном наборе тестовых данных. Вследствие этого основным недостатком является невозможность анализа всех возможных путей исполнения и состояний программы. Преодоление этих недостатков видится в использовании методов символьного исполнения программы (symbolic execution), что является темой дальнейших исследований. Стоит отметить, что комбинация динамического анализа по трассе и символьного исполнения лежит в основе ряда зарубежных исследований, направленных на разработку практических систем анализа программного обеспечения [5].

## 2. Требования к содержимому трассы

Для обеспечения корректности проведения анализа потока управления и потока данных в трассе должна содержаться информация, полностью определяющая процесс преобразования, выполняемый над ячейками памяти и регистрами процессора:

- информация о непрерывная последовательность выполненных процессором инструкций;
- информация о смене режимов работы процессора и значений системных регистров, влияющих на декодирование инструкций, а также информация, требуемая для корректного дизассемблирования инструкции (см. раздел 2).
- информация о возможность восстановления содержимого любого регистра общего назначения на любом шаге;
- информация о факте прямой модификации содержимого ячейки памяти (запись в режиме DMA), т.е. модификации памяти, вызванной не выполнением процессорной инструкции, а работой других устройств;
- информация о факте асинхронного вызова обработчика прерывания, т.е. вызова обработчика прерывания не по причине выполнения специальной процессорной инструкции, а по причине возникновения аппаратного события;
- информация о содержимом ячеек памяти, определяющих поведение конкретных инструкций. Особенно важно знать содержимое ячейки, если она влияет на зависимости по данным, формируемые инструкцией. В случае системы команд x86 примером такой инструкции является CMPXCHG/CMPXCHG8B/CMPXCHG16B;
- информация о некоторых случаях требуется возможность восстановления содержимого ячеек памяти на момент выполнения конкретного шага трассы (см. раздел 4 поиск закладок);
- информация о случаях, когда процессор различает виртуальную и физическую адресацию памяти, требуется механизм отождествления физически одинаковых ячеек памяти, имеющих различающиеся виртуальные адреса, по которым эти ячейки адресуются процессором;
- информация о случаях, когда процессор и/или операционная система поддерживает многозадачность, требуется механизм идентификации задач для того, чтобы уметь определять, какой из множества выполняемых задач принадлежит текущая инструкция.

Не всю эту информацию необходимо получать непосредственно на этапе получения трассы. Часть информации может быть восстановлена на этапе обработки уже полученной трассы.

## 3. Декодирование и дизассемблирование инструкций процессора

Инструкции процессора имеют числовое представление и хранятся в памяти компьютера. В случае компьютеров, построенных на базе архитектуры фон-Неймана инструкции хранятся в той же памяти, что и данные, и по сути ничем от данных не отличаются. Так же как и обычные данные, код инструкций может быть прочитан и модифицирован. Это приводит к известной проблеме неразличимости инструкций и данных. Кроме того, в некоторых процессорных архитектурах размер инструкций может различаться, в результате чего даже внутри линейного блока кода невозможно декодировать одну инструкцию, не декодировав все ей предшествующие. В зависимости от режима работы процессора интерпретация одного и того же числового значения может различаться. На интерпретацию инструкции может влиять ее положение в памяти — например в случае, когда код инструкции содержит относительный адрес, т.е. когда адрес ячейки памяти, к которой обращается инструкция, должен быть вычислен относительно положения инструкции в памяти. Еще одна проблема — различение констант и адресов. Одна и та же, корректно декодированная инструкция, с точки зрения представления на языке ассемблера может быть представлена по-разному, например, как инструкция помещения в регистр числовой константы, либо как инструкция помещения в регистр адреса ячейки памяти.

Подводя итог сказанному, будем различать термины декодирование и дизассемблирование инструкции.

Декодирование — процесс интерпретации кода инструкции, при котором мы получаем информацию о функциональности инструкции. Декодирование — неоднозначное преобразование, если нет информации о состоянии процессора. При динамическом анализе декодирование — однозначно, при статическом — неоднозначно.

Дизассемблирование — это получение корректного ассемблерного представления инструкции, при котором выполняется декодирование и получение информации о смысле инструкции. Рассмотрим случай с инструкцией, помещающей константу в регистр. Константа возможно является адресом. Вне зависимости от этого ее значение будет помещено в регистр. В случае корректного декодирования такой инструкции, но некорректного дизассемблирования будут корректно учтены функциональные зависимости по данным, и некорректно — адресные зависимости. Корректность распознавания адресов можно обеспечить только в рамках конкретной трассы, причем в общем случае с ошибками второго рода. Например, с помощью слайсинга (см. раздел 3) можно отследить использование константы для поиска ситуации, когда она явным образом используется в качестве адреса. Но если константа пройдет через серию преобразований, результатом которых будет исходное значение — анализ кода не поможет, потребуется полноценный анализ алгоритмов.

Единственный способ решения — явный контроль обращений по адресам и проверка — совпадает ли значение проверяемой константы с адресом. При таком подходе все константные адреса, к которым шло обращение в трассе, будут обнаружены (нет ошибок первого рода), но могут быть ложные отождествления константы с адресом (возможны ошибки второго рода).

## *4. Принципы анализа трассы. Слайсинг и taint-анализ*

В основе существующих подходов к решению задач анализа потока управления и потока данных лежит предположение о том, что для исследуемой программы известны корректный граф потока управления — **CFG** (Control Flow Graph) и граф зависимостей программы — **PDG** (Program Dependence Graph). Неформально — CFG описывает поток управления, т.е. возможный порядок исполнения инструкций, его вершинами являются операторы программы, а ориентированные связи между вершинами показывают допустимый порядок исполнения операторов. PDG описывает отношения зависимостей по данным и управлению между операторами на множестве вершин GFG. Связь первого типа идет от одного оператора к другому, если результаты вычислений одного оператора используются в качестве входных данных для вычислений другого. Связь второго типа идет в том случае, если выполнение одного оператора определяет будет ли выполняться другой оператор. На практике, особенно при работе с исполняемым кодом, зачастую неизвестно ни полное множество инструкций, ни связи между ними по управлению и по данным.

Основой подхода, описываемого в данной работе, является получение знаний об инструкциях исследуемой программы, порядке их выполнения и зависимостях между ними по данным напрямую, путем наблюдения за работой программы в интервале времени, в который укладывается работа интересующего аналитика алгоритма. Результатом такого наблюдения является **трасса исполнения программы**, представляющая собой описание последовательности шагов выполнения программы, для каждого из которых сохранен код выполненной инструкции и состояние всех регистров процессора. Основная идея – свести трассу к представлению в виде CFG и PDG и адаптировать к их обработке известные алгоритмы. Преимуществом данного подхода является то, что он позволяет получать гарантированные и точные результаты. Недостатком — то, что эти результаты относятся только к той части алгоритма, которая была выполнена. Оставшиеся незадействованными ветки алгоритма анализироваться не будут. Тем не менее, предлагаемый подход предпочтителен по сравнению с существующими, поскольку для выполнившейся части алгоритма анализ полон и не содержит неточностей, которые возникают у других подходов. Для оставшейся непроанализированной части алгоритма можно адаптировать существующие подходы к статическому и динамическому анализу, суммарно получив более полные результаты анализа.

В предлагаемом подходе различаются четыре типа **зависимостей**:

- функциональная зависимость — когда результат вычисления на одном шаге напрямую зависит от результата вычисления на другом;
- адресная зависимость — когда выбор ячейки, от которой функционально зависит результат вычисления на одном шаге, зависит от результата вычисления на другом шаге;
- зависимость по управлению;
- зависимость по коду инструкции — когда код инструкции, выполняющейся на одном шаге, зависит от результата вычисления на другом. Рассмотрение данной зависимости важно при анализе динамически изменяемого кода программы.

Суть слайсинга по трассе — отбор из трассы шагов в соответствии с зависимостями, позволяющими отнести эти шаги к интересующему аналитика алгоритму преобразования входных ячеек в выходные. Шаг трассы отбирается в слайс, если выполненная на этом шаге инструкция **зависит** хотя бы от одной указанной аналитиком входной ячейки, или от результата работы инструкции **зависит** хотя бы одна выходная ячейка.

Taint-анализ при анализе трассы по сути мало отличается от слайсинга, только отбираются не шаги (инструкции), а данные (ячейки памяти и регистры). Перед началом анализа аналитик помечает некоторые данные, исходя из своих соображений, после чего осуществляется продвижение пометок (taint propagation) вверх или вниз по трассе в соответствии с зависимостями по помеченных данным: если входом инструкции на некотором шаге являются помеченные данные, то выходы инструкции также помечаются для анализа на следующем шаге. Если выходом инструкции на некотором шаге являются помеченные данные, то входы инструкции помечаются для анализа на предыдущем шаге.

Отличия между слайсингом и taint-анализом проявятся в предлагаемом для поиска закладок подходе, когда с этапом продвижения меток может быть связана дополнительная обработка, рассмотренная в следующем разделе.

## *5. Поиск закладок*

В настоящее время наблюдается "эпидемия" вредоносных программ, ориентированных на кражу конфиденциальных данных и тем самым наносящих вред конкретному пользователю. Под них могут маскироваться и шпионские программы, разработанных спецслужбами различных государств, которые причиняют ущерб уже на государственном уровне. Для выявления вредоносного кода и оценки возможного ущерба иногда полезно выяснить цель атаки — конкретные данные, на добычу которых ориентирована такая программа.

Вначале требуется формально определить, что является закладкой. Без такого определения не ясно, что требуется найти, а потому невозможно предложить методы поиска. В Российских стандартах имеется два сходных и довольно общих определения:

1. Программная закладка — Преднамеренно внесенный в программное обеспечение функциональный объект, который при определенных условиях инициирует реализацию недекларированных возможностей программного обеспечения. Программная закладка может быть реализована в виде вредоносной программы или программного кода — ГОСТ Р 51275-2006 [6].

2. Программная закладка — скрытно внесенный в программное обеспечение функциональный объект, который при определенных условиях способен обеспечить несанкционированное программное воздействие. Программная закладка может быть реализована в виде вредоносной программы или программного кода [7].

Проанализируем эти определения с точки зрения возможности их использования для формирования принципов обнаружения закладок.

Первое определение:

- преднамеренность — методами анализа исполняемого кода доказать проблематично;
- функциональный объект это присуще только модели программной системы, при анализе бинарного кода такого понятия нет;
- недекларированные возможности ПО — в соответствии с ГОСТ Р 51275-2006 [6] — функциональные возможности программного обеспечения, не описанные в документации. При отсутствии модели программной системы, описанной ее разработчиком — понятие бессмысленное.

Второе определение:

- скрытное внесение функциональности — так же как и для преднамеренности трудно доказуемо. Скрытность может являться следствием применения механизмов защиты легального кода от анализа;
- несанкционированное программное воздействие — наиболее близкое определение дано в п.2.6.6. ГОСТ Р 50922-2006 [8]: "несанкционированное воздействие на информацию: воздействие на защищаемую информацию с нарушением установленных прав и (или) правил доступа, приводящее к утечке, искажению, подделке, уничтожению, блокированию доступа к информации, а также к утрате, уничтожению или сбою функционирования носителя информации". Это определение подразумевает наличие модели программы, описывающей правила обработки информации. При

анализе бинарного кода такого понятия нет, требуется построение модели программы, отражающей политику безопасности, что само по себе является сложной задачей.

Вывод — данные определения давались под конкретный подход к анализу программного обеспечения по исходным текстам при наличии формального описания модели. Такие данные для анализа бинарного кода использованы быть не могут.

Сформулируем более приемлемое определение на основе стандартизованного понятия безопасности информации.

В настоящее время под безопасностью информации принято понимать состояние защищенности информации, при котором обеспечиваются её конфиденциальность, доступность и целостность [8]. Иногда выделяют и другие признаки безопасности: невозможность отказа от авторства (non-repudiation), подотчётность (accountability), достоверность (reliability), аутентичность или подлинность (authenticity).

В соответствии с п. 2.6.5. ГОСТ Р 50922-2006 [8] вредоносная программа: программа, предназначенная для осуществления несанкционированного доступа к информации и (или) воздействия на информацию или ресурсы информационной системы.

По сути, вредоносная программа ориентирована на нарушение безопасности информации путем устранения одного или нескольких перечисленных выше признаков безопасности.

Наиболее очевидным нарушением безопасности, на которое ориентирована значительная часть вредоносных программ, является нарушение конфиденциальности, а по сути — кража информации.

**Конфиденциальность** — одно из свойств безопасности информации (свойство обеспечивающее безопасное состояние информации). В соответствии с ГОСТ 17799-2005 [[9]] будем понимать его как "обеспечение доступа к информации только авторизованным пользователям".

В данной работе предлагается подход к поиску вредоносного кода, предназначенного для компрометации секретных данных пользователя, т.е. к нарушению конфиденциальности. Неформально будем называть такой код **закладкой, нарушающей конфиденциальность**, далее по тексту — **закладкой**. Рассматриваемый подход может быть распространен и на поиск других типов закладок, но это тема отдельной работы.

Объектом анализа в случае динамического анализа исполняемого кода является последовательность инструкций, выполняемых процессором. Инструкции выполняют некоторые преобразования данных, содержащихся в процессорных регистрах или ячейках памяти. Каждая инструкция может иметь входные данные, сформированные ранее выполненными инструкциями, и выходные данные, являющиеся результатом выполнения инструкции. Получение данных извне процессора и выдача данных вовне процессора

реализуется специальными инструкциями, ячейками памяти и портами ввода/вывода. Любая информация, обрабатываемая процессором, должна быть введена в него извне, и после обработки выведена вовне. Зависимости между инструкциями по входным/выходным данным образуют поток данных. Поток данных имеет несколько начальных точек — инструкций, вводящих данные в процессор, и несколько конечных точек — инструкций, выводящих данные из процессора. Вычислительную систему можно рассматривать и более широко — не только как процессор, но и как все аппаратное обеспечение, составляющее вычислительную систему (компьютер), или даже вычислительную систему из нескольких взаимодействующих компьютеров. В этом случае требуется обеспечить непрерывность отслеживание потока данных между различными аппаратными компонентами вычислительной системы, например передачу данных из процессора на жесткий диск и обратно.

Если снабдить функционирующую в вычислительной системе информацию специальной пометкой, отражающей ее конфиденциальность, и контролировать потоки данных, обрабатывающие эту информацию, можно выявить все факты выдачи такой информации вовне вычислительной системы, после чего аналитик должен решить — что из них будет являться нарушением конфиденциальности. На этом принципе основан **метод анализа помеченных данных (taint-анализ)**.

Закладка должна нарушать конфиденциальность некоторых данных, для чего эти данные посредством выполнения процессорных инструкций должны попасть вовне вычислительной системы. Очевидно, что какая-то часть таких инструкций может принадлежать легальному коду операционной системы. Это необходимое, но недостаточное условие для того, чтобы отнести некоторый код к закладке. Например, код отображения конфиденциальных данных на экран может быть вполне легальным, если делает это по запросу пользователя. Легален или нет доступ к конфиденциальным данным, должно описываться политикой безопасности исследуемой системы. При начале анализа исполняемого кода она, как правило, неизвестна аналитику, и его задачей является восстановить ее в ходе анализа, и в соответствии с ней принимать решение о допустимости или недопустимости конкретного обращения к данным. Вопрос восстановления политики безопасности по коду программы является темой отдельных исследований.

С точки зрения стойкости закладки к обнаружению методами проверки работоспособности легального алгоритма на контрольных примерах, закладка не должна влиять на результаты использования компрометируемых секретных данных легальным алгоритмом.

Передача секретных данных вовне вычислительной системы может быть связана с их преобразованием, т.е. вычислением некоторой функции от исходных секретных данных. Чтобы нарушать конфиденциальность, данная функция должна быть обратима при условии контроля атакующей стороной переданных вовне вычислительной системы данных.

Если секретные данные попали вовне вычислительной системы — они скомпрометированы, при условии, что они **доступны**.

Здесь под **доступностью** понимается возможность вычисления обратной функции, т.е. возможность вычисления исходных секретных данных. Если это функция симметричного шифрования с секретным ключом, должен быть доступен секретный ключ и выход прямого алгоритма. При этом ключ шифрования либо был прошит в реализацию закладки, либо получен извне анализируемой системы. Такой ключ может быть извлечен аналитиком посредством анализа потоков данных, после чего ему становятся доступны компрометируемые закладкой секретные данные. Если используется функция асимметричного шифрования — ключ должен принадлежать атакующему, причем извлечение ключа прямого преобразования из программной реализации не позволит вычислять обратное преобразование. Иными словами, компрометируемые секретные данные будут доступны только владельцу ключа (которым автор закладки может даже и не являться). На данном принципе основаны некоторые схемы депонирования ключей.

Имеет также смысл говорить о **частичной доступности**, если компрометируется только часть секретных данных.

Проверка обратимости в общем случае является криптоаналитической задачей, и методами анализа потоков данных решена быть не может. Однако можно сформулировать частный случай, когда такую задачу можно свести к задаче анализа потоков данных — когда функция преобразования является известной криптографической функцией шифрования или комбинацией таких функций. В этом случае для доказательства доступности компрометируемых данных достаточно показать доступность ключевых данных алгоритма шифрования.

Наиболее подходящим подходом для решения задачи поиска рассматриваемого типа закладок является taint-анализ. В известных определениях taint-анализа при продвижении меток обратимость вычислений не учитывается. Для исправления этого недостатка операции, анализируемые алгоритмом продвижения меток, должны быть отнесены к классу прямого или обратного преобразования конкретного алгоритма. При прохождении помеченных данных через такую операцию ее результат также помечается, однако при этом пометка "обертывается" признаком выполненного преобразования (функция + ключ). Если в дальнейшем такой "конверт" попадает на операцию, отнесенную к обратному преобразованию той же функции, при совпадении ключей "обертка" снимается и остаются содержащиеся в ней помеченные данные. Разновидность такого подхода — для некоторых реализаций криптоалгоритмов ключи должны также снабжаться пометками "прямой/обратный ключ".

Важным вопросом является проверка совпадения ключей. Если между созданием и снятием "конверта" ключ был преобразован некоторым алгоритмом, необходимо ответить на вопрос — была ли произведена реальная

модификация ключа, либо выполненное преобразование оставило ключ неизменным — что может быть при намеренном запутывании алгоритма для усложнения анализа. В общем случае ответ на такой вопрос не может быть получен анализом потоков данных и управления. Однако при динамическом анализе можно просто сравнить содержимое ключа на момент выполнения прямого и обратного преобразования. При анализе трассы это предъявляет дополнительное требование к системе получения и обработке трассы — необходимо обеспечить **возможность восстановления содержимого памяти на момент выполнения конкретного шага трассы** (см. раздел 1 требования к содержимому трассы).

Рассмотренные выше ситуации позволяют дать определение понятия закладки в терминах, близких к анализу потоков данных и управления.

Определим закладку как обратимый алгоритм преобразования секретных данных, порождающий доступные выходные данные, помимо указанного аналитиком (санкционированного) выхода.

**Консервативный анализ** считает любой алгоритм преобразования секретных данных обратимым, а любой способ выдачи секретных данных вовне системы — несанкционированным доступом (политика безопасности по умолчанию). По мере выдачи системой анализа ложных, по мнению аналитика, сообщений о найденных закладках, аналитик уточняет политику безопасности, описывая конкретное обращение к данным как разрешенное. Задача восстановления политики безопасности по коду программы является темой отдельного исследования.

Поскольку окончательное решение о том, является найденный алгоритм закладкой или нет, принимает аналитик, процесс такого поиска правильнее называть **поиском мест, подозрительных на наличие закладок**.

Если закладка влияет на выход легального алгоритма — она потенциально **ослабляет этот алгоритм**.

**Если одновременно с ослаблением отсутствует компрометация секретных данных — закладки как алгоритма не существует, поскольку не существует выходных данных никакого другого алгоритма, кроме ослабляемого легального алгоритма.** В этом случае мы имеем дело со слабостью, присущей самому легальному алгоритму. Слабость являющаяся частью легального алгоритма, методами анализа потоков данных не выявляется, выявляется методами криптоанализа.

Выявление закладки в сформулированном выше смысле средствами только динамического анализа возможно только в том случае, когда закладка функционировала в момент проведения анализа. Более точно — поток данных, образованный кодом закладки, должен "ответвляться" от помеченных секретных данных в интервале времени снятия трассы, и иметь хотя бы один выход вовне контролируемой системы. Во временном интервале с момента ответвления кода закладки от секретных данных до момента вывода

преобразованных секретных данных вовне системы должна быть обеспечена непрерывность контроля над потоком данных. Такая непрерывность естественным образом обеспечивается в случае анализа по трассе исполнения программы. В случае статического анализа машинного кода в силу неполноты CFG и PDG (некорректность и неполнота множества инструкций, множества связей по управлению и множества связей по данным) непрерывность анализа обеспечить очень сложно.

В ситуации, когда аналитик не знает происхождение некоторых входных данных и потому не может выставить для них пометки (сбор трассы начался, когда известные аналитику секретные данные уже были неподконтрольно преобразованы), проведение анализа потока данных невозможно.

В ситуации, когда помеченные данные находятся в обработке, но не попали вовне системы (раннее прекращение контроля), возможно проведение дальнейшего анализа. Подходы к такому анализу могут быть различными, например рестарт вычислительной системы с состояния, сохраненного на момент завершения анализа (возможно на симуляторах), эмуляция или статический анализ незадействованных веток кода на предмет возможного доступа к помеченным данным.

Преимуществами подхода, основанного на taint-анализе, является относительная простота задания политики безопасности в виде пометок входных данных на некотором шаге трассы и ее последующей модификации. Необходимо задать начальную разметку секретных данных, что при известном интерфейсе вычислительной системы с внешним миром не является слишком сложной задачей. Например, если помеченными данными является содержимое файла на жестком диске, началом отслеживаемого потока данных могут являться функции ОС доступа к файлу, или, если интерфейс ОС неизвестен — низкоуровневые процессорные инструкции доступа к диску. Если осуществляется ввод помеченных данных с клавиатуры — началом отслеживаемого потока данных могут являться функции ОС получения данных с клавиатуры, из окна пользовательского интерфейса, или низкоуровневые процессорные инструкции чтения данных с клавиатуры. Точки вывода информации вовне вычислительной системы могут также определяться либо соответствующими функциями ОС, либо процессорными инструкциями, осуществляющими низкоуровневое взаимодействие с периферийным оборудованием.

В случае консервативного анализа требуется только начальная разметка. Для проведения полного анализа потребуется восстановление модели алгоритма, составляющего поток помеченных данных. Даже в этом случае это более простая задача, чем построение модели системы в целом, что необходимо для проведения анализа в соответствии с руководящими документами [7].

Очевидными подходами к затруднению анализа на наличие закладок являются:

- порождение большого числа выходных данных, зависящих от секретной информации, кроме существенных выходных данных;
- использование нестандартных криптографических преобразований, чтобы вынудить проводить криптоанализ для определения доступности выходных секретных данных;
- смягчение требования к обратимости алгоритма — алгоритм может быть необратим, но иметь ограниченное количество эффективно вычисляемых обратных значений;
- расширение границ анализируемой системы.

Рассмотрим подробнее последний подход. Предположим, анализ потока данных осуществляется в пределах процессора. Если помеченные данные будут сохранены на диск, а затем заново прочитаны с него, то в рамках анализа зависимостей между процессорными инструкциями сохранить пометку у прочитанных с диска данных не получится. Потребуется формализовать взаимодействие с диском. Например, определить правила распространения пометок при прохождении потока управления через функции чтения/записи с диска. Проблема — при использовании различных способов чтения/записи. Например, запись на файловом уровне, а чтение — на секторном. Другой пример — передача помеченных данных "в конверте", ключ которого не доступен вовне системы, на другой компьютер, а затем возврат таких данных обратно, "снятие конверта" и передача результата вовне системы. Для анализа такой ситуации требуется проводить "сквозной" анализ потоков данных через вычислительную систему из двух компьютеров. При анализе только одного из них, имеем незамкнутую систему — наблюдаем только часть потока управления.

В качестве примера рассмотрим схему, используемую в некоторых реальных системах — см. рис. 1.

В принципе, анализ мог быть начат с любой части потока данных, изображенного на рисунке — с записи в файл, генерации ключа или чтения открытого текста из файла — в зависимости от того, какую из этих точек аналитик обнаружит первой. Предположим, анализ начался с момента, когда аналитик обнаружил функцию AES-шифрования, преобразующую открытый текст *ОТ* в шифр-текст *ШТ* на основе ключа *Ключ1*. Шифр-текст впоследствии сохраняется в файле на диске, т.е. в нашем понимании поступает вовне анализируемой системы. Аналитик объявляет *ОТ* и *Ключ1* секретными данными (помечает их), и описывает модель функции AES (входные параметры — шифруемый буфер и ключ, выходные параметры — зашифрованный буфер, функция обратима при известных ключе и зашифрованном буфере). После этого аналитик выполняет анализ потока данных, состоящий из комбинации восходящего и нисходящего анализа. При восходящем анализе определяется, как были получены данные, при нисходящем — как они будут обработаны.

Анализ начинается с проверки преобразований над секретными данными (*ОТ* и *Ключ1*):

- восходящий анализ по трассе для *ОТ* показывает, что данные были прочитаны из файла, по всем промежуточным результатам чтения нисходящим анализом проверяется, что ни в каких других вычислениях они участия не принимали
- нисходящий анализ для *ОТ* показывает, что больше ни в каких вычислениях данные участие не принимали
- восходящий анализ по трассе для *Ключ1* показывает, что данные были получены как результат вызова функции setkey (инициализация ключа из ключевого материала), в свою очередь функция setkey получает данные из псевдослучайного генератора rand.
- нисходящий анализ для *Ключ1* показывает, что он шифруется на ключе *Ключ2* алгоритмом AES, результат сохраняется в файл.

По аналогии восстанавливается полная схема.

Обнаружив, что значение имеющего пометку ключа *Ключ1* зависит от *состояния генератора случайных чисел (ГСЧ)*, система анализа пометит и *состояние ГСЧ*. Поскольку по умолчанию система анализа рассматривает любой алгоритм как обратимый (для консервативности анализа), а на помеченное *состояние ГСЧ* влияли все предыдущие состояния, все они также будут помечены. Поскольку часть из этих состояний в итоге попадают в виде заполнителя в файл на диске, соответствующий код объявляется как алгоритм, подозреваемый на наличие закладки. В дальнейшем аналитик должен выполнить анализ алгоритма ГСЧ, и если он будет обратим — найденное место действительно является закладкой.

Анализируя часть потока данных, где помеченный *Ключ1* шифруется с помощью алгоритма AES на ключе *Ключ2*, а результат попадает в файл, в итоге помечаются *Ключ2* и *пароль*. В ходе дальнейшего анализа проверяется, что никаких результатов преобразований над помеченными данными вовне системы не попадает.

Приведенный пример сильно упрощен — в случае реального анализа реализации сходного алгоритма количество проверочных точек (фактически — вершин графа потока данных), в которых осуществляется восходящий-нисходящий анализ, может достигать десятков и сотен тысяч.

## 6. Заключение

В работе предложен подход к поиску вредоносного кода, предназначенного для компрометации секретных данных пользователя, т.е. к нарушению конфиденциальности. В работе такой код назван закладкой, нарушающей конфиденциальность, или просто — закладкой. Необходимость в таком

определении закладки возникла в силу малопригодности существующих в ГОСТ определений к формулированию подходов к ее поиску средствами анализа потоков управления и данных. Показано, что в общем случае задача поиска закладок средствами анализа потоков данных сводится к алгоритмически неразрешимым задачам. Однако в случаях, когда аналитику известны применяемые в программе алгоритмы преобразования данных, такая задача разрешима. Подобный подход может представлять практическую ценность, поскольку многие разработчики ПО при разработке используют стандартные криптоалгоритмы, которые достаточно просто распознаются в коде. Рассматриваемый подход может быть распространен и на поиск некоторых других типов закладок.

Предложенный в работе подход является развитием динамического taint-анализа и реализуется в рамках системы динамического анализа программного обеспечения по трассе исполнения программы TREX.

**Литература**


[1] А. Ю. Тихонов. Ключевые технологии, способствующие решению задач восстановления и анализа алгоритмов. Журнал Безопасность Информационных Технологий № 2008-1.

[2] Тихонов А.Ю., Аветисян А.И., Падарян В.А. Извлечение алгоритма из бинарного кода на основе динамического анализа. // Труды XVII общероссийской научно-технической конференции «Методы и технические средства обеспечения безопасности информации». Санкт-Петербург, 07-11 июля 2008 г. Стр. 109.

[3] В.А. Падарян, А.И. Гетьман, М.А. Соловьев. Программная среда для динамического анализа бинарного кода. Труды Института системного программирования РАН, том 17, 2009 г. Стр. 51-72.

[4] Balakrishnan, G. and Reps, T., WYSINWYX: What You See Is Not What You eXecute. In ACM Transactions on Programming Languages and Systems, vol.32 Issue 6, August 2010.

[5] Edward J. Schwartz, Thanassis Avgerinos, David Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask), Proceedings of the 2010 IEEE Symposium on Security and Privacy.

[6] ГОСТ Р 51275-2006 — Защита информации. Объект информатизации. Факторы, воздействующие на информацию.

[7] Руководящий документ "Защита от несанкционированного доступа к информации. Часть 1. Программное обеспечение средств защиты информации. Классификация по уровню контроля отсутствия недекларированных возможностей". Утверждено решением председателя Государственной технической комиссии при Президенте Российской Федерации от 4 июня 1999 г. № 114.

[8] ГОСТ Р 50922-2006 Защита информации Основные термины и определения.

[9] ГОСТ Р 17799-2005 Информационная технология. Практические правила управления информационной безопасностью.

# Development of taint-analysis methods to solve the problem of searching of undeclared features

*A.Y. Tichonov {fireboo@ispras.ru}*
*A.I. Avetisyan {arut@ispras.ru}*

**Abstract.** Program analysis problem will be viewed as the process of extracting some properties of analyzed program. It might be the description of implemented algorithms, output data formats, such as file formats, network packet formats and structures of data in memory, or information of existing bugs. In this article we will consider an approach to program analysis to identify some types of program beetles which break confidentiality. In this approach, programs are presented in the form of stripped executables without source code. According to Russian National Standard (ГОСТ Р 51275-2006) a program beetle is intentionally introduced into the software functional object which under certain circumstances can initialize an implementation of software's undeclared features. A program beetle may be implemented either as some malicious program, or as a part of the software code. The proposed approach described in the fifth section of the article. This approach was implemented in our program analysis tool named TREX (TRace EXplorer). The first four sections briefly describe TREX implementation.

**Keywords**: software security, undeclared features, malicious code, taint-analysis

# Использование аппаратной виртуализации в контексте информационной безопасности

*Д. В. Силаков*
*silakov@ispras.ru*

**Аннотация.** Статья посвящена возможностям использования аппаратной виртуализации для решения различных задач информационной безопасности. Предлагается обзор подходов к повышению безопасности программных систем, основанных на использовании виртуализации. Также приводится обзор возможных сценариев использования виртуализации злоумышленниками. Указываются области применения и ограничения существующих решений и дальнейшие перспективы развития рассматриваемой области.

**Ключевые слова:** информационная безопасность; виртуализация; гипервизор.

## *1. Введение*

Задачами информационной безопасности (обеспечением конфиденциальности, целостности и доступности информации) человечество уделяет внимание уже не одно столетие. На сегодняшний день основными средствами обработки и передачи информации являются компьютеры и связанные с ними технологии, в особенности - глобальные сети передачи данных. Соответственно, основные исследования сосредоточены именно в области обеспечения безопасности обработки и передачи данных в электронном виде.

В наиболее распространенных на сегодняшний день системах, основанных на процессорах архитектуры Intel x86 и ее 64-битного расширения, одна из ключевых ролей в обеспечении информационной безопасности принадлежит операционной системе (ОС). Традиционно, ОС работает на наивысшем аппаратном уровне привилегий (в кольце 0) и потенциально имеет доступ ко всем другим компонентам системы. В частности, ей доступны данные и исполняемый код работающих на машине приложений, а также подконтрольны каналы связи приложений с внешним миром. Как следствие, компрометация ОС и внедрение в нее вредоносного кода представляет собой одну из основных угроз безопасности всей системы. Несмотря на то, что за время развития индустрии ИТ было разработано немало подходов к повышению качества ПО (обзор таких методов можно найти в [1]), полностью

избавиться от ошибок в таких крупных продуктах, как современные ОС общего назначения, на сегодняшний день не представляется возможным.

Обнаружение вредоносного кода, внедренного в ОС, затруднено тем, что он работает с теми же привилегиями, что и средства обнаружения и предотвращения угроз (инструменты обнаружения вторжений, антивирусы и прочие). Таким образом, вредоносные программы имеют возможность скрывать следы своей деятельности – например, манипулируя системными журналами или проводя атаки непосредственно на средства защиты [2].

Такая ситуация, когда вредоносное ПО и средства защиты системы играют в «кошки-мышки» на уровне ОС, сохранялась на протяжении долго времени и, по большому счету, сохраняется сейчас. Однако представленные в 2005-2006 годах средства аппаратной поддержки виртуализации архитектуры x86 открывают новые горизонты в этой области.

## *2. Аппаратная поддержка виртуализации*

Изначально, аппаратная поддержка виртуализации была добавлена в процессоры Intel и AMD с целью обойти ограничения архитектуры x86, сильно усложнявшие задачу запуска нескольких виртуальных машин (ВМ) на одной физической. Согласно сформулированному еще в 1974 году критерию Попека-Голдберга [3], эффективный монитор виртуальных машин (гипервизор) может быть построен, если все инструкции, способные изменить состояние ресурсов виртуальной машины, а также инструкции, поведение которых зависит от конфигурации этих ресурсов, являются привилегированными. Соблюдение этого условия позволяет разместить ВМ на непривилегированном уровне, оставив на привилегированном только гипервизор. Выполнение привилегированных инструкций в ВМ вызывает аппаратное исключение, перехватываемое гипервизором, который эмулирует необходимое поведение.

Практика показала, что в реальных системах доля таких перехватов достаточно мала и большинство инструкций может напрямую исполнятся на аппаратуре реальной машины, что обеспечивает эффективность гипервизора в плане производительности.

Основной проблемой архитектуры x86 является наличие непривилегированных инструкций, способных изменить состояние ресурсов виртуальной машины, а также инструкций, которые ведут себя по-разному, будучи выполнены на разных уровнях привилегий. Для подобных инструкций метод перехвата аппаратных исключений не работает (поскольку никаких исключений при их выполнении на непривилегированном уровне не возникает) и необходимы другие подходы – такие, как бинарная трансляция (используемая в VMware Workstation и VirtualBox) или паравиртуализация (Xen, KVM) [4].

Технологии виртуализации от Intel (VT-x) и AMD (AMD-V) подразумевают дополнительное разграничение привилегий на аппаратном уровне. Более точно, вводятся два новых уровня привилегий – корневой (“root”) и некорневой (“nonroot”). “Старые” кольца x86 располагаются внутри некорневого уровня, а в корневом уровне может размещаться гипервизор. ВМ располагаются в некорневом уровне (при этом ОС внутри них работают в кольце 0, как на “обычной” машине). При возникновении определенных ситуаций, определяемых посредством соответствующих управляющих структур, ВМ останавливается, а управление передается гипервизору, который выполняет необходимые преобразования инструкций и их аргументов и возвращает управление ВМ.

Первые реализации аппаратной поддержки виртуализации были сосредоточены на перехвате инструкций, то есть нацеливались, прежде всего, на виртуализацию процессора. Однако со временем производители добавили средства работы с памятью и с устройствами, предоставив возможность создавать гипервизоры, способные контролировать практически все активности системы.

Важным фактом является то, что гипервизор – это обычная программа, и его функциональность не обязательно должна ограничиваться управлением виртуальными машинами. Фактически, аппаратная виртуализация позволяет запускать на физической машине программу, привилегии которой выше, чем привилегии ОС. Такая программа может останавливать работу системы в случае возникновения определенных событий, совершать произвольные манипуляции с адресным пространством ОС и ее компонентов, и возвращать ей управление. Именно на этом аспекте и основываются различные методы, как повышения безопасности программных систем, так и их компрометации, о которых пойдет речь ниже.

## *3. Обнаружение вредоносного ПО*

Одним из способов использования нового уровня привилегий, лежащего ниже нулевого кольца x86, является помещение на этот уровень инструментов обнаружения вредоносного ПО. Такой подход существенно затрудняет возможность компрометации средств защиты со стороны кода, работающего внутри ОС. Кроме того, злоумышленникам становится сложнее скрывать следы своей деятельности, поскольку все действия с системными журналами, файлами и другими ресурсами могут перехватываться и анализироваться гипервизором.

При этом сложную функциональность по анализу событий, происходящих в системе, не обязательно реализовывать непосредственно в гипервизоре. Вместо этого можно запустить еще одну виртуальную машину, невидимую для пользователя основной системы, в которой и будет проводиться анализ. Внутри такой вспомогательной машины может быть развернута одна из массовых ОС (например, Linux), что позволит использовать уже имеющиеся

инструменты анализа системных событий и выявления вредоносных действий [5].

Помимо использования традиционных средств анализа, ряд исследователей (см., например, [6], [7]) предлагают новые способы анализа и контроля происходящих в системе событий, основанные на использовании гипервизора, а точнее – его возможности приостанавливать работу ВМ при наступлении определенных событий с последующим анализом адресного пространства и ресурсов системы.

Следует отметить, что гипервизор (и, соответственно, вспомогательная ВМ) «видит» операционную систему, ее компоненты и приложения на достаточно низком уровне и не обладает никакой семантической информацией о процессах ОС, структуре ее файловой системы и других деталях. В то же время наличие некоторых семантических сведений активно используется рядом средств защиты системы – например, инструментами обнаружения вторжений. Проблема предоставления такой информации также активно исследуется. Часть предлагаемых решений ([7], [8]) сводятся к добавлению в гипервизор возможности реконструкции данных о системе на основе анализа ее ресурсов – адресного пространства, жесткого диска и других. Альтернативой является внедрение в каждую ВМ специального агента, который бы сообщал гипервизору необходимые сведения – подобный подход используется, например, в технологии VMware VMsafe [9]. Однако в такой схеме на гипервизор ложится дополнительная задача по защите адресного пространства своих агентов внутри ВМ с целью защиты их от компрометации.

Что касается производительности, то многие исследовательские прототипы работают медленнее своих аналогов, не использующих виртуализацию, на 5-20%, что в большинстве случаев является вполне приемлемым. Важно отметить, что инструменты, работающие на уровне гипервизора, могут работать быстрее своих аналогов внутри ВМ – например, подтверждающие этот факт экспериментальные данные по ряду известных антивирусов приведены в [8]. Этот аспект важен, в частности, в контексте активно развивающихся облачных вычислений, где уже существуют соответствующие промышленные решения – так, инструменты Catbird могут быть использованы клиентами Amazon EC2 для аудита своих систем. Пионер и один из лидеров в области виртуализации архитектуры x86 – компания VMware – разработала уже упоминавшуюся технологию VMsafe, упрощающую разработку и использование средств обеспечения безопасности в виртуализированных средах, построенных на продуктах VMware, посредством предоставления API для доступа к гипервизору.

## *4. Внедрение вредоносного ПО*

Возможность иметь программу, неподконтрольную ОС, заинтересовала исследователей в области безопасности не только с точки зрения создания программ, обнаруживающих вредоносные действия, но и с точки зрения

осуществления этих самых действий - ведь теоретически, вредоносное ПО, расположенное на корневом уровне привилегий, крайне сложно обнаружить с использованием средств защиты изнутри ОС.

Основная трудность использования такого способа компрометации заключается во внедрении гипервизора в систему. Как Intel, так и AMD предлагают возможность запуска гипервизора из уже работающей системы посредством специальных инструкций SENTER [10] и SKINIT [11] соответственно. Эти инструкции осуществляют приостановку работающей на машине системы и запускают некоторую программу — например, гипервизор. После чего система продолжает работу, но уже внутри ВМ, управляемой гипервизором. Для ОС и приложений внутри нее запуск гипервизора проходит прозрачно (при условии, что гипервизор не закроет доступ к каким-либо ресурсам и устройствам, используемым системой, не предоставив виртуализированного аналога).

Целью Intel TXT и AMD SVM является предоставление возможности запускать доверенное ПО в скомпрометированной системе (именно запуска — защита ПО в процессе работы не осуществляется). Реализуемая схема запуска программ получила название «поздний запуск» («late launch»). Естественно, выполнение такого запуска может осуществляться только из привилегированного кольца 0 – то есть, как правило, от лица ОС. Поэтому для прозрачного (скрытого от пользователя) внедрения гипервизора необходимо использовать какую-либо брешь в ОС, либо рассчитывать на некомпетентные действия администратора системы, которые приведут к запуску гипервизора. К тому же, запустить гипервизор один раз мало – необходимо предусмотреть его запуск при перезагрузке системы. Таким образом, применение виртуализации для компрометации системы не избавляет от необходимости обращаться к «традиционным» способам атаки.

Впрочем, как и практически все программное обеспечение, реализации Intel TXT и AMD SVT могут содержать уязвимости, дающие возможность их компрометации. Например, одна из уязвимостей, подтвержденная (и, естественно, исправленная) специалистами Intel описана в работе [12].

Также стоит упомянуть о подходах к компрометации системы с помощью атаки на System Management Mode (SMM) – специальный режим работы процессора, в котором приостанавливается работа всей системы (включая гипервизор, если он есть), и запускается программа-обработчик из аппаратной прошивки (какой именно обработчик – определяется причиной, по которой был осуществлен переход в SMM). Соответственно, компрометация программы, работающей в SMM, потенциально ведет к компрометации всей системы. Несмотря на то, что SMM появился еще в процессорах Intel 386, подходы к его компрометации появились лишь в последние годы [13], [14] – практически совпав с началом исследований возможности применения виртуализации для компрометации системы.

Наконец, компрометации могут быть подвергнуты и другие низкоуровневые компоненты машины — например, BIOS [15] или Intel Active Management Technology [16].

В настоящее время использование виртуализации (равно как и SMM и других низкоуровневых компонентов) для компрометации системы существует скорее в виде исследовательских работ. Существуют инструменты, демонстрирующие возможность соответствующих атак на систему – из наиболее известных можно выделить BluePill [17], а также SubVirt, разработанный в Microsoft Research [18]. Однако до массового применения подобных средств дело еще не дошло. Не в последнюю очередь это обусловлено техническими сложностями, а также сильной зависимостью от аппаратного обеспечения, используемого на целевой системе [19].

Кроме того, несмотря на сложность обнаружения вредоносного гипервизора из ОС, эта задача не является абсолютно неразрешимой – точнее, существуют подходы к определению того, работает ли ОС на «голом» железе или внутри виртуальной машины. Соответственно, если выполнение ОС в виртуальной машине не является легитимным, то администратор системы должен принять меры по выявлению и устранению гипервизора. Большинство подходов основано на том факте, что работа гипервизора, так или иначе, замедляет работу системы. Даже если гипервизор будет манипулировать часами, которые видит ОС, всегда есть возможность воспользоваться внешними (по отношению к системе) часами [20].

Наконец, если пользователю не нужна поддержка аппаратной виртуализации, ее можно просто отключить средствами BIOS. А если виртуализация используется, то в момент атаки на машине уже может работать гипервизор, вытеснение которого вредоносным монитором будет, очевидно, замечено пользователем. Обработка ситуаций, когда «виртуализировать» надо не только работающую ОС, но еще и гипервизор, на данный момент изучена достаточно слабо, и до реальных применений имеющихся теоретических разработок дело пока не дошло.

Впрочем, если в целевой системе уже работает какой-либо монитор виртуальных машин, то создавать и внедрять собственный гипервизор не обязательно. Можно провести атаку на уже существующий, получив в случае успеха доступ ко всем виртуальным машинам. Компрометация существующего гипервизора даже более выгодна для злоумышленника, чем внедрение собственного – ведь наличие существующего гипервизора легитимно (нет необходимости пытаться скрыть его существование от пользователя) и его запуск, вероятно, будет осуществляться и после перезапуска машины.

Возможность компрометации гипервизора допускалась еще разработчиками первого коммерческого монитора – CP-67, работавшего на платформах IBM System/360-370. Однако отмечалось, что поскольку код гипервизора существенно меньше и проще, чем код операционной системы, то и

уязвимостей в нем существенно меньше [21]. Тем не менее, в последователе CP-67 – гипервизоре VM/370 [22] – было найдено несколько критических уязвимостей.

В IBM этот опыт учли, и при разработке последующих гипервизоров безопасности уделялось особое внимание. В частности, уже для KVM/370 использовались формальные методы для спецификации и верификации гипервизора – возможность и целесообразность применения таких методов обусловлена, опять же, относительно небольшим размером кода. Современный PR/SM – монитор для IBM System z, считающийся одним из самых безопасных гипервизоров – сертифицирован по уровню 5 Common Criteria Evaluation Assurance (EAL5) и уровню E4 ITSEC.

К сожалению, гипервизорам для массовой архитектуры x86 (таким как Xen, KVM или VMware ESX) до таких высот еще далеко, несмотря на наличие различных подходов к повышению качества подобного ПО (в том числе с использованием формальных методов – см., например, [23]). Причем, по мнению исследователей из IBM, проблемы в этих гипервизорах лежат не просто в недостаточно качественной реализации, а на уровне архитектуры [24]. Неудивительно, что время от времени в существующих решениях виртуализации обнаруживаются уязвимости, позволяющие скомпрометировать гипервизор и получить контроль над виртуальными машинами – из наиболее известных исследований, стоит выделить работы [25], [26] и [27]. Следует иметь в ввиду возможность компрометации такого рода и не считать гипервизор заведомо неуязвимым.

## *5. Изоляция приложений*

Еще одним направлением, для которого аппаратная виртуализация открывает новые перспективы, является изолированное выполнение приложений и защита адресного пространства приложения от несанкционированного доступа. В принципе, этой задачей должна заниматься операционная система. Однако современные ОС достаточно крупны, сложны, и в них постоянно обнаруживаются уязвимости, используемые злоумышленниками в недобросовестных целях.

Изолировать приложения друг от друга пользователь ПК может посредством запуска отдельной виртуальной машины для каждого приложения (или группы приложений) – современные средства виртуализации позволяют делать это достаточно просто. Однако использование аппаратной виртуализации позволяет перенести задачу изоляции адресного пространства приложения в гипервизор, который может гарантировать изолированность не только от других приложений, но и от компонентов ОС – подобный подход, например, используется в технологии Overshadow [28], предложенной исследователями из VMware совместно с учеными университетов Стэнфорда и Принстона и Массачусетского технологического института.

Размер кода гипервизора, необходимого для выполнения задач изоляции, существенно меньше размера кода современных ОС общего назначения. В частности, в таком гипервизоре отсутствует необходимость в драйверах различных устройств, а именно драйвера являются одной из наиболее уязвимых частей современных монолитных ОС. Как было показано в предыдущем разделе, малый размер кода сам по себе не гарантирует отсутствия ошибок. Тем не менее, многие исследователи активно эксплуатируют гипотезу о том, количество уязвимостей в относительно небольшом гипервизоре существенно меньше, чем в сложной ОС общего назначения.

Как и в случае с реализацией средств обнаружения вредоносного ПО, при необходимости предоставления гипервизором какой-либо дополнительной функциональности (например, передачи управления доступом в сеть от ОС гипервизору), такая функциональность может быть реализована не в самом гипервизоре, а во вспомогательной виртуальной машине. При этом вспомогательная ВМ может полностью располагаться в оперативной памяти и быть невидимой для пользователя основной ВМ. Такой подход используется в работах, проводимых в ИСП РАН для создания среды запуска приложений в недоверенной ОС [29], [30].

Ряд исследований совмещают запуск отдельных ВМ для различных групп приложений и контроль над действиями приложений и компонентов ОС со стороны гипервизора. Одной из наиболее интересных разработок в этой области является операционная система Qubes [31]. В основе системы лежит разбиение приложений на изолированные домены, каждый из которых работает в отдельной ВМ (управляемой Linux). В основе Qubes лежит гипервизор Xen. В целях повышения безопасности коды непосредственно гипервизора и привилегированного домена (Dom0) сохраняются максимально простыми, а практически все драйвера (в том числе средства работы с сетью и с дисковыми накопителями) вынесены в отдельные вспомогательные домены.

В ОС Qubes используется специализированный X-сервер, позволяющий отображать окна приложений из всех доменов на экране пользователя и осуществлять обмен данными (“copy-paste”) между приложениями различных доменов. В итоге у пользователя складывается ощущение, что он работает с одним экземпляром ОС Linux.

По состоянию на сегодняшний день, ОС Qubos находится в состоянии разработки и не готова для промышленного использования — равно как и большинство прототипов, разработанных в этой области.

## *6. Заключение*

Реализация аппаратной поддержки виртуализации в массовых процессорах дала толчок большому числу исследований по возможностям применения виртуализации в контексте информационной безопасности — как в области

защиты информации, так и в области компрометации программных систем. Однако, несмотря на достаточно большое количество теоретических работ и прототипов, говорить о массовом использовании виртуализации для решения проблем безопасности пока рано.

Одной из основных причин такого положения является техническая сложность реализации предлагаемых решений и зависимость от аппаратной платформы — несмотря на схожесть концепций, реализации Intel и AMD имеют ряд серьезных отличий. К тому же, история аппаратной поддержки виртуализации насчитывает немногим более пяти лет, на протяжении которых технология активно дорабатывалась и расширялась. Многие работы, предлагающие использовать виртуализацию для решения различных задач, опираются на второе поколение аппаратной виртуализации (Intel Extended Page Table, AMD Rapid Virtualization Indexing), представленное в процессорах только в 2009-2010 годах. Возможно, через несколько лет — по мере увеличения доли процессоров с поддержкой виртуализации на машинах пользователей — мы увидим и рост числа приложений, использующих виртуализацию для решения задач информационной безопасности.

**Литературы**


[1] Липаев В.В. Методы обеспечения качества крупномасштабных программных средств. М.: СИНТЕГ, 2003. 520 с.

[2] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. // Proc. of the Internet Society's 2003 Symposium on Network and Distributed System Security. 2003. Pp. 163-176.

[3] G.J. Popek, R.P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. // Communications of the ACM, Volume 17, Issue 7, July 1974, pp. 412-421.

[4] Касперски К. Аппаратная виртуализация или эмуляция "без тормозов". // InsidePro, 2007. [HTML] http://www.insidepro.com/kk/159/159r.shtml

[5] A. Dinaburg, P. Royal, M. Sharif, W. Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. // Proc. of the 15th ACM conference on Computer and communications security. 2008. Pp. 51-62.

[6] S. Krishnan, K.Z. Snow, F. Monrose. Trail of bytes: efficient support for forensic analysis. // Proc. of the 17th ACM conference on Computer and communications security. 2010. Pp. 50-60.

[7] T. Garfinkel, M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. // Proc. of the Symposium on Network and Distributed System Security (NDSS'03). 2003. [PDF] http://suif.stanford.edu/papers/vmi-ndss03.pdf

[8] X. Jiang, X. Wang, D. Xu. Stealthy Malware Detection Through VMM-Based “Out-of-the-Box” Semantic View Reconstruction. // Proc. of the 14th ACM conference on Computer and communications security. 2007. Pp. 128-138.

[9] New VMware VMsafe Technology Allows the Virtual Datacenter to Be More Secure Than Physical Environments. // Press release. 2007. [HTML] http://www.vmware.com/company/news/releases/vmsafe_vmworld.html

[10] Intel Trusted Execution Technology Architectural Overview. // Intel White Paper. 2008. [PDF] http://www.intel.com/technology/security/downloads/arch-overview.pdf

[11] G. Strongin. Trusted Computing Using AMD «Pacifica» and «Precidio» Secure Virtual Machine Technology. // Information Security Technical Report. 2005. Volume 10, Issue 2, pp. 120-132.
[12] R. Wojtczuk, J. Rutkowska, A.Tereshkin. Another Way to Circumvent Intel Trusted Execution Technology. // Invisible Things Lab. December, 2009. [PDF] http://invisiblethingslab.com/resources/misc09/Another%20TXT%20Attack.pdf
[13] R. Wojtczuk, J. Rutkowska. Attacking Intel Trusted Execution Technology. // Black Hat DC 2009. [PDF] http://invisiblethingslab.com/resources/bh09dc/Attacking%20Intel%20TXT%20-%20paper.pdf
[14] S. Embleton, S. Sparks, C. Zou. SMM Rootkits: A New Breed of OS Independent Malware. // Proc. of the 4th international conference on Security and privacy in communication networks. Istanbul, Turkey, 2008. Article #11, pp. 1-12.
[15] R. Wojtczuk, A. Tereshkin. Attacking Intel BIOS. // Black Hat USA 2009. [PDF] http://invisiblethingslab.com/resources/bh09usa/Attacking%20Intel%20BIOS.pdf
[16] R. Wojtczuk, A. Tereshkin. Introducing Ring -3 Rootkits. // Black Hat USA 2009. [PDF] http://invisiblethingslab.com/resources/bh09usa/Ring%20-3%20Rootkits.pdf
[17] J. Rutkowska. Subverting Vista Kernel For Fun And Profit. // Black Hat USA 2006. [PDF] http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf
[18] S.T. King, P.M. Chen, Y.M. Wang, C. Verbowski, H.J. Wang, J.R. Lorch. SubVirt: Implementing malware with virtual machines. // Proc. of the 2006 IEEE Symposium on Security and Privacy. 2006. Pp. 314–327.
[19] A. Liguori. Debunking Blue Pill myth. // Interview to Virtualization.info. August, 2006. [HTML] http://virtualization.info/en/news/2006/08/debunking-blue-pill-myth.html
[20] T. Garfinkel, K. Adams, A. Warfield, J. Franklin. Compatibility is Not Transparency: VMM Detection Myths and Realities. // 11th Workshop on Hot Topics in Operating Systems (HotOS-XI), 2007. [PDF] http://www.stanford.edu/~talg/papers/HOTOS07/vmm-detection-hotos07.pdf
[21] S.E. Madnick, J.J. Donovan. Application and analysis of the virtual machine approach to information system security and isolation. // Proc. of the workshop on virtual computer systems. ACM, 1973, pp. 210-224.
[22] C.R. Attanasio, P. W. Markstein, Ray J. Phillips. Penetrating an Operating System: A Study of VM/370 Integrity. // IBM Systems Journal, Volume 15, 1976. Pp. 102-116.
[23] И.Б. Бурдонов, А.С. Косачев, В.В. Кулямин. Безопасность, верификация и теория конформности. // Материалы Второй международной научной конференции по проблемам безопасности и противодействия терроризму. Москва, МНЦМО, 2007. С. 135-159.
[24] P.A. Karger, T.J. Watson. Is Your Virtual Machine Monitor Secure? // Materials of Third Asia-Pacific Trusted Infrastructure Technologies Conference, 2008. Pp. 5-5.
[25] T. Garfinkel, M. Rosenblum. When Virtual is Harder than Real: Security Challenges in Virtual Machine Based Computing Environments. // 10th Workshop on Hot Topics in Operating Systems (HotOS-X), 2005. [PDF] http://www.stanford.edu/~talg/papers/HOTOS05/virtual-harder-hotos05.pdf
[26] J. Rutkowska. Security Challenges in Virtualized Environments. // RSA Conference, 2008. [PDF] http://www.invisiblethingslab.com/resources/rsa08/Security%20Challanges%20in%20Virtualized%20Enviroments%20-%20RSA2008.pdf
[27] R. Wojtczuk. Subverting the Xen Pypervisor. // Black Hat USA 2008. [PDF] http://invisiblethingslab.com/resources/misc08/xenfb-adventures-10.pdf

[28] X. Chen, T. Garfinkel, E.C. Lewis, P. Subrahmanyam, C.A. Waldspurger, D. Boneh, J. Dwoskin, D.R.K. Ports. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. // Proc. of the 13th international conference on Architectural support for programming languages and operating systems (ASPLOS XIII). 2008. Pp. 2-13.
[29] Яковенко П.Н. Прозрачный механизм удаленного обслуживания системных вызовов. // Труды Института системного программирования РАН. Том 18. 2010. С. 221-241.
[30] I. Burdonov, A. Kosachev, P. Iakovenko. Virtualization-Based Separation of Privilege: Working With Sensitive Data In Untrusted Environment. // Proc. Of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems. 2009. Pp. 1-6.
[31] Qubes Architecture Specification. Version 0.3. 2010. [PDF] http://qubes-os.org/files/doc/arch-spec-0.3.pdf

## Using Hardware-assisted Virtualization in the Information Security Area

*D. V. Silakov*
*silakov@ispras.ru*

**Abstract.** The paper describes possible ways of using hardware-assisted virtualization for solving different information security problems. An overview of virtualization-based approaches for increasing software security is presented, as well as overview of possible ways to compromise the system that can take advantage of virtualization. The paper analyzes possible applications of existing approaches, their limitations and possible future directions of further developments.

**Keywords:** information security; virtualization; hypervisor.

# Оптимизация динамической двоичной трансляции

*Кирилл Батузов, Алексей Меркулов*
*batuzovk@ispras.ru, steelart@ispras.ru*

**Аннотация.** Двоичная трансляция — это процесс получения по заданной программе P программы Q, удовлетворяющей заданным требованиям, если обе программы записаны в виде машинных кодов. Если двоичная трансляция производится во время выполнения программы, то она называется динамической двоичной трансляцией. В данной статье рассматриваются возможности применения различных оптимизаций во время динамической двоичной трансляции, а именно: мы улучшили алгоритм поиска блоков трансляции в кэше трансляций в QEMU, проанализировали влияние алгоритма распределения регистров на быстродействие кода в QEMU, реализовали простые машинно-независимые оптимизации в QEMU и планировщик инструкций в Valgrind. Изменения алгоритма поиска блоков трансляций в кэше дали наибольший эффект, планировщик инструкций также является многообещающей оптимизацией.

**Ключевые слова:** оптимизации программы, динамическая двоичная трансляция, QEMU, Valgrind.

## *1. Введение*

Двоичная трансляция — это процесс получения по заданной программе P программы Q, удовлетворяющей заданным требованиям, если обе программы записаны в виде машинных кодов. Примерами требований могут быть «программа Q должна делать то же, что и программа P, но выполняться на процессоре с другой архитектурой» или «программа Q должна делать все то же, что и программа P, но дополнительно должна проверять корректность всех операций работы с памятью». Если двоичная трансляция производится во время выполнения программы, то она называется динамической двоичной трансляцией.

Различают два подхода к динамической двоичной трансляции: *копирование и аннотирование* и *дизассемблирование и перегенерация*. Первый подход подразумевает, что код транслируемой программы копируется в том виде, в котором он уже существует, и к нему добавляется некоторый дополнительный, анализирующий код. Во втором подходе программа дизассемблируется в некоторое внутреннее представление. В этом

представлении в неё вносятся изменения, после чего, для полученного внутреннего представления генерируется машинный код, который и будет исполняться [1]. В данной статье будет рассматриваться только второй подход.

Целью данной работы является исследование возможностей применения различных оптимизаций во время динамической двоичной трансляции.

Динамическая двоичная трансляция имеет несколько особенностей, которые необходимо учитывать во время разработки алгоритмов оптимизаций:

- оптимизации производятся во время выполнения, время, затраченное на оптимизации, также входит в результирующее время работы программы,
- во время оптимизаций отсутствует информация, которую компилятор получает из высокоуровневого представления (например, граф потока управления).

Для практических экспериментов в данной работе используются два инструмента, использующих динамическую двоичную трансляцию: QEMU [2] и Valgrind [3]. Оба эти инструмента имеют открытый исходный код.

## *2. Оптимизация двоичной трансляции в QEMU*

QEMU — эмулятор процессоров и целых вычислительных систем. QEMU поддерживает большое количество различных процессоров и устройств. У него есть два режима работы: эмуляция системы и эмуляция приложения. В первом случае эмулируется вся вычислительная система: процессор и внешние, по отношению к нему, устройства. Динамической трансляции подвергаются код программы, всех задействованных библиотек и операционной системы. Во втором случае транслируется только код программы и нужных библиотек. Функции операционной системы (в частности обработку системных вызовов) берет на себя QEMU. QEMU также поддерживает использование аппаратной виртуализации, однако она не будет рассматриваться в рамках данной работы.

**Единицей двоичной трансляции в QEMU** является расширенный базовый блок — ациклический участок кода с одним входом. Трансляция происходит при первом обращении к соответствующему коду. Полученные оттранслированные блоки кэшируются и затем используются при повторных выполнениях того же самого кода.

Профилирование QEMU при эмуляции процессора ARM на архитектуре x86 с помощью OProfile [4] выявило, что оттранслированный код гостевой программы и код самого QEMU выполняются сравнимое количество времени. Поэтому в данной работе будут рассмотрены оба аспекта: оптимизации кода

во время динамической двоичной трансляции и улучшение кода самого QEMU.

## 2.1. Оптимизация поиска в кэше оттранслированных блоков.

Первая рассматриваемая в данной работе оптимизация повышает эффективность поиска оттранслированных блоков в кэше.

В QEMU поддерживаются два уровня кэша для оттранслированных блоков. Оба уровня реализованы с помощью хэш-таблиц. В первом, более быстром, уровне хэш берётся от значения счётчика команд. На этом уровне каждому хэшу может соответствовать не более одного блока, а при возникновении коллизий один блок вытесняет другой. В случае отсутствия нужного блока в первом уровне кэша поиск начинается во втором. На втором уровне хэш берётся от физического адреса блока трансляции. Каждому хэшу соответствует список блоков, среди которых ведётся линейный поиск нужного блока. Если нужный блок не был найден, то вызывается процедура двоичной трансляции, которая его генерирует. После того, как блок найден или сгенерирован, оба уровня кэша обновляются соответствующим образом.

Использование счётчика команд для хэширования на первом уровне кэша позволяет быстро выполнять поиск без необходимости преобразования виртуальных адресов в физические. Это является весьма дорогой операцией, поскольку выполняется полностью программно. В свою очередь, использование физических адресов на втором уровне кэша позволяется нескольким приложениям использовать одну и ту же разделяемую библиотеку без необходимости несколько раз транслировать код этой библиотеки.

Нами было экспериментально установлено, что первый уровень кэша успешно обрабатывает от 75% запросов, для сложных систем с большим количеством активных графических приложений и задействованных библиотек, до почти 100% для небольших программ, занимающихся какими-либо вычислениями без привлечения стороннего кода из разделяемых библиотек. В результате загрузки и работы операционной системы на базе ядра Linux с большим набором графических приложений, количество элементов списка, просматриваемых на втором уровне кэша, на каждом запросе в среднем превосходило 20. Профилирование показало, что функция поиска в кэше второго уровня при этом работала более 60% общего времени работы.

Предложенная нами оптимизация переносит только что найденный на втором уровне кэша блок в начало соответствующего списка. Таким образом, если данный блок в скором времени опять будет запрошен, он будет найден быстрее. В сочетании с тем, что выполняющийся код обладает очень высокой степенью локальности, можно ожидать, что такая оптимизация снизит количество просматриваемых в среднем элементов списка и, тем самым, уменьшит время поиска в кэше второго уровня.

Эксперименты показали, что среднее количество просмотренных элементов списка при загрузке и последующей работе той же самой операционной

системы в результате применения данной оптимизации снизилось на порядок и стало равно 1.5. Кэш первого уровня при этом остался без изменений. Время загрузки используемой в эксперименте операционной системы снизилось в 2 раза. Также было заметно существенное улучшение времени отклика графических приложений. Время работы небольших вычислительных программ остались без изменений.

Мы отправили патч, добавляющий данную оптимизацию разработчикам QEMU, и 5 декабря 2010 года он был включен в основную ветку разработки [5].

## 2.2. Распределение регистров в QEMU.

В QEMU используется чрезвычайно простой и быстрый алгоритм распределения регистров. Если в некоторый момент требуется поместить некоторое значение на регистр, алгоритм выбирает «первый» свободный регистр из допустимых. Допустимость регистров определяется из ограничений самой инструкции, а также условия несовпадения выбранного регистра с зарезервированными. Если свободного регистра не нашлось, то «первый» среди допустимых сбрасывается в память и, затем, используется для хранения нового значения.

В данной части мы рассмотрим влияние порядка просмотра регистров целевой архитектуры на эффективность данного алгоритма распределения регистров. В частности, от этого порядка будет зависеть, какой регистр будет находиться как «первый» допустимый регистр при генерировании сброса регистра в память.

В QEMU версии 0.12.50, в которой начинались данные эксперименты, при генерации кода для процессоров архитектуры x86, регистры просматривались в следующем порядке при распределении: EAX, EDX, ECX, EBX, ESI, EDI, EBP. Глядя на этот порядок можно ожидать, что регистры EAX, EDX и ECX будут использоваться и сбрасываться в память значительно чаще, чем остальные, поскольку они не только являются приоритетными для алгоритма распределения регистров, но также участвуют в соглашениях о вызовах функций и к ним жёстко привязаны многие инструкции.

Действительно, эксперименты показали, что за время загрузки операционной системы на базе ядра Linux и пользовательской графической оболочки было сгенерировано 1818624 сброса регистров в память. Из них на EAX приходятся 1129590 сбросов, на EDX — 521281, на ECX — 110387, а на все остальные — 57366.

Легко видеть существенный перекос в сторону регистров EAX, ECX и EDX. Пытаясь исправить этот перекос и, за счет этого, снизить общее количество сбросов регистров в память, мы понизили приоритет регистров EAX, ECX и EDX. Таким образом, был получен следующий порядок просмотра регистров: EBX, ESI, EDI, EBP, ECX, EDX, EAX.

Эксперименты выявили следующее: за время загрузки той же самой операционной системы было сгенерировано 1327104 сброса регистров в память, из них EBX — 677100 сбросов, ESI — 406587, EDI — 140361, EAX — 49363, ECX — 38612, EDX — 15081. Результаты показывают, что общее количество сбросов регистров в память уменьшилось на 27%, а также, что количество сбросов значительно выровнялось между регистрами по сравнению с тем, что было изначально. С точки зрения производительности, наибольший эффект данная оптимизация даёт на рекурсивных программах — до 20% уменьшения времени работы. Дальнейшие изменения порядка просмотра регистров не привели к заметным улучшениям генерируемого кода.

Этот результат также был независимо получен Ричардом Хендерсоном. Сделанные им изменения были включены в основную ветку разработки и попали в QEMU версии 0.13 [6].

## 2.3. Машинно-независимые оптимизации над внутренним представлением в QEMU.

В результате профилирования QEMU удалось выяснить, что весь процесс динамической двоичной трансляции занимает очень мало времени по сравнению со временем, затрачиваемым на выполнение полученного кода. Это является следствием практически полного отсутствия оптимизаций во время двоичной трансляции и быстрым, но неэффективным алгоритмом распределения регистров. В QEMU присутствуют только две оптимизации: удаление мёртвого кода и подстановка констант, которая выполняется во время распределения регистров.

В данной работе мы опишем наши эксперименты по добавлению оптимизаций по продвижению констант и копий, а также сворачивания константных выражений в QEMU. Прежде чем перейти к деталям реализации данных оптимизаций, опишем внутреннее представление, используемое в QEMU, на котором эти оптимизации производятся, и источники неэффективности в генерируемом коде.

Несмотря на то, что код гостевой программы в большинстве случаев был хорошо оптимизирован компилятором, код, получаемый из него при двоичной трансляции, может быть улучшен машинно-независимыми оптимизациями. Причиной этому служат два факта:

- гостевая программа может компилироваться для одной архитектуры, а выполняться после динамической трансляции на другой,
- трансляция в QEMU производится с помощью замены каждой инструкции гостевой программы на последовательность инструкций машины, на которой программа будет выполняться.

На стыке инструкций как раз и возникает неэффективный код, который может быть улучшен. Например, пересылка константы 120034h может быть выполнена только в две инструкции на архитектуре ARM:

**mov r5, #0x34**
**orr r5, #0x120000**

Причиной этого является особенность кодирования констант в двоичном представлении инструкций на ARM, накладывающая ограничение на возможные значения констант. На x86 таких ограничений нет и того же результата можно добиться с помощью одной инструкции:

**mov $0x120034, %edi**

Рассмотрим теперь особенности внутреннего представления QEMU. Код, содержащийся в блоке трансляции, представляется в виде двух массивов: в первом хранятся коды операций, а во втором — аргументы. Коды операций всегда 16-битные числа, аргументы — 32-битные или 64-битные целые числа, в зависимости от того, для какой архитектуры генерируется код. Как трактуются числа, записанные в массиве аргументов, полностью зависит от кода операции, к которому они относятся. Набор кодов операций тоже зависит от целевой архитектуры. Однако есть некоторый набор операций, которые всегда присутствуют во внутреннем представлении. Этот набор включает в себя основную целочисленную 32-битную арифметику (сложение, вычитание, умножение, сдвиги, побитовые логические операции; деление может отсутствовать), операции загрузки и сохранения значений, а также операции перехода.

При такой структуре внутреннего представления важно уметь отвечать на следующие вопросы:

- К какой операции относится данный аргумент?
- Что означает записанное в нем число?

Основным вопросом является первый, так как ответ на второй может быть получен из семантики операции. Ответ на первый получается из свойства упорядоченности: коды операции упорядочены так, как они будут выполняться, аргументы упорядочены в соответствии с порядком операций, к которым они относятся, — аргументы первой операции идут первыми в порядке, заданном семантикой этой операции, затем идут аргументы второй операции и так далее.

Таким образом, при последовательном просмотре кода в прямом или обратном порядке легко определить, к какой операции относится данный аргумент. Проблема операций с переменным числом параметров (например, вызов вспомогательной функции) решается введением фиктивных первого и последнего аргумента, в которых записано общее количество аргументов данной операции.

В таблице 1 приведён пример внутреннего представления кода, который вычисляет значение выражения r11 + 12, где r11 — регистр гостевой системы, а 12 — константа. В первой строке таблицы записаны коды операций в мнемоническом виде, во второй строке записаны их аргументы также в мнемоническом виде, а в третьей — те же самые аргументы, но в том виде, в

котором они хранятся во внутреннем представлении QEMU. Аргументы и коды операций выровнены так, чтобы аргументы операции всегда находились под её кодом. Легко видеть, что аргумент, равный 12, в случае операции mov_i32 интерпретируется как r11, а в случае операции movi_i32 — как константа 12.

| опкод | mov_i32 | | movi_i32 | | add_i32 | | |
|---|---|---|---|---|---|---|---|
| Аргу-менты | tmp8 | r11 | tmp9 | 12 | tmp8 | tmp8 | tmp9 |
| | 30 | 12 | 31 | 12 | 30 | 30 | 31 |

*Таблица 1. Пример внутреннего представления QEMU.*

Основным недостатком такого внутреннего представления является сложность внесения локальных изменений, поскольку они будут приводить к сдвигу обоих массивов.

Опишем детали реализации оптимизаций по сворачиванию константных выражений и продвижению копий. Обе эти оптимизации будут производиться в рамках одного базового блока. Они будут выполнены в результате одного линейного просмотра кода в прямом порядке.

Для каждой переменной внутреннего представления будем хранить её текущее состояние, которое будет принимать одно из трёх значений: *является константой*, *является копией*, *не является ни копией, ни константой*. Для переменных, являющихся константами, будем хранить значение этой константы; для переменных, являющихся копиями других переменных, будем хранить идентификатор переменной, копией которой является данная переменная.

Очевидно, что в любой заданной точке кода отношение «переменная A является копией переменной B» (обозначим его EQ) является отношением эквивалентности на множестве переменных и задаёт классы эквивалентности в этом множестве. В ходе оптимизаций, в каждой точке кода в каждом таком классе выберем представителя и обеспечим, чтобы все остальные переменные данного класса считались копиями этого одного представителя. Если в классе эквивалентности существуют внутренние переменные, не соответствующие регистрам эмулируемой системы, то в качестве представителя класса берётся одна из таких переменных. В противном случае берётся любая переменная из данного класса. Такие приоритеты вызваны тем, что работа с переменными, соответствующими регистрам эмулируемой системы, приводит к операциям чтения и записи структуры, содержащей значения этих регистров. Эта структура расположена в памяти реальной машины и доступ к ней может быть достаточно медленным в случае промаха мимо кэша данных первого уровня.

Остальные же переменные чаще всего располагаются на регистрах реальной машины и попадают в память только при нехватке регистров. Доступ к ним, как правило, быстрый.

Опишем саму оптимизацию. Пусть у нас есть правильно вычисленные состояния всех переменных на момент начала выполнения операции внутреннего представления. На момент начала выполнения первой инструкции базового блока будем считать, что все переменные находятся в состоянии *не является ни копией, ни константой*, а также что состояние всех переменных на момент начала выполнения инструкции из середины базового блока совпадает с их состоянием на момент конца выполнения предыдущей инструкции. Покажем, как вычислить состояния всех переменных на момент конца выполнения данной инструкции, а также какие изменения должны быть внесены в инструкцию.

Если операция является операцией пересылки переменных «A = B», принадлежащих одному классу эквивалентности относительно отношения EQ, данная инструкция является излишней и может быть заменена на пустую операцию (nop). Состояния переменных не изменяются.

Если операция является операцией пересылки переменных «A = B» из разных классов, то переменная A переходит из одного класса эквивалентности в другой. При этом если переменная A являлась представителем в своём классе эквивалентности, в этом классе эквивалентности выбирается новый представитель по описанным выше правилам и обновляются пометки для всех переменных из этого класса, чтобы они содержали нового представителя. Также устанавливается пометка, что переменная A *является копией* представителя класса эквивалентности, содержащего переменную B, а переменная B заменяется на представителя своего класса.

Если операция является операцией пересылки «A = B», где B — константа, то A также помечается как константа, операция заменяется на операцию пересылки константы. Если A была представителем своего класса эквивалентности, то в этом классе эквивалентности выбирается новый представитель и состояния всех переменных этого класса обновляются.

Если все аргументы арифметической операции «A = B op C» являются константами, то значение A вычисляется и операция заменяется на операцию пересылки константы. Обновление состояния других переменных происходит аналогично предыдущему пункту. При вычислении значения переменной A необходимо следить за несколькими вещами:

- Хотя во внутреннем представлении все константы считаются беззнаковыми, операция может трактовать их как знаковые. В этом случае необходимо приведение типов.

- 32-битные операции на 64-битных системах используют только младшие 32-бита констант. При реализации таких операции, как сдвиги, необходимо обнулять старшие биты входных констант.

Сворачивание константных выражений было реализовано для основных 32-битных арифметических операций и их 64-битных аналогов.

Во всех остальных случаях, все входные переменные заменяются на представителей их классов, все выходные переменные операции помечаются как *не является ни копией ни константой*. Если какие-то из выходных переменных являлись представителями классов эквивалентности, то в этих классах выбираются новые представители и состояния всех переменных соответствующих классов обновляются, чтобы отразить изменение представителя класса.

Данный алгоритм является консервативным: эквивалентность всех изменений может быть легко доказана на основе имеющейся у оптимизации информации на момент внесения изменений в инструкцию. Сложность данного алгоритма — O(количество_инструкций * количество_переменных).

Рассмотрим, как производится изменение инструкций во время проведения данной оптимизации на внутреннем представлении QEMU. Заметим, что количество операций не меняется, меняются их типы. Такое изменение внутреннее представление поддерживает. Изменение количества аргументов требует дополнительных действий. Для того чтобы обеспечить его, будем просто строить новый массив аргументов. Поскольку операции просматриваются в прямом порядке, это не представляет никакой алгоритмической сложности. Далее заметим, что в ходе данной оптимизации число аргументов текущей операции либо остаётся неизменным, либо уменьшается (вместе с изменением самой операции). Значит, новый массив аргументов можно строить в том же физическом массиве, что и имеющиеся аргументы, затирая уже прочитанные и, следовательно, ненужные аргументы.

Данная оптимизация была реализована и протестирована в QEMU. QEMU эмулировал архитектуру ARM в режиме приложения и запускался на процессоре Intel Xeon E5520 2.27ГГц. Изучение получающегося кода показывает, что оптимизация работает и успешно продвигает копии и константы, а также сворачивает константные выражения. Удаление мёртвого кода и подстановка констант в инструкции во время распределения регистров заканчивают дело по улучшению промежуточного кода.

Мы протестировали оптимизации на целочисленных тестах из набора SPEC CPU2000 [7]. Тесты запускались под QEMU в режиме эмуляции приложения. Тесты 254.gap и 255.vortex не работали ожидаемым образом в этом режиме. Результаты работы остальных тестов приведены в таблице 2. Приведённые числа являются медианой по 5 запускам. Можно видеть небольшое увеличение производительности на всех тестах, кроме 176.gcc и 186.crafty. При этом изменения скорости работы теста 176.gcc находятся в пределах точности измерений. На тесте 186.crafty наблюдается ухудшение производительности.

| Тест | без оптимизаций (время в секундах) | с оптимизациями (время в секундах) | % |
|---|---|---|---|
| 164.gzip | 754.45 | 751.19 | 0.43 |
| 175.vpr | 728.84 | 719.84 | 1.23 |
| 176.gcc | 459.47 | 459.95 | -0.1 |
| 181.mcf | 119.4 | 118.99 | 0.35 |
| 186.crafty | 705.12 | 712.58 | -1.06 |
| 197.parser | 1709.33 | 1687.56 | 1.27 |
| 252.eon | 1496.51 | 1490.66 | 0.39 |
| 253.perlbmk | 1173.34 | 1164.48 | 0.76 |
| 256.bzip2 | 665.53 | 659.77 | 0.87 |
| 300.twolf | 1479.2 | 1467.94 | 0.76 |

*Таблица 2. Результаты работы QEMU на целочисленных тестах из набора SPEC CPU2000.*

## *3. Оптимизация двоичной трансляции в Valgrind*

Valgrind — инструментальное программное обеспечение, предназначенное для отладки и обнаружения ошибок в программе, а также для профилирования. Под названием Valgrind скрывается целое множество различных инструментов по анализу программ. Например, инструмент Memcheck анализирует утечки памяти и обращения к невыделенной памяти.

Valgrind сначала транслирует тестируемую программу в промежуточное представление (Intermediate Representation, сокращённо IR), которое машинно-независимо и находится в SSA-форме. Затем Valgrind инструментирует код дополнительными инструкциями (в зависимости от инструмента) и на последней стадии транслирует это представление обратно в машинный код.

В этой части данной работы мы опишем оптимизацию по планированию инструкций, которая была добавлена в Valgrind.

Как и в случае с QEMU, единицей двоичной трансляции является расширенный базовый блок. До и после аннотирования кода отладочными

инструкциями Valgrind производит машинно-независимые оптимизации на уровне промежуточного представления.

Из промежуточного представления в фазе выбора инструкций генерируется низкоуровневое промежуточное представление, очень близкое к ассемблеру целевой машины, но использующее виртуальные регистры в большинстве случаев. Реальные регистры используются в тех случаях, когда не предусмотрено альтернативы. Например, в архитектуре ARM, по соглашениям о вызовах, параметры в функцию передаются через первые четыре регистра (r0, r1, r2, r3). Поэтому инструкция передачи аргумента в функцию транслируется как копирование значения из виртуального регистра в соответствующий реальный.

Виртуальные регистры имеют только одно определение (впрочем, это определение может занимать несколько ассемблерных инструкций, которые обязательно выполняются до первого использования). Реальные регистры могут иметь сколько угодно определений.

После фазы выбора инструкций идёт фаза распределения регистров, которая заменяет виртуальные регистры реальными. Между этими двумя фазами нами была добавлена оптимизация по планированию инструкций. Она является реализацией алгоритма списочного планирования (list scheduling [8]), рассмотренного также в [9,10]. Эта оптимизация строит граф зависимостей между инструкциями, после чего присваивает им веса так, чтобы вес был тем больше, чем больше длина цепочки зависимости от этой инструкции до последней. После этого планировщик, в соответствии с моделью процессора, потактово выдаёт одну или несколько инструкций. Для этого он на каждом такте строит список доступных инструкций, то есть инструкций, вычисление всех аргументов которых уже завершилось. Далее из этого списка планировщик выбирает нужное количество инструкций с наивысшим приоритетом.

Опишем модель процессора ARM Cortex-A8, на котором мы тестировали нашу оптимизацию. На данных процессорах отсутствует аппаратное переупорядочивание инструкций, поэтому эффект от планирования должен быть существенен. В процессорах Cortex-A8 имеется два конвейера и, следовательно, на один такт можно помещать до двух инструкций, если они удовлетворяют ограничениям: две инструкции работы с памятью, две инструкции перехода не могут быть выданы на одном такте, а также операция умножения всегда выполняется на первом конвейере. Если между инструкциями на разных конвейерах возникает зависимость, то это может приводить к простою конвейера. В архитектуре ARM возможно адресовать 16 регистров общего назначения. Из них занято 4 регистра: r15 — счётчик команд, r14 — адрес возврата, r13 — указатель стека, r8 использует для адресации своих внутренних данных Valgrind. Помимо них, регистры r0-r3 участвуют в соглашениях о вызовах. Остальные 8 регистров могут быть использованы произвольным образом.

Поскольку количество свободных регистров ограничено, слишком агрессивное переупорядочивание инструкций приводит к сбросу регистров в память, что негативно сказывается на производительности. Поэтому планировщик оценивает на каждом такте регистровое давление (количество живых регистров в данной точке программы) [8] и старается планировать инструкции так, чтобы это давление не превышало 6 регистров. Данная оценка была получена экспериментально.

Однако эффект от одного лишь планировщика снижался по двум причинам. Во-первых, в силу архитектурных причин, в низкоуровневом внутреннем представлении в Valgrind вместе с обычными командами ассемблера ARM использовались также виртуальные команды, состоящие из нескольких реальных команд. Это снижало эффективность планировщика из-за того, что он не всегда мог оперировать реальными инструкциями и поэтому рассчитать латентность и такты процессора (см. также схожие проблемы в [11]). К тому же, виртуальные команды оперировали фиксированными реальными регистрами для промежуточных вычислений, что мешало планировщику оценивать текущее давление регистров. Поэтому пришлось реализовать дополнительное преобразование, которое работает перед планировщиком инструкций и транслирует виртуальные команды в эквивалентную последовательность реальных, заменяя эти лишние фиксированные регистры на виртуальные.

Во-вторых, алгоритм распределения регистров был устроен так, что если требовалось назначить на место очередного виртуального регистра реальный, он искал свободный реальный регистр в массиве, просматривая с начала до конца. Из-за этого, если регистр умирал на одной инструкции, он мог быть тут же распределён в инструкции, непосредственно следующей за ней. В результате между этими инструкциями возникает зависимость, и они не могут быть выполнены параллельно на двух разных конвейерах. Однако планировщик работает с виртуальными регистрами, и считает, что между этими инструкциями зависимостей нет. В сумме, это приводит к неоптимальному планированию. Поэтому, алгоритм распределения регистров был модифицирован таким образом, чтобы освобождающийся реальный регистр попадал в конец списка поиска.

Мы протестировали получившийся планировщик инструкций на целочисленных тестах из набора SPEC CPU2000 [7]. На тестах 164.gzip, 181.mcf и 256.bzip2 было получено ускорение (3.30%, 2.07% и 2.32% соответственно), а на тестах 176.gcc, 186.crafty, 253.perlbmk, 300.twolf было получено замедление (4.31%, 5.45%, 1.47% и 5.24% соответственно). Это говорит о том, что данная оптимизация выглядит многообещающей, но требует дополнительной, более тщательной настройки.

## 4. Заключение

В данной статье были рассмотрены некоторые направления оптимизации динамической двоичной трансляции в QEMU и Valgrind, реализованные в ИСП РАН. Наилучшего результата удалось добиться за счёт увеличения быстродействия инфраструктуры двоичной трансляции. Простые машинно-независимые оптимизации в QEMU не приносят больших результатов, но и накладные расходы на их выполнение минимальны, поэтому они полезны в среднем. Также многообещающими выглядят оптимизации по улучшению распределения регистров в QEMU и планированию инструкций в Valgrind. Данные оптимизации заслуживают более тщательного исследования и настройки.

**Литература**


[1] Nicolas Nethercote, Julina Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. Proceedings of ACM SIGPLAN 2007 Conference on Programming Languages Design and Implementation, 2007.

[2] QEMU – Open Source Processor Emulator. http://wiki.qemu.org/Main_Page. Дата обращения: 09.12.2010.

[3] Valgrind. http://www.valgrind.org/. Дата обращения: 09.12.2010.

[4] Oprofile – A System Profiler For Linux. http://oprofile.sourceforge.net/. Дата обращения: 10.12.2010.

[5] qemu.git – the QEMU master repository. http://git.qemu.org/qemu.git/commit/?id=2c90fe2b71df2534884bce96d90cbfcc93aeedb8. Дата обращения 16.12.2010.

[6] qemu.git – the QEMU master repository. http://git.qemu.org/qemu.git/commit/?id=6648e29608ce17f6109d5696fb01f056238e2628. Дата обращения 16.12.2010.

[7] SPEC CPU2000. http://www.spec.org/cpu2000. Дата обращения 21.12.2010.

[8] Steven Muchnick. Advanced compiler design and implementation. Morgan Kaufmann Publishers Inc., 1997.

[9] А.Белеванцев, Д.Журихин, Д.Мельник. Компиляция программ для современных архитектур. Труды института системного программирования РАН, том 16, 2009, стр. 31-50.

[10] Andrey Belevantsev, Alexander Chernov, Maxim Kuvyrkov, Vladimir Makarov, Dmitry Melnik. Improving GCC instruction scheduling for Itanium. In Proceedings of GCC Developers' Summit 2005, Ottawa, Canada, June 2005, pp.1-13.

[11] Andrey Belevantsev, Maxim Kuvyrkov, Vladimir Makarov, Dmitry Melnik, Dmitry Zhurikhin. An interblock VLIW-targeted instruction scheduler for GCC. In Proceedings of GCC Developers' Summit 2006, Ottawa, Canada, June 2006, pp.1-12.

# Optimizations in Dynamic Binary Translation

*Kirill Batuzov <batuzovk@ispras.ru>*
*Alexey Merkulov <steelart@ispras.ru>*

**Abstract.** Binary translation is a process of constructing program Q's binary code from program P's binary code according to a certain specification. If a binary translation is performed in runtime it is called dynamic binary translation. We evaluate application of different optimizations during dynamic binary translation. We improve lookup of existing translation block in translation cache in QEMU, evaluate impact of register allocation algorithm on program performance in QEMU, implement simple machine independent optimizations in QEMU and implement an instruction scheduler in Valgrind. The improvement of translation block lookup gives the greatest speedup among all these optimizations. Instruction scheduler in Valgrind is promising too.

**Keywords:** program optimizations, dynamic binary translation, QEMU, Valgrind.

# Декларативный интерфейс поиска дефектов по синтаксическим деревьям: язык KAST

С. В. Сыромятников
syrom@ispras.ru

**Аннотация.** Во многих случаях дефекты программного кода могут быть выявлены путём анализа соответствующего синтаксического дерева. В данной статье рассматриваются преимущества и недостатки данного подхода, в сравнении с более сложными видами статического анализа, и обосновывается необходимость предоставления пользователю интерфейса для написания собственных обнаружителей дефектов. Рассматриваются различные подходы к реализации подобного интерфейса. Описывается новый декларативный язык, позволяющий пользователю описывать дефекты кода, которые требуется обнаруживать, в виде шаблонов для синтаксических деревьев, и рассматриваются некоторые аспекты работы анализатора этого языка.

**Ключевые слова:** синтаксическое дерево; дефект кода; чекер; программа; код; декларативный интерфейс; язык; шаблон

## *1. Введение*

На сегодняшний день существует достаточное количество средств, позволяющих анализировать исходный код программ и выявлять в нём те или иные дефекты. Как правило, анализу кода предшествует синтаксический разбор, в результате которого строится *синтаксическое дерево*, структура которого отражает синтаксическое строение исходного кода ([1]). Далее для некоторых узлов данного дерева вычисляются соответствующие им семантические характеристики.

Для поиска наиболее существенных дефектов кода необходимо выявлять возникающие в процессе его исполнения зависимости между структурными элементами. Это требует моделирования и анализа потоков команд и данных, что является существенно более сложной задачей по сравнению с анализом статических синтаксических структур ([2, 3]). Однако анализ синтаксических деревьев как таковых тоже оказывается полезным для выявления определённой категории дефектов (*СД-дефектов*). Во многих случаях дефекты такого рода суть отступления от некоторых принятых правил написания программ (*Coding Style*), которые регламентируют именование языковых сущностей, наличие или отсутствие скобок, использование тех или

иных системных функций и т. п. Примером типичного Coding Style СД-дефекта может служить следующий пример.

***Пример 1.*** *Переменная неявно объявлена как целочисленная:*

```
const x;
```

Однако не следует полагать, что неверно расставленными скобками исчерпывается множество дефектов, потенциально находимых по синтаксическим деревьям:

***Пример 2.*** *Слишком большие структуры данных передаются по значению в качестве аргументов функции.*

Интересен также:

***Пример 3.*** *В декларации класса определяется конструктор копирования, но не определяется оператор присваивания*.

Характерной особенностью последнего примера является то, что данный дефект может быть обнаружен и при помощи анализаторов потоков данных, причём более точно, поскольку наличие конструктора копирования при отсутствии оператора присваивания критично практически только в тех случаях, когда класс содержит данные, копируемые нетривиальным образом, например, С-строки. Однако анализом дерева также можно выявить значительное количество подобных дефектов; в силу того, что такой анализ является гораздо менее ресурсоёмким по сравнению с анализом потока данных, во многих случаях его применение оказывается оправданным.

Нередко бывает так, что написать абсолютно универсальный обнаружитель того или иного дефекта, выявляющий все его потенциальные разновидности, не представляется возможным. Подкласс дефектов, достоверное выявление которых оказывается достаточным, часто определяется потребностями конкретного пользователя. Если же говорить о Coding Style, то своды подобных правил, которые имеются во многих крупных компаниях, занимающихся разработкой программного обеспечения, тем более являются специфичными именно для данных организаций. Поэтому представляется логичным, чтобы программные средства, предназначенные для выявления СД-дефектов, давали пользователю возможность самостоятельно описывать ошибки, которые он хочет искать, в терминах синтаксических деревьев.

Одним из вариантов решения этой задачи является предоставление пользователю некоего программного интерфейса (API) для доступа к структурам дерева, при этом выявители дефектов (*чекеры*, или, в данном случае, *СД-чекеры*) пишутся на соответствующем языке программирования (например, С) и требуют компиляции. Как показывает практика, при этом накладные расходы на написание чекера оказываются довольно велики, что в случае несложных дефектов является неудобным. Поэтому более привлекательным представляется другой подход: описывать поддеревья синтаксического дерева, соответствующие тем или иным дефектам, на

некотором специальном языке в виде *шаблонов*, что позволит пользователю во многих случаях обходиться без создания собственных скомпилированных библиотек. Подобный язык, получивший название *KAST*, был нами разработан на базе известного языка поиска на деревьях XPath ([4, 5]).

## 2. Синтаксические деревья: основные понятия

Понятие *дерева абстрактного синтаксиса* (*ДАС*) определяется, например, в работе [1]. Напомним, что ДАС — это древовидная структура данных, отражающая синтаксическую структуру программного кода. *Узлы* ДАС соответствуют синтаксически завершённым элементам кода - целочисленным константам, бинарным выражениям, условным операторам, определениям функций и так далее. Поскольку более сложные синтаксические конструкции включают в себя более примитивные, у узла ДАС могут быть ссылки на другие узлы, называемые его *дочерними узлами*; так, узел, соответствующий бинарному выражению, имеет ссылки на узлы-операнды.

Именно наличие дочерних узлов делает структуру ДАС истинно древовидной (а не линейной); корень соответствующего дерева обыкновенно соответствует максимальной единице синтаксического анализа (то есть одной единице компиляции). Следует различать *простые* (условие в условном операторе) и *списочные* (набор параметров функции) дочерние ссылки. Последние представляют собой однородный массив, вообще говоря, произвольной длины.

Однако ДАС, будучи хорошим объектом для теоретических исследований, несколько отличается от того, что обыкновенно получается в результате работы синтаксических анализаторов (вполне конкретных синтаксических деревьев). Так, с каждым узлом синтаксического дерева связан ряд характеристик, описывающих свойства соответствующей языковой конструкции (*атрибутов*), например, значения литералов или конкретные типы выражений. Кроме того, после построения дерева на нём осуществляется работа семантического анализатора, в результате чего некоторые узлы оказываются дополнительно снабжены *семантической информацией*. Так, выражение - функциональный вызов дополняется ссылкой на информацию о вызываемой функции. Практика показывает, что нередко использование семантической информации в СД-чекерах делает их более эффективными по скорости и расходу памяти, однако ещё чаще оказывается, что создание тех или иных чекеров без использования семантической информации оказывается попросту невозможным.

Стоит также отметить, что часто разные типы узлов синтаксического дерева обладают сходными свойствами и сходным набором атрибутов и могут использоваться в одних и тех же синтаксических позициях. К примеру, узлы, соответствующие бинарным выражениям (скажем, сложению) имеют две дочерних ссылки — на левый и правый операнд. Эти операнды могут быть выражениями достаточно произвольного вида — идентификаторами, константами, скобочными выражениями и так далее, то есть представляться

узлами, вообще говоря, различных типов. Сказанное означает, что систему типов узлов дерева целесообразно организовывать по иерархическому принципу, и, таким образом, для них вводятся понятия *подтипа* и *супертипа*.

## 2.1. Существующие системы

Существует достаточно много систем, осуществляющих, в том числе, и поиск дефектов кода по синтаксическим деревьям, как свободно распространяемых - Checkstyle, RATS, Flawfinder и так далее, так и коммерческих - AppPerfect, Jstyle, Coverity и других. ([6-12]). Приведём несколько примеров дефектов, потенциально находимых СД-анализом и обнаруживаемых данными системами (все примеры, кроме последнего, касаются языка java):

- Пустые catch-блоки.
- Лишние конструкторы (конструктор в классе единственный, публичный, без аргументов и без тела).
- Использование оператора 'continue'.
- Использование оператора 'new' для строк.
- Использование потенциально небезопасных функций ('gets' вместо 'fgets').

Большинство этих программных систем не имеет возможностей расширения; небольшое их число предоставляют пользователю API для написания чекеров, а из систем, реализующих декларативный язык описания дефектов, можно упомянуть только PMD и некоторые системы, её интегрирующие (Coverity).

Система PMD ([11]), ориентированная исключительно на язык java, предоставляет пользователю как API для написания чекеров в виде классов java, так и возможность создания шаблонов дефектов на языке Xpath. Рассмотрим следующий пример кода на java:

```
class Foo {
}
class Example {
  void bar(Foo x, Foo y) {
    if (y == null)
      y = x;
    else
      return;
  }
}
```

и попытаемся создавать Xpath-шаблоны, соответствующие тем или иным фрагментам данного кода.

***Пример 4.*** *Найти все параметры методов, имеющие тип 'Foo':*

```
// MethodDeclaration / MethodDeclarator / FormalParameters /
FormalParameter / Type / ReferenceType / ClassOrInterfaceType
[@Image='Foo']
```

Здесь уже видны некоторые недостатки подхода PMD:

- не различаются типы узлов и имена дочерних ссылок (что может провоцировать пользователя на синтаксические ошибки при написании шаблонов);
- отсутствует возможность адресации конкретного члена для списочной дочерней ссылки. Приведённому шаблону, вообще говоря, соответствуют оба параметра метода из примера. Если нас интересует только первый параметр, нам поможет такой шаблон:

```
// MethodDeclaration / MethodDeclarator / FormalParameters /
FormalParameter [ following-sibling::FormalParameter ] / Type /
ReferenceType / ClassOrInterfaceType [@Image='Foo'],
```

но подобное решение никак не может быть признано универсальным.

Кроме того, у пользователя практически отсутствует доступ к семантической информации, что делает реализацию ряда чекеров неэффективной, а многих — просто невозможной. Так, Xpath-интерфейс PMD не позволяет отслеживать вызовы статических методов класса.

Стоит также отметить, что иногда неудобство использования описываемого интерфейса PMD обусловлено недостатками в структуре используемой реализации синтаксического дерева. Так, нижеприведённому шаблону будут соответствовать обе ветви условного оператора — then и else:

```
// IfStatement / Statement
```

## *3. Интерфейсы описания дефектов*

В ряде случаев программная реализация дерева не позволяет прикладным программам удобно работать с ним напрямую при поиске дефектов. Это означает, что для доступа к его структурам может потребоваться более или менее сложный программный интерфейс (API); будучи хорошо спроектированным, такой интерфейс обладает очень большой выразительной мощностью. Схема использования API, так или иначе, остаётся следующей. Используя функции или классы интерфейса, пользователь пишет программный код, выявляющий интересующие его дефекты в синтаксическом дереве (СД-чекер). Это код компилируется в модуль известного формата, например, динамически подгружаемую библиотеку. Впоследствии специальный анализатор, работающий после стадий синтаксического и семантического анализа, подгружает нужные модули и запускает требуемые функции (методы).

Таким образом, объём работ по написанию даже примитивных чекеров оказывается довольно велик, и в случае простых дефектов подобная технология оказывается не очень удобной. Более привлекательным в этом плане представляется другой подход: описывать поддеревья синтаксического дерева, соответствующие тем или иным дефектам, на некотором языке в виде шаблонов. В отличие от описания алгоритмов поиска дефектов, шаблоны могут быть помещены непосредственно в некоторый конфигурационный файл, который не требует компиляции и прочитывается анализатором при каждом запуске. Это приводит к дополнительному усложнению анализатора, но позволяет пользователю во многих случаях обходиться без сложной процедуры создания собственных программных модулей. Недостатком подобного подхода является потенциально меньшая по сравнению с API выразительная мощность, однако, как показывает практика, эта проблема может быть решена, если язык шаблонов будет расширяем.

Синтаксис такого языка шаблонов может быть достаточно произвольным; так, разработанный нами язык KAST основан на языке поиска на деревьях XPath.

## *4. Описание языка KAST*

Основой для языка описания шаблонов был выбран язык XPath ([4,5]). Первоначально XPath был разработан как язык поиска заданных шаблонов в xml-документах, а xml-документы, как известно, представимы в виде деревьев. Типичный запрос на языке XPath, предназначенный для поиска заданных поддеревьев во всём xml-документе (с любым расположением относительно корня соответствующего дерева), имеет следующий вид (#):

*// Node1 {**[conditions1]**} / {axis2::}Node2 {**[conditions2]**} / {axis3::}Node2 {[conditions2]} ...*

Здесь элементы языкового синтаксиса выделены **полужирным** начертанием, имена метапеременных – *курсивом*, а необязательные элементы заключены в фигурные скобки *{}*.

Рассмотрим для примера следующий Xpath-шаблон:

**// A [ @id = 2 ] / parent::B**

Если запрос с данным шаблоном применить к приведённому ниже xml-документу, будут найдены узлы, выделенные полужирным шрифтом:

```
<B>
 <A id="2"/>
 <C>
  <A id="2"/>
  <B>
   <A id="2"/>
   <A name="foo"/>
  </B>
```

```
    </C>
</B>
```

Однако непосредственное использование XPath для поиска шаблонов в синтаксических деревьях оказалось не вполне удобным, главным образом потому, что этот язык не предоставляет возможности указывать имена дочерних ссылок узла в рамках шаблона. Смешивание имён дочерних ссылок с именами типов узлов, как это сделано в PMD, существенно затруднит практическую реализацию, заведомо ограничит выразительную мощность языка и, по-видимому, приведёт к излишней путанице среди пользователей. Кроме того, далеко не все стандартные *оси* (спецификаторы, обозначенные в (#) как *axisi*) и встроенные функции XPath представляются актуальными для целей поиска по деревьям рассматриваемого типа.

Поэтому стандартный XPath был модифицирован. В частности, имена дочерних ссылок стали, по сути, разновидностью осей: $Node_1$ / *ChildName*::$Node_2$ ; была добавлена поддержка переменных, увеличившая выразительную мощность языка, и произведён ряд других модификаций, преимущественно синтаксического характера.

Шаблоны поддеревьев синтаксического дерева, написанные на KAST'e, имеют следующий вид (##):

*// Tun1 [* ***Спецификация1,1*** *] ... [* ***Спецификация1,k1*** *] /*
*Квалификатор1 :: Tun2 [* ***Спецификация2,1*** *] ... [* ***Спецификация2,k2*** *] ...*
*Tunn [* ***Спецификацияn,1*** *] ... [* ***Спецификацияn,kn*** *]*

Наличие в начале шаблона двух символов '//' обязательно: оно означает, что поиск данного шаблона будет происходить в дереве повсеместно, вне зависимости от положения сопоставляемых узлов относительно корня. $Tun_i$ – имя одного из типов узлов или звёздочка ('*'), соответствующая произвольному имени типа. *Квалификаторы* – это имена дочерних ссылок узла или некоторые предопределённые оси. *Спецификации* бывают двух видов: присваивания (в которых определяются переменные) и ограничения (суживающие множество подходящих узлов). Данный шаблон соответствует некоторому поддереву синтаксического дерева, если:

1. Каждому $Tun_i$ из шаблона соответствует узел дерева, тип которого есть $Tun_i$ или является подтипом $Tun_i$ ($1 \le i \le n$).
2. Для всякого узла, сопоставленного некому $Tun_i$, выполнены все соответствующие ограничения $Спецификация_{i,j}$ ($1 \le i \le n$, $1 \le j \le ki$).
3. Для всякой пары ($Tun_i$, $Tun_{i+1}$) соответствующие узлы ($N_i$, $N_{i+1}$) должны быть связаны согласно квалификатору $Квалификатор_i$. $Квалификатор_i$ может быть именем дочерней ссылки; в этом случае

вершина $N_i$ должна ссылаться на вершину $N_{i+1}$ именно данной дочерней ссылкой. Кроме того, в качестве квалификатора могут использоваться следующие *оси*:

- *parent.* Запись "$Тип_1$ / ***parent*** :: $Тип_2$" означает, что вершина с типом $Тип_2$ является родительской по отношению к вершине с типом $Тип_1$.
- *ancestor.* Запись "$Тип_1$ / ***ancestor*** :: $Тип_2$" означает, что вершина с типом $Тип_2$ является предком (необязательно непосредственным) по отношению к вершине с типом $Тип_1$.
- *descendant.* Запись "$Тип_1$ / ***descendant*** :: $Тип_2$" означает, что вершина с типом $Тип_2$ является потомком по отношению к вершине с типом $Тип_1$.
- *following-sibling.* Запись "Тип1 / **following-sibling** :: Тип2" означает, что вершина с типом Тип2 является следующей по отношению к вершине с типом Тип1 в некотором однородном списке вершин (например, соответствующем списку параметров функции).

*Ограничения* могут содержать числовые значения и операции с ними, логические операции, строковые выражения и *подшаблоны*. Подшаблоны в ограничениях соответствуют формату (##) за тем исключением, что начинаются сразу с квалификатора. Такое изменение формата обусловлено тем, что сопоставление подшаблонов происходит не "где-то в дереве", а начиная непосредственно с текущего узла. Кроме того, в ограничениях могут использоваться вызовы функций, как предопределённых, так и определяемых пользователем. *Присваивания* служат одновременно и объявлениями *переменных*; переменной может присваиваться любое выражение, удовлетворяющее условиям на синтаксис ограничений. Переменная видна всюду в пределах данного шаблона после своего определения и может использоваться в ограничениях и других присваиваниях наравне с другими выражениями.

Поиск шаблона (##) в синтаксическом дереве происходит согласно нижеследующему алгоритму:

1. В дереве ищутся узлы, имеющие тип $Тип_1$ или тип которых является подтипом типа $Тип_1$. Найденный такой узел полагается *текущим*.
2. Тип текущего узла сравнивается с $Тип_i$ (если это не было сделано раньше). Если они совпадают или $Тип_i$ является супертипом для типа текущего узла, данный шаг признаётся успешным и происходит переход к пункту 3.

3. Происходит вычисление значений всех переменных и всех ограничений для текущего узла в порядке их следования в шаблоне. В частности, ограничение-подшаблон считается истинным, если его сопоставление прошло успешно. Если хотя бы одно из ограничений не выполняется, сопоставление признаётся неуспешным. Если все ограничения истинны и $Тип_i$ – последний в цепочке, исходное сопоставление считается успешным. Если же ограничения истинны, но данный тип не последний, происходит переход к следующему узлу согласно пункту 4.
4. Следующий текущий узел определяется согласно квалификатору $Квалификатор_i$. Если это имя дочерней ссылки, происходит переход по соответствующей ссылке. Если это какая-то из осей, происходит переход к одной из вершин, соответствующих данной оси. Таковые вершины будут перебираться до тех пор, пока это возможно или пока не будет успешно сопоставлена оставшаяся часть шаблона. Если текущий узел найден, сопоставление продолжается согласно пункту 2, иначе оно считается неуспешным.

Следует отметить, что при нахождении удовлетворяющего шаблону узла поиск не прекращается и ищутся остальные аналогичные дефекты в данном исходном коде.

Чтобы не ограничивать пользователя жёсткими рамками декларативного шаблона, была реализована возможность определения пользовательских функций, которые пишутся на процедурном API для доступа к синтаксическим деревьям. Это делает выразительную мощность нашего декларативного языка сопоставимой с выразительной мощностью API.

## *5. Реализация*

Анализ существующих систем с открытым исходным кодом, поддерживающих **XPath** ([13, 14]), показывает, что обычно в них используется следующий алгоритм поиска на деревьях:

1. xml-документ рассматривается как множество узлов, каждый из которых имеет не более одной ссылки на отца и некоторое количество ссылок на дочерние узлы. В процессе работы анализатор оперирует неким подмножеством этого множества, которое мы будем называть *активным множеством*. На начальном этапе активное множество полагается равным всему множеству узлов документа.
2. Когда анализатору передаётся запрос, он вычисляет подмножество активного множества, которое соответствует первому элементу запроса. После этого от запроса отрезается его первый элемент.

3. Если вычисленное подмножество непусто и оставшаяся часть запроса также непуста, активное множество полагается равным найденному подмножеству и вычисление повторяется согласно пункту 2.
4. Если вычисленное подмножество пусто, делается вывод, что узлов, соответствующих данному запросу, в документе нет.
5. Если вычисленное подмножество непусто, а оставшаяся часть запроса пуста, найденное подмножество и есть результат применения данного запроса к данному документу.

Кроме того, ввиду ограниченного объёма xml-документов для реализации анализаторов XPath нередко используются функциональные языки программирования ([15]). Размеры же синтаксических деревьев нередко бывают чрезвычайно велики (порядка миллионов узлов), а число шаблонов поиска может достигать сотен. Это делает описанные выше манипуляции с множествами вершин неэффективными, а применение рассмотренного алгоритма и функциональных языков для решения нашей задачи, как следствие, - неудобным и нецелесообразным. Более логичным представляется другой подход, основанный на планомерном обходе дерева сверху вниз и слева направо:

1. Обход начинается с корня дерева, каковым обычно является узел, представляющий единицу трансляции (исходный файл, с включёнными заголовочными файлами в случае C/C++).
2. В каждой вершине происходит попытка применить все правила, для которых данный тип вершин является стартовым. Если сопоставление правила прошло успешно, этот факт фиксируется.
3. Аналогично обходятся все дочерние ссылки данной вершины.

Если обозначить размер дерева через *A*, а количество шаблонов через *p*, то, очевидно, сложность такого алгоритма можно будет оценить снизу величиной *O(Ap)*. Данная оценка будет достигаться, если ни для какого из шаблонов не нашлось вершин, тип которых являлся бы для этого шаблона стартовым. Если же вершина с нужным типом в дереве всё же нашлась и реальное сопоставление шаблона с поддеревом началось, следует учитывать свойства самого шаблона, которые мы назовём его *сложностью*.

На первый взгляд кажется, что сложность определяется главным образом максимальным количеством вершин, которые придётся обойти при сопоставлении с данным шаблоном. Однако опыт написания реальных шаблонов показывает, что многие из них ограничены весьма небольшими поддеревьями (не более 4-5 вершин), поэтому наибольший вклад в величину сложности вносят как раз условия, не требующие обхода поддерева. Так или иначе, сложность шаблона, очевидно, может быть оценена количественно, и поэтому, имея некоторый их набор, мы можем определить максимальную сложность шаблона *l*. Тогда сложность нашего алгоритма можно будет оценить сверху величиной *O(Apl)*.

Стоит, однако, отметить, что сложность шаблонов практически может быть оценена сверху некоторым константным значением, не зависящим от их конкретного набора. СД-дефекты обычно таковы, что шаблоны для их описания требуются сравнительно небольшие. Дефект, содержащий в своём описании очень большое количество ограничений, перестаёт быть некоторой универсальной категорией и становится экзотическим случаем. Соответствующий чекер с большой вероятностью будет исключительно труден для понимания и сопровождения. Поэтому для большинства случаев верхнюю оценку сложности нашего алгоритма можно также записать как *O(Ap).*

## *6. Пример работы*

Проиллюстрируем использование языка KAST следующим примером (для языка C++):

*Найти все публичные члены класса, имеющие тип* int.

Сама по себе эта задача, по-видимому, не имеет большого практического значения, но может являться подзадачей при обнаружении некоторого действительно значимого дефекта. Мы будем пользоваться программным инструментом под названием *Checker Studio*, входящим в состав продуктов компании Klocwork. Данная программа графически изображает синтаксическое дерево для заданного фрагмента исходного кода и подсвечивает фрагменты кода и узлы дерева, соответствующие заданному KAST-чекеру. Код, на котором мы будем искать наш дефект, выглядит так:

```
class Zoo {
        private:
        int deers;
};
class Forest {
       public:
        int bears;
};
```

Итак, поиск членов класса начнём с поиска самих определений классов:

```
// ClassType
```

Этому шаблону, естественно, соответствуют оба определения класса из нашего примера (рис. 1):

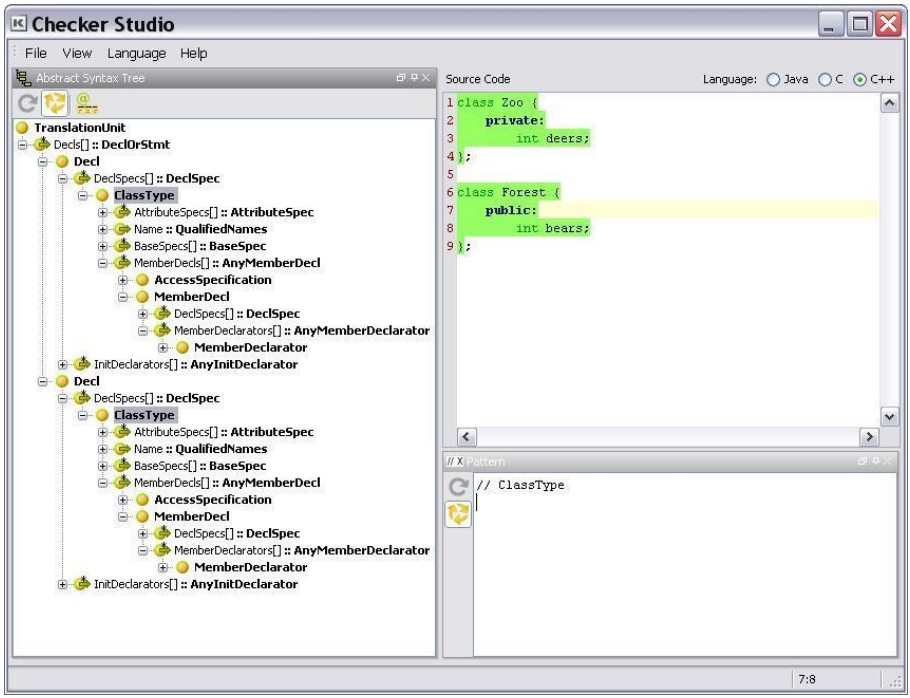



*Рис. 1.*

Но нас интересуют не сами классы, а их члены-переменные:

*// ClassType / MemberDecls[*]::MemberDecl*

Звёздочка означает, что данный член может находиться на произвольном месте в списке членов класса. Если бы нам было важно, чтобы он был, например, третьим, мы бы вместо звёздочки указали число 2 (нумерация элементов списка начинается с 0).

Для данного шаблона также нашлось два соответствия (рис. 2):

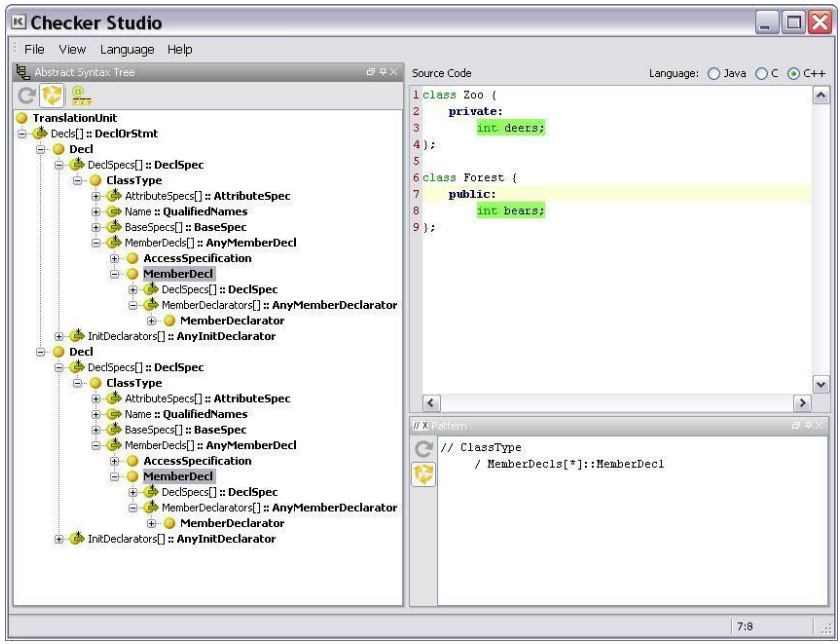

Checker Studio
File View Language Help
Abstract Syntax Tree
TranslationUnit
Decls[] :: DeclOrStmt
Decl
DeclSpecs[] :: DeclSpec
ClassType
AttributeSpecs[] :: AttributeSpec
Name :: QualifiedNames
BaseSpecs[] :: BaseSpec
MemberDecls[] :: AnyMemberDecl
AccessSpecification
MemberDecl
DeclSpecs[] :: DeclSpec
MemberDeclarators[] :: AnyMemberDeclarator
MemberDeclarator
InitDeclarators[] :: AnyInitDeclarator
Source Code
Language: Java C C++
1 class Zoo {
2 private:
3 int deers;
4 };
5
6 class Forest {
7 public:
8 int bears;
9 };
Pattern
// ClassType
/ MemberDecls[*]::MemberDecl
7:8

*Рис. 2.*

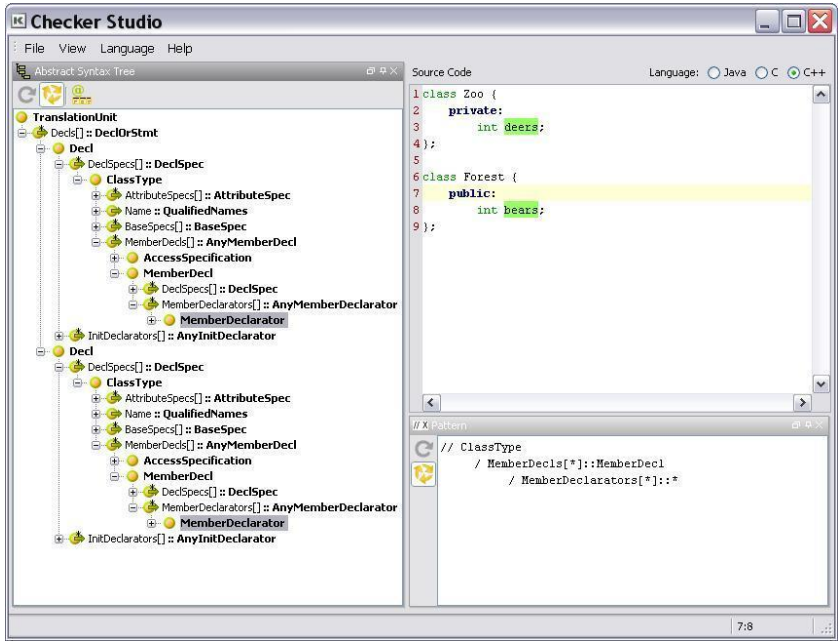

Checker Studio
File View Language Help
Abstract Syntax Tree
TranslationUnit
Decls[] :: DeclOrStmt
Decl
DeclSpecs[] :: DeclSpec
ClassType
AttributeSpecs[] :: AttributeSpec
Name :: QualifiedNames
BaseSpecs[] :: BaseSpec
MemberDecls[] :: AnyMemberDecl
AccessSpecification
MemberDecl
DeclSpecs[] :: DeclSpec
MemberDeclarators[] :: AnyMemberDeclarator
MemberDeclarator
InitDeclarators[] :: AnyInitDeclarator
Source Code
Language: Java C C++
1 class Zoo {
2 private:
3 int deers;
4 };
5
6 class Forest {
7 public:
8 int bears;
9 };
Pattern
// ClassType
/ MemberDecls[*]::MemberDecl
/ MemberDeclarators[*]::*
7:8

*Рис. 3.*

Далее мы перейдём непосредственно к декларации конкретного члена:

*// ClassType / MemberDecls[*]::MemberDecl /*
*MemberDeclarators[*]::**

Как видно, для нас не важна не только позиция данного члена в данной декларации (а их там может быть несколько, например *int x, *y;*), но и тип соответствующего узла дерева. В данном случае отсутствие проверки типа скомпенсируется ограничениями, которые будут наложены на данный узел далее. Результат поиска по данному шаблону выглядит так (рис. 3):

Осталось среди найденных членов класса выбрать искомые. Во-первых, они должны иметь тип int:

*// ClassType / MemberDecls[*]::MemberDecl /*
*MemberDeclarators[*]::* [ getTypeName() = 'int' ]*

Этому ограничению по-прежнему удовлетворяют два члена класса и два узла дерева (рис. 4):

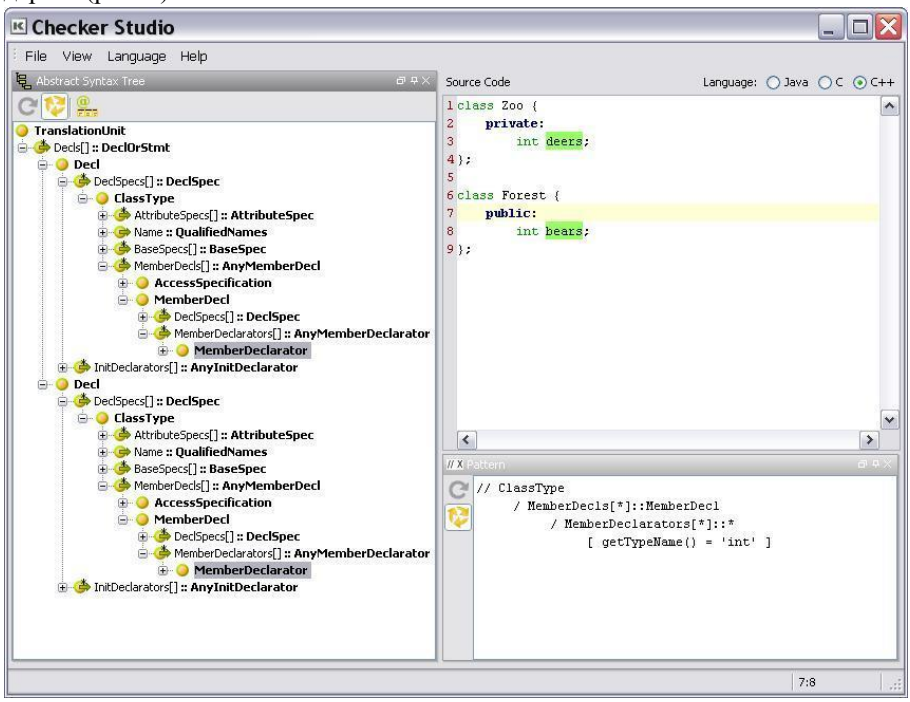



*Рис. 4.*

Во-вторых, они должны быть публичны:

*// ClassType / MemberDecls[*]::MemberDecl /*
*MemberDeclarators[*]::* [ getTypeName() = 'int' ] [ isPublic() ]*

Это ограничение позволяет отсеять один из первоначально найденных членов (рис. 5):

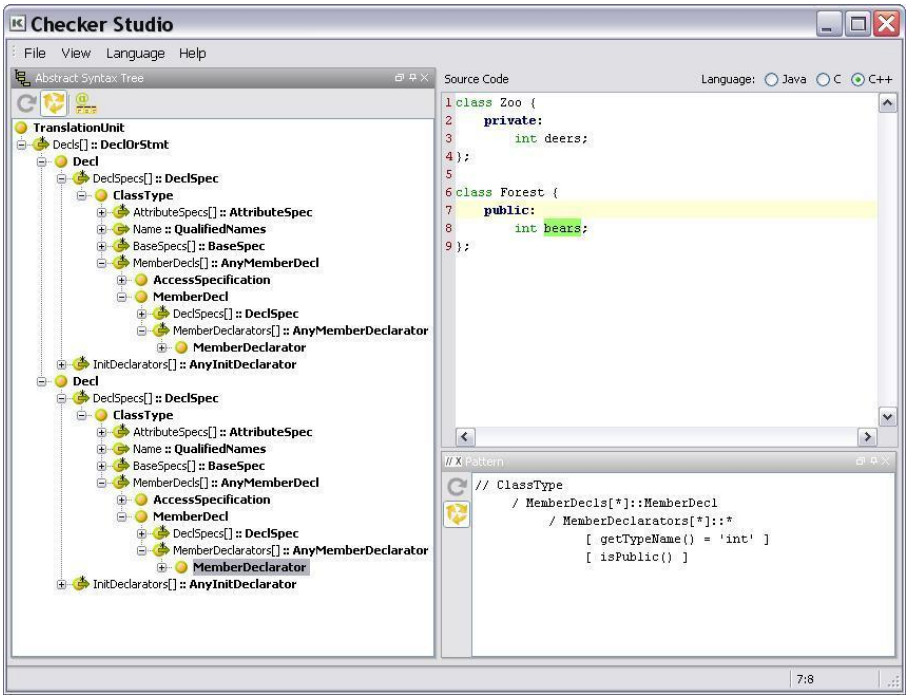



*Рис. 5.*

Подобно тому, как мы последовательно строили наш шаблон, работает и сам сопоставитель шаблонов. Для данного шаблона сначала ищутся определения классов (рис. 1); найдя такое определение, сопоставитель проверяет, выполняются ли наложенные на это определение ограничения (в данном случае их нет) и в соответствии со спецификацией (/ *MemberDecls[*]::*) переходит к следующему узлу (рис. 2). Аналогично, после проверки ограничений второго узла, происходит переход к третьему узлу (рис. 3) и проверяются его ограничения (рис. 4, 5).

## *7. Результаты*

В настоящее время поддержка KAST реализована (с некоторым достаточным набором встроенных функций) для языков программирования C/C++ и Java; в значительной степени также поддержан язык C#. В частности, на первоначальном этапе для языков C и C++ в среде Unix было написано порядка 25 чекеров, как простых, ориентированных преимущественно на выявление несущественных дефектов кода, так и более сложных. При помощи этих чекеров были проанализированы исходные коды проектов

Mozilla Firefox 3.5, Apache http server (httpd) 2.2.14 и postgresql 8.4.2. Результаты анализа представлены в таблице 1.

| | **Всего потенциальных дефектов** | **Из них реальных** | **Доля реальных дефектов** |
|---|---|---|---|
| Firefox | 393 | 226 | 58% |
| httpd | 7 | 3 | 43% |
| postgresql | 33 | 21 | 64% |
| Всего | 433 | 250 | 58% |

*Табл. 1. Результаты работы KAST-чекеров на некоторых проектах.*

При этом нужно отдавать себе отчёт, что процент ложных срабатываний для простых и (часто, но не всегда) не очень важных дефектов был заметно ниже, чем для более сложных. Также следует отметить, что для 50% всех имеющихся чекеров не было отмечено ни одного срабатывания, истинного либо ложного, что объясняется главным образом спецификой анализируемых проектов.

При общем времени работы программы над данным проектом порядка 50 минут (оно включает в себя, главным образом, время, затрачиваемое на построение синтаксических деревьев, работу синтаксического анализатора и работу не-KAST чекеров) время работы собственно KAST-анализатора составило всего несколько минут.

## *8. Заключение*

Разработанный нами язык оказался удобным средством описания несложных дефектов кода, находимых по синтаксическому дереву. Он широко используется в компании Klocwork при написании поставляемых конечным клиентам чекеров. Более того, практика показывает, что и сами клиенты активно пользуются предоставленной возможностью по разработке собственных обнаружителей дефектов.

Основная возможность для усовершенствования нашего языка шаблонов — это добавление новых встроенных функций, при этом исключительно полезным оказывается анализ отзывов и пожеланий конечных пользователей, которые в процессе анализа собственного кода сталкиваются с отсутствием той или иной функциональности в языке KAST. Кроме того, ввиду

относительной новизны описанной технологии и, как следствие, отсутствия некоего канонического варианта синтаксиса языка, возможно добавление новых осей. Не все оси, имеющиеся в стандартном Xpath, на данный момент поддерживаются в KAST, и возможно, наличие некоторых из них было бы полезно при поиске дефектов в синтаксических деревьях.

Перспективы расширения KAST за счёт поддержки новых языков программирования представляются достаточно туманными, поскольку поддержка для основных коммерчески используемых процедурных языков (C/C++, Java, C#) уже реализована.

## Приложение.

### Примеры чекеров на KAST

Приведём в качестве примеров чекеры, написанные на KAST и предназначенные для поиска дефектов, описанных во введении.

***Пример 1'.*** Переменная неявно объявлена как целочисленная

```
// Decl [ not InitDeclarators::Null ]
        [ not DeclSpecs[*]::AnyTypeOf ]
        [ not DeclSpecs[*]::ReservedTypeSpec ]
        [ not DeclSpecs[*]::ClassType ]
        [ not DeclSpecs[*]::EnumType ]
        [ not DeclSpecs[*]::TypeName ]
```

***Пример 2'.*** Слишком большие структуры данных передаются по значению в качестве аргументов функции.

```
// FuncDeclarator  [ ancestor::FuncDef ]
                [ $1 := getName() ]
               / Params[*]::Decl [ InitDeclarators[*]::InitDeclarator
                                [ $type := getInitialDefinedType() ]
                                    [ not $type.isArray() ]
                                    [ not $type.isPointer() ]
                                    [ not $type.isReference() ]
                                    [ $type.getTypeSize() > 128 ] ]
```

***Пример 3'.*** В декларации класса определяется конструктор копирования, но не определяется оператор присваивания.

*// ClassType*

*[ MemberDecls[*]::MemberDecl*

*/ MemberDeclarators[*]::MemberDeclarator*

*[ isCopyConstructor() ]*

*| MemberDecls[*]::MemberFunc / FuncDef::FuncDef*

*/ Declarator::Declarator [ isCopyConstructor() ] ]*

*[ not MemberDecls[*]::MemberDecl*

*/ MemberDeclarators[*]::MemberDeclarator*

*[ isAssignmentOperator() ] ]*

*[ not MemberDecls[*]::MemberFunc / FuncDef::FuncDef*

*/ Declarator::Declarator [ isAssignmentOperator() ] ]*

**Литература**


[1] Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. Т. 1. Синтаксический анализ. М., 1978. 613 с.

[2] S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, 1997. 888 с.

[3] A. Belevantsev, O. Malikov. Using data flow analysis for detecting security vulnerabilities. Сборник трудов Института системного программирования РАН. Под ред. чл.-корр. РАН Иванникова В.П. Т. 11. М., ИСП РАН, 2006. 128 с., с. 83-98.

[4] http://www.zvon.org/xxl/XPathTutorial/General/examples.html

[5] http://www.w3schools.com/XPath/default.asp

[6] http://www.mmsindia.com/jstyle.html

[7] http://www.fortify.com/security-resources/rats.jsp

[8] http://www.dwheeler.com/flawfinder

[9] http://www.coverity.com

[10] http://www.appperfect.com/products/java-code-test.html

[11] http://pmd.sourceforge.net

[12] http://checkstyle.sourceforge.net

[13] http://tinyxpath.sourceforge.net

[14] http://xerces.apache.org

[15] Д. А. Лизоркин. Оптимизация вычисления обратных осей языка XML Path при его реализации функциональными методами. Сборник трудов Института системного программирования РАН. Под ред. чл.-корр. РАН Иванникова В.П. Т. 8, ч. 2. М., ИСП РАН, 2004. 214 с., с. 93-119.

[16] R. Cole, R Hariharan, P. Indyk. Tree pattern matching and subset matching in deterministic O(n log3m) time. Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, Baltimore, MD, 1999, pp. 245-254.

# Declarative Interface of Detecting Defects on Syntax Trees: KAST Language

S. V. Syromyatnikov
syrom@ispras.ru

**Summary.** In many cases source code defects can be detected by analyzing the corresponding syntax trees. In this article we investigate advantages and drawbacks of this approach in comparison with more complex types of static code analysis and prove that a user needs an interface for writing his own defect detectors. Different ways of implementing such an interface are considered. We also introduce a new declarative language for describing code defects that a user wants to detect, in the form of syntax tree patterns. Some aspects of the language analyzer implementation are also discussed.

**Keywords:** syntax tree; source code defect; checker; program; source code; declarative interface; language; pattern

# О моделировании счётчиков с бесконечным числом значений в обыкновенных сетях Петри.

*Л. В. Дворянский*
*leo@ispras.ru*

**Аннотация.** В статье проведен анализ моделирования в обыкновенных сетях Петри счетчиков с бесконечным числом состояний. Обоснован выбор отношения эквивалентности симуляции готовности в качестве отношения реализации для моделирования счётчиков. Показано, что в сетях Петри невозможно промоделировать счётчики с бесконечным числом состояний. Представлена минимальная модель счётчика с конечным числом значений.
**Ключевые слова:** счётчиковые машины; сети Петри; спектр фон Глэббика; отношение эквивалентности симуляции готовности.

## *1. Введение*

При моделировании поведения дискретных динамических систем часто явно или неявно возникает необходимость моделирования дискретного счётчика. Счётчик один из основных элементов семантики исполнения дискретных динамических систем. Счётчиковые машины Минского с двумя счётчиками являются универсальными вычислителями. В других моделях счётчики неявно присутствуют в виде стека, множества управляемых состояний, буфера сообщений, определенным образом занумерованных строк и пр.

При моделировании динамических систем счётчики используются для выражения зависимости исполнения одних событий от числа произошедших других событий. К примеру, счётчики используются для ограничения длительности исполнения системы заданным числом шагов или выполнения повторяющихся событий с заданным периодом.

При моделировании счётчика с помощью сетей Петри, простейшим способом является использование позиции в качестве счётчика, при котором значению счётчика соответствует значение функции разметки на позиции. В работе [1] используется подобный подход для моделирования машин Минского с помощью сетей Петри, называемый в статье «моделирование в слабом смысле» («simulate in a weak sense»). Под «моделированием в слабом смысле» понимается то, что возможны «ложные» («cheating») срабатывания переходов,

которые не могут присутствовать в исполнении оригинальной системы со счётчиком. Неизбежность ложных срабатываний объясняется свойством монотонности входных условий переходов в операционной семантике сетей Петри (см. [2]). Такая конструкция полезна для доказательства некоторых свойств (см. [1]) формализма сетей Петри, но не подходит для моделирования исполнения системы со счётчиком.

Целью данной работы является изучение проблемы моделирования счётчиков без ложных срабатываний с помощью сетей Петри. Вначале, осуществляется выбор подходящего критерия корректности и отношения реализации для моделирования счётчиков сетями Петри. Затем рассматривается возможность моделирования счётчика со счётным множеством значений в сетях Петри. В конце приводится простая (и минимальная) модель счётчика с конечным множеством значений.

## *2. Предварительные сведения*

### 2.1. Отношение квазипорядка.

**Определение 1.** Бинарное отношение называется квазипорядком или предпорядком, если оно рефлексивно и транзитивно.

**Определение 2.** Квазипорядок называется фундированным, если всякое непустое подмножество множества-носителя имеет минимальный элемент.

**Определение 3.** Квазипорядок $\leq$ на множестве $X$ называется правильным, если для любой бесконечной последовательности $x_0, x_1, x_2, \ldots$ элементов из $X$ существуют индексы $i \leq j$ такие, что $x_i \leq x_j$.

**Утверждение 1.** Квазипорядок $\leq$ на множестве $X$ является правильным в том, и только в том, случае, когда одновременно выполняются два следующих условия (см. [3], стр. 19.):

1. $X$ не содержит бесконечных строго убывающих цепей $x_0 > x_1 > x_2 > \ldots$;
2. всякая содержащаяся в $X$ антицепь конечна.

### 2.2. Операции над цепочками символов.

**Определение 4.** Операция конкатенации $\bigcirc: V^{*^*} \to V^*$ для последовательности цепочек $\alpha = \langle \alpha_1, \alpha_2, \alpha_3 \ldots \rangle$ это операция последовательного склеивания цепочек:

$$\forall i \in I : \forall j \in \overline{1, |\alpha_i|} : l = \sum_{k=1, i-1} |\alpha_k| + j : \left( \bigcirc(\alpha)_l = \alpha_{ij} \right) \qquad (0.1)$$

где $I$ – множество индексов набора $\alpha$, $\alpha_i \in V^*$ – цепочки над алфавитом $V$, $V^{**}$ – множество последовательностей таких цепочек, $\alpha_{ij}$ - $j$-ый элемент $i$-ой цепочки набора цепочек $\alpha$.

**Определение 5.** Проекция $\rho : A^* \times 2^A \to A^*$ цепочки $\alpha \in A^*$ на алфавит $B \subseteq A$ это цепочка полученная исключением из $\alpha$ всех элементов, кроме тех которые принадлежат алфавиту $B$.

$$\rho(\alpha, B) = \bigcirc\left(\left\langle \left\langle \alpha_i \right\rangle \mid \alpha_i \in B \right\rangle\right) \tag{0.2}$$

## 2.3. Дискретный счётчик.

Счётчик - одна из основных абстракций, используемых в теории вычислимости для описания поведения дискретных по времени абстрактных машин. Регистр это сущность, которая хранит одно натуральное число. Счётчик является регистром с ограниченным набором команд. Одна из первых и простейших вычислительных моделей, использующих счётчики, является регистровая (или счётчиковая) машина Минского (см.[4], стр.244).

Каждый счётчик обладает значением, определяющим его состояние в наблюдаемый момент времени. Множество возможных значений счётчика является начальным отрезком множества натуральных чисел, т.е. ограничено снизу нулевым значением и может быть ограничено или неограниченно сверху. В последнем случае значение счётчика может являться любым натуральным числом из $\overline{0,\infty}$. Наблюдение за состоянием счётчика ограничено проверкой на ноль и на положительное значение. Т.е. модель, использующая счётчик, может только различать содержит ли счётчик 0 или положительное целое число.

Определим конфигурацию счётчика (или просто счётчик) $C$ как

$$C \rightleftharpoons \langle V, v \rangle \tag{0.3}$$

где

$V = \overline{1,N}$ - множество возможных значений счётчика, где $N \in \mathbb{N} \cup \{\infty\}$;

$v \in V$ - значение счётчика.

**Определение 6.** Обозначим через $\mathbb{C}_V$ множество всех конфигураций счётчиков с множеством значений $V$.

**Определение 7.** Функция $\nu : \mathbb{C}_V \to V$ определяет значения счётчика в наблюдаемый момент времени.

Над счётчиком определен набор операций

Операция инкремента $+ : \mathbb{C}_V \to \mathbb{C}_V$ увеличивает значение счётчика на единицу.

$$\forall C \in \mathbb{C}_V : \nu(C) \neq N : \nu\left(+\left(C\right)\right) = \nu\left(C\right) + 1 \qquad (0.4)$$

Операция декремента $- : \mathbb{C}_V \to \mathbb{C}_V$ уменьшает значение счётчика на единицу:

$$\forall C \in \mathbb{C}_V : \nu(C) \neq 0 : \nu\left(-\left(C\right)\right) = \nu\left(C\right) - 1 \qquad (0.5)$$

Пусть $B = \{true,\ false\}$, тогда

Операция проверки счётчика на нулевое значение $0_? : \mathbb{C}_V \to B$ имеет значение истина, когда счётчик имеет нулевое значение

$$\forall C \in \mathbb{C}_V : \left(\nu\left(C\right) = 0\right) \Rightarrow 0_?\left(C\right) = true \qquad (0.6)$$

Операция проверки счётчика на положительное значение $\overline{0_?} : \mathbb{C}_V \to B$ имеет значение истина, когда счётчик имеет положительное значение

$$\forall C \in \mathbb{C}_V : \left(\nu\left(C\right) > 0\right) \Rightarrow \overline{0_?}\left(C\right) = true \qquad (0.7)$$

Значение счётчика меняется под воздействием операций. С каждым шагом исполнения модели значение счётчика может увеличиться на единицу, уменьшиться на единицу или остаться неизменной. Значение счётчика натуральное число, т.е. при нулевом значении счётчика операция декремента заблокирована (см.[5], стр.13). Состояние счётчика полностью определяется его значением.

## 2.4. Сеть Петри

В различных источниках предложены определения различающиеся способом обозначения составляющих формализма, хотя на семантическом уровне подобные формализмы эквивалентны. Подобные синтаксические и семиотические вопросы влияют лишь на удобство использования формализма. Зафиксируем одно из распространенных определений, приведенных в книге (см. [6], стр.14), сделав его чуть более строгим.

**Определение 8.** Сеть Петри $N$ определяется, как кортеж

$$N =< P, T, F, W, m_0 > \quad (0.8)$$

где

- $P$ – непустое конечное множество позиций сети;
- $T$ – непустое конечное множество переходов сети;
- $F$ – отношение инцидентности;

$$F \subseteq P \times T \cup T \times P \quad (0.9)$$

- $\mathbb{N}$ – множество натуральных чисел $\overline{0,\infty}$;
- $W$ – функция кратности дуг $W : F \to \mathbb{N} \setminus \{0\}$;
- $m_0$ – начальная разметка $m_0 : P \to \mathbb{N}$.

При этом выполняются следующие условия:

1) Множества позиций и переходов не пересекаются:

$$P \cap T = \emptyset \quad (0.10)$$

2) Любой элемент сети инцидентен хотя бы одному элементу другого типа:

$$\forall x \in P \cup T : \exists y \in P \cup T : xFy \vee yFx \quad (0.11)$$

**Определение 9.** Обозначим через $\Pi$ класс всех обыкновенных сетей Петри.

В некоторых случаях удобней использовать функцию инцидентности $F : P \times T \cup T \times P \to \mathbb{N}$, которая задается на основе отношения инцидентности $F$ и функции кратности дуг $W$:

$$F(x, y) = \begin{cases} W(x, y), & \text{если } xFy \\ 0, & \text{если } \neg xFy \end{cases} \quad (0.12)$$

### *2.4.1. Состояние сети Петри.*

Для определения состояния сети Петри $N$ необходимо ввести определение разметки.

**Определение 10.** Разметка это функция $m : P \to \mathbb{N}$, которая ставит в соответствие каждой позиции $p \in P$ целое число $n \in \mathbb{N}$.

**Определение 11.** Множество всех возможных разметок сети $N$ обозначим, как $\mathfrak{M}_N = \mathbb{N}^P$.

**Определение 12.** Так как, множество $P$ конечно, то пронумеровав позиции произвольным образом, разметку можно представить в виде вектора

$$m_v =< m_1, ..., m_{|P|} > \tag{0.13}$$

Такой вектор полностью определяет функцию-разметку сети Петри [7].

Определим порядок $\leq$ на разметках сети Петри используя векторное представление.

**Определение 13.** Порядок на разметках сетей Петри задается отношением $\leq \subseteq \mathfrak{M}_N \times \mathfrak{M}_N$ таким, что $m \leq m'$ тогда и только тогда, когда значение $m$ для каждой позиции меньше или равен значению $m'$ для этой позиции.

$$\begin{aligned} &\forall m, m' \in \mathfrak{M}_N : m_v =< m_1, ..., m_n >, m_v' = \\ &=< m_1', ..., m_n' >: m \leq m' \Leftrightarrow \forall i \in \overline{1,n} : m_i \leq m_i' \end{aligned} \tag{0.14}$$

**Утверждение 2.** Пусть отношение $\leq$ - правильный квазипорядок на множестве $X$. Тогда отношение $\leq^n$ на множестве векторов размерности $n$ из $X^n$ также является правильным квазипорядком (см. [3], стр.22).

**Утверждение 3.** Отношение $\leq \subseteq \mathfrak{M}_N \times \mathfrak{M}_N$ является правильным квазипорядком.

**Доказательство.** Рассмотрим (Утверждение 2), если в качестве множества $X$ рассматривать $\mathbb{N}$, а в качестве квазипорядка $\leq$ на $X$ будем рассматривать естественный порядок «меньше или равно» на натуральных числах, то отношение порядка на разметках сетей Петри введенного в (**Определение 13**) соответствует отношению $\leq^n$.

Так как классическое отношение «меньше или равно» на множестве натуральных чисел $\mathbb{N}$ является правильным квазипорядком, то в силу (Утверждение 2) отношение $\leq \subseteq \mathfrak{M}_N \times \mathfrak{M}_N$ является правильным квазипорядком. □

### *2.4.2. Исполнение сетей Петри.*

Для формального определения исполнения сети Петри опишем входную и выходную функцию переходов. Для каждого перехода $t \in T$ входная функция $^\bullet F(t) : T \to \mathfrak{M}_N$ задает разметку (входное условие перехода, охрана перехода) на основе функции инцидентности:

$$^\bullet F(t) =< f_1, ..., f_n >: \forall i \in \overline{1,n} : f_i = F(p_i, t) \tag{0.15}$$

Выходная функция $F^{\bullet}(t): T \to \mathfrak{M}_N$ определяется аналогично:

$$F^{\bullet}(t) =< f_1, ..., f_n >: \forall i \in \overline{1,n}: f_i = F(t, p_i) \qquad (0.16)$$

**Определение 14.** Переход $t$ в сети Петри $N$ называется активным при разметке $m \in M_N$, когда

$${}^{\bullet}F(t) \le m \qquad (0.17)$$

Если переход активен он может быть выполнен. Исполнение перехода $t$ переводит сеть Петри из разметки $m$ в следующую разметку $m'$:

$$m' = m - {}^{\bullet}F(t) + F^{\bullet}(t) \qquad (0.18)$$

Исполнение активного перехода называется шагом исполнения сети Петри.

1. **Определение 15.** Множеством достижимости $R(N)$ сети Петри $N$ называется множество всех разметок, достижимых из начальной разметки по правилу (0.18).

Пусть $t$ переход в сети Петри, такой что ${}^{\bullet}F(t) \le m$. Определим операцию $\varepsilon: \mathfrak{M}_N \to \mathfrak{M}_N$ исполнения перехода $t$ в сети Петри

$$\varepsilon(m,t) = m - {}^{\bullet}F(t) + F^{\bullet}(t) \qquad (0.19)$$

Исполнение $\mu$ сети Петри $N$ определяется последовательным выполнением активных переходов из начальной разметки $m_0$. Обозначим последовательность сработавших переходов как $\mu_t$, таким образом, что $\mu_t(i)$ равен переходу $t \in T$ сработавшему на $i$-ом шаге. Каждый раз при выполнении перехода, текущая разметка и множество активных переходов могут измениться. Последовательность исполнения переходов задает последовательность разметок $\mu_m$, первой из которых является $\mu_m(0) = m_0$, а все последующие соответствуют разметкам после срабатывания соответствующего перехода.

$$\forall i \in \overline{1,|\mu|}: \mu_m(i) = \varepsilon(\mu_m(i-1), \mu_t(i)) \qquad (0.20)$$

**Утверждение 4.** Свойство монотонности исполнения переходов сетей Петри (см. **[6]**, стр.19).

$$\forall m_1, m_2, m' \in \mathfrak{M}_N : m_2 = \varepsilon(m_1, t) \Rightarrow \\ (m_2 + m') = \varepsilon(m_1 + m', t) \tag{0.21}$$

## 2.5. Системы помеченных переходов

Системы помеченных переходов получили широкое распространение в качестве унифицированного средства представления процессов дискретных систем в различных задачах верификации и тестирования основанного на моделях[8].

**Определение 16.** Система помеченных переходов $LTS$ (Labelled Transition System) определяется, как множество состояний системы и отношение помеченных переходов на этом множестве (см. [9])

$$LTS \rightleftharpoons \langle \mathbb{S}, s_0, \Lambda, \rightarrow \rangle \tag{0.22}$$

где $\mathbb{S}$ - множество состояний (возможно бесконечное), $s_0$ - начальное состояние, $\Lambda$ - конечный алфавит пометок (имен действий), $\rightarrow$ - тернарная операция соответствующая возможным переходам системы из одного состояние в другое в ответ на помеченные действия.

$$\rightarrow \subseteq \mathbb{S} \times \Lambda \times \mathbb{S} \tag{0.23}$$

**Определение 17.** Обозначим через $\mathfrak{L}$ класс всех систем помеченных переходов.

Состояния представляются явно в виде элементов множества $\mathbb{S}$. В отличие от конечных автоматов, множество $\mathbb{S}$ может быть бесконечно. Последнее свойство позволяет представлять в виде систем помеченных переходов системы, как с конечным, так и с бесконечным числом состояний, а также как конечно, так и бесконечно ветвящиеся процессы.

# *3. Контекст задачи*

В рамках подхода к разработке программного обеспечения основанного на моделях (Model-Driven Development[10]), областях моделирования и анализа различных потоковых моделей бизнес-, технологических и других подобных процессов, а также в области координирования Web-сервисов стали использоваться модели с бесконечным числом состояний обладающих параллелизмом и недетерминизмом. Многие языки моделирования (UML2 Activity Diagrams, BPMN, BPEL4People и др.) используют в качестве основы для определения операционной семантики выполнения хорошо изученный формализм сетей Петри. Преимущества использования традиционного формализма заключается в строгости семантики и изученности вопросов,

связанных с разрешимостью и сложностью многих задач анализа таких моделей. Конечно, при построении таких моделей руководствуются удобством моделирования, поэтому в семантику вносятся расширения, которые увеличивают выразительную мощность моделей и модели выходят за рамки обыкновенных сетей Петри. Однако некоторые конструкции удается выразить и на языке сетей Петри. Для таких конструкций можно применять все известные алгоритмы анализа на сетях Петри.

Данная работа посвящена анализу одной из сложностей при моделировании реальных моделей с помощью обыкновенных сетей Петри, а именно моделирования счётчика с бесконечным числом состояний.

С другой стороны развитие теории алгебры процессов привело к развитию понятия соответствия процессов. То, что один процесс ведет себя аналогично другому (моделирует его) может быть выражено различными отношениями эквивалентности и предпорядка (см. например [11], [12]). Выбор конкретного отношения зависит от рассматриваемой задачи и зависит от семантики языков процессов и ожидаемого соответствия. В данной работе предложен и обоснован выбор отношения симуляции готовности (ready simulation) в качестве такого отношения.

## *4. Выбор отношения реализации*

Корректность моделирования динамической системы обычно обуславливается тем, что модель в некотором определенном смысле ведет себя аналогично моделируемой системе. Критерий корректности определяется в зависимости от решаемой задачи (см.[11]), и обычно задается в виде некоторого отношения между моделью и моделируемым объектом. Такое отношение называется отношением реализации (implementation relation). Отношение реализации может быть установлено между процессами, на основе отношения между множествами трасс процессов, множествами возможных переходов, множествами переходов, которые не могут выполниться (refusal set) и другими артефактами исполнения процессов. Выбор подходящего отношения зависит от особенностей задач и объектов моделирования.

### 4.1. Процессы счётчика и сетей Петри

Процессы рассматриваются с позиции унифицированного параллелизма (uniform concurrency) [13], т.е. все операции рассматриваются как атомарные и нам не важен внутренний характер действий.

Процессы обеих моделей допускают бесконечные исполнения, поэтому не являются конечными. В случае счётчика простейшим бесконечным исполнением является бесконечное повторное исполнение операции инкремента и операции декремента. Пример бесконечного исполнения в сетях Петри - бесконечное исполнение перехода с одним входящим и одним исходящим ребрами, инцидентными одной позиции.

Обе модели задают конечно-ветвящиеся процессы. В случае счётчиков, множество возможных действий ограничено операциями, заданными на счётчиках. В случае сетей Петри, число ветвлений ограничено конечным множеством переходов, т.к. каждое действие моделируется переходом. Из конечности ветвления следует, что обе модели также являются конечно-отображаемыми (image finite) процессами.

Счётчик является детерминированной моделью, т.к. каждое последующее состояние однозначно определяется текущим значением счётчика и выполняемой операции, т.е.

$$\begin{aligned} &\forall a \in \{+, -, 0_?, \overline{0_?}\}: \\ &:\left(a\left(C_1\right) = C_2\right) \& \left(a\left(C_1\right) = C_3\right) \to C_2 = C_3 \end{aligned} \tag{0.24}$$

Помеченные сети Петри относят к недетерминированным моделям, т.к. одним действием могут быть помечены несколько переходов и в результате выполнения одноименных переходов модель может переходить в различные состояния.

Мы не будем рассматривать внутренние события, которые могут осуществляться во время исполнения процесса, но не влияют на его функционирование. Поэтому наблюдаемые процессы являются непрерывными (concrete), т.е. такими, между явными действиями которых ничего не происходит. (Для систем помеченных переходов формально это означает, что все возможные цепочки $\tau$ событий конечны и игнорируются).

Таким образом, счётчик определяется классом не конечных, конечно-ветвящихся, детерминированных и непрерывных процессов. Формализм сеть Петри лежит в классе не конечных, конечно-ветвящихся, недетерминированных и непрерывных процессов.

Для того, чтобы иметь возможность сравнивать поведение порождаемое счётчиком и сетевыми моделями, необходимо привести счётчик и сети Петри к единому представлению. Существует много различных способов представления систем – системы помеченных переходов (labeled transition system), процессные графы (process graph), структуры заданные на множестве событий (event structure), сетевые модели, проективные пределы на рядах конечных термов и другие. Отношение эквивалентности может быть определено на любом из этих представлений. Представление процессов в виде систем помеченных переходов является удобным и распространенным способом представления произвольных процессов, чье поведение может быть выражено с помощью отношения действий.

Определим отображение $L_C : \mathbb{C} \to \mathfrak{L}$ счётчиков в системы помеченных переходов, как

$$L_C\left(C\right)=\left\langle \mathbb{S}_C, s_{C0}, \Lambda_C, \rightarrow_C \right\rangle \tag{0.25}$$

где $\mathbb{S}_C$ класс состояний, занумерованный множеством значений счётчика

$$\mathbb{S}_C=\left\langle s_i : i\in V\right\rangle \tag{0.26}$$

$$s_{C0}=s_{v_0} \tag{0.27}$$

$$\Lambda_C=\left\{+,-,0_?,\overline{0_?}\right\} \tag{0.28}$$

Отношение действия $\rightarrow_C$ определено, как

$$\forall i,\, j\in V:\forall a\in\{+,-\}:\left(< s_i, a, s_j >\in\rightarrow_C\right)\Leftrightarrow\left(j=a(i)\right) \tag{0.29}$$

$$\forall i,\, j\in V:\forall a\in\{0_?,\overline{0_?}\}:\left(< s_i, a, s_j >\in\rightarrow_C\right)\Leftrightarrow\left(i=j\,\&\,a\left(i\right)\right) \tag{0.30}$$

Определим отображение $L_N:\Pi\rightarrow\mathfrak{L}$ сетей Петри в системы помеченных переходов, как

$$L_N\left(N\right)=\left\langle S_{\mathfrak{M}_N}, s_{m_0}, \Lambda_N, \rightarrow_N \right\rangle \tag{0.31}$$

$$\mathbb{S}_{\mathfrak{M}_N}=\left\{s_i : i\in\mathfrak{M}_N\right\} \tag{0.32}$$

$$\Lambda_N=\Lambda_C \tag{0.33}$$

Введем функцию $\lambda:T\rightarrow\Lambda_N$ пометки переходов сети Петри, которая ставит в соответствие переходу пометку (действие), которое он моделируют.

1. Отношение действия $\rightarrow_N$ определено, как

$$\begin{aligned}&\forall i,\, j\in\mathfrak{M}:\forall a\in\Lambda_N:\left(\left\langle s_i, a, s_j\right\rangle\in\rightarrow_N\right)\Leftrightarrow\\&\Leftrightarrow\left(\exists t\in T: t\le i\,\&\,\lambda\left(t\right)=a\,\&\,j=\varepsilon\left(i,t\right)\right)\end{aligned} \tag{0.34}$$

Можно установить отображение $L_{NT}:\rightarrow_N\rightarrow T_N$ между элементами отношения $\rightarrow_N$ и переходами сети Петри $N$:

$$\forall \left\langle s_{m_1}, a, s_{m_2} \right\rangle \in \rightarrow_N : L_{NT}\left(\left\langle s_{m_1}, a, s_{m_2} \right\rangle\right) = t_a \Leftrightarrow$$
$$\Leftrightarrow m_2 = \varepsilon\left(m_1, t_a\right) \qquad (0.35)$$

Используя отношения $L_C$ и $L_N$ можно устанавливать отношение между множеством процессов порождаемых счётчиком $C$ и сетью Петри $N$ сравнивая системы помеченных переходов $L_C\left(C\right)$ и $L_N\left(N\right)$.

## 4.2. Отношение реализации (implementation relation).

Два процесса считаются эквивалентными, если они показывают одинаковое наблюдаемое поведение при одинаковых воздействиях, как следствие принципа равенства неразличимого. Отношения эквивалентности различаются способом наблюдения за поведением процесса. Способ наблюдения выбирается с учетом особенностей решаемой задачи.

В нашей задаче, единственной операцией позволяющей наблюдать за состоянием счётчика является проверка на ноль. Поэтому, при одинаковом воздействии на счётчик и на сетевую модель, они должны обеспечивать одинаковые значения операции $0_?$ на всем протяжении исполнения. При этом внешняя система может обратиться к операции проверки на ноль значения счётчика на любом шаге. Т.е. процессы счётчика и модели счётчика должны давать одинаковое значение проверки на ноль в любой наблюдаемый момент. Операция $0_?$ моделируется, как дополнительный переход на нулевом состоянии счётчика.

Окружение счётчика может воздействовать на счётчик с помощью операций инкремента и декремента. Таким образом, множество возможных действий системы помеченных переходов должно соответствовать множеству возможных действий самого счётчика в заданный момент исполнения. Т.е. всегда, когда над счётчиком можно выполнить операции инкремента или декремента, те же операции должны быть выполнимы над его моделью. Так как окружение счётчика может обратиться к любой из указанных операций определенных над счётчиком, возникает следующее требование.

**Требование 1.** В каждый наблюдаемый момент процесс корректной модели должен разрешать все, и только те действия, которые разрешены на процессе счётчика.

Так как окружение определяет исполнение каждого следующего действия, то счётчик и его сетевая модель должны быть эквивалентны на каждом шаге исполнения.

**Требование 2.** (Требование 1) должно исполняться на каждом шаге наблюдаемого исполнения процессов счётчика и модели.

Под вторым требованием понимается, что в результате одинакового воздействия на счётчик и на модель на каждом шаге будет выполняться **Требование 1**.

Отношение реализации, заданное этими неформальными требованиями назовем $\leq_L$.

**Определение 18.** Отношение реализации $\leq_L \subseteq \mathfrak{L} \times \mathfrak{L}$ это бинарное отношение такое, что для процессов $p, q \in \mathfrak{L}$ выполняется $p \leq_L q$, если

$$\mathrm{I}(p) = \mathrm{I}(q) \tag{0.36}$$

$$(p \xrightarrow{a} p') \Rightarrow (\forall q' \in \mathfrak{L} : (q \xrightarrow{a} q') \Rightarrow (p' \leq_L q')) \tag{0.37}$$

где функция $I : \mathfrak{L} \to 2^{\Lambda}$ задает множество разрешенных действий процесса.

В работе [11] представлен спектр отношений эквивалентности и предпорядков для конечно-ветвящихся, непрерывных процессов. Частичный порядок «устанавливает строго больше отождествлений на процессах, чем» задает на отношениях эквивалентности решетку спектра (см. Рис. 1).

Счётчик порождает детерминированные процессы, а для детерминированных процессов спектр фон Глэббика сильно упрощается, что было показано в работе [14]. Но класс сетей Петри являются классом недетерминированных моделей. Отношение реализации должно учитывать возможность недетерминированного поведения модели, так как возможно, что сеть Петри, моделирующая бесконечный счётчик, будет необходимо являться недетерминированной.

Покажем, что отношение реализации является более точным, чем отношения эквивалентности завершенной симуляции (complete simulation) и завершенных трасс (complete trace).

Рассмотрим контрпример (Рис. 2), предложенный в работе [11] для установления неравенства между отношениями эквивалентности завершенной симуляции, завершенных трасс и симуляции готовности (ready simulation). Пусть действие $a$ моделирует операцию декремента, $b$ операцию инкремента, а $c$ операцию $0_?$. Тогда процесс $p$ на рисунке слева может представлять счётчик в состоянии, когда его значение равно 1, а процесс $q$ на рисунке справа может соответствовать сети Петри моделирующей такой счётчик.

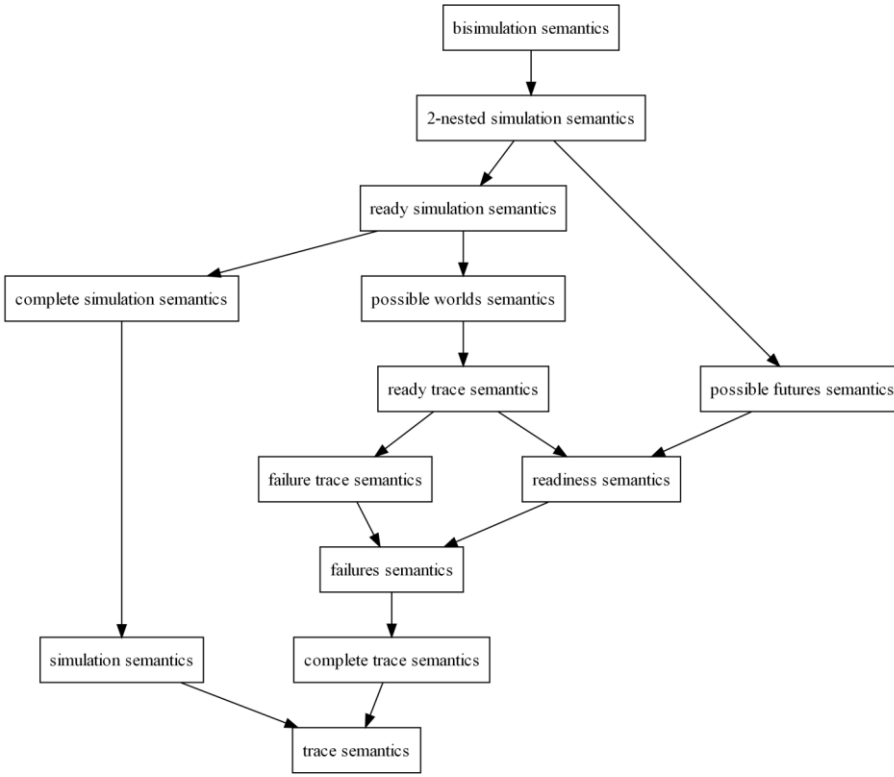



*Рис. 1 Спектр линейного-ветвящегося времени.*

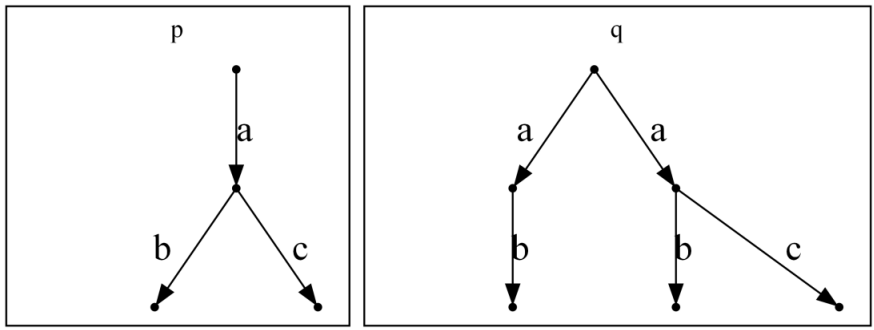



*Рис. 2 Процессы эквивалентные относительно отношений завершенных трасс и завершенной симуляции, и неэквивалентные относительно отношения симуляции готовности.*

Как было показано в работе [11], эти процессы эквивалентны относительно отношений завершенной симуляции и завершенных трасс. Но процесс $q$ допускает исполнение $\sigma = (a)$, после которого действие $c$ не разрешено. Тогда как процесс $p$ всегда разрешает действие $c$ после исполнения $\sigma = (a)$. Таким образом, нарушается условие (0.37). Из этого следует несоответствие отношения завершенной симуляции и завершенных трасс установленным требованиям к критерию корректности моделирования. Это распространяется на все менее точные отношения, которые лежат ниже эквивалентности завершенной симуляции и завершенных трасс на спектре (см. Рис. 1).

В то же время легко заметить, что определение отношения реализации включает в себя определения отношения симуляции (simulation) и отношение готовности (readiness). Отношение симуляции готовности (ready simulation) является наименьшей верхней гранью этих отношений, относительно порядка «строго больше отождествлений, чем».

**Определение 19.** Отношение эквивалентности симуляции готовности это бинарное отношение $=_{RS} \subseteq \mathfrak{L} \times \mathfrak{L}$ на системах помеченных переходов, которое ставит в соответствие процессы $p \in \mathfrak{L}$ и $q \in \mathfrak{L}$, если

$$I(p) = I(q) \tag{0.38}$$

$$\forall a \in Act : (p \xrightarrow{a} p') \Rightarrow (\exists q' \in \mathfrak{L} : q \xrightarrow{a} q' \,\&\, p' =_{RS} q') \tag{0.39}$$

$$\forall a \in Act : (q \xrightarrow{a} q') \Rightarrow (\exists p' \in \mathfrak{L} : p \xrightarrow{a} p' \,\&\, p' =_{RS} q') \tag{0.40}$$

Покажем, что для случая, когда один процесс является детерминированным, отношение реализации $\leq_L$ совпадает с отношением эквивалентности $=_{RS}$.

**Утверждение 5.** Пусть $p \in \mathfrak{L}$ детерминированный процесс, тогда $p \leq_L q \Leftrightarrow p =_{RS} q$.

$\Rightarrow$: Покажем, что отношение $\leq_L$ является отношением $=_{RS}$. Пусть $p \leq_L q$ и $p \xrightarrow{a} p'$, тогда

1). $\mathrm{I}(p) = \mathrm{I}(q)$

Следует из определения $\leq_L$.

2). $\forall a \in Act : (p \xrightarrow{a} p') \Rightarrow (\exists q' \in \mathfrak{L} : q \xrightarrow{a} q' \& \; p' \leq_L q')$

$\left(p \xrightarrow{a} p'\right) \Rightarrow a \in \mathrm{I}(p) \Rightarrow a \in \mathrm{I}(q) \Rightarrow \left(\exists q'' : q \xrightarrow{a} q''\right)$

$(p \leq_L q) \& \left(p \xrightarrow{a} p'\right) \& \left(q \xrightarrow{a} q''\right) \Rightarrow p' \leq_L q''$ по второму условию определения $\leq_L$.

3). $\forall a \in Act : (q \xrightarrow{a} q') \Rightarrow (\exists p' \in \mathfrak{L} : p \xrightarrow{a} p' \& \; p' \leq_L q')$

$\left(q \xrightarrow{a} q'\right) \Rightarrow a \in \mathrm{I}(q) \Rightarrow a \in \mathrm{I}(p) \Rightarrow \left(\exists p'' : p \xrightarrow{a} p''\right)$

$(p \leq_L q) \& \left(p \xrightarrow{a} p''\right) \& \left(q \xrightarrow{a} q'\right) \Rightarrow p'' \leq_L q'$ по второму условию определения $\leq_L$.

Таким образом, $\leq_L$ соответствует определению отношения $=_{RS}$.

$\Leftarrow$: Покажем, что отношение $=_{RS}$ является отношением $\leq_L$. Пусть $p =_{RS} q$, тогда

1). $\mathrm{I}(p) = \mathrm{I}(q)$

Следует из определения $=_{RS}$.

2). $\forall a \in Act : (p \xrightarrow{a} p') \rightarrow (\forall q' \in \mathfrak{L} : (q \xrightarrow{a} q') \rightarrow (p' =_{RS} q'))$

Пусть $\left(p \xrightarrow{a} p'\right)$, тогда

$\forall q' \in \mathfrak{L} : \left(q \xrightarrow{a} q'\right) \rightarrow (\exists p'' : p \xrightarrow{a} p'' \& \; p'' =_{RS} q')$ по третьему условию определения $=_{RS}$.

$\left(p \xrightarrow{a} p'\right) \& \left(p \xrightarrow{a} p''\right) \Rightarrow (p' = p'') \Rightarrow p' =_{RS} q'$ в силу детерминированности процесса $p$.

Таким образом, $=_{RS}$ соответствует определению отношения $\leq_L$. $\square$

Отношение эквивалентности симуляции готовности $=_{RS}$ удовлетворяет требованиям к отношению реализации формализованным в (**Определение 18**), и может быть принято в качестве критерия корректности моделирования счётчика сетью Петри.

## 5. Моделирование счётчиков с помощью сетей Петри

Рассмотрим несколько способов моделирования счётчиков в сетях Петри. Простейшим способом моделирования счётчика является позиция сети Петри. Если поставить в соответствие значение счётчика и значения разметки на позиции, то можно рассматривать данную позицию как примитивный счётчик. Его можно инкрементировать, увеличивая значение разметки на позиции, и аналогично декрементировать. Но в силу монотонности семантики сетей Петри [2] невозможно промоделировать с помощью одной позиции событие, которое срабатывает при заданном значении счётчика, но не срабатывает при большем значении счётчика. Таким образом, в приведенной модели, мы не можем промоделировать действие $0_?$, которое разрешено при нулевом значении счётчика, но запрещено при положительном значении счётчика.

### 5.1. Необходимое условие моделирования счётчика сетями Петри.

Рассмотрим тривиальный счётчик $C_2$ с множеством возможных значений $V = \{0,1\}$ и нулевым начальным состоянием.

$$C_2 = \langle \{0,1\}, 0 \rangle \tag{0.41}$$

Процессный граф, соответствующий системе помеченных переходов $L_C(C_2)$, представлен на (Рис. 3).

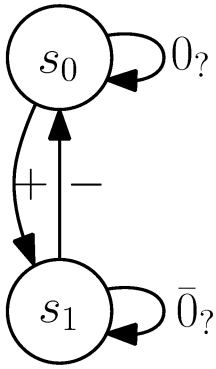



*Рис. 3 Процессный граф счётчика с двумя значениями.*

Поведение моделирующей сети Петри $N_2$ должно быть эквивалентно поведению моделируемого счётчика $C_2$ относительно $=_{RS}$. Состоянию $s_0$

должно соответствовать некоторое состояние $s_{\mu_0}$, такое что $s_0 =_{RS} s_{\mu_0}$. В силу определения отношения $=_{RS}$ $\mathrm{I}(s_0) = \mathrm{I}(s_{\mu_0})$. Следовательно, в сети $N_2$ при разметке $\mu_0$ должны быть активны переходы соответствующие операциям $\{+, 0_?\}$ и пассивны переходы соответствующие операциям $\{-, \overline{0_?}\}$. Т.е.

$$\forall t \in T_{N_2} : \left(\lambda(t) \in \{+, 0_?\}\right) \Rightarrow \left(^\bullet F(t) \le \mu_0\right) \qquad (0.42)$$

$$\forall t \in T_{N_2} : \left(\lambda(t) \in \{-, \overline{0_?}\}\right) \Rightarrow \left(^\bullet F(t) \not\le \mu_0\right) \qquad (0.43)$$

Так как $s_1$ достижимо из $s_0$ ($s_0 \xrightarrow{+} s_1$), то, в силу определения отношения $=_{RS}$ (0.39) должно существовать состояние $s_{\mu_1}$, такое что $s_1 =_{RS} s_{\mu_1}$. Т.е. в $N_2$ должна существовать разметка $\mu_1$, такая что

$$\forall t \in T_{N_2} : \left(\lambda(t) \in \{-, \overline{0_?}\}\right) \Rightarrow \left(^\bullet F(t) \le \mu_1\right) \qquad (0.44)$$

$$\forall t \in T_{N_2} : \left(\lambda(t) \in \{+, 0_?\}\right) \Rightarrow \left(^\bullet F(t) \not\le \mu_1\right) \qquad (0.45)$$

Из приведенных неравенств следует, что $\mu_1 \not\le \mu_0$ и $\mu_0 \not\le \mu_1$, т.е. разметки $\mu_0$ и $\mu_1$ несравнимы. Будем обозначать несравнимость разметок, как $\mu_0 \| \mu_1$. Таким образом, пространство состояний сети Петри $N_2$, моделирующей счётчик с двумя состояниями, будет содержать две несравнимые разметки.

Рассмотрим счётчик конечного объема $C_n$ с $n$ возможными состояниями (Рис. 4).

**Утверждение 6.** Множество достижимости сети Петри $N_n$, моделирующей счётчик $C_n$ с $n$ возможными различимыми состояниями, необходимо содержит $n$ несравнимых разметок.

Пусть сеть Петри $N_n$ корректно моделирует счётчик $C_n$ с $n$ различными состояниями $N_n =_{RS} C_n$ и ее множество достижимых разметок не содержит $n$ несравнимых разметок.

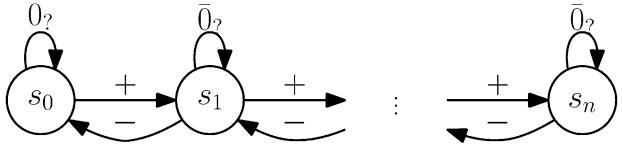



*Рис. 4. Процессный граф счётчика с $n$ значениями.*

Рассмотрим исполнение счётчика $\sigma = +^{n-1} -^{n-1}$, т.е. сначала $n-1$ раз осуществляется операция инкремента, затем $n-1$ раз осуществляется операция декремента. Счётчик проходит все свои возможные состояния, при этом внутреннее значение счётчика на каждом шаге равно

$$\forall i \in \overline{0,|\sigma|} : \nu(C_n) = \rho(\sigma_i, +) - \rho(\sigma_i, -) \qquad (0.46)$$

где $\sigma_i$ исполнение на $i$ шагу. Следовательно, счётчик принимает нулевые значения на 0 и последнем шаге. Это означает, что только на 0 и последнем шаге предикат $0_?$ истинен, а в системе помеченных переходов $L_C(C_n)$ действие $0_?$ разрешено, а $0_?$ запрещено. Для промежуточных шагов предикат $0_?$ ложен, действие $0_?$ запрещено и $\overline{0_?}$ разрешено.

Т.к. в силу критерия корректности моделирования $L_C(C_n) =_{RS} L_N(N_n)$, то на каждом шаге исполнения множество разрешенных и запрещенных действий процессов систем $L_C(C_n)$ и $L_N(N_n)$ должны совпадать.

Рассмотрим первый отрезок исполнения $\sigma_{n-1}$. Пусть нулевому состоянию $s_0$ системы $L_C(C_n)$ соответствует нулевое состояние $s_{\mu_0}$ системы $L_N(N_n)$, т.е. нулевое состояние счётчика моделируется разметкой $\mu_0$. Отрезок исполнения $\sigma_{n-1}$ задает $n$ состояний $s_0 \ldots s_n$ системы $L_C(C_n)$. Состояниям

$s_0 \ldots s_n$ соответствуют состояния $s_{\mu_0} \ldots s_{\mu_n}$ системы $L_N\left(N_n\right)$, которые в свою очередь имеют прообразами разметки сети Петри $\mu_0 \ldots \mu_n$. В силу допущения о том, что множество достижимых разметок $N_n$ не содержит $n$ несравнимых разметок, имеются две разметки $\mu_1$ и $\mu_2$, такие, что либо $\mu_1 \le \mu_2$, либо $\mu_1 \ge \mu_2$. Рассмотрим оба случая.

Пусть $\mu_1 \xrightarrow{+^l} \mu_2$ и $\mu_1 \le \mu_2$. На (Рис. 5, а) схематично изображено исполнение сети Петри с такой парой. Допустим $\mu_0 \xrightarrow{+^k} \mu_1$. Тогда $\mu_1 \xrightarrow{-^k} \mu_0'$ и $^\bullet F\left(t_{0_?}\right) \le \mu_0'$. Но, в силу монотонности исполнения сетей Петри, $\mu_1 \le \mu_2 \Rightarrow \left(\mu_2 \xrightarrow{-^k} \mu_3\right) \& \left(\mu_0' \le \mu_3\right)$, т.е. $^\bullet F\left(t_{0_?}\right) \le \mu_3$. Так как исполнение $\mu_0 \xrightarrow{\delta} \mu_3$, содержит $l+k$ инкрементов и $k$ декрементов, то в силу (0.46) значение счётчика в этом состоянии будет равняться $\nu\left(L_C^{-1}\left(s_3\right)\right) = l$, и следовательно действие $\overline{0_?}$ будет разрешено, а $^\bullet F\left(t_{\overline{0_?}}\right) \le \mu_3$. Таким образом, в состоянии $s_{\mu_3}$ будут разрешены одновременно действия $0_?$ и $\overline{0_?}$, однако в состоянии $s_0 \xrightarrow{\delta} s_3$ действие $0_?$ будет запрещено. Т.е. $0_? \in \mathrm{I}\left(\delta\left(s_{\mu_0}\right)\right) \setminus \mathrm{I}\left(\delta\left(s_0\right)\right)$ и следовательно $s_0 \not\simeq_{RS} s_{\mu_0}$.

Теперь допустим, что $\mu_1 \xrightarrow{+^l} \mu_2$ и $\mu_1 \ge \mu_2$. На (Рис. 5, б) схематично изображено исполнение сети Петри с такой парой. Допустим $\mu_0 \xrightarrow{+^k} \mu_1$. Тогда $\mu_2 \xrightarrow{-^k} \mu_3$ и $^\bullet F\left(t_{\overline{0_?}}\right) \le \mu_3$. Но, в силу монотонности исполнения сетей Петри, $\mu_1 \ge \mu_2 \Rightarrow \left(\mu_1 \xrightarrow{-^k} \mu_0'\right) \& \left(\mu_0' \ge \mu_3\right)$, т.е. $^\bullet F\left(t_{\overline{0_?}}\right) \le \mu_0'$. Так как исполнение $\mu_0 \xrightarrow{\delta} \mu_0'$, содержит $k$ инкрементов и $k$ декрементов, то в силу (0.46) значение счётчика в этом состоянии будет равняться $\nu\left(L_C^{-1}\left(s_0'\right)\right) = 0$, и следовательно действие $0_?$ будет разрешено, а $^\bullet F\left(t_{0_?}\right) \le \mu_0'$. Таким образом, в состоянии $s_{\mu_0'}$ будут разрешены

одновременно действия $0_?$ и $\overline{0_?}$, однако в состоянии $s_0 \xrightarrow{\delta} s_0'$ действие $\overline{0_?}$ будет запрещено. Имеем $\overline{0_?} \in \mathrm{I}\left(\delta\left(s_0\right)\right) \setminus \mathrm{I}\left(\delta\left(s_{\mu_0}\right)\right)$ и следовательно $s_0 \not\approx_{RS} s_{\mu_0}$. □

Утверждение 7. Не существует сети Петри $N_{\aleph}$, моделирующей счётчик $C_{\aleph}$ со счетным множеством возможных значений.

В силу (**Утверждение 6**), множество достижимости сети Петри, моделирующей счетчик, должно содержать множество несравнимых разметок, мощности большей или равной мощности возможных значений моделируемого дискретного счётчика.

Если множество возможных значений счётчика счётное, тогда множество достижимости моделирующей его сети Петри $N_{\aleph}$ должно содержать счётное множество несравнимых разметок.

Множество несравнимых разметок является антицепью относительно порядка на разметках сети Петри. Но порядок на разметках сети Петри является правильным в силу (**Утверждение 3**).

По (**Утверждение 1**), множество-носитель правильного порядка не может содержать бесконечной антицепи. Таким образом, множество достижимости любой сети Петри и, в частности $N_{\aleph}$, не может содержать счётного множества несравнимых разметок. □

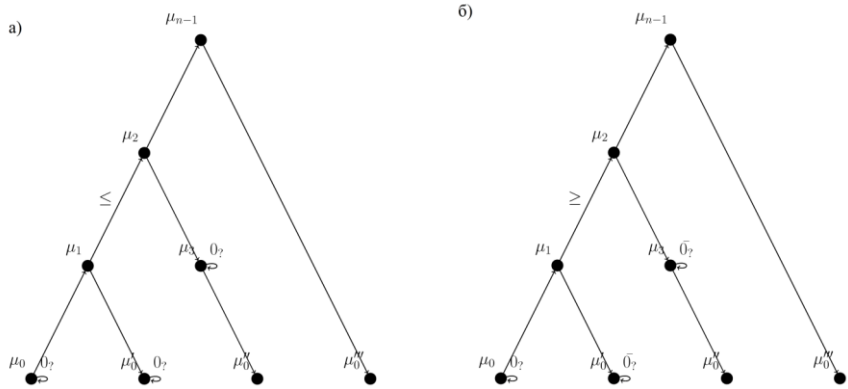



*Рис. 5 Схема исполнения модели счётчика с а) возрастающей и б)убывающей парой состояний.*

## 5.2. Модель счётчика конечного объема.

Покажем, что существует сеть Петри с $n$ несравнимыми разметками во множестве достижимых состояний, которая моделирует счётчик с $n$ возможными состояниями.

Это хорошо известная конструкция, которая часто явно и неявно применяется при моделировании процессов с помощью сетей Петри.

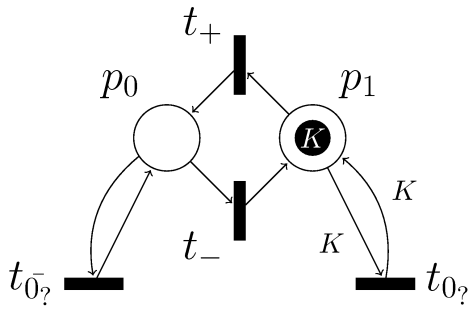



*Рис. 6 Модель счётчика конечного объема.*

Рассмотрим сеть Петри $N_F$ изображенную на (Рис. 6). Множество позиций состоит из двух элементов $P_F = \{p_0, p_1\}$. Четыре перехода $T = \left\{t_+, t_-, t_{0_?}, t_{0_?}^-\right\}$ моделируют операции счётчика. Начальная разметка $\mu_{F0} = \langle 0, K \rangle$ определяет $K+1$ возможных значений счётчика. Множество достижимых состояний такой сети Петри состоит из $K+1$ разметок $R(N_F) = \left\{\mu_{Fi} : i \in \overline{0,K} \,\&\, \mu_i = \langle i, K-i \rangle\right\}$, причем легко заметить, что все разметки несравнимы друг с другом, т.е. имеется $K+1$ несравнимых разметок. Переход $t_{0_?}$ будет активен только в нулевой разметке $\mu_{F0} = \langle 0, K \rangle$, т.к. значение любой разметки с индексом не равным нулю, для позиции $p_1$ равно $K-i$. Переход $t_{0_?}^-$ не активен при нулевой разметке, т.к. значение разметки на $p_0$ равно $0$, и активен при любой другой разметке $\mu_i$ имеющей на $p_0$ значение $i$.

## 6. Заключение

В данной статье рассмотрены проблемы моделирования счётчиков с помощью сетей Петри. В рамках работы осуществлен выбор критерия корректности моделирования счётчика сетями Петри. Доказано, что для моделирования счётчика с $n$ значениями необходимо $n$ несравнимых разметок сети Петри. Так же показано, что невозможно выразить счётчик с бесконечным счётным числом значений традиционной сетью Петри.

Причиной ограниченности выразительной мощности сетей Петри и подобных сетевых и потоковых моделей является монотонность входной функции по отношению к правильному порядку на разметках сети. Первый способ моделирования - оценка числа несравнимых состояний в пространстве состояний моделируемой системы. Если мощность такого множества конечна, то система выразима на языках сетей Петри и эквивалентна по мощности разрешимости сетям Петри. Если множество различимых состояний не конечно, то необходимо использовать конечную аппроксимацию счётчиков с помощью предложенной модели, либо использовать более мощные модели. Однако в последнем случае необходимо учитывать, что теряется мощность разрешимости модели.

**Литература**


[1] Decidability questions for bisimilarity of Petri nets and some related problems. **Jancar, Petr.** 1994. Proc. of STACS'94. Т. 775, стр. 581-592.

[2] **Ломазова И.А.** Вложенные сети Петри: моделирование и анализ распределенных систем с объектной структурой. Москва : Научный мир, 2004.

[3] **Кузьмин Е. В., Соколов В. А.,.** Структурированные системы переходов. Москва : ФИЗМАТЛИТ, 2006.

[4] **Минский М.** Вычисления и автоматы. Москва : Мир, 1971.

[5] **Бурдонов И.Б., Косачев А.С., Кулямин В.В.** Теория соответствия для систем с блокировками и разрушениями. Москва : ФИЗМАТЛИТ, 2008.

[6] **Котов, В. Е.** Сети Петри. Москва : Наука, 1984. стр. 160 стр.

[7] **Питерсон Дж.** Теория сетей Петри и моделирование систем. Москва : Мир, 1984.

[8] Тестирование конформности на основе соответствия состояний. **Бурдонов, И. Б. и Косачев, А. С.** Москва : ИСП РАН, 2010 г., Труды Института системного программирования РАН, Т. 18. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print).

[9] **Roscoe, A. W.** A mathematical theory of communicating processes. 1982.

[10] Collaborative Software Engineering Using Metamodel-Driven Approach. **Semenov, V.** 16th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2007), Paris : IEEE, 2007 г., стр. 178-179.

[11] The Linear Time - Branching Time Spectrum I. **Glabbeek, R. J.** Berlin : Springer, 1990. Lecture Notes in Computer Science. Т. 458.

[12] Полное тестирование с открытым состоянием ограниченно недетерминированных систем. **Бурдонов, И. Б. и Косачев, А. С.** Москва : б.н., 2009 г., Труды Института системного программирования РАН, Т. 17, стр. 193-208. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print).

[13] Contrasting themes in the semantics of imperative concurrency. **J.W. de Bakker, J.N. Kok, J.-J.Ch. Meyer, E.-R. Olderog, J.I.Zucker.** New York : Springer, 1986. Current Trends in Concurrency, LNCS. T. 224, стр. 51-121.
[14] Concurrency and automata on infinite sequences. **Park, D.M.R.** [ред.] P. Deussen. London : Springer, 1981. 5th GI Conference, LNCS 104. T. 104, стр. 167-183.

## "On the modeling of infinite counters by ordinary Petri nets"

*L. V. Dvoryansky,*
*leo@ispras.ru*

**Annotation.** This paper provides the analysis of the express power of Petri nets to model counters with infinite state space. The implementation relation for the counters modeling is suggested. A lack of Petri nets expression power for such modeling is shown. The minimal Petri net model of a counter with a finite state space is provided.

**Keywords:** counter machines; Petri nets; Glabbeek spectrum; ready simulation preorder.

# Кросс-система программирования ЯУЗА-6 для специализированных ЭВМ реального времени (в 70 – 80-е годы прошлого века)

*В.В. Липаев*
*lip@ispras.ru*

**Аннотация.** Рассмотрены особенности и проблемы эффективного создания в 70-е годы сложных комплексов программ оборонного применения, базирующихся на специализированных ЭВМ. Представлены методы, требования и реализация адаптируемых кросс-систем автоматизации программирования и тестирования для различных типов сложных комплексов программ объектных ЭВМ. Изложены результаты многолетнего применения в ряде предприятий кросс-системы автоматизации программирования и отладки ЯУЗА-6 с использованием БЭСМ-6.

**Ключевые слова.** Кросс-система программирования; комплексы программ; автоматизация программирования и отладки программ; специализированные ЭВМ; трансляторы и интерпретаторы программ; автокоды и макроязыки программирования.

## *1. Введение*

В 70-е годы программная инженерия оказалась наиболее востребованной для решения крупных задач в оборонной технике и для государственных административных систем [1]. Массовое прикладное программирование в академических организациях, промышленных институтах и в вузах в это время оставалось на уровне индивидуального создания небольших программ и обычно не требовало применения методов программной инженерии. Опыт предприятий оборонной промышленности, накопленный в 60-е годы, позволил обобщить особенности и проблемы развития программной инженерии для эффективного создания наиболее сложных комплексов программ оборонного применения, базирующихся на специализированных ЭВМ.

Быстрый рост сложности функциональных задач и требований к ресурсам ЭВМ для их решения в ***60-е годы***, не могли быть удовлетворены при доступной в то время технической и элементной базе вычислительных машин. Эту проблему разработчики систем стремились решать путем адаптации архитектуры ЭВМ к особенностям и характеристикам сложных

функциональных задач. Поэтому в ряде предприятий оборонной промышленности еще в конце ***50-х годов*** начали разрабатываться многочисленные специализированные ЭВМ. При подготовке требований к таким объектным ЭВМ военного назначения для эффективного использования их ограниченных вычислительных ресурсов, необходим был детальный анализ алгоритмов и программ, предназначенных для функционирования в реальном времени.

Относительно узкая ориентировка каждого типа ЭВМ на совершенно определенные функциональные задачи, давала возможность значительной экономии оборудования и улучшения использования характеристик ЭВМ по памяти и производительности ***на имевшейся элементной базе***. С другой стороны, в то время от программистов требовалась максимальная эффективность использования доступных ресурсов и знаний тонкостей архитектуры специализированных ЭВМ при реализации алгоритмов, что, в частности, определило широкое применение машинно-ориентированных языков программирования – автокодов.

Ресурсы объектных ЭВМ позволяли размещать программы только функциональных задач оборонных систем, а для автоматизации их программирования и отладки приходилось использовать ***стационарные универсальные ЭВМ с большими ресурсами***. Принципиально новые возможности открылись в начале ***70-х годов*** в связи с появлением в НИИ-5 (МНИИПА) ЭВМ БЭСМ-6. Началось создание ***системы автоматизации разработки и отладки программного обеспечения (САРПО)*** ЯУЗА-6 (под руководством В.В. Липаева, Л.А. Серебровского) [1, 2]. Эти работы были активно поддержаны со стороны руководства института и министерства. Министерством радиопромышленности СССР было открыто достаточное финансирование работ и значительно увеличен коллектив специалистов.

Были сформулированы следующие ***основные концептуальные особенности САРПО ЯУЗА-6***:

- возможность автоматизированной адаптации системы на структуру и архитектуру команд различных специализированных ЭВМ;
- применение трех входных языков программирования для специализированных ЭВМ – автокода, макроязыка и алгоритмического языка;
- обеспечение высокого качества программ по использованию памяти и производительности ЭВМ, транслированных с алгоритмического языка (коэффициент расширения в пределах 1,1 – 1,2);
- автоматизированную стыковку программных компонентов по глобальным переменным и по передачам управления;
- автоматизированный контроль структурного построения и

использования памяти программными компонентами;

- автоматизированное тестирование и отладка программ с использованием стационарной универсальной ЭВМ на уровне входного языка программирования с интерпретацией команд специализированной ЭВМ;

- автоматизированный выпуск документации на весь комплекс программ и на отдельные компоненты в соответствии с ГОСТами и пригодной для ввода программ в специализированную ЭВМ.

Перечисленные особенности отличали САРПО ЯУЗА-6 от обычных трансляторов как ***комплексную систему программной инженерии***, предназначенную для автоматизации основных этапов разработки крупных комплексов программ, при условии высокой эффективности создаваемых программ по использованию ресурсов специализированной ЭВМ. Комплексная автоматизация должна была в несколько раз повысить производительность труда специалистов при разработке больших управляющих программ (размером порядка 100 тыс. команд). При этом должны были быть исследованы, и учитываться принципиальные особенности специализированных ЭВМ и комплексов программ оборонных систем того времени [1, 2].

Значительное расширение функций автоматизации разработки программ на мощных технологических ЭВМ (БЭСМ-6), а также быстрый рост числа различных структур специализированных ЭВМ, для которых была необходима автоматизация программирования, приводит к сокращению изменяемой части инструментальных систем разработки программ. В результате ***экономически целесообразно*** выделять в инструментальных системах программирования ***машинно-ориентированную на специализированные ЭВМ часть программ***, и отдельно автоматизировать их разработку. Примером может служить система ЯУЗА-6, в которой из 400 тысяч команд, для адаптации на специализированные ЭВМ в ней оказалось необходимо изменение около ***3 – 5% команд***. Для автоматизации такой настройки целесообразно было разработать дополнительную подсистему размером около 20 тысяч команд и соответствующие инструкции по её применению.

***Основное достоинство разработанного и исследованного оригинального метода адаптируемых кросс-систем*** состояло в том, что эти системы могли быть ориентированы практически на любые типы специализированных ЭВМ, поддерживая при этом эффективность результатов программирования на достаточном уровне. Процесс адаптации кросс-систем был формализован и автоматизирован. При разработке адаптируемой кросс-системы в её машино-зависимые технологические программы были введены проблемно-ориентированные средства, предназначенные для обеспечения автоматизации адаптации. Адаптируемость кросс-системы достигалась совокупным

применением принципов параметризации, конфигурирования и функционального расширения. Методы подготовки адаптируемой кросс-системы и условия её применения, к которым она должна была адаптироваться, определялись классом специализированных ЭВМ, базовым автокодом и характеристиками программного продукта оборонной системы.

Для формирования подсистемы адаптации было ***исследовано множество характеристик и параметров различных специализированных ЭВМ***. При этом был использован не только статистический материал, накопленный при рассмотрении специализированных ЭВМ, но и априорные ***обобщенные характеристики широкого класса объектных вычислительных машин***. В пределах этого класса определялись типы данных, с которыми работают специализированные ЭВМ, специфика систем команд, их форматы и времена исполнения; состав памяти данных и команд; системы адресации и т. д. Данные характеризовались типом, формой представления, способом кодирования и точностью. В форматах команд учитывались длина, топология и значение полей форматов. Память данных и команд характеризовалась составом, емкостью и способом расположения данных и команд. В классе специализированных ЭВМ фиксировались также способы вычисления контрольных разрядов команды и виды кодограмм, которые определяют формат упаковки готовых программ в канале телекоммуникации, соединяющем технологическую ЭВМ со стендовым оборудованием и специализированной ЭВМ.

## 2. Требования к системе автоматизации программирования и отладки

К программным комплексам специализированных ЭВМ реального времени ***в 70-е годы*** предъявлялись особенно жесткие требования к использованию их вычислительных ресурсов, что вызывало необходимость построения и применения специфических методов программирования и соответствующих языков для систем реального времени. Для сложных комплексов программ было характерно разнообразие алгоритмов решения функциональных и вспомогательных задач. При этих условиях трудно было удовлетворить требования высокой эффективности получаемых программ и одновременно высокой производительности труда специалистов, если проводить программирование на одном языке. В то же время для обеспечения преемственности алгоритмов, решающих одинаковые задачи в различных системах, было целесообразно унифицировать их описания и структуру модулей программ и данных. Проведенные ***исследования показали, что перспективным является путь автоматизации программирования на базе трех взаимно связанных входных языков***, различающихся глубиной машинной и проблемной ориентации, применяемых в зависимости от типов задач в комплексе программ. По своей ориентации языки программирования ***полезно было разделить*** на:

- ***автокоды***, предназначенные для записи программ с полным учетом структуры и системы команд конкретной специализированной ЭВМ;

- ***макроязыки***, предназначенные для записи программ с использованием унифицированных операторов, состав и содержание которых не полностью учитывают структуру каждой специализированной ЭВМ и особенности ее системы команд;

- ***алгоритмические языки***, запись программ, на которых почти полностью унифицирована и производится с учетом лишь структуры памяти специализированной ЭВМ или класса ЭВМ, либо вообще без учета особенностей конкретной ЭВМ, предназначенной для работы по этой программе.

Для специализированных ЭВМ указанные языки должны были образовывать взаимосвязанную ***систему языков программирования***, в которой обеспечивался преемственный ***вертикальный переход*** от нижних уровней на более высокий уровень путем ввода новых конструкций в язык и, наоборот, исключением конструкций, не соответствующих задачам уровня. Разработчик функциональной программы или модуля специализированной ЭВМ мог выбирать уровень языка в зависимости от требований к эффективности использования ресурсов памяти и производительности ЭВМ. Удовлетворение этих требований зависело от характеристик трансляторов с того или иного языка. Было исследовано расширение текста транслированных программ при программировании на языках высокого уровня. Для специализированных ЭВМ по отношению к программе, записанной на автокоде, трансляторы ***с макроязыка давали расширения не более 10%, а с алгоритмического языка (модификация и русификация Алгола-60) около 20 – 30%***.

Для обобщенной записи системы команд специализированных ЭВМ, определенных в классе этих машин был разработан ***базовый автокод***. От традиционных автокодов он отличался унифицированным построением синтаксиса и лексики на базе русского языка. Универсальность обобщенного формата в рамках класса специализированных ЭВМ обеспечивала инвариантность синтаксиса базового автокода к изменению этих машин. Лексика базового автокода по составу терминов разделялась на две части: постоянную и переменную. Переменная часть лексики формировалась в процессе идентификации кодов операций, устройств, регистров и других программно-доступных ресурсов специализированной ЭВМ.

***Описание базового автокода*** представляло собой унифицированный документ, который использовался при программировании любой специализированной ЭВМ. В описание базового автокода введен специальный раздел, содержащий правила формирования представления автокода конкретной специализированной ЭВМ, которые регламентировал действия по обозначению машинно-зависимых элементов автокода. Все ***функции транслятора с автокода***, кроме генерации машинных команд, следовало

адаптировать к явным значениям параметров специализированной ЭВМ, автокода и характеристикам проекта программ оборонной системы. Программы, реализующие перечисленные функции, следовало проектировать по методу параметризации.

В большинстве проектов ***макрогенератор*** не зависел от параметров условий применения. Однако в специальных проектах в него могли быть встроены машинно-зависимые функции распределения памяти и масштабирования. В этом случае подготовка макрогенератора осуществлялась аналогично подготовке компилятора алгоритмического языка.

***Компилятор алгоритмического языка*** необходимо было настраивать на структуру памяти и форматы специализированной ЭВМ, а также на правила программирования операторов алгоритмического языка, учитывающие специфику системы команд специализированной ЭВМ. Сложность семантики не позволяла применять для проектирования компилятора принцип параметризации. Поэтому при подготовке компилятора алгоритмического языка, допускалось частичное перепрограммирование семантик. Кроме формирования последовательности команд, при подготовке компилятора алгоритмического языка решались дополнительные задачи, связанные с распределением регистров, масштабированием, контролем точности, выполняемыми машинно-зависимыми процедурами.

***Характеристики проекта комплекса программ*** служили для описания функционального распределения ресурсов специализированной ЭВМ под различные компоненты комплекса программ. В этом описании должны были закрепляться соглашения о связях, т. е. указываться номера регистров, которые отводятся для хранения адресов передачи управления от вызывающей программы к вызываемой и адресов возврата; адресов начала зон переменных; для распределения областей памяти под глобальные и локальные переменные.

***Адаптируемость интерпретатора*** для отладки программ специализированной ЭВМ ***с исполнением на универсальной ЭВМ***, при выбранных показателях качества по производительности, обеспечивалась совместным применением трех принципов проектирования адаптируемых программ. Для параметризованной компоненты интерпретатора в базу данных для настройки включались информационные модули, содержащие всю информацию о структуре специализированной ЭВМ, а именно: таблицы регистров и триггеров; аппаратных прерываний; описания системы команд, т.е. кодов операций, типов операндов, времени исполнения команд, слова состояния процессора. Конфигурируемую компоненту интерпретатора, которую составляют процедуры, моделирующие арифметические, логические и управляющие команды, можно было записывать на макроязыке. В результате проведенных исследований были сформулированы основные ***требования к системе автоматизации программирования и отладки***,

которые практически полностью были реализованы в середине ***70-х годов*** в системе ЯУЗА-6, построенной на технологической ЭВМ БЭСМ-6.

Требования к САРПО ЯУЗА-6 для специализированных ЭВМ зависели от особенностей этих машин, а также от функциональных задач ряда оборонных систем и ***состояли в следующем*** [1, 2].

***Единство системы автоматизации программирования и отладки*** комплексов программ должно было обеспечивать разработку высококачественных программных продуктов при минимальных затратах совокупного труда специалистов на их создание. Система должна была иметь средства общения человека со средствами автоматизации на языках задания и программирования, построенных по единым правилам. Вследствие ***ограниченных ресурсов специализированных ЭВМ***, технологические задачи автоматизации программирования и отладки программ следовало реализовать на стационарной, универсальной ЭВМ, сохраняя на специализированных машинах только минимум обслуживающих программных средств, необходимых для комплексной отладки в реальном времени.

***Высокая эффективность получаемых программ*** по использованию памяти и производительности мобильных ЭВМ определялась стабильностью решаемых задач, перечень и объем которых обычно устанавливался на ранних стадиях разработки оборонной системы управления. Экономичность программ должна была обеспечиваться не только по количеству машинных команд, но и по объему оперативной памяти, используемой для хранения переменных и констант, что приводило к необходимости работы с величинами, занимающими часть слова, с масштабированными переменными и к плотной упаковке величин в одной ячейке памяти.

***Многоязыковый вход в систему автоматизации*** представлял перспективный путь эффективного использования ресурсов мобильных ЭВМ при программировании для управляющих систем на базе нескольких взаимосвязанных входных языков, различающихся глубиной машинной и проблемной ориентировки, применяемых в зависимости от типов описываемых задач. По степени машинной ориентации весь ***комплекс программ управляющей системы*** целесообразно было делить на следующие ***три группы***:

- программы обмена информацией и машинные тесты;
- программы организации вычислительного процесса, управления задачами и данными (операционные системы реального времени);
- программы решения функциональных задач конкретной оборонной системы.

Наибольшей проблемной и наименьшей машинной ориентировкой обладают функциональные задачи оборонной управляющей системы. Программы этих задач имеют обычно наибольший удельный вес в комплексе программ

системы, составляя 70 – 80% общего размера программ. Их разработку целесообразно было производить на проблемно-ориентированном алгоритмическом языке высокого уровня (в данном случае на модификации языка Алгол-60).

Программы организации вычислительного процесса, управления задачами обмена информацией и тесты весьма сильно связаны с архитектурой вычислительной машины и с конкретной задачей управления вычислениями в разрабатываемой системе. Эти обстоятельства снижали эффективность использования алгоритмического языка высокого уровня и его применяемость для записи алгоритмов в различных системах. Для разработки этих программ целесообразно было иметь машинно-ориентированный язык типа автокода с макрокомандами.

***Адаптация системы автоматизации программирования и отладки*** на логику систем команд специализированных ЭВМ должна была быть унифицирована и сделана независимой от типа и системы команд ЭВМ. Вся специфика, обусловленная системой команд конкретной ЭВМ, была вынесена в сменные или настраиваемые программные и информационные модули. Кроме состава, структуры и логики исполнения команд объектной ЭВМ, реализация заменяемых элементов зависит от методов адресации, объемов и структуры памяти различных типов, состава и типов внешних устройств, характеристик и способов обмена с внешними устройствами и абонентами. Настройка должна была осуществляться автоматизировано по формальному описанию логики команд и структуры мобильной ЭВМ, представленному на специальном языке.

Сменная (машинно-ориентированная) часть системы автоматизации программирования и отладки должна была обеспечивать, с одной стороны, машинную ориентацию процесса трансляции с каждого из входных языков на язык системы команд конкретной специализированной ЭВМ, а с другой – позволять исполнять программы этой ЭВМ на универсальной машине при отладке программ по тестам. При этом входные языки программирования, методы и правила структурного построения управляющих программ, язык отладки, состав и правила оформления документации на алгоритмы и программы должны быть едиными для всех систем управления. Единая технологическая база автоматизации должна в значительной степени обеспечивать инвариантность алгоритмических и программных разработок относительно изменения управляющих, специализированных вычислительных машин.

Должно было ***обеспечиваться автоматизированное сопряжение*** отдельных модулей и компонентов в большие комплексы управляющих программ. Для управляющих программ характерно мультипрограммное функционирование комплексов взаимодействующих функциональных компонентов объемом в десятки и сотни тысяч команд, использующих общую память глобальных переменных. ***Модульное, иерархическое построение*** должно было позволять

получать обозримое описание решаемых задач и их взаимосвязей в сложном комплексе программ, а также существенно облегчать автоматизацию всех этапов процесса разработки и в первую очередь контроля компонентов и их сопряжения в единый комплекс. Достаточно эффективным средством обеспечения сопряжения подпрограмм являлось использование ***единого описания глобальных переменных***, с применением которого транслируются или загружаются все функциональные компоненты программы системы.

Должен осуществляться ***автоматизированный расчет длительности исполнения*** функциональных компонентов и всего комплекса программ в реальном времени. Функционирование управляющих систем в реальном времени связано с возможностью реализовать исполнение комплекса всех необходимых программ в заданные интервалы времени. На конечных этапах разработки, когда функционирует вся управляющая система или основные функциональные задачи, длительность исполнения программ может быть получена непосредственным измерением времени исполнения каждого компонента программы и всего комплекса в процессе решения задач в реальном времени.

На всех этапах разработки, значения длительности реализации компонентов и всего комплекса программ, могут быть получены аналитически, последовательным расчетом по блок-схемам алгоритмов или по готовым программам. Длительность реализации каждого линейного участка подпрограммы следовало рассчитывать непосредственно по операциям в программе или по операторам в блок-схеме. Вероятности условных переходов и количества прохождений циклов разработчик мог задать на основе представлений о функционировании программы в средних и в предельных условиях. При таких исходных данных аналитически должна была рассчитываться длительность реализации каждой программы.

Должен был производиться ***автоматизированный контроль корректности*** текстов, структуры компонентов программ и сопряжения всего комплекса программ. Во всех задачах контроля следовало иметь эталон – систему правил, которым должна соответствовать программа в целом и каждая ее часть. Несоответствие эталонным правилам надо регистрировать одной из систем контроля с указанием места нарушения правил (ошибки). Задача синтаксического контроля – проверка входного текста программы на соответствие формальному описанию синтаксиса языка программирования. Задачей семантического контроля должен быть анализ текстов программ на правильность применения конструкций входного языка для записи алгоритмов и соответствия правилам взаимосвязи элементов конструкций, главным образом между операторами и описаниями глобальных данных.

***Структурный контроль построения компонентов*** должен был выявлять некоторые виды зацикливания, наличие тупиковых и лишних участков в алгоритмах и другие нарушения правил построения структуры компонентов программ. Структурный контроль взаимодействия функциональных

подпрограмм в комплексе программ в принципе подобен контролю компонентов, однако охватывает значительно больший объем программ, и каждая подпрограмма при этом участвует как единое целое. При этом должны контролироваться сопряжение подпрограмм по локальной информации, по управлению, и использованию глобальных переменных в динамике их записи и чтения.

***Автоматизированная отладка автономных модулей и подпрограмм*** на уровне входного языка предназначена для локализации ошибок в системе, путем контроля процесса вычислений при заданном тесте с различной степенью подробности. Они должны были обеспечивать:

- обработку задания на отладку, представленного на языке, близком по уровню к языку программирования;
- трансляцию значений переменных и констант в составе теста в величины и масштабы данных, используемых в специализированной ЭВМ;
- исполнение задания на отладку при заданных исходных тестах;
- информирование разработчика о результатах исполнения отладочного задания.

Значительная часть отладки при программировании на алгоритмических языках, должна была предварительно проводиться с использованием метода компиляции, т. е. по программам, представленным в командах технологической ЭВМ. Это позволяло существенно повысить скорость исполнения программ и снизить общую трудоемкость отладки. Исполнение отладочных заданий на заключительных стадиях отладки должно было производиться ***методом интерпретации*** на уровне команд специализированной ЭВМ. Введение различных уровней детализации в системе отладки логики исполнения программ и изменения переменных должно было позволять селектировать необходимую для отладки информацию, в процессе исполнения задания и избавлять разработчика программы от избыточной информации. Тем самым каждая операция по локализации ошибки должна была содержать всю необходимую информацию.

***Комплексная отладка систем управляющих программ реального времени*** должна была учитывать, что существует противоречие между стремлением обеспечить функционирование комплекса управляющих программ в условиях, максимально близких к реальным, и ограниченными возможностями управляющих ЭВМ для размещения и исполнения технологических программ, обслуживающих отладку. Кроме того, специализированные ЭВМ весьма ограничены по составу внешних устройств, необходимых для общения человека с машиной в процессе отладки комплекса программ в реальном времени.

Для обеспечения решения задач управления в реальном времени при одновременном решении задач: имитации информации внешней среды, накопления и статистической обработки результатов, информирования операторов о ходе процесса исполнения программ и других функций, обслуживающих отладку, следует использовать технологическую ЭВМ. В этом случае на управляющую ЭВМ дополнительно к ее основным функциональным задачам должно возлагаться исполнение операторов отладочного задания и выдача в технологическую ЭВМ результатов их реализации, которые осуществляются при весьма малых затратах памяти и производительности управляющей ЭВМ. Кроме того, это способствует единству методов и языковых средств в течение всего процесса разработки управляющих программ и использования средств автоматизации разработки программ без применения трудоемких ручных методов при переходе от одного этапа к другому.

***Автоматизированный выпуск и корректировка технической документации*** на комплекс программ необходимы, для того чтобы успешно разработать, отладить и ввести в эксплуатацию сложный программный продукт. Документация на комплексы управляющих программ должна была строиться по иерархическому принципу и состоять из нескольких уровней детализации. Такая структура должна позволять в удобной и наглядной форме проводить анализ программ как от общего к частному, так и от частного к общему. Наибольший объем документации соответствует представлению комплекса программ в детальном виде на уровне текстов модулей и компонентов программ на разных языках и описаний переменных и констант. Значительный объем документов представляют также блок-схемы с расшифровкой операторов, блоков и подпрограмм. Вся необходимая исходная информация для этой части документов должна быть в архивах и библиотеках системы автоматизации программирования и отладки на технологической ЭВМ.

Вторая часть документов должна представлять комплекс программ в обобщенном и укрупненном виде, описывать общие принципы их функционирования и иерархическую схему связей компонентов. Эта документация не может быть изготовлена автоматически, так как для обобщений необходимо творческое участие разработчиков, и укрупненная исходная информация отсутствует в технологической ЭВМ. Положение упрощалось тем, что объем такой документации относительно невелик и находится в пределах 10 – 20% общего объема.

Кроме того, должен был обеспечен оперативный автоматический выпуск документов, отражающих изменения в программах, переменных и константах с корректировкой всей документации на завершающих этапах комплексной отладки, а также в процессе эксплуатации и при модернизации системы. Автоматический выпуск основной документации на программы позволяет исключать множество ошибок, как в документах, так и непосредственно в

управляющих программах, и существенно влияет на качество и эксплуатационные характеристики программного продукта.

***Автоматизированный контроль процесса разработки*** и технологических характеристик комплекса управляющих программ необходим руководителям проекта для учета объективных характеристик и тенденций изменения состояния комплекса. В зависимости от степени детализации и аспекта анализа, эти характеристики должны служить основой для принятия различных решений по корректировке программ, технологии их разработки и распределению усилий специалистов с целью повышения качества проектирования и снижения его трудоемкости. Исходные данные для получения большинства частных и обобщенных характеристик содержатся в архивах и библиотеках системы автоматизации программирования и отладки и могут быть выделены, упорядочены и отредактированы. Однако ряд величин является промежуточными, не регистрируются и теряются, если не предусмотрены специальные программные средства для их фиксирования в процессе выполнения функциональных задач по программированию и отладке.

## *3. Структура системы автоматизации программирования и отладки ЯУЗА-6.*

Система ЯУЗА-6 делилась на ***три крупных компонента*** (рис. 1) [2]:

- организующую систему;
- систему автоматизации программирования;
- систему автоматизации отладки.

Эти три системы использовали развитую информационную систему – ***базу данных***, которая включала архив символьной информации исходных текстов программ, библиотеку паспортов модулей и компонентов, архив оттранслированных программ на языке программирования и в машинных кодах, тесты и результаты отладки, технологические и эксплуатационные документы программных продуктов.

**Организующая система** была предназначена для формирования режимов функционирования САРПО, для управления хранением и обработкой данных, для подготовки системы к эксплуатации, а также для сбора и подготовки информации контроля процесса разработки. В соответствии с назначением она состояла из четырех частных систем.

***Монитор*** служил для связи пользователей с системой ЯУЗА-6 и являлся управляющей системой, обеспечивающей общую организацию прохождения работ, выполняемых САРПО ЯУЗА-6 на технологической ЭВМ БЭСМ-6 с ОС ДИСПАК. Монитор обеспечивал ввод и контроль информации, и вызов отдельных компонентов системы ЯУЗА-6.

***Система управления данными проекта*** обеспечивала запись, корректировку и считывание, распечатку и каталогизацию входной символьной информации архивов САРПО, организованных на магнитных лентах или дисках БЭСМ-6.

***Система автоматизации адаптации*** обеспечивала выполнение комплекса процедур, формировавших версию САРПО ЯУЗА-6 для конкретной специализированной, управляющей ЭВМ и конкретного проекта автоматизированной системы. Для адаптации пользователь использовал, так называемую ***базовую систему,*** представлявшую собой совокупность машинно-независимых программ и блоков САРПО ЯУЗА-6 (95% всей системы). Машинно-зависимая часть САРПО была сосредоточена в ограниченном числе машинно-зависимых программных модулях.

***Система контроля разработки*** обеспечивала сбор, обобщение и редактирование информации о текущем состоянии разработки комплекса программ проекта и об их характеристиках. Эта информация являлась исходной для руководителей при контроле и управлении ходом разработки комплекса программ.

**Система автоматизации программирования** обеспечивала получение синтаксически, семантически и структурно-корректных записей компонентов и комплекса программ и описаний переменных, на входных языках САРПО ЯУЗА-6 и в машинных кодах программ, а также технической документации на них и временных характеристик исполнения.

***Транслятор описаний глобальных переменных***. В системе ЯУЗА-6 составление описания глобальных переменных, которыми пользовались (без дополнительных описаний) все или часть программных компонентов, было выделено в отдельный этап. Описания глобальных переменных и/или глобальные константы были объединены в структуры в различных формах представления, принятых для специализированной ЭВМ. Язык описаний глобальных переменных являлся фрагментом алгоритмического языка ЯУЗА. Транслятор описаний состоял из двух просмотров и обработки входного текста, предварительно прошедшего лексический контроль, и перекодированного во внутренний код транслятора.

ЯУЗА-6 имел ***три взаимосвязанных языка программирования***: автокод, макроязык и алгоритмический язык. В качестве базового языка был использован модифицированный и русифицированный Алгол-60, дополненный средствами раздельной компиляции, возможностями указания типов данных с ограничениями и конкретным представлением их в памяти ЭВМ.

***Трансляторы с алгоритмических языков*** для специализированных ЭВМ состояли из пяти просмотров и решали следующие задачи:

- лексический и синтаксический контроль входного текста;
- распределение памяти переменных;

- семантический контроль;
- масштабирование;
- оптимизация программы;
- формирование последовательности автокодных команд или операторов макроязыка.

***Транслятор с макроязыка*** строился по обычной схеме макрогенераторов. При этом макроязык рассматривался как макросредства над автокодом.

***Транслятор с автокода*** для определенной специализированной ЭВМ отличался выбором при адаптации изобразительных средств (буквенных и символьных) для составляющих элементов команд. Глобальные переменные в модуле программы не описывались, они считались объявленными во всем комплексе программ. Трансляция с автокода состояла из двух просмотров.

***Система структурного контроля*** осуществляла контроль построения компонентов на соответствие общим правилам, предъявляемым к структуре подпрограмм и вложенных модулей, а также производила контроль размещения переменных в памяти специализированной, управляющей ЭВМ. Система подразделялась на две подсистемы: построения модели программы (операторной схемы) и собственно контроля структуры на отсутствие зацикливаний, тупиков и неподключенных участков в компонентах программы.

***Система расчета временных характеристик*** позволяла получать средние и предельные значения длительностей исполнения отдельных компонентов и комплекса взаимосвязанных программ без их непосредственного исполнения. Объектом анализа для системы являлась взвешенная графовая модель программы, автоматически построенная по модели программы и приведенная к эквивалентному ациклическому виду.

***Система выпуска документации*** была предназначена для выпуска технологической и эксплуатационной документации по информации, хранящейся в базе данных системы ЯУЗА-6. Такими документами являлись: распределение памяти ОЗУ и ДЗУ; паспорта программ; макроописания; каталоги библиотек и характеристики проекта. Все перечисленные документы печатались на стандартных бланках.

**Система автоматизации отладки** (система детерминированного тестирования) была предназначена для исполнения программ управляющей ЭВМ на технологической машине БЭСМ-6 с использованием программного интерпретатора, который моделировал работу специализированной ЭВМ. Система ***состояла из трех компонентов***.

***Транслятор отладочных заданий и тестов*** производил обработку отладочного задания на программу, представляющую собой последовательность операторов отладки и тесты.

***Интерпретатор команд специализированной ЭВМ*** осуществлял моделирование покомандного исполнения программы на БЭСМ-6 в кодах специализированной ЭВМ в условном масштабе времени. Память специализированной ЭВМ (ОЗУ и ДЗУ) отображалась однозначно на память БЭСМ-6, загрузка которой всеми программами и константами отлаживаемого комплекса программ производилась из архива САРПО. В результате формировалась информация о типе и адресе исполненной команды, адресе команды, которая будет исполняться следующей, и содержании всех моделируемых регистров управляющей ЭВМ.

***Информирующая система отладки*** была предназначена для получения и выдачи на экран и/или на печать информации о результатах исполнения отлаживаемых программ. Система опиралась на план отладки, в котором были зафиксированы данные, интересующие пользователя, и на результаты моделирования исполнения каждой команды, выданные интерпретатором. Используя паспорта программ, производился перевод адресов в символьное представление имен величин, а содержимое ячеек (регистров) переводилось с учетом типов и масштабов величин в значения в терминах входного языка.

## *4. Заключение*

При применении ЯУЗА-6 был налажен и апробирован ***процесс накопления и использования наборов готовых испытанных программных и информационных модулей и компонентов*** для формирования новых комплексов программ и/или их модернизированных версий. Библиотеки наборов таких компонентов и унифицированные межмодульные интерфейсы обеспечили ***эффективное конфигурационное управление*** при создании из них новых комплексов программ и при модернизации эксплуатируемых версий программных продуктов.

Первая версия **САРПО была** ***передана для эксплуатации разработчикам в середине 1975-го года***, адаптированной на ***ЭВМ 5Э26*** для разработки комплекса программ радиолокационного узла «Основа», а затем системы ПВО «Байкал». Эта массовая высокопроизводительная ЭВМ для мобильных оборонных систем была создана в ИТМ и ВТ [1]. Было произведено около 1,5 тысячи этих машин, с программными комплексами, разработанными с использованием САРПО ЯУЗА-6. Для обеспечения работ по отладке системных и пользовательских ***программ в реальном времени*** впоследствие был создан имитационный комплекс на БЭСМ-6 в многомашинном комплексе АС-6.

***Полная версия САРПО ЯУЗА-6 эксплуатировалась в ряде организаций почти 20 лет с 1979 года*** (в одной организации до 2000 года) и имела суммарный объем около 400 тыс. слов БЭСМ-6. В разработке этой версии системы принимали участие около 60 специалистов при средней производительности труда около 5 команд в день на человека. Процессы разработки компонентов системы производились сверху вниз по техническим

заданиям и спецификациям требований небольшими группами специалистов на языке БЕМШ. Регистрировались затраты на различных этапах и была произведена оценка общей трудоемкости разработки САРПО ЯУЗА-6, составившей около 300 человеко-лет. Для системы было выпущено техническое описание, инструкции по адаптации и эксплуатации.

Наиболее активно САРПО ЯУЗА-6 применялась для различных типов бортовых ЭВМ класса «Аргон», а также в следующих организациях, где была адаптирована для приведенного количества типов специализированных ЭВМ:

МНИИ приборной автоматики – 6 типов;

НИИ автоматической аппаратуры – 5 типов;

НПО автоматизации приборостроения – 11 типов;

НИ электромашиностроительный институт – 2 типа;

НПО ЭЛАС (г. Зеленоград) – 3 типа;

НПО им. Лавочкина – 2 типа;

НПО ПРОГРЕСС (г. Самара) и другие.

В сумме это определило использование ЯУЗА-6 более чем в 13 организациях, для свыше 30 типов мобильных, специализированных ЭВМ. Общий ***объем разработанных программ с применением ЯУЗА-6 к 1985 году превысил 5 млн. команд***. В НПО АП ЯУЗА-6 использовалась, в частности, для разработки программ: орбитальной станции САЛЮТ-7; межпланетных станций Венера и Марс, спутников Экран, Радуга, Горизонт и ряда стратегических ракет. Эксплуатация ЯУЗА-6 прекращалась в конце ***90-е годов***, в основном, вследствие технического старения машин БЭСМ-6.

**Литература**


[1] Липаев В.В. Отечественная программная инженерия: фрагменты истории и проблемы. – М.: СИНТЕГ. 2007. 312 с.

[2] Липаев В.В., Серебровский Л.А., Филиппович В.В. Система автоматизации программирования и отладки комплексов программ управления (ЯУЗА-6). «Программирование», 1977, № 3.

# Cross programming system YAUZA-6 for specialized real time computers (70s–80s of the last century)

*V.V. Lipaev*
*lip@ispras.ru*

**Abstract.** The paper presents an overview of features and challenges of efficient creation in 70s of sophisticated complexes of programs for military use based on specialized computers. Methods, requirements, and implementation of adaptive cross systems for automating of programming and testing for various types of sophisticated complexes of programs for target computers are demonstrated. The paper also describe the results of long-term usage of the YAUZA-6 on the platform of BESM-6 at a number of enterprises.

**Keywords:** Cross programming system, complex of programs, automation of programming and testing, specialized computer, translator and interpreter, assembler language and macro language.

# Планирование строго периодических задач в системах реального времени

*С.В. Зеленов*
*zelenov@ispras.ru*

**Аннотация.** Одним из важнейших аспектов функционирования систем реального времени является планирование задач. Классические алгоритмы планирования периодических задач работают лишь в случае, когда время запуска каждой задачи может варьироваться внутри разных периодов ее выполнения. Однако, в настоящее время имеется потребность в составлении таких расписаний, в которых время между соседними запусками одной периодической задачи было бы фиксировано и равнялось бы длине периода. Такое дополнительное требование не позволяет использовать в планировщике классические алгоритмы планирования. В работе предлагается алгоритм построения расписаний, близких к оптимальным в смысле минимизации общего количество прерываний задач. Оптимизация производится с учетом того, чтобы за приемлемое время строилось как можно более оптимальное расписание.

**Ключевые слова:** система реального времени; периодическая задача; планирование; авионика; непересекающиеся арифметические прогрессии.

## *1. Введение*

Задачи реального времени составляют одну из сложнейших и крайне важных областей применения вычислительной техники. Как правило, они связаны с контролем и управлением процессами, являющимися неотъемлемой частью современной жизни. Управление прокатными станами, роботами, движение на автомагистралях, контроль за состоянием окружающей среды, управление атомными и космическими станциями, авиационными системами, GRID-системы и многое другое – область задач реального времени. Эти задачи предъявляют такие требования к аппаратному и программному обеспечению, как надежность, высокая пропускная способность передающей среды в распределенных системах, своевременная реакция на внешние события и т.д. Для выполнения этих требований и создаются системы реального времени [1].

Одним из важнейших аспектов работы систем реального времени является своевременное обеспечение процессорными ресурсами всех выполняющихся в системе задач. Этим аспектом занимается специальная подсистема планирования задач. Проблема планирования как зависимых, так и

независимых задач в однопроцессорной или многопроцессорной системах реального времени достаточно хорошо изучена. Многочисленные результаты, полученные в этой области, приведены, например, в работах [2], [3], [4], [5], [6], [7], [8].

Наиболее частым случаем задач реального времени являются *периодические* задачи, каждая из которых должна быть запущена и полностью выполнить свою работу на каждом соответствующем этой задаче периоде времени. Примерами периодических задач являются разнообразные датчики, функции управления оборудованием, и пр. В реальных ситуациях, как правило, редко удается спланировать работу периодических задач так, чтобы каждая задача на каждом своем периоде работала непрерывно. Обычно постановка проблемы планирования предполагает, что любая задача в любой момент может быть прервана для выполнения более приоритетной задачи. В теории планирования предполагается, что прерывание задачи и передача управления другой задаче является бесплатной операцией. Однако, очевидно, в реальности такая операция требует некоторых ресурсов системы.

В классической постановке проблемы планирования задач время запуска периодической задачи не привязывается к определенному моменту времени внутри периода, но может варьироваться. Однако, в настоящее время в ряде случаев, например, при разработке систем управления авионикой, возникает необходимость обеспечить, чтобы запуски каждой периодической задачи выполнялись через строго определенные моменты времени (такие задачи мы будем называть *строго периодическими*). Это дополнительное очень сильное требование не позволяет напрямую использовать классические алгоритмы составления расписаний, такие как *Rate Monotonic* (RM), в котором преимущество отдается задачам с самыми короткими периодами выполнения, и *Earliest Deadline First* (EDF), в котором предпочтение отдается задачам с наиболее ранним предельным временем завершения выполнения [9].

В настоящей работе предлагается алгоритм построения расписания для строго периодических задач в однопроцессорной системе, так чтобы минимизировалось общее количество прерываний задач. Поскольку, вообще говоря, подобная проблема минимизации предполагает переборное решение, основные усилия в настоящей работе были направлены на то, чтобы алгоритм как можно быстрее находил достаточно хорошее приближенное решение.

## 2. Формальная постановка задачи

Пусть имеется один процессор, на котором требуется выполнять несколько независимых задач. Каждой задаче для своего завершения требуется некоторое процессорное время. При этом процессор может иногда прерывать выполнение любой задачи $T$, начать выполнять другую задачу $T'$ с более высоким приоритетом, а через некоторое время продолжить выполнение задачи $T$. Процессорное время разбито на равные отрезки – *тики*, так что на каждом тике может выполняться только одна задача, т.е. запуск, завершение,

прерывание и возобновление выполнения (после прерывания) любой задачи возможно только в начале тика. Переключение с выполнения одной задачи на другую считается мгновенным.

Будем считать, что точки, являющиеся началами тиков, последовательно перенумерованы неотрицательными целыми числами начиная с нуля. Везде далее под словом «*точка*» будем подразумевать эти перенумерованные точки, а также будем отождествлять точки с их номерами. Будем говорить, что некоторая точка *x принадлежит* задаче *T*, если *x* является началом тика, на котором выполняется задача *T*.

Задача $T(d, p)$, требующая для своего выполнения $d$ тиков, называется *строго периодической*, если ее запуск периодически повторяется, причем между соседними точками запуска проходит ровно $p$ тиков, и выполнение задачи на каждом периоде должно завершиться до следующей точки запуска. Величина $p$ называется *периодом*, $d$ – *длительностью* задачи *T*. Всегда верно соотношение $p \geq d \geq 1$.

Для данной строго периодической задачи $T(d, p)$ точка $r$ называется *начальной точкой запуска*, если $p > r \geq 0$, и задача *T* запускается только в точках $r + kp$ для всех целых $k \geq 0$. В этих обозначениях условия строгой периодичности задачи *T* формально записываются так:

```
задача T запускается в каждой точке r + kp,
на каждом участке [ r + kp, r + kp + (p – 1) ]
включительно имеется ровно d точек, принадлежащих T.
```

Обратим внимание на то, что завершение выполнения строго периодической задачи на очередном периоде *не является* прерыванием этой задачи. Действительно, в начале следующего периода будет произведен *новый запуск* этой задачи (но *не* возобновление выполнения после прерывания).

В рамках веденных здесь определений формальная постановка основной задачи формулируется следующим образом.

*Пусть заданы n строго периодических задач* $\{ T_i(d_i, p_i) \}$*, выполняющихся на одном процессоре. Требуется найти для них начальные точки* $\{ r_i \}$ *и распределить время выполнения задач между собой так, чтобы минимизировать общее количество прерываний задач.*

Как видно из приведенной формулировки, задача распадается на две подзадачи:

```
найти начальные точки { ri }, так чтобы для любых двух
задач никакие их точки запуска не совпадали;
распределить время выполнения задач между собой, т.е.
построить расписание.
```

## 3. Предварительные сведения

*Наибольший общий делитель* (НОД) $n$ целых чисел $a_1, \ldots, a_n$ обозначается $(a_1, \ldots, a_n)$. Два целых числа $a$ и $b$ называются *сравнимыми по модулю* целого числа $c$ (записывается: $a \equiv b \pmod{c}$), если разность $(a - b)$ делится на $c$.

Из теории чисел известна следующая теорема (см., например, [10]):

**Теорема 1.** Пусть даны целые числа $a$, $b$ и $c$, и дано сравнение с одним неизвестным $x$:

$$ax \equiv b \pmod{c}.$$

Тогда если $b$ не делится на $(a, c)$, то сравнение не имеет решений, иначе – сравнение имеет $(a, c)$ решений по модулю $c$, причем все эти решения сравнимы друг с другом по модулю $c / (a, c)$.

Из Теоремы 1 вытекает следующая теорема:

**Теорема 2.** Пусть даны целые числа $t$, $c$, $c'$. Тогда для всевозможных целых $l$ решения сравнения

$$x \equiv t + cl \pmod{c'} \tag{1}$$

удовлетворяют соотношению

$$x - t \equiv 0 \pmod{(c, c')}. \tag{2}$$

**Доказательство.** По Теореме 1, для любого $l$ сравнение (1) имеет единственное решение. Решения сравнения (1) для разных $l$ и $l'$ совпадают, если $c(l - l') \equiv 0 \pmod{c'}$, т.е. (по Теореме 1) если $l$ и $l'$ сравнимы друг с другом по модулю $c' / (c, c')$. Отсюда следует, что существует $c' / (c, c')$ различных решений сравнения (1) для разных $l$. Следовательно, поскольку для любого $l$ разность $x - t = cl$ делится на $(c, c')$, и число $c'$ делится на $(c, c')$, то все $c' / (c, c')$ различных решений сравнения (1) для разных $l$ удовлетворяют соотношению (2). ►

Если даны $n$ периодических задач, то расписание для них достаточно построить на участке длины *наименьшего общего кратного* (НОК) их периодов. Планировщик в этом случае будет выполнять задачи, циклически проходя по построенному расписанию.

Необходимым условием возможности построения расписания на однопроцессорной системе для $n$ периодических задач с длительностями $d_i$ и периодами $p_i$ является условие на суммарную загруженность процессора:

$$\sum (d_i / p_i) \leq 1$$

Для алгоритма EDF это условие является и достаточным [9]. Однако, в случае строго периодических задач это условие уже не является достаточным.

**Пример.** Пусть даны три строго периодические задачи: $T_1(1, 2)$, $T_2(1, 4)$, $T_3(1, 6)$. Тогда суммарная загруженность процессора строго меньше 1, а НОК периодов этих задач равен 12. Без ограничения общности будем считать, что задача $T_1$ стартует в точке 0. Тогда она будет запускаться во всех четных точках. В какой бы из оставшихся свободных точек 1 или 3 ни стартовала задача $T_2$, она будет запускаться каждый четвертый тик, а значит свободными для запуска задачи $T_3$ останутся точки, образующие арифметическую прогрессию с периодом 4. Однако на такой прогрессии невозможно уместить прогрессию с периодом 6. Таким образом, задачу $T_3$ спланировать не удастся.

## 4. Алгоритм перебора потенциальных начальных точек

Сопоставим каждой задаче $T_i(d_i, p_i)$ множество $R_i$, состоящее из $p_i$ точек, последовательно перенумерованных числами от 0 до $(p_i - 1)$. Каждое множество $R_i$ содержит все потенциальные начальные точки для задачи $T_i$. Однако, в ходе выбора начальных точек для некоторых из задач, множества потенциальных начальных точек для остальных задач будут постепенно сокращаться. Действительно, если для некоторой задачи $T_k$ выбрана начальная точка $r_k$, то для любого $m \neq k$ нужно удалить из отрезка $R_m$ все такие точки $r$, что

$$r - r_k \equiv 0 \quad (\bmod (p_k, p_m)), \qquad (3)$$

поскольку если в качестве начальной точки для $T_m$ будет выбрана точка со свойством (3), то тогда в некоторый момент, по Теореме 2, точки запуска задач $T_k$ и $T_m$ совпадут.

**Замечание.** Иногда для некоторых задач может и не произойти реального сокращения множеств потенциальных начальных точек в силу того, что все точки, которые должны быть удалены согласно приведенному выше правилу, уже были удалены в результате выбора начальных точек для других задач. Поясним это на следующем примере. Пусть даны три задачи: $T_1(1, 4)$, $T_2(1, 8)$, $T_3(1, 10)$. Пусть для первой задачи выбрана начальная точка $r_1 = 0$. Тогда из множества $R_2$ будут удалены точки 0 и 4 (поскольку (4, 8) = 4), а из множества $R_3$ – точки 0, 2, 4, 6, 8 (поскольку (4, 10) = 2). Тогда если для второй задачи выбрать начальную точку $r_2 = 2$, то из множества $R_3$ нужно будет удалить точки 0, 2, 4, 6, 8 (поскольку (8, 10) = 2), однако все эти точки уже были удалены из этого множества.

**Замечание**. В том же примере из предыдущего замечания, если для второй задачи выбрать начальную точку $r_2 = 1$, то реальное сокращение множества $R_3$ произойдет, а именно нужно будет удалить точки 1, 3, 5, 7, 9.

Будем считать, что выбор начальных точек происходит последовательно по порядку, начиная с $r_1$ и заканчивая $r_n$. Из сделанных выше замечаний следует,

что для любого $m$ состав множества $R_m$ зависит как от периодов задач, так и от выбора всех предыдущих начальных точек $r_1, \ldots, r_{m-1}$.

Установим на кортежах начальных точек $\{ r_i \}$ лексикографический порядок: будем говорить, что кортеж $( r_1, \ldots, r_n )$ предшествует кортежу $( r'_1, \ldots, r'_n )$, если для некоторого $m$ выполнены соотношения:

$$r_m < r'_m;$$

$$r_k = r'_k \text{ для всех } k < m.$$

Алгоритм перебора потенциальных начальных точек в первом приближении состоит в переборе кортежей $\{ r_i \}$ в лексикографическом порядке; при этом для любого $k$ перебор $r_k$ осуществляется по актуальному множеству $R_k$ (т.е. полученному в соответствии с выбором начальных точек $r_1, \ldots, r_{k-1}$).

В реальных приложениях количество задач $n$ может достигать значения 20, 30, и даже 40. При этом реальные периоды задач могут достигать значения в несколько тысяч тиков. На подобных входных данных приведенный алгоритм будет работать неприемлемо долго. Однако, на деле нам не требуется перебирать все варианты кортежей $\{ r_i \}$. Требуется как можно быстрее найти такой вариант кортежа начальных точек, при котором количество прерываний задач будет как можно меньше.

Исходя из этого, мы модифицируем алгоритм так, чтобы в первую очередь перебирались те варианты кортежей $\{ r_i \}$, на которых наиболее вероятно будет происходить непрерывное выполнение задач. Более точно, мы сконцентрируемся на обеспечении как можно более длительного непрерывного выполнения первых тиков после запуска каждой задачи. Для этого мы воспользуемся двумя следующими приемами.

*Первый прием* состоит в следующем. Заметим, что при выборе очередной начальной точки $r_k$ мы фактически резервируем один начальный тик выполнения задачи $T_k$. Однако можно таким же образом резервировать для этой задачи $s_k$ начальных тиков, где $d_k \geq s_k \geq 1$. При этом для перебора следующих начальных точек $r_m$ (при $m > k$) придется из $R_m$ удалять не только точки, соответствующие (в смысле соотношения (3)) точке $r_k$, но все точки, соответствующие всем точкам из участка $[r_k, r_k + s - 1]$.

Очевидно, наилучшим вариантом будет такое резервирование, когда для всех задач $s_k = d_k$, поскольку при этом все задачи будут выполняться без прерываний. Однако такое возможно далеко не всегда. Действительно, после такого резервирования, проведенного лишь для нескольких первых задач, может оказаться, что для какой-то из оставшихся задач множество потенциальных начальных точек стало пустым.

Заметим, что чем меньше период задачи, тем чаще она будет запускаться, а, следовательно, приоритетным является зарезервировать $s_k = d_k$ для задач с наименьшими периодами. Исходя из этого, поскольку задачи у нас

отсортированы по возрастанию их периодов, а резервирование осуществляется по порядку номеров задач от 1 до $n$, будем резервировать начальные тики для задач по остаточному принципу: для текущей задачи $T_k$ будем резервировать $s_k$ как можно больше, чтобы для остальных задач $T_m$ (при $m > k$) можно было зарезервировать хотя бы по одному тику (будем говорить, что в этом случае для задачи $T_k$ резервирование *корректно*).

Алгоритм поиска наибольшего корректного резервирования $s_k$ основан на методе половинного деления отрезка [ 1, $d_k$ ], так что на каждом шаге деления левая граница отрезка позволяет корректное резервирования, а правая – нет.

*Второй прием* для обеспечения как можно более длительного непрерывного выполнения первых тиков после запуска каждой задачи состоит в следующем. Заметим, что после осуществления резервирования для первых $k$-1 задач и удаления всех соответствующих точек из $R_k$, множество $R_k$ представляет собой участок целых чисел от 0 до $p_k - 1$, в котором некоторые точки «выколоты», а оставшиеся точки образуют несколько непрерывных участков. Для того, чтобы обеспечить наибольшее резервирование $s_k$, следует в первую очередь перебирать левые концы этих оставшихся непрерывных участков из $R_k$ в порядке убывания длины соответствующего непрерывного участка.

## 5. Алгоритм построения расписания для данного набора точек старта

Алгоритм построения расписания является модификацией алгоритма EDF. Модификация направлена на то, чтобы обеспечить как можно более длительное непрерывное выполнение первых тиков после запуска каждой задачи.

В алгоритме EDF задачам динамически назначаются приоритеты, так что наибольший приоритет имеет та (еще не завершенная) задача, для которой ближе всего очередная следующая точка запуска.

Наша модификация этого алгоритма состоит в следующем. Рассмотрим две последовательные точки $x$ и $y$ запусков задач ($x < y$). Пусть в точке $x$ запущена задача $T$, для которой зарезервировано $s$ начальных тиков. Построим по алгоритму EDF предварительное расписание на участке [ $x + s$, $y - 1$ ]. После этого, если на этом участке имеются точки, принадлежащие задаче $T$, то перенесем их в начало этого участка. В результате все точки, принадлежащие задаче $T$ на участке [ $x$, $y - 1$ ], будут следовать непрерывно в начале этого участка. Таким же образом будем поступать для всех последовательных точек запусков задач.

## 6. Результаты экспериментов

Предложенный в настоящей работе алгоритм предполагается использовать в реальных условиях, когда количество задач может достигать значения 20, 30 и

даже 40. При этом расчетная априорная загрузка процессора составляет порядка 50–70%.

Предложенный алгоритм испытывался на множествах задач (от 16 до 44 задач) с псевдослучайными параметрами, удовлетворяющими следующим свойствам:

Период каждой задачи является делителем числа 29·3·53 = 192000 (с тем, чтобы, из соображений потребления ресурсов алгоритмом, НОК периодов всех задач не превышал бы 192000);

Длительность для каждой задачи равна (с учетом округлений) либо ¼, либо ½, либо ¾ от периода этой задачи, деленного на количество задач (так что максимальная загрузка процессора не будет превышать 75%).

Эксперименты проводились на ПК с процессором Intel с тактовой частотой 2,4 GHz и ОЗУ объемом 1 GB. Для каждого множества задач осуществлялся запуск предложенного алгоритма для поиска наилучшего (в смысле общего количества прерываний задач) расписания в течение первых 10 секунд работы алгоритма. Результаты экспериментов приведены в таблице (под *лучшим решением* здесь понимается лучшее, найденное за 10 с).

| Число задач | НОК периодов задач | Общая загрузка | Время нахождения первого решения (мс) | Время нахождения лучшего решения (мс) | Количество прерываний на участке НОК | Среднее количество прерываний (на 1000 тиков) |
|---|---|---|---|---|---|---|
| 16 | 48000 | 56.43% | 18 | 451 | 455 | 9.48 |
| 17 | 96000 | 51.90% | 34 | 5045 | 1807 | 18.82 |
| 18 | 192000 | 52.81% | 125 | 8559 | 2047 | 10.66 |
| 19 | 96000 | 57.93% | 66 | 1204 | 1409 | 14.68 |
| 20 | 48000 | 56.05% | 26 | 6412 | 941 | 19.60 |
| 21 | 192000 | 53.98% | 105 | 7650 | 2425 | 12.63 |
| 22 | 96000 | 53.69% | 89 | 3635 | 1397 | 14.55 |
| 23 | 96000 | 59.19% | 37 | 1573 | 1941 | 20.22 |
| 24 | 192000 | 54.58% | 103 | 4758 | 1859 | 9.68 |
| 25 | 48000 | 52.87% | 68 | 6635 | 738 | 15.37 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 26 | 96000 | 54.74% | 75 | 3325 | 1363 | 14.20 |
| 27 | 192000 | 55.47% | 206 | 6437 | 2160 | 11.25 |
| 28 | 96000 | 61.22% | 66 | 235 | 2559 | 26.66 |
| 29 | 192000 | 59.37% | 238 | 3480 | 2819 | 14.68 |
| 30 | 96000 | 64.09% | 106 | 9782 | 2560 | 26.67 |
| 31 | 192000 | 51.29% | 189 | 2818 | 2301 | 11.98 |
| 32 | 96000 | 56.58% | 96 | 3730 | 1878 | 19.56 |
| 33 | 192000 | 56.19% | 240 | 6497 | 2108 | 10.98 |
| 34 | 96000 | 64.68% | 130 | 3706 | 2308 | 24.04 |
| 35 | 192000 | 56.28% | 290 | 9993 | 2978 | 15.51 |
| 36 | 192000 | 65.39% | 193 | 2848 | 4652 | 24.23 |
| 37 | 96000 | 58.30% | 129 | 1134 | 2109 | 21.97 |
| 38 | 192000 | 60.03% | 302 | 1698 | 4348 | 22.65 |
| 39 | 96000 | 66.39% | 155 | 9234 | 2628 | 27.37 |
| 40 | 192000 | 61.80% | 328 | 2961 | 4717 | 24.57 |
| 41 | 192000 | 58.73% | 229 | 4857 | 4531 | 23.60 |
| 42 | 96000 | 57.96% | 181 | 2527 | 1678 | 17.48 |
| 43 | 192000 | 69.38% | 385 | 8249 | 6032 | 31.42 |
| 44 | 192000 | 58.58% | 781 | 5073 | 4320 | 22.50 |

*Табл. 1. Результаты экспериментальных испытаний алгоритма.*

Как видно из результатов испытаний, предложенный алгоритм позволяет в условиях с реальными общими характеристиками достаточно быстро строить расписания со средним количеством прерываний, как правило, не превышающим величины 25 прерываний на 1000 тиков, что является вполне приемлемым результатом.

## *7. Заключение*

В работе предложен алгоритм построения расписания, близкого к оптимальному, для строго периодических задач на однопроцессорной системе, где в качестве оптимизационного критерия рассматривается требования минимизации общего количества прерываний задач.

Эксперименты показывают, что алгоритм позволяет за приемлемое для человека время (порядка 10 секунд) получать достаточно оптимальное расписание. При количестве задач 20–40, когда периоды задач могут достигать значения в несколько тысяч тиков, алгоритм позволяет получать расписания со средним количеством прерываний не более 25 прерываний на 1000 тиков.

В настоящее время алгоритм активно используется в реальных проектах по разработке систем управления авионикой.

**Литература**


[1] А.С.Косачев, И.Б.Бурдонов, В.Н.Пономаренко. Операционные системы реального времени. // Препринт Института системного программирования РАН, 2006, № 14. http://citforum.ru/operating_systems/rtos/

[2] Liu J.W.S. Real-Time Systems. // Prentice Hall, Englewood Cliffs, NJ, 2000. 600 p.

[3] Cottet F., Kaiser J., Mammeri Z. Scheduling in Real-Time Systems. // John Wiley & Sons Ltd. 2002. 282 p.

[4] N.N. Kuzjurin. Multiprocessor scheduling and expanders. Information Process. Letters, 1994, v. 51, № 6, 315–319.

[5] Н. Н. Кузюрин. Многопроцессорные расписания и комбинаторные конфигурации. Дискретная математика, 1995, т. 7, № 2, 77–87.

[6] S. Zhuk, A. Tchernykh, N. Kuzjurin, A. Pospelov, A. Shokurov, A. Avetisyan, S. Gaissaryan, D. Grushin. Comparison of Scheduling Heuristics for Grid Resource Broker. Proc. of the Third International Conference on Parallel Computing Systems (PCS2004). IEEE Computer Society Press, 2004, 388–392.

[7] A. Tchernykh, J. M. Ramirez, A. Avetisyan, N. Kuzjurin, D. Grushin, S. Zhuk. Two Level Job-Scheduling Strategies for a Computational Grid. Proc. of the Second Grid Resource Management Workshop. LNCS 3911, 2006, 774–781.

[8] A. Tchernykh, U. Schwiegelsohn, R. Yahyapour, N. Kuzjurin. On-line Hierarchical Job Scheduling in Grids with admissible allocation. J. of Scheduling. 2010, v. 13, № 5, 545–552.

[9] Liu C. and Layland J.W. Scheduling algorithms for multiprogramming in a hard real-time environment. // Journal of ACM , 20(1): 46–61, 1973.

[10] И.М.Виноградов. Основы теории чисел. // М.-Л.: Гостехиздат, 1952. 180 стр.

# Scheduling of Strictly Periodic Tasks in Real-Time Systems

*S.V. Zelenov*
*zelenov@ispras.ru*

**Abstract.** The very important subsystem in a real-time system is a task scheduler. Classical algorithms for periodic tasks scheduling imply that release points of each task may vary inside different periods of the task. However, nowadays, some systems require a scheduler that can build schedules where release points of each task form an arithmetic progression. Such an additional requirement does not allow using classical scheduling algorithms. In this paper, we present a scheduling algorithm that takes into account this additional requirement and builds a schedule close to optimal (in the sense of minimization of total number of tasks interruptions) in acceptable time.

**Keywords:** real-time system; periodic task; scheduling; avionics; disjoint arithmetic progressions.

# Тестирование протоколов электронной почты Интернета с использованием моделей

*Н. В. Пакулин, А. Н. Тугаенко*
*{npak, tugaenko}@ispras.ru*

**Аннотация.** В статье рассматриваются вопросы тестирования почтовых протоколов с использованием формальных моделей протоколов: предложен метод моделирования почтовых протоколов, рассмотрены особенности почтовых протоколов в контексте тестирования, представлены результаты тестирования популярных почтовых серверов с открытым кодом. В качестве примера представлена разработка тестовых наборов для протоколов SMTP и POP3 на языке JavaTESK – расширении языка Java, реализующем тестирование с формальными методами. Тестовые наборы состоят из двух частей: независимого тестирования соответствия протоколов спецификациям и совместного теста, имитирующего работу почтовых протоколов в сети.

**Ключевые слова.** Формальные методы, верификация, тестирование, электронная почта, UniTESK.

## *1. Введение*

Электронные письма являются основой современного взаимодействия между людьми. Миллионы писем перемещаются ежедневно в сети Интернет. Надежность и корректность инфраструктуры почтовых сообщений чрезвычайно важна для современного информационного общества. В этой статье мы коснемся двух аспектов этих вопросов: надежности (1) передачи почты в сети Интернет и (2) доставки писем конечным адресатам.

На своем пути от отправителя к получателю почтовое сообщение проходит через цепочку промежуточных серверов. Часто промежуточные сервера поставляются разными разработчиками, и поэтому они имеют различные реализации почтовых протоколов. Общая надежность механизма передачи электронной почты в значительной степени определяется совместимостью различных реализаций серверов между собой, а также корректностью функциональной части каждой реализации.

В настоящее время тестирование соответствия реализации стандартам служит основным методом обеспечения их совместимости. Логическое обоснование такого предложения основано на предположении хорошего проектирования протокола: если две реализации соответствуют спецификации протокола, то

они совместимы. В этой статье мы не обсуждаем, выполняется ли это предположение для почтовых протоколов сети Интернет, мы предлагаем подход для тестирования соответствия таких протоколов.

Несмотря на более чем двадцатилетнюю историю почтовых протоколов и существования десятков реализаций протоколов SMTP[1], POP3[2] и IMAP4[3], до сих пор нет открытого и независимого от реализации тестового набора для проверки их соответствия стандартам. Мы считаем, что есть несколько причин объясняющих это:

1. почтовые протоколы выглядят простыми;
2. разработчики при тестировании фокусируются на аспектах разработки почтовых серверов, не имеющих отношения к соответствию стандартам: настройка параметров, безопасность, внутреннее хранилище сообщений, быстродействие и так далее.

Рассмотрим эти причины более детально. Во-первых, простота почтовых протоколов кажущаяся. Можно легко увидеть, что протоколы SMTP, POP3 и IMAP4 содержат ряд нетривиальных особенностей:

1. почтовые протоколы недоспецифицированы, существенная часть функциональности оставлена на усмотрение разработчика, в спецификации описаны несколько вариантов возможного дальнейшего поведения системы;
2. почтовые протоколы недетерминированы: стандарт допускает различные варианты поведения системы, включая отказ в доставке почтовых сообщений или разрыв соединений;
3. функции почтовых протоколов различаются по степени обязательности (MUST, SHOULD, MAY и прочие);
4. архитектура протоколов расширяемая: реализация протокола может использовать различные расширения, как дополняющие функциональность протокола, так и изменяющие её.

Перечисленные особенности определяют фактическую сложность разработки тестовых наборов для тестирования реализаций протоколов на соответствие спецификациям.

Во-вторых, дополнительная сложность в разработке почтовых серверов связана с поддержкой большого числа настроек, необходимых для практической эксплуатации – параметров пересылки, аутентификации, безопасности, хранилища писем и так далее. Мы изучили тестовые наборы нескольких реализаций, построенных по модели открытого кода: Apache James, Sendmail, Dovecot, Qmail, Postfix. Как оказалось, тесты сильно связаны с конкретными реализациями и непереносимы для тестирования серверов от других разработчиков, так как используют особенности реализации: настройки параметров, доступ к внутреннему состоянию, выполнение в одном

процессе вместе с реализацией. Использование тестов, разработанных для одной реализации, не предоставляется возможным для проверки других реализаций. Более того, как показал анализ, эти тесты не предназначены для тестирования соответствия, они проверяют, прежде всего, корректность реализации многочисленных настроек почтового сервера, не относящихся напрямую к стандартам SMTP, POP3 и IMAP4. Проблема заключается в том, что такой подход к тестированию не гарантирует обеспечения совместимости сервера со стандартом. В частности, именно тестирование соответствия позволило выявить в почтовом сервере James серьезную функциональную ошибку – зацикливание передачи сообщений в определенной конфигурации.

Строгая связь между тестами и требованиями стандартов позволяет оценить полноту тестирования с точки зрения внешнего пользователя почтового сервиса. Как показывает пример с сервером James, тщательное тестирование внутренних функций реализации не гарантирует качества функционирования реализации в реальном окружении.

Мы считаем, что задача тестирования соответствия стандартам почтовых протоколов обладает существенным практическим значением. Целью нашего исследования является разработка новой архитектуры тестовой системы, а также методики для разработки элементов системы, пригодных для учета особенностей почтовых протоколов. В данной работе представлена новая технология разработки тестов для тестирования соответствия почтовых протоколов. Данная технология включает несколько этапов разработки тестов, причем этапы сконструированы таким образом, что каждый этап может быть завершающим. Новизна метода тестирования соответствия почтовых протоколов заключается в последовательной трансформации модели тестирования, при которой на каждом шаге получаются отлаженные тесты (модель и программа зависят от шага). В начале разработки тестового набора пишется набор элементарных испытаний (test cases) или берется уже имеющийся. На следующих шагах строится интерфейсная модель, в построенную модель добавляются состояния. После выполнения этих шагов получается тот же по функциональности тестовый набор, что был после первого шага. Но на данном этапе уже намного проще расширять тестовый набор путем расширения формальной спецификации тестируемого протокола. Предложенный подход, основанный на методологии тестирования на основе моделей, фокусируется только на тестировании соответствия и не рассматривает другие аспекты проверки почтовой инфраструктуры, такие как тестирование взаимодействия, производительности, надежности и прочие.

В качестве примера применения предложенного метода с его помощью был разработан открытый тестовый набор для проверки соответствия основной функциональности протоколов SMTP, POP3 и IMAP4 их стандартам. Также был разработан тест, использующий композицию спецификаций протоколов SMTP и POP3. Данный тест используется для проверки корректности взаимодействия реализаций различных почтовых серверов.

Предложенный подход позволяет последовательно трансформировать тесты, написанные как элементарные испытания, в тесты, сгенерированные на основе моделей, с сохранением функциональности и работоспособности (отлаженности) тестов. На первых шагах происходит только трансформация подхода тестирования, формируется модель тестируемого протокола, в то время как последующие шаги метода позволяют легко расширять набор тестов для проверки определенной функциональности.

Структура статьи: в разделе 2 представлен краткий обзор современных почтовых протоколов, в разделе 3 кратко рассмотрены существующие подходы к тестированию протоколов и обсуждается выбор тестирования на основе моделей и технологии автоматизированного тестирования UniTESK для предлагаемого подхода. Раздел 4 содержит краткое введение в технологию UniTESK, которая лежит в основе предлагаемого подхода. В разделе 5 описан подход, использующийся для разработки тестовых наборов для SMTP, POP3 и IMAP4. В разделе 6 представлены разработанные к настоящему моменту тестовые наборы для протоколов SMTP, POP3 и IMAP4. В разделе 7 обсуждаются плюсы и минусы предложенного подхода. Раздел 8 содержит полученные к данному моменту результаты и описывает направления будущих исследований.

## 2. Краткое введение в почтовые протоколы

Большинство почтовых сообщений в сети Интернет передаются посредством протокола SMTP [1] – сетевого протокола верхнего уровня стека протоколов TCP/IP. Это текстовый протокол, состоящий из двух частей: клиента и сервера. Клиент подает команды и сервер исполняет их, возвращает код ответа и другие данные, если это необходимо. Протокол SMTP имеет свою подсеть в стеке TCP/IP, содержащую множество почтовых серверов и агентов пересылки, используемых для передачи сообщений между различными сетевыми узлами. Особенность SMTP состоит в том, что каждый физический сервер может функционировать и как SMTP клиент, и как SMTP сервер: будучи сервером он принимает входящее сообщение и становится клиентом для пересылки сообщения следующему узлу.

Протокол SMTP используется для отправки сообщений, но когда реализация SMTP сервера идентифицирует себя как конечного получателя почтового сообщения, пересылка прекращается, а письмо помещается во внутреннее, зависящее от реализации, хранилище. Для получения сообщений из хранилища конечные пользователи используют другие протоколы: POP3 [2] и IMAP4 [3]. Оба протокола также являются текстовыми, поддерживающими роли клиента и сервера. Клиенты получают доступ к хранилищу, подавая протокольные команды серверам, и сервера предоставляют необходимую информацию в ответах на эти команды. Обычно реализации POP3 и IMAP4 поддерживают только одну роль в одно время – либо клиента, либо сервера.

Также стоит отметить, что реализации почтовых серверов могут функционировать на одном сетевом узле или даже в одном почтовом сервере. В частности, почтовые сообщения, пересылаемые по протоколу SMTP, доступны клиентам по протоколам POP3 и IMAP4. По этой причине при тестировании необходимо рассматривать почтовый сервер как совокупность реализаций нескольких протоколов, причем тестовые воздействия возможны сразу по нескольким протоколам.

## *3. Тестирование почтовых протоколов*

В современной индустрии тестирование реализаций протоколов на соответствие стандартам большей частью основано на ручной разработке тестовых наборов, состоящих из отдельных программ, написанных на специализированных или обычных языках программирования. Такие программы называются элементарными испытаниями (test case), в них реализуется генерация последовательности стимулов, подача сгенерированной последовательности в целевую систему, считывание и анализ реакций [4].

Создателями реализаций почтовых серверов разработали многочисленные тестовые наборы для функционального тестирования своих реализаций. Эти тестовые наборы написаны, как правило, на том же языке, который использовался для создания сервера (C, C++, Java и т.д.). Они нацелены прежде всего на тестирование внутренних механизмов реализаций – обработки всевозможных опций настройки сервера, обработки ошибок, взаимодействия с пользователем и т.п. Тесты, созданные разработчиками серверов, не содержат четко определенного подмножества тестов соответствия спецификациям; кроме того, тесты не переносимы между реализациями почтовых приложений, так как существенно используют знание о внутреннем устройстве конкретной реализации.

Стоит отметить разработанный в Apache Project инструмент Mail Protocol Tester (MPT) [5], предназначенный для тестирования произвольных протоколов с тестовыми сообщениями. Тесты задаются на языке XML; спецификация тестов не зависит от какой-то конкретной реализации. Тест представляет собой последовательность строковых команд и описаний ожидаемых ответов реализаций. Ожидаемые ответы описываются регулярным выражением — если ответ сервера не соответствует регулярному выражению, MPT останавливает работу и выдает сообщение об ошибке. MPT не поддерживает ветвления в тестах, циклы и использование параметров.

Исследователи обычно рассматривают элементарные испытания как один из традиционных методов тестирования [6-8], в то время как тестирование на основе моделей рассматривается как новый метод, решающий множество неразрешаемых с помощью традиционных методов проблем (например, неточность в вычислении покрытия функциональности тестируемой системы или ручная разработка прослеживаемости требований). Предложенный в этой статье метод рассматривает тестирование на основе моделей как расширение

метода, основанного на элементарных испытаниях. В качестве демонстрации предложенного метода представлены разработанные тестовые наборы для протоколов SMTP, POP3 и IMAP4. Также проводилась разработка тестового набора для проверки корректности взаимодействия реализаций почтовых протоколов SMTP и POP3.

Рассмотрим требования к тестовому набору. Тестовый набор для тестирования соответствия должен обладать следующими свойствами:

1. прослеживаемостью требований. Тесты (полученные из формальной спецификации) должны соотноситься с требованиями стандарта, должно быть наглядно видно, какие требования какими тестами покрываются.

2. многообразием настроек на особенности реализаций (MAY, SHOULD, MUST и другие). Должна быть опция для определения множества требований, поддерживаемых тестируемой реализацией. В это множество не должны попадать требования, не поддерживаемые тестируемой реализацией.

3. полнотой тестового набора в смысле покрытия требований. Полученный в итоге тестовый набор должен покрывать как минимум все выделенные из стандарта требования, помеченные в стандарте как обязательные для каждой реализации.

Тестирование на основе элементарных испытаний не предоставляет возможностей явной прослеживаемости требований, полнота тестирования в смысле покрытия всех требований при таком подходе также затруднена. Походы, основанные на TTCN3 [9] и JUnit [10], требуют внешних инструментов для установления соединения между тестами и требованиями, таких как матрицы прослеживаемости (traceability matrixes). Слабость таких инструментов в том, что требования не являются составной частью теста.

Кроме того, разработка большого числа тестов приводит к дублированию кода или сложным и запутанным связям. Нужно использовать методы, позволяющие декомпозировать тестовый набор и обеспечить прослеживаемость требований.

Необходимые возможности дают инструменты, построенные на формальных методах. Выделение формальных спецификаций позволяет:

1. задавать формальные связи между требованиями и тестами, автоматически отслеживать качество тестирования в терминах покрытия спецификации;

2. использование формальной спецификации дает возможность многократно использовать модель для проверки правильности поведения реализаций;

3. наличие спецификации позволяет генерировать тестовые воздействия в терминах модели и автоматически фильтровать избыточные воздействия.

Также при выборе метода для построения тестовых последовательностей важно учитывать особенности почтовых протоколов. В частности, из-за недетерминированного поведения протоколов и из-за недоспецифицированности протоколов для тестирования следует выбирать подходы, обеспечивающие построение тестовых последовательностей с учетом откликов целевой системы. Автоматическое вынесение вердикта из постусловий спецификации может решить проблему проверки корректности поведения целевой системы.

Существует много инструментов и подходов для тестирования. NModel [11] описывает модельную систему на языке C# и предоставляет основные возможности автоматической генерации тестов («на лету», on-the-fly testing) и максимизации покрытия согласно специально написанному критерию. Но для использования автоматической генерации тестов необходимо написать отдельную программу, описывающую сложный алгоритм обхода автомата теста.

Предыдущая версия SpecExplorer (2004) [12] предоставляла возможность тестирования «на лету», но эта версия больше не поддерживается. В последней версии SpecExplorer(2010) [13] тестирование «на лету» не документировано. Инструмент Conformiq Qtronic [14] не поддерживает тестирование «на лету», вместо этого он генерирует тестовые сценарии TTCN-3.

Технология UniTESK [15,16] поддерживает нотацию формальных спецификаций, автоматическую генерацию («на лету») тестовых воздействий (на уровне кода, нет необходимости писать отдельную программу) и автоматический анализ результатов. Поэтому мы решили использовать эту технологию в нашем проекте. UniTESK предоставляет средства для формальной спецификации асинхронных систем [17] в виде контрактных спецификаций переходов расширенного конечного автомата [18-20]. Помимо программных систем средствами UniTESK возможно специфицировать и тестировать аппаратные системы, такие как микропроцессоры [21,22].

В технологии UniTESK формальные спецификации, формализующие требования в виде пред- и постусловий, используются для генерации тестовой последовательности [23]. Также для генерации тестовых последовательностей должен быть определен конечный автомат (автомат теста).

Процесс тестирования в UniTESK представляет собой автоматический обход автомата теста, при котором наблюдаемое поведение реализации автоматически верифицируется тестовыми оракулами, сгенерированными из формальной спецификации. Использование формальных спецификаций позволяет автоматизировать проверку корректности поведения реализации и оценку полноты тестирования, а представление теста как автомата дает

возможность автоматически генерировать длинные и разнообразные последовательности тестовых событий.

В связи с тем, что реализации различных почтовых протоколов часто функционируют на одном и том же сетевом узле или в одном почтовом сервере, в дополнение к тестированию соответствия имеет смысл проводить интеграционное тестирование, направленное на проверку корректности взаимодействия реализаций различных протоколов, в частности, на корректность обработки данных. Инструмент JavaTESK [24] позволяет при наличии уже разработанной программы для генерации тестовых наборов для тестирования соответствия почтовых протоколов без особых дополнительных усилий создавать тесты для проведения интеграционного тестирования. При таком подходе используется более половины кода, написанного для проведения тестирования соответствия.

Авторы использовали представленный метод для разработки тестовых наборов для тестирования соответствия реализаций протоколов SMTP, POP3 и IMAP4, а затем и интеграционного тестирования реализаций различных почтовых протоколов, функционирующих на одном компьютере. Из инструментов, реализующих подход UniTESK, был выбран JavaTESK. JavaTESK использует язык программирования Java с набором расширений для записи формальных спецификаций и задания тестов.

## *4. Обзор технологии UniTESK*

Стандартным форматом нормативной документации протоколов Интернет являются документы RFC (Request for Comment). Требования в этих документах изложены на английском языке и представляют собой неформальный текст, описывающий желаемое поведение системы. В рамках технологии UniTESK для формальной записи требований используются спецификационные расширения языков программирования – Java или C.

Запись неформальных требований нормативной документации на формальном языке представляет собой модель протокола. В подходе UniTESK формальная модель протокола строится в терминах конечных автоматов. Переходы между состояниями могут задаваться как в явном виде, так и в неявном. В случае явного задания перехода модель содержит алгоритм вычисления следующего состояния и реакции протокола; неявное задание перехода представляет собой предикат, который накладывает ограничения на допустимые конечные состояния и реакции протокола.

Спецификация на языке JavaTESK обычно состоит из одного или нескольких спецификационных классов, которые описывают состояния и переходы моделируемого протокола. Переходы протокола представляются как методы класса специального вида (спецификационные методы), кроме того поддерживается возможность задать ограничения на множество допустимых

состояний посредством инвариантов типов (ограничений на значения типов) и инвариантов переменных состояния.

Неявное задание переходов реализуется в виде пред- и постусловий. В предусловии содержатся ограничения на допустимые значения параметров воздействия на целевую систему и на состояние, в котором воздействие оказывается. На воздействия целевая система может реагировать изменением состояния, подачей реакции или и тем, и другим. Постусловие проверяет допустимость продемонстрированного поведения системы.

При моделировании поведения тестируемой системы используется набор структур данных, которые называются абстрактными состояниями. Для вынесения вердикта о корректности поведения целевой системы UniTESK использует данные, которые содержатся в абстрактном состоянии модели.

В UniTESK воздействия на целевую систему и реакции целевой системы описываются в терминах модели, содержащейся в спецификациях. Поэтому для взаимодействия модельной и целевой систем используется некоторый посредник – медиатор, – который переводит параметры воздействия из модельного представления в представление реализации, реакции целевой системы в модельное представление, а также при необходимости вносит изменения состояния целевой системы в абстрактное состояние.

Тестовый сценарий определяет последовательность воздействий, оказываемых на целевую систему. В качестве теоретической основы для построения сценариев используется метамодель конечных автоматов. В JavaTESK автомат теста задается в сценарном классе, который содержит процедуру определения текущего состояния автомата теста и итераторы тестовых воздействий. Инструмент JavaTESK предоставляет набор обходчиков, которые строят цепочки тестовых воздействий из описания автомата теста.

## 5. Предложенный метод тестирования почтовых протоколов

Формально процесс тестирования почтовых протоколов можно разбить на две части: независимое тестирование реализаций на соответствие стандартам (тестирование соответствия) и интеграционное тестирование реализаций различных протоколов. Первая часть заключается в тестировании соответствия различных реализаций серверов каждого протокола требованиям, независимо от реализации данным сервером других протоколов. Здесь внимание уделяется корректности функциональных частей протоколов. Вторая часть представляет собой интеграционное тестирование реализаций различных почтовых протоколов. В интеграционном тестировании внимание уделяется взаимодействию реализаций разных протоколов, в частности, корректности обработки данных.

Почтовые протоколы могут находиться в нескольких состояниях. При получении определенных стимулов они генерируют и передают отклики, а

также переходят в другие состояния либо остаются в тех же. С учетом этого на базе тестирования UniTESK был разработан метод для тестирования соответствия почтовых протоколов. Данный метод основан на методе тестирования, изложенном в работе [25], но в отличие от разработанного там метода позволяет разрабатывать тестовые наборы путем последовательного перехода от тестирования на основе элементарных испытаний (test cases) к тестированию на основе моделей, если уже имеется набор элементарных испытаний, или как метод последовательного создания тестов на основе моделей. Разработанный метод содержит следующие основные шаги:

1. знакомство с предметной областью, разработка примеров и элементарных тестов. Данный шаг не дает никаких видимых результатов, но он важен для более детального понимания работы протоколов и помогает при реализации последующих шагов метода.

2. составление каталога требований. Каталог требований представляет собой базу данных или таблицу с описанием требований. При выделении требования из RFC в таблицу или базу данных вместе с текстом требования заносятся идентификатор требования, а также тип требования (синтаксическое или функциональное), обязательность выполнения данного требования (в RFC часть требований помечена ключевыми словами MUST, SHOULD, MAY и другими, характеризующими обязательность выполнения требования реализациями), ссылка на место в RFC и, возможно, какие-то дополнительные атрибуты.

3. построение упрощенной модели протокола, создание экспериментального теста – автомата теста с единственным состоянием. Упрощенная модель протокола включает в себя знания о командах протокола, а также о возможных реакциях на эти команды. Экспериментальный тест состоит из спецификационного, медиаторного и сценарного классов. Спецификационный класс на данном этапе содержит только сигнатуры методов, а все проверки производятся в сценарном классе. Такой тест называется линейным, подача стимулов и считывание реакций в нем производятся в определенном порядке, указанном в сценарии.

4. построение концептуальной модели поведения протокола; выделение основных состояний протокола; построение автомата теста с выделенными состояниями; добавление в экспериментальный тест блока, ответственного за перевод модельной системы между состояниями. Концептуальная модель описывает внешне наблюдаемое поведение системы как операции над некоторым набором абстрактных компонентов и объектов, составляющих её. Эти компоненты используются только для моделирования поведения и могут не соответствовать разбиению самой системы на компоненты.

Добавление в тест блоков, переводящих систему из одного состояния в другое, преобразует тест из линейного в автоматный. В таком тесте построение цепочки тестовых воздействий происходит при обходе автомата, а подача стимулов и считывание реакций производятся только из разрешенных состояний.

5. формализация требований, перенос проверки полученных от реализации откликов в отдельный блок – спецификационный класс. При формализации требований проверяется их полнота и непротиворечивость. Итогом шага является формальная спецификация протокола, написанная на расширении одного из языков программирования.

6. расширение сценария и спецификации для покрытия всех требований. В сценарном классе описываются стимулы, которые будут подаваться системе. Порядок подачи стимулов формируется при обходе автомата и зависит от условий подачи стимулов в определенных состояниях. Обычно один сценарный класс отвечает за какой-либо конкретный раздел требований. Для покрытия всех формальных требований может потребоваться несколько сценарных классов.

7. выполнение тестов и анализ полученных результатов. Анализ результатов может показать, что не все требования покрываются сгенерированным тестовым набором. Если это так, то шаг 6 повторяется до тех пор, пока все требования не будут покрыты

Если на момент начала написания тестов уже имеется набор элементарных испытаний (test cases), а также если тестирование проводит человек, уже имеющий опыт создания тестов на основе моделей, то первый шаг можно опустить. Также стоит отметить, что данный порядок шагов не является единственным. Так, шаги 3 и 4 могут быть объединены, а шаг 5 может выполняться параллельно с шагами 3 и 4. Также возможны итерации в рамках подцепочек приведенной последовательности.

Интеграционное тестирование проводится после того, как готовы спецификационные и медиаторные классы для тестирования соответствия протоколов, участвующих в интеграционном тестировании. При разработке тестовых наборов для тестирования интеграции в качестве спецификации используется композиция спецификаций тестируемых протоколов, а медиаторные классы остаются без изменений. Сценарный класс описывает определенную последовательность действий, которая позволяет проверить корректность передачи, получения и обработки данных различными протоколами.

# 6. Применение метода для тестирования протоколов SMTP, POP3 и IMAP4

В данном разделе описывается применение метода для тестирования почтовых протоколов. Сначала описывается разработка тестового набора для тестирования соответствия почтовых протоколов, затем – разработка тестов для проведения интеграционного тестирования.

## 6.1. Независимое тестирование реализаций протоколов

Первым шагом в написании тестов для реализаций протоколов SMTP, POP3 и IMAP4 было изучение предметной области, отправление и получение писем напрямую из командных строк серверов. Затем были выделены требования из RFC и классифицированы по типам: команды и отклики, маршрутизация, уведомления, серверные настройки, заголовки письма, тело письма и прочие. На основании этого были построены упрощенные модели протоколов в которых автоматы тестов подавали в определенном порядке команды:

- для протокола SMTP: EHLO, HELO, MAIL, RCPT, DATA и другие;
- для протокола POP3: USER, PASS, LIST, STAT, RETR, DELE, TOP и другие;
- для протоколаIMAP4: тег плюс командное слово LOGIN, EXAMINE, CREATE, DELETE, RENAME, SELECT и другие,

а затем принимали отклики серверов: для протокола SMTP – трехзначные числа (коды откликов); для POP3 – "+OK" или "-ERR" отклики; для IMAP4 – необязательный тег плюс "OK", "NO", "BAD", "PREAUTH" или "BYE". На этом шаге реализация генерации тестовых наборов имела формальный интерфейс, спецификационные классы состояли только из сигнатур методов, все проверки корректности поведения тестируемых реализаций находились в сценарных классах. Сценарные классы состояли из методов, подающих воздействия целевым системам (посредством медиаторных классов), считывающих ответы серверов и возвращающих вердикты о корректности поведения серверов. Медиаторные классы переводили стимулы из формата тестируемой реализации в формат модели и обратно.

Затем были выделены основные состояния протокола. На данном шаге в сценарные классы были добавлены новые блоки, ответственные за перевод системы между состояниями. Спецификационные классы не менялись.

На следующем шаге блоки, ответственные за проверку корректности поведения тестируемой системы, и блоки, переводящие систему между состояниями, были перенесены из сценарных классов в спецификационные. В сценарных классах остались только стимулы, подающиеся тестируемой реализации. С этого момента определенные команды могли подаваться только из разрешенных состояний автомата. Возможность такой проверки

достигается записью разрешенных состояний в предусловия спецификации. Итератор каждый раз проверяет, в каком состоянии находится система, и разрешена ли подача данной команды в текущем состоянии. Также это позволяет проверить тот факт, что сервер не посылает команды из неразрешенных для данных команд состояний.

### 6.2. Тестирование взаимодействия реализаций протоколов

Для интеграционного тестирования реализаций протоколов создавался отдельный сценарий, который использует спецификации и медиаторы, разработанные для тестирования соответствия. Сценарий может содержать как сценарные методы, разрабатываемые для тестирования соответствия, так и модифицированные или специально разработанные непосредственно для тестирования интеграции. В частности, сценарии, не содержащие ветвления, могут быть просто скопированы в новый сценарный класс, в то время как сценарии, проверяющие реакцию протокола на различные параметры в командах, следует модифицировать, чтобы не проверять в интеграционном тесте несущественные для интеграции детали. Например, один из разработанных сценариев содержит следующие действия: для SMTP – последовательность команд для отправки письма, для POP3 и IMAP4 – последовательность команд для проверки количества писем в почтовом ящике, а также для чтения нового письма и сравнения его с письмом, отправленным в этом же тесте через протокол SMTP.

## *7. Анализ предложенного подхода*

К недостаткам метода, основанного на формальных спецификациях, можно отнести отсутствие быстрой возможности обновления тестовых наборов. Для разработки нового теста необходимо тщательно изучить требования, формализовать и классифицировать их. Только после этих приготовлений можно начинать писать спецификационный, медиаторный и сценарный классы. Из-за стадии приготовления, включающей разработку спецификации, период разработки нового теста увеличивается. С другой стороны, после того, как написаны спецификационные, медиаторные и сценарные классы получается не один тест, а набор тестов, отвечающий за определенный класс требований. Также следует отметить, что длительная стадия приготовления присутствует только в случае неполной формальной спецификации. Если формальная спецификация доступна, тесты могут быть сгенерированы быстро, просто путем изменения сценарного класса. Более того, если задача состоит в написании сложных тестов, то использование формальных спецификаций даст результат быстрее и проще, чем в случае подхода, основанного на элементарных испытаниях.

Основной недостаток предложенного метода заключается в обязательности существования хотя бы одной реализации: на каждом шаге тесты и спецификация проверяются на реальной реализации. Как результат, этот

метод нельзя применять напрямую для протоколов, для которых еще нет какой-либо реализации.

Одним из важных достоинств этого метода является отделение блока, ответственного за вынесения вердикта, от блока генерации тестовых последовательностей. Оракул, который генерируется из постусловий спецификации, ответственен за вынесение вердикта. Благодаря этому разделению разработчик тестов не должен обрабатывать вердикты во время проверки корректности поведения системы. Наличие определенного блока (спецификации), в котором производятся и хранятся все проверки корректности поведения системы, упрощает переиспользование оракула в различных тестовых сценариях.

Также использование формальных спецификаций позволяет сформулировать точный недвусмысленный критерий полноты тестирования – тестирование может быть завершено, когда все элементы соответствующей формальной спецификации покрыты. UniTESK предоставляет поддержку динамического доступа к покрытым требованиям и позволяет задавать критерии отбора для сценариев – если применение конкретного сценария не увеличивает тестового покрытия, то система пропускает данный сценарий и переходит к следующему.

## *8. Результаты и дальнейшие исследования*

В данной работе рассматривались только основные функции протоколов, другая функциональность, например, уведомления о доставке или сбоях, почтовые шлюзы в не-SMTP домены, выходят за рамки данного проекта. Для протокола SMTP было выделено 51 требование, 43 требования относятся к серверным командам и откликам (11 из них обязательные и 4 опциональные), 8 относятся к маршрутизации (все обязательные). Для протокола POP3 было выделено 58 требований на основные команды, 5 из них обязательные и 6 опциональные. Для протокола IMAP4 было выделено 43 требования на основные команды, 7 из них обязательные и 2 опциональные.

Все требования покрыты разработанными тестовыми наборами. Тесты применялись для почтовых серверов, разрабатываемых по модели открытого исходного кода – Apache James, hMail Server, Postfix и Dovecot. Сгенерированные тесты обнаружили несколько несоответствий между реализациями протоколов и стандартами [1-3]. При проведении интеграционного тестирования между реализациями протоколов SMTP и POP3 ошибок не обнаружено. Несоответствия, обнаруженные при тестировании соответствия, приведены ниже:

- отсутствие поддержки обязательных команд в отдельных реализациях;
- нарушение правил протокола (подача команд в недопустимых состояниях);

- неверные значения кодов откликов на команды протокола;
- отсутствие блокировки почтового ящика при аутентификации на сервере;
- зацикливание при пересылке сообщения.

## 8.1. Дальнейшие исследования

Общая характеристика почтовых протоколов заключается в том, что почтовые протоколы расширяемые. Некоторые расширения просто добавляют новую функциональность, то есть добавляют новые требования, которые не противоречат требованиям основного стандарта. Но также существуют расширения, которые радикально меняют структуру протокола, отвергая или изменяя отдельные требования базового стандарта. Тестирование расширяемых протоколов требует расширить инструментарий UniTESK средствами, которые предоставят возможность формально специфицировать расширения и генерировать тестовые последовательности с учетом набора расширений, поддерживаемых тестируемой реализацией. Направление нашего дальнейшего исследования – разработать соответствующие программные средства на базе инструмента JavaTESK.

# *9. Заключение*

В статье представлен подход для тестирования почтовых протоколов, основанный на формальных спецификациях. Новизна метода заключается в последовательном переходе от тестов, написанных по методике элементарных испытаний, к формальной модели и тестам, сгенерированным с использованием этой модели как оракула. Более того, на каждом шаге получается работоспособный тестовый набор, проверяющий ту же функциональность, что проверялась на предыдущем шаге, или даже содержащую больше проверок.

Подход относится к области тестирования на основе моделей, он использует контрактные спецификации для формализации спецификаций протокола и генерации тестовой последовательности «на лету». Реализация подхода базируется на технологии UniTESK. Отличительными чертами данного метода являются автоматизированная генерация тестовой последовательности на основе формальных спецификаций, подсчет покрытия тестов, позволяющий оптимальным образом генерировать стимулы, а также наличие отдельного компонента – оракула – ответственного за вынесение вердикта о корректности поведения тестируемой системы.

Разработанный метод применялся для тестирования давно используемых реализаций почтовых протоколов. В одной реализации (Apache James Server) была найдена критическая ошибка – при определенной конфигурации DNS сервера во время пересылки сообщения сервер пересылал сообщение себе,

таким образом, сообщение никогда не достигало адресата. При этом уведомление отправителя о том, что сообщение не доставлено, отсутствует. Разработчики James признали выявленную проблему и планируют ее исправление в одном из ближайших релизов.

**Литература**


[1] IETF RFC 5321. J. Klensin. Simple *Mail Transfer Protocol*. 2008, 95 с.
[2] IETF RFC 1939. J. Myers, M. Rosem, *Post Office Protocol — Version 3*. 1996, 23 с.
[3] IETF RFC 3501. M. Crispin. *Internet Message Access Protocol – version 4rev1*. 2003, 108 с.
[4] ISO/IEC 9646. *Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 1: General concepts.* Geneva: ISO (1994).
[5] *Apache James Mail Protocol Tester*. http://james.apache.org/mpt/antlib/
[6] Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann, San Francisco (2007).
[7] Blackburn, M., Busser, R., Nauman, A.: *Why Model-Based Test Automation is Different and What You Should Know to Get Started*. Software Productivity Consortium, NFP (2004).
[8] Dalal, S.R., Jain, A., Karunanithi, N., Leaton, J.M., Lott, C.M., Patton, G.C., Horowitz, B.M.: *Model-Based Testing in Practice*. In: Proceedings of the ICSE 1999 (May 1999).
[9] ETSI ES 201 873-1 V3.1.1. *Methods for Testing and Specifcation (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language*. Sophia-Antipolis, France: ETSI (2009).
[10] Библиотека для тестирования программного обеспечения на языке Java. http://www.junit.org
[11] Jacky, J., Veanes, M., Campbell, C., Schulte, W.: *Model-based Software Testing and Analysis with C#*. Cambridge University Press, Cambridge (2008)
[12] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, Lev Nachmanson. *Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer*. Microsoft Research, Redmond, WA, USA. http://research.microsoft.com/pubs/77383/bookChapterOnSE.pdf
[13] Тестирование на основе моделей с помощью инструмента Spec Explorer. http://research.microsoft.com/en-us/projects/specexplorer/
[14] *End-to-End Testing Automation in TTCN-3 environment using Conformiq Qtronicand Elvior MessageMagic* (2009).
[15] Баранцев А.В., Бурдонов И.Б., Демаков А.В., Зеленов С.В., Косачев А.С., Кулямин В.В., Омельченко В.А., Пакулин Н.В., Петренко А.К., Хорошилов А.В. *Подход UniTesK к разработке тестов: достижения и перспективы*. // Труды ИСП РАН, №5, 2004. http://www.citforum.ru/SE/testing/unitesk
[16] UniTESK: индустриальная технология надежного тестирования. http://www.unitesk.com
[17] Н.В. Пакулин, А.В. Хорошилов. Разработка формальных моделей и тестирование соответствия для систем с асинхронными интерфейсами и телекоммуникационных протоколов. // Журнал "Программирование" № 5, 2007 г., ISSN 0132-3474, с. 1-29.
[18] V. V. Kuliamin, A. K. Petrenko, N. V. Pakoulin, A. S. Kossatchev, I. B. Bourdonov. Integration of Functional and Timed Testing of Real-time and Concurrent Systems. // Proceedings of the 5-th International Conference on Perspectives of System Informatics, July 9-12, 2003, Novosibirsk, Russia; LNCS 2890, Springer, 2003, pp. 450-461.

[19] V. V. Kuliamin, A. K. Petrenko, N. V. Pakoulin. Practical Approach to Specification and Conformance Testing of Distributed Network Applications. // Proceedings of the 2-nd International Service Availability Symposium, April 25-26, 2005, Berlin, Germany; LNCS 3694, Springer, 2005, pp. 68-83.
[20] V. V. Kuliamin, A. K. Petrenko, N. V. Pakoulin. Extended Design-by-Contract Approach to Specification and Conformance Testing of Distributed Software. // Proceedings of the 9-th World Multiconference on Systemics, Cybernetics, and Informatics, Model Based Development and Testing Workshop, July 10-13, 2005, Orlando, Florida, USA, pp. 65-70
[21] В. П. Иванников, А. С. Камкин, В. В. Кулямин, А. К. Петренко. Применение технологии UniTesK для функционального тестирования моделей аппаратного обеспечения. // Препринт 8 ИСП РАН, 2005.
[22] Иванников В. П., Камкин А. С., Косачев А. С., Кулямин В. В., Петренко А. К. Использование контрактных спецификаций для представления требований и функционального тестирования моделей аппаратуры. // Программирование, том 33, №5. МАИК «Наука/Интерпериодика», 2007, с. 47-61.
[23] I. B. Bourdonov, A. S. Kossatchev, V. V. Kuliamin, A. K. Petrenko. UniTesK Test Suite Architecture.// Proceedings of the International Symposium of Formal Methods Europe, July 22-24, 2002, Copenhagen, Denmark; LNCS 2391, 2002, pp. 121-152.
[24] I. B. Bourdonov, A. V. Demakov, A. A. Jarov, A. S. Kossatchev, V. V. Kuliamin, A. K. Petrenko, S. V. Zelenov. *Java Specification Extension for Automated Test Development.* // Proceedings of the 4-nd International Andrei Ershov Memorial Conference Perspectives of System Informatics, July 2-6, 2001, Novosibirsk, Russia; LNCS 2244, 2001, pp. 301-307.
[25] В. В. Кулямин, Н. В. Пакулин, О. Л. Петренко, А. А. Сортов, А. В. Хорошилов. Формализация требований на практике. // Препринт 13 ИСП РАН, 2006.

## Model-based testing of Internet Mail Protocols

*N.V. Pakulin, A.N. Tugaenko*
*{npak, tugaenko}@ispras.ru*

**Abstract.** The paper discusses model-based testing of the modern Internet e-mail protocols, including the method of protocol modeling by means of formal notations, peculiarities of e-mail protocols in the context of testing. The paper presents results of model-based testing of a number of popular open-source e-mail implementations. JavaTESK, an extension of Java programming language, was used to develop formal functional and test specifications for SMTP and POP3 protocols. The developed test suite includes separate tests for SMPT and POP3 as well as integration test for composition of SMTP and POP3 in a single implementation.

**Keywords:** formal methods, verification, testing, email protocols, UniTESK.

# Механизмы поддержки функционального тестирования моделей аппаратуры на разных уровнях абстракции

*А.С. Камкин, М.М. Чупилко*
*{kamkin,chupilko}@ispras.ru*

**Аннотация.** На разных этапах проектирования аппаратуры используются разные представления целевой системы (общее описание архитектуры в начале проекта шаг за шагом конкретизируется вплоть до разработки топологии интегральной схемы на кристалле). Соответственно, в зависимости от зрелости проекта применяются разные методы верификации, в частности, разные методы построения эталонных моделей, используемых для оценки корректности проектируемой аппаратуры (в начале разработки используются абстрактные модели, но ближе к завершению, для повышения качества проверки, точность моделей повышается). Различие в уровнях абстракции усложняет переиспользование тестовых систем, созданных в начале проектирования, для верификации аналогичных компонентов, но на более поздних этапах. В статье предлагается подход к построению эталонных моделей аппаратуры и тестовых оракулов на их основе, который упрощает повторное использование тестовых систем, тем самым снижая затраты на верификацию.

**Ключевые слова:** проектирование аппаратуры, функциональное тестирование, моделирование на уровне транзакций

## *1. Введение*

Системы аппаратуры постоянно усложняются, требуя все больше ресурсов для проектирования и верификации. Двумя основными способами преодоления сложности являются *декомпозиция* и *абстракция*. В общих словах, декомпозиция – это разделение системы на множество слабо связанных компонентов, в то время как абстракция (или абстрагирование) – это выявление существенных свойств системы относительно некоторого интересующего нас аспекта. Проектирование аппаратуры можно представить как эволюционный процесс декомпозиции системы и ее уточнения (понижения уровня абстракции), управляемый целевыми требованиями и имеющимися ресурсами.

Для того чтобы проверить соответствие результата проектирования требованиям, используется *верификация*. Часто верификация рассматривается

как завершающий этап проектирования. Однако, это не совсем так. Верификация тесно интегрирована в процесс разработки и осуществляется на всех без исключения этапах, обеспечивая своевременную обратную связь и делая проектирование более контролируемым и предсказуемым. В соответствии с вышесказанным, верификация применяется как для систем в целом, так и для отдельных компонентов. Кроме того, на разных этапах проектирования для верификации используются разные типы *эталонных моделей* (*reference models*)[1] – более абстрактные на ранних этапах, более точные в конце.

В статье рассматривается верификация *отдельных компонентов* аппаратуры. При этом предполагается, что состав и общая функциональность компонентов системы в процессе проектирования не меняются. Решаемая в работе проблема заключается в определении архитектуры *тестовой системы* (*testbench* [1]), которая, с одной стороны, является *универсальной* для разных уровней абстракции эталонных моделей, а, с другой стороны, обеспечивает *повторное использование* тестовых систем при эволюционном развитии проекта (иными словами, позволяет варьировать уровень абстракции проверяемых свойств). Если говорить более точно, в статье рассматривается архитектура не всей тестовой системы, а лишь ее части, связанной с автоматической проверкой корректности реакций целевого компонента в ответ на подаваемые стимулы (эта часть обычно называется *тестовым оракулом* или просто *оракулом* [2]).

Предлагаемый подход к построению тестовых оракулов (включающих в свой состав и эталонные модели) основан на так называемом *моделировании на уровне транзакций* (*TLM, Transaction Level Modeling*). Основной чертой TLM является отделение *коммуникаций* (деталей передачи данных между компонентами) от *вычислений* (функциональных преобразований данных) [3]. Связи между компонентами моделируются программно с помощью *каналов*, а *транзакции* (единичные пересылки данных) осуществляются посредством вызова интерфейсных функций каналов. Описанная в статье архитектура тестовых оракулов была апробирована в ряде промышленных проектов, где показала свою универсальность и гибкость.

Статья организована следующим образом. В разделе 2 делается обзор подходов к построению тестовых оракулов для моделей аппаратуры. Раздел 3 описывает основные понятия TLM и определяет основные уровни абстракции,

[1] Под *эталонной моделью* понимается формальным образом представленные требования, на соответствие которым проверяется результат проектирования. Как правило, эталонная модель – это программа на языке программирования общего назначения, эмулирующая работу целевой системы или ее компонента, а результат проектирования – описание (модель) схемы (системы или компонента) на специализированном языке (HDL, Hardware Description Language), например, Verilog или VHDL.

используемые при проектировании аппаратуры. В разделе 4 предлагается подход к организации тестовых оракулов и эталонных моделей. Раздел 5 описывает частные случаи предлагаемого подхода для разных уровней абстракции. В разделе 6 дается классификация ошибок в аппаратуре, основанная на рассмотренной схеме работы тестового оракула. Раздел 7 содержит обобщенное описание нашего опыта разработки тестовых систем для моделей аппаратуры. В разделе 8 делается заключение, и указываются направления дальнейших исследований.

## 2. Обзор подходов к построению тестовых оракулов

Существует два основных класса подходов к верификации – *формальные методы* (*formal methods*) и *методы тестирования*, основанные на имитационном моделировании (*simulation-based methods*) [4]. Известно, что формальные методы являются исчерпывающими (в некотором смысле), но они плохо масштабируются на сложные системы. Как правило, такие методы применяются для сравнительно простых компонентов, причем на достаточно поздних стадиях проектирования, когда требования достаточно стабильны. Методы тестирования не являются исчерпывающими, но они гораздо более гибкие и, следовательно, могут быть использованы на разных этапах проектирования.

Для автоматизации тестирования используют специализированные программы, называемые *тестовыми системами*. Типичная тестовая система включает в себя три базовых компонента: (1) *генератор стимулов*, (2) *тестовый оракул* и (3) *сборщик покрытия*. Генератор стимулов создает последовательность тестовых воздействий (стимулов), подаваемых на входы тестируемого компонента. Тестовый оракул оценивает корректность реакций, выдаваемых компонентом в ответ на поданные стимулы. Сборщик покрытия отслеживает достигнутый уровень тестового покрытия. Объектом исследования данной статьи являются тестовые оракулы. Существуют три основных метода построения оракулов: (1) *тесты со встроенными проверками* (*self-checking tests*), (2) *утверждения* (*assertions*) и (3) *ко-симуляция* (*co-simulation*).

Тесты со встроенными проверками – достаточно старый метод проверки реакций. Каждый стимул (точнее, тестовый пример) снабжается кодом, осуществляющим проверку выдаваемого компонентом результата на корректность [5]. Подход имеет очевидные неудобства. Во-первых, написать процедуру проверки реакций компонента для сложного теста достаточно трудно, а число тестов, как правило, велико. Во-вторых, тестовые примеры требуют постоянной поддержки, чтобы быть согласованными с изменениями в проекте (из-за большого объема тестов сопровождение может потребовать значительных ресурсов). В-третьих, для этого подхода характерна неполнота выполняемых проверок – каждый тестовый пример нацелен на достижение

определенной ситуации и обычно проверяет лишь те аспекты поведения тестируемого компонента, которые существенны для этой ситуации.

Утверждениями называются предикаты на поведение компонента, которые должны быть выполнены [6]. При таком подходе проверки отделяются от стимулов и пишутся либо в коде тестируемого компонента, либо отдельно. Это позволяет использовать для верификации автоматизированные генераторы стимулов, что является ключевым преимуществом подхода по сравнению с тестами со встроенными проверками. Как правило, утверждения описывают наиболее важные или наиболее очевидные свойства компонента. Тем самым, проверку на основе утверждений нельзя назвать полной. Следует отметить, что при использовании утверждений, встроенных в код, невозможно разработать тестовый оракул до того, как компонент будет полностью описан.

Ко-симуляция – это подход к проверке реакций, в котором вместе с моделью тестируемого компонента используется независимо разработанная эталонная модель [5]. На две модели подаются одинаковые тестовые последовательности, а результаты их работы сравниваются на соответствие. При расхождении результатов определяется, какая из моделей некорректна, ошибочная модель исправляется, после чего тестирование продолжается. Использование эталонной модели позволяет генерировать стимулы автоматически. Однако, создание программной модели, эмулирующей работу тестируемого компонента на всех допустимых тестовых последовательностях, является сложной задачей, которая в некоторых случаях равносильна повторному проектированию компонента.

Проанализируем подходы, описанные выше. Тесты со встроенными проверками не обеспечивают высокого уровня автоматизации тестирования и страдают от неполноты проверки реакций. Кроме того, их сложно сопровождать при наличии частых изменений в проекте. Утверждения являются идеальным решением для проверки небольшого числа свойств, но они не подходят для исчерпывающего тестирования сложных компонентов аппаратуры. Ко-симуляция выглядит наиболее многообещающим подходом, но разработка эталонной модели может потребовать много ресурсов. Для упрощения разработки и сопровождения эталонных моделей необходима специальная методология. На наш взгляд, за основу подобной методологии можно взять TLM.

Схожие идеи лежат в основе методологии верификации OVM (Open Verification Methodology), широко известном и применяемом на практике подходе [7, 8]. Согласно OVM, тестовая система должна быть разделена на несколько функциональных компонентов, интерфейсы между которыми должны быть определены с помощью TLM. Эта методология описывает общую архитектуру тестовой системы, но ничего не говорит о внутренней организации компонентов, в частности, не отвечает на вопрос, как генерировать тестовые последовательности и как проверять поведения сложной аппаратуры.

## 3. TLM и различные уровни абстракции

TLM – это подход к программному моделированию аппаратуры, в котором описание коммуникаций между компонентами отделено от их функциональности [3]. Механизмы коммуникации (такие как шины и буферы) моделируются каналами, инкапсулирующими низкоуровневые детали передачи данных. Транзакции (то есть пересылки данных) осуществляются путем вызова интерфейсных функций этих каналов. В общих словах, TLM фокусируется на функциональности передачи данных, а не ее действительной реализации. При использовании этого подхода легко экспериментировать с различными реализациями коммуникационных шин без необходимости изменения взаимодействующих компонентов.

В процессе проектирования аппаратуры используется множество промежуточных моделей, которые, с одной стороны, являются объектами верификации, а, с другой стороны, могут быть использованы в качестве эталонных моделей при последующей верификации. Для классификации уровней абстракции проектных моделей можно использовать *граф системного моделирования* (*system modeling graph*), показанный на Рис. 1 [3]. Ось X соответствует точности моделирования времени в вычислениях (функциональных преобразованиях данных), а ось Y – точности моделирования времени в коммуникациях (пересылках данных). На каждой оси отмечены три уровня абстракции: *без учета времени*, *с учетом времени* и *с потактовой точностью*.

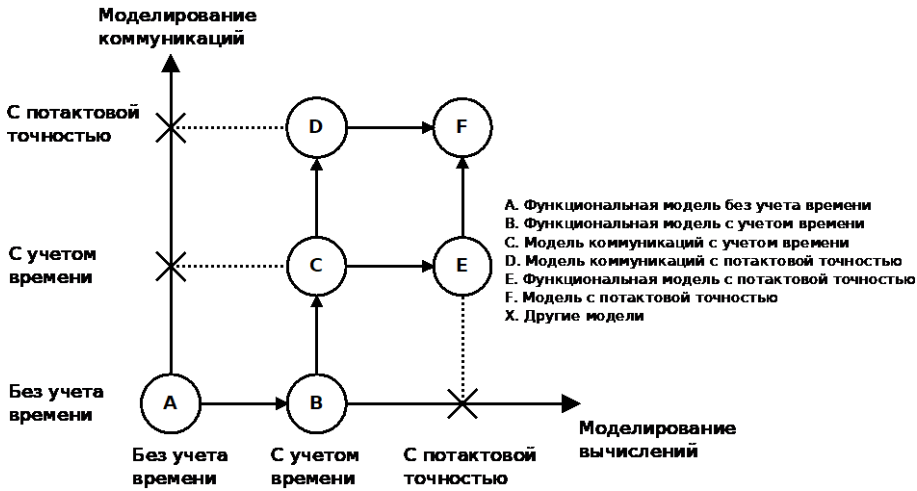



*Рис. 1. Граф системного моделирования*

В соответствии с [3], существуют 6 основных уровней абстракции: (A) *функциональные модели без учета времени*, (B) *функциональные модели с учетом времени*, (C) *модели коммуникаций с учетом времени*, (D) *модели коммуникаций с потактовой точностью*, (E) *функциональные модели с потактовой точностью* и (F) *модели с потактовой точностью* (терминология немного отличается от используемой в [3]). В таблице 1 приведены характеристики приведенных уровней абстракции.

| **Название уровня абстракции** | **Моделирование времени в коммуникациях** | **Моделирование времени в вычислениях** | **Используемая схема коммуникации** |
|---|---|---|---|
| Функциональная модель без учета времени (A) | Отсутствует | Отсутствует | Общие данные |
| Функциональная модель с учетом времени (B) | Отсутствует | Приблизительное | Каналы передачи сообщений |
| Модель коммуникаций с учетом времени (C) | Приблизительное | Приблизительное | Абстрактная шина (арбитр) |
| Модель коммуникации с потактовой точностью (D) | Точное | Приблизительное | Уточненная шина (протокол) |
| Функциональная модель с потактовой точностью (E) | Приблизительное | Точное | Абстрактная шина (арбитр) |
| Модель с потактовой точностью (F) | Точное | Точное | Сигналы |

*Таблица 1. Характеристики основных уровней абстракции*

## *4. Архитектура тестового оракула*

В данной статье рассматривается моделирование и верификация отдельных компонентов аппаратуры. Это сделано намеренно для того, чтобы абстрагироваться от коммуникаций внутри тестируемого устройства. Говоря о разработке тестовой системы (точнее, об организации тестового оракула),

основной интерес представляют внешние интерфейсы (через которые осуществляется взаимодействие с тестовой системой). Обобщенная структура тестовой системы представлена на Рис 2.

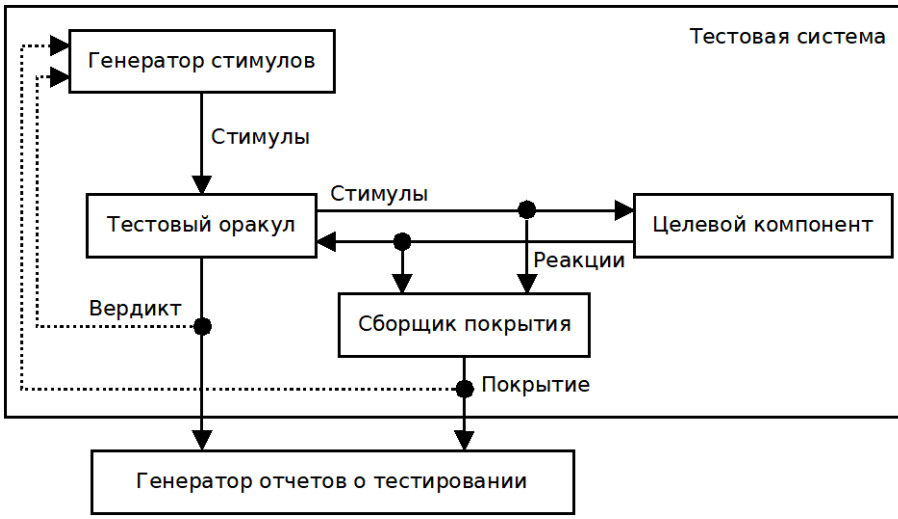


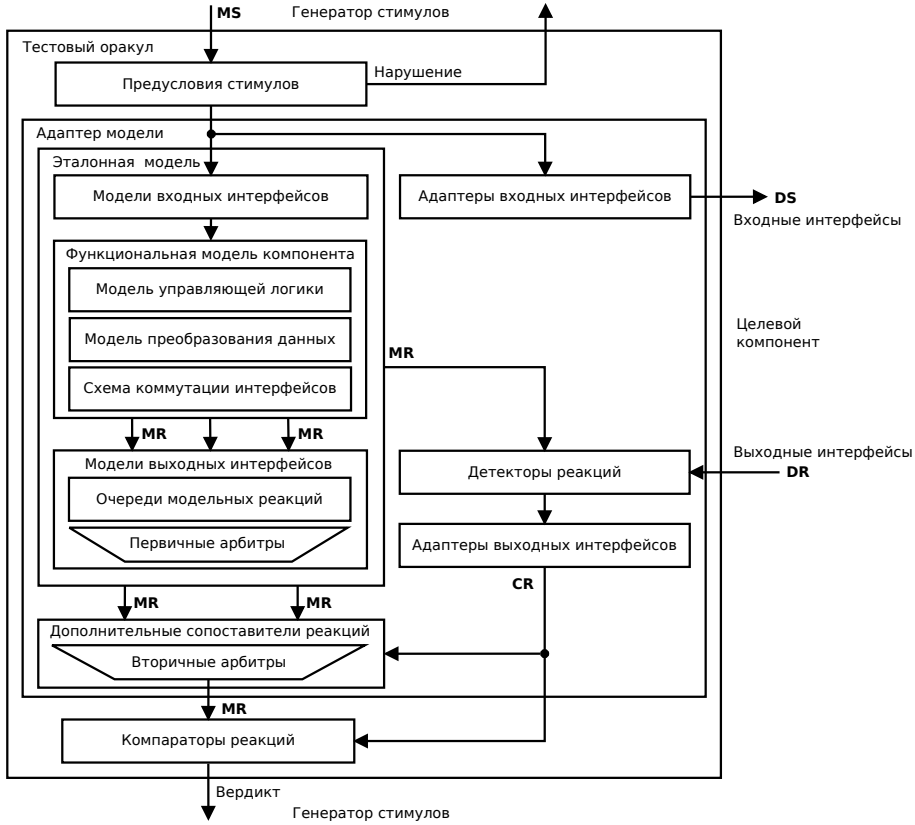

*Рис. 2. Обобщенная структура тестовой системы*

Генератор стимулов создает последовательность стимулов и передает их тестовому оракулу, который, в свою очередь, переводит их в представление тестируемого компонента и осуществляет их подачу. Реакции тестируемого компонента передаются оракулу, который оценивает их корректность (основываясь на утверждениях или эталонной модели). Стимулы и реакции также подаются сборщику покрытия, который оценивает завершенность тестирования, используя некоторые метрики или эвристики.

Эта схема отражает взаимодействие основных компонентов тестовой системы, не раскрывая их внутреннюю организацию. Между тем, внутренняя архитектура компонентов тестовой системы имеет решающее значение для обеспечения эффективного сопровождения и повторного использования тестов. Следует отметить, что, как правило, самые большие ресурсы, связанные с сопровождением тестовой системы, тратятся на поддержание тестового оракула в согласованном состоянии с целевым компонентом. Ниже перечислены основные требования, сформулированные нами для методов построения тестовых оракулов.

- Возможность использования абстрактной эталонной модели для тестирования компонента аппаратуры на уровне регистровых передач (RTL, Register Transfer Level).

- Простая адаптация тестовой системы к изменениям входных и выходных интерфейсов тестируемого компонента.
- Простая адаптация к изменениям временны*ы*х свойств тестируемого компонента и возможность уточнения эталонной модели вплоть до потактовой точности.
- Обеспечение настолько точной диагностики ошибок, насколько это возможно для данного уровня абстракции.




MS (Model Stimulus) - модельный стимул (абстрактное сообщение)
DS (Design Stimulus) - реализационный стимул (сериализация MS на входы компонента)
MR (Model Reaction) - модельная реакция (эталонные данные или ограничения)
DR (Design Reaction) - реализационная реакция (последовательность значений выходов компонента)
CR (Checking Reaction) - проверяемая реакция (десериализация DR в абстрактное сообщение)

*Рис. 3. Архитектура тестового оракула для компонента аппаратуры*

На Рис. 3 приведена архитектура тестового оракула, учитывающая сформулированные требования. Оракул содержит *предусловия стимулов*,

*компараторы реакций*, *эталонную модель* и *адаптер эталонной модели*. Эталонная модель состоит из *моделей входных и выходных интерфейсов* и *функциональной модели*, а адаптер эталонной модели – из *адаптеров входных и выходных интерфейсов*. Для каждого выходного интерфейса имеется *очередь модельных реакций* (часть модели интерфейса) *детектор реакций* (часть адаптера модели) и *арбитры*, предназначенные для сопоставления реакций тестируемого компонента и реакций эталонной модели: *первичный арбитр* (часть эталонной модели) и *вторичный арбитр* (часть адаптера модели). Совокупность первичного и вторичного арбитров для выходного интерфейса называется *сопоставителем реакций* для данного интерфейса. Заметим, что предложенная архитектура основана на TLM – в ней имеется явное разделение коммуникаций (моделей входных и выходных интерфейсов) от вычислений (функциональной модели).

Теперь рассмотрим, как работает тестовый оракул. При получении очередного *модельного стимула* (*MS, Model Stimulus* на Рис. 3) от генератора стимулов проверяется соответствующее предусловие. Если предусловие нарушено (например, если нарушен протокол передачи входных данных), оракул фиксирует ошибку в тестовой системе. В противном случае модельный стимул передается в адаптер эталонной модели, который передает его как на эталонную модель, так и на тестируемый компонент. Во втором случае используется адаптер входного интерфейса, который *сериализует* абстрактное сообщение, представляющее модельный стимул, в потактово точную последовательность значений входных сигналов – *реализационный стимул* (*DS, Design Stimulus*). Эталонная модель эмулирует обработку стимула на некотором уровне абстракции, вычисляет набор *модельных реакций* (в явном виде или в форме ограничений[2]) (*MR, Model Reaction*) и передает их в очереди реакций соответствующих выходных интерфейсов.

Как только на каком-нибудь выходном интерфейсе тестируемого компонента детектор реакций обнаруживает реакцию (*DR, Design Reaction*), она с помощью адаптера интерфейса *десериализуется* в модельное представление (*CR, Checking Reaction*) – *проверяемая реакция*. После этого сопоставитель реакций пытается найти модельную реакцию из очереди реакций интерфейса, соответствующую проверяемой реакции. Первичный арбитр вычисляет множество реакций-кандидатов, основываясь на данных, предоставляемых эталонной моделью (временем поступления модельной реакции в очередь, ее приоритетом и др.). В некоторых ситуациях он выбирает ровно одну реакцию, но в общем случае, когда эталонная модель является достаточно абстрактной, первичный арбитр возвращает множество реакций (например, когда эталонная модель ожидает несколько реакций на одном выходном интерфейсе, но их

[2] *Ограничения* (в частности, *неопределенные значения*) используются, если требования допускают несколько корректных альтернатив для данных, содержащихся в реакции.

порядок не определен). В таких случаях дополнительно используется вторичный арбитр, который знает, как выглядит проверяемая реакция, и выбирает модельную реакцию из множества кандидатов, которая наиболее близка к проверяемой. Выбранная модельная реакция называется *эталонной реакцией* (с ней сравнивается проверяемая реакция).

Для сопоставления проверяемой реакции с одной из реакций-кандидатов вторичный арбитр использует *подсказку* – специальную функцию, определенную на множестве модельных реакций, возвращающую *идентификатор* сообщения (обычно в качестве подсказки используется поле сообщения, контрольная сумма или сообщение в целом). Пусть $CR$ – это проверяемая реакция, $\mathbf{C} = \{MR_i\}$ – множество реакций-кандидатов, предоставленное первичным арбитром, $h$ – функция подсказки. Если множество **C** пусто, фиксируется ошибка. Если множество **C** содержит ровно один элемент, вторичный арбитр возвращает этот элемент. В остальных случаях он вычисляет $h(CR)$ и пытается найти модельную реакцию $MR \in \mathbf{C}$ такую, что $h(MR) = h(CR)$[3]. Если таких реакций нет, фиксируется ошибка. Если найдена ровно одна реакция, она возвращается. Если найдено несколько реакций (возвращаемый подсказкой идентификатор не является уникальным), вторичный арбитр выбирает одну из них произвольным образом.

Выбранная эталонная реакция удаляется из очереди модельных реакций соответствующего выходного интерфейса и вместе с проверяемой реакцией передается компаратору реакций. В большинстве случаев компаратор проверяет равенство $CR = MR$, однако нередки ситуации, когда выполняемые проверки являются более сложными (например, когда определенность одних полей сообщения зависит от значений других полей). Если сравниваемые реакции не идентичны, фиксируется ошибка. В противном случае тестирование продолжается.

Ключевым в данной схеме являются механизмы сопоставления реакций – именно они позволяют использовать абстрактные модели для верификации компонентов на уровне RTL. Предлагаемый подход легко справляется с изменениями входных и выходных интерфейсов (для этого используются адаптеры интерфейсов). Следует особо отметить, что подход применим для всех уровней абстракции и позволяет уточнять временные свойства тестового оракула вплоть до потактовой точности (см. раздел «*Применение подхода на разных уровнях абстракции*»). Кроме того, алгоритм работы тестового оракула обеспечивает хорошую диагностику ошибок (см. раздел «*Классификация ошибок в аппаратуре*»).

[3] Вместо функции подсказки можно использовать и более общий подход, в котором на множестве модельных реакций вводится метрика ρ, а в качестве эталонной реакции выбирается $argmin_{MR \in \mathbf{C}}\{\rho(MR, CR)\}$.

## 5. Применение подхода на разных уровнях абстракции

В соответствии с предложенной архитектурой тестового оракула существует два основных параметра, определяющих уровень абстракции эталонной модели (и тестового оракула в целом): (1) *механизм обнаружения реакций* и (2) *механизм сопоставления реакций*. Первый из них принимает решение о том, выдал ли тестируемый компонент реакцию на том или ином выходном интерфейсе. В случае если реакции есть, второй механизм ищет модельную реакцию соответствующую обнаруженной реализационной реакции.

Механизм обнаружения реакций для одного выходного интерфейса может быть описан как булева функция $d$ (*detection*), зависящая от состояния эталонной модели ($S \in \mathbf{MS}$) и выходов устройства, относящихся к данному интерфейсу ($O \in \mathbf{DO}$)[4]:

$$d\colon \mathbf{MS} \times \mathbf{DO} \to \{true,\ false\}.$$

В общих словах, $d$-функции вычисляются на каждом такте тестирования и, как правило, проверяют значения *управляющих сигналов* (*стробов*). Если некоторая функция возвращает *true*, тестовая система считает, что обнаружена реакция и запускает механизм сопоставления реакций (осуществляет поиск эталонной реакции соответствующей обнаруженной реакции). Механизм сопоставления реакций (включающий первичные и вторичные арбитры) описывается как функция $a$ (*arbitration*), которая возвращает модельную реакцию или специальное значение *failed* (при невозможности сопоставления), в зависимости от состояния эталонной модели ($S \in \mathbf{MS}$) и проверяемой реакции ($CR \in \mathbf{MR}$):

$$a\colon \mathbf{MS} \times \mathbf{MR} \to \mathrm{MR} \cup \{failed\}.$$

Используя формализм $d$- и $a$-функций можно определить основные уровни абстракции, выделенные выше. В *моделях с потактовой точностью* для обнаружения реакций не используются выходы тестируемого компонента. Эталонная модель сама определяет моменты времени, когда на выходах устройства появляются реакции. Иными словами, $d$-функции выглядят следующим образом:

$$d\colon \mathbf{MS} \to \{true,\ false\}.$$

Другой отличительной чертой моделей с потактовой точностью является то, что эти модели не используют вторичные арбитры. Первичные арбитры (которые являются частью модели) всегда могут выбрать ровно одну реакцию

[4] Обычно используется подход на основе «черного ящика», поэтому внутреннее состояние тестируемого компонента здесь не рассматривается.

для того, чтобы проверить корректность обнаруженной реакции (точнее, очереди модельных реакций всех выходных интерфейсов содержат не более одной реакции, причем, как только модельная реакция заносится в очередь, сразу делается запрос на получения реакции от тестируемого компонента). Это свойство означает, что *a*-функции не зависят от обнаруженных реакций и никогда не возвращают *failed*:

$$a\colon \mathbf{MS} \to \mathbf{MR}.$$

На Рис. 4 показан фрагмент тестового оракула для потактово точной эталонной модели. Можно видеть, что детекторы реакций и вторичные арбитры отсутствуют.

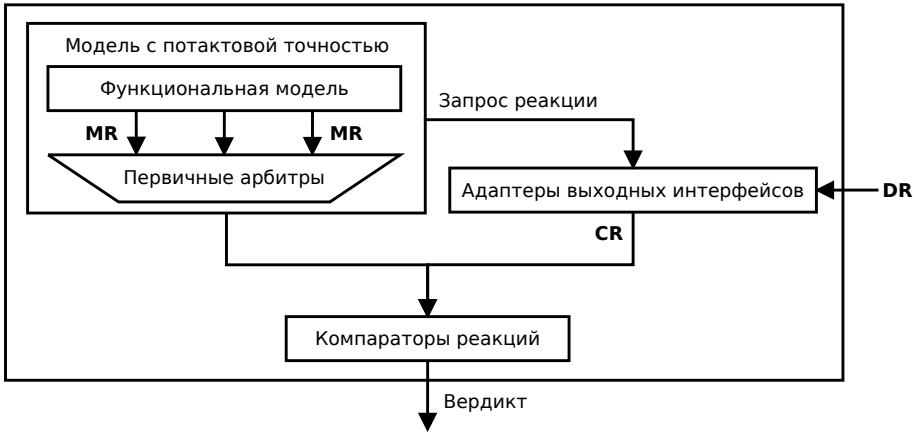



*Рис. 4. Тестовый оракул для эталонной модели с потактовой точностью*

В противоположность моделям с потактовой точностью, *модели без учета времени* не описывают временны*е* свойства компонентов аппаратуры. В частности, это означает, что у них нет первичных арбитров и, следовательно, вся работа по сопоставлению реакций выполняется вторичными арбитрами. Чтобы определить моменты времени, в которые возникают реакции, тестовые оракулы, основанные на таких моделях (и на моделях с приближенным моделированием времени) используют детекторы реакций. *d*-функции зависят только от выходов тестируемого устройства:

$$d\colon \mathbf{DO} \to \{true,\ false\}.$$

Тестовый оракул для моделей без учета времени представлен на Рис. 5. Можно видеть, что первичные арбитры отсутствуют, а все модельные реакции поступают напрямую к вторичным арбитрам.

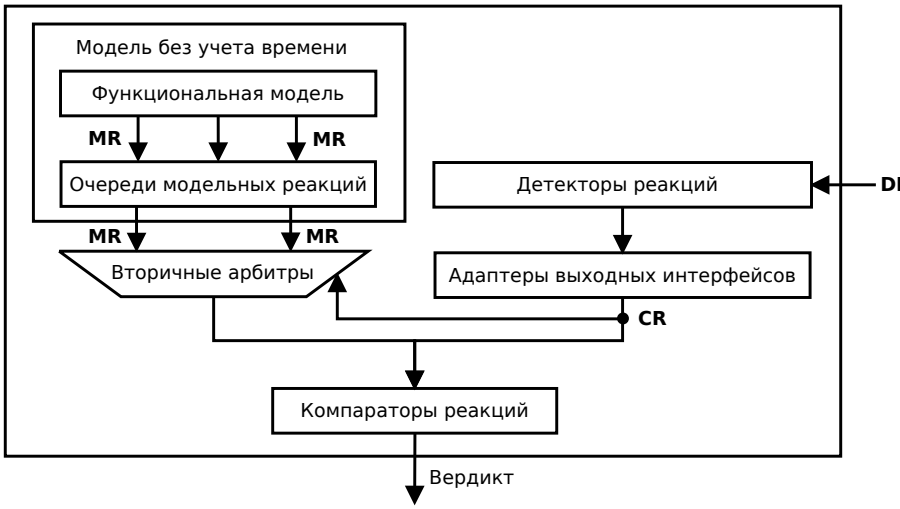



*Рис. 5. Тестовый оракул для эталонных моделей без учета времени*

Эталонные модели других уровней абстракции (*функциональных моделей с учетом времени*, *моделей коммуникации с учетом времени* и др.) находятся где-то посередине между моделями с потактовой точностью и моделями без учета времени. Поскольку они не являются достаточно точными в плане временны*х* свойств, они используют детекторы реакций, но в отличие от моделей без учета времени, в них применяются первичные арбитры. Разница между моделями промежуточных уровней в основном связана с различной точностью первичных арбитров.

Существует два основных класса арбитров: *детерминированные* (возвращающие не более одной реакции из очереди модельных реакций) и *недетерминированные* (которые в некоторых ситуациях возвращают несколько реакций). Детерминированные арбитры (в отличие от недетерминированных) задают линейный порядок модельных реакций для соответствующих выходных интерфейсов. Если степень недетерминизма ограничена (в некотором смысле), можно рассматривать промежуточный класс арбитров, определяющих приблизительный порядок модельных реакций.

Скажем несколько слов о переходах между уровнями абстракции. В начале процесса проектирования, когда требования существенно неполны, обычно используются функциональные модели без учета времени (A на Рис. 1). При уточнении требований эталонные модели естественным образом детализируются. Главная роль в повторном использовании тестовых систем отводится имеющейся в них модели коммуникаций (в частности, первичным арбитрам). В самом деле, коммуникации между компонентами обычно более подвержены изменениям, нежели вычисления.

В большинстве случаев эталонных моделей с приближенным учетом времени (B и C на Рис. 1) достаточно для достижения высокого качества тестирования. В некоторых случаях (например, для критичных компонентов или компонентов со сложной управляющей логикой) требуются модели с потактовой точностью (D, E, F) [9, 10]. Следует отметить, что уточнение эталонных моделей с приближенным учетом времени до потактовой точности требует значительных усилий (в этом случае необходимо учесть многие нюансы синхронизации, реализованные в тестируемом компоненте). Однако, это необходимо для ограниченного числа компонентов и только на завершающих этапах проектирования. В других случаях уточнение времени обычно связано с более точной настройкой первичных арбитров.

Рассмотрим фрагмент метода, являющегося частью эталонной модели, который описывает некоторую операцию некоторого компонента аппаратуры:

```
void DUV::operation(Interface &input, Message &stimulus, ...){
                                    ...
      // Применить стимул через адаптер входного интерфейса
                        recv(input, stimulus);
       // Вычислить реакции на некотором уровне абстракции
                                    ...
        // Сообщить тестовой системе об ожидаемой реакции
                        send(output, reaction);
                                    ...
  }
```

У приведенного метода есть, по крайней мере, два параметра: (1) *входной интерфейс* и (2) *модельный стимул* (*входное сообщение*). В начале метода вызывается метод `recv()`, который перегружается в адаптере эталонной модели и с помощью адаптера соответствующего интерфейса сериализует входное сообщение и подает его на тестируемый компонент. После этого вычисляются модельные реакции (на некотором уровне абстракции), которые добавляются в очереди соответствующих выходных интерфейсов путем вызова метода `send()`. В моделях с потактовой точностью можно использовать конструкцию типа `cycle()` для эмуляции такта работы компонента. Вызов `send()` в таких моделях приводит к немедленному чтению (десериализации) реакции с соответствующих выходов устройства и сравнению ее с эталонным значением. Для моделей без учета времени и моделей, учитывающих время приближенно, при вызове `send()` создается отдельный процесс, который, ожидает реакцию устройства, считывает ее, находит соответствующую ей модельную реакцию и сравнивает полученную реакцию с сопоставленной ей модельной реакцией.

## 6. Классификация ошибок в аппаратуре

Предложенная схема проверки реакций дает основу для классификации ошибок и их диагностики. Для определения различных типов ошибок в

компонентах аппаратуры рассмотрим *граф проверки реакций*, показывающий возможные альтернативы, имеющиеся в процессе проверки (см. Рис. 6).

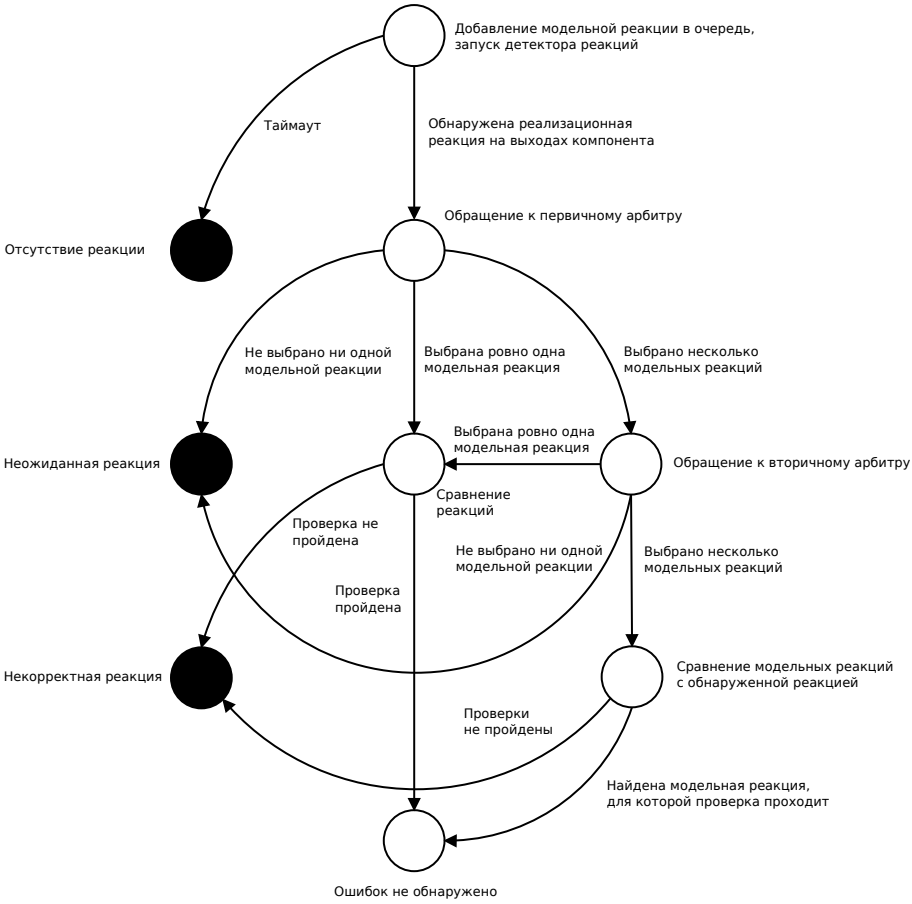



*Рис. 6. Граф проверки реакций*

Граф, показанный на Рис. 6, несколько упрощен, но он, тем не менее, демонстрирует основные классы ошибок в аппаратуре (они показаны черными кружками). Если тестовая система ожидает реакцию на некотором выходном интерфейсе тестируемого компонента, но реакция не возникает в течение определенного времени, фиксируется ошибка *отсутствия реакции* и сообщается информация об ожидаемой реакции (сообщение, интерфейс, ожидаемое время возникновения реакции). При обнаружении реакции, запрашивается сопоставитель реакций (первичные и вторичные арбитры).

Если он не может найти соответствующую модельную реакцию, тестовая система сообщает о *неожиданной реакции*, указывая также интерфейс и время возникновения реакции. Если сопоставитель выбирает ровно одну модельную реакцию, она считается эталонной и сравнивается с принятой реакцией. В противном случае тестовая система предупреждает о *недетерминированном поведении* и сравнивает все отобранные модельные реакции с реакцией устройства. Если есть хотя бы одно совпадение, проверка считается успешной. Иначе тестовая система сообщает о *некорректной реакции* и выводит список наиболее похожих модельных реакций.

## 7. Опыт применения подхода

Предложенный подход к организации тестовых оракулов был использован в ряде промышленных проектов по тестированию моделей аппаратуры (в основном отдельных модулей микропроцессоров [11]). Подход на практике показал возможность обнаружения достаточно сложных ошибок (включая трудно обнаруживаемые ошибки в управляющей логике), приемлемые трудозатраты и возможность повторного использования компонентов тестовой системы. Опыт использования подхода отражен в таблице 2.

| **Название тестируемого компонента** | **Максимальный используемый уровень абстракции** | **Минимальный используемый уровень абстракции** | **Стадия проектирования компонента** |
|---|---|---|---|
| Буфер трансляции виртуальных адресов | Детальный учет времени | Потактовая точность | Поздняя/завершающая |
| Модуль арифметики для чисел с плавающей точкой | Без учета времени | — | Поздняя/завершающая |
| Неблокируемая кэш-память второго уровня | Приблизительный учет времени | Детальный учет времени | Средняя/поздняя |
| Коммутатор данных северного моста | — | Потактовая точность | Завершающая |
| Устройство доступа к оперативной памяти | Без учета времени | Потактовая точность | Ранняя/средняя |
| Системный контроллер прерываний | Без учета времени | Приблизительный учет времени | Ранняя/средняя |

*Таблица 2. Опыт применения предлагаемого подхода*

На Рис. 7 показаны усредненные оценки трудозатрат на разработку тестовых оракулов на разных уровнях абстракции. Это достаточно грубое приближение, но оно позволяет планировать ресурсы на верификацию. Например, нам

потребовалось около 100% от начальных усилий для того, чтобы специфицировать приближенные временные свойства системного контроллера прерываний (около одного человеко-месяца). Добавленные свойства описывают порядок модельных реакций на каждом из двух выходных интерфейсов. Это позволило найти дополнительную ошибку в устройстве (одну из восьми). Другим примером является буфер трансляции адресов. Потребовалась одна человеко-неделя из трех для того, чтобы сделать эталонную модель потактово точной. При этом были найдены три дополнительные ошибки из десяти.


*Рис. 7. Трудозатраты на разработку тестовых оракулов для разных уровней абстракции*

## 8. Заключение

Очевидно, что использование более точных эталонных моделей позволяет находить более сложные ошибки в проектируемой аппаратуре. Также очевидно, что разработка более точных эталонных моделей требует привлечения бо́льших ресурсов. Выбор правильного уровня абстракции для тестирования того или иного компонента является трудной задачей, при решении которой необходимо учитывать множество факторов (критичность компонента, доступные ресурсы, затраты на исправление ошибок и многие другие). В ситуации, когда разные компоненты аппаратуры проверяются с разной тщательность и, более того, одни и те же компоненты тестируются по-разному в зависимости от этапа проектирования, полезно иметь универсальный подход к построению тестовых систем, применимый для разных уровней абстракции и позволяющий повторно использовать уже разработанные тестовые системы. Данная статья посвящена созданию такого подхода и рассматривает важный аспект разработки тестовых систем – создание эталонных моделей и основанных на них тестовых оракулов. Предложенная в работе архитектура тестового оракула позволяет

использовать абстрактные эталонные модели для верификации RTL-моделей аппаратуры и адаптировать тесты к изменениям интерфейсов и временны*х* свойств аппаратуры. В будущем мы планируем улучшить диагностику ошибок, чтобы адекватно описывать такие ошибки, как нарушение порядка реакций на одном интерфейсе и ошибки демультиплексирования.

## Литература


[1] J. Bergeron. “Writing testbenches: functional verification of HDL models”. Kluwer Academic Publishers, 2000.

[2] I. Bourdonov, A. Kossatchev, V. Kuliamin, A. Petrenko. “UniTesK test suite architecture”. In Proc. Formal Methods Europe (FME) 2002, pp. 77-88, 2002.

[3] L. Cai, D. Gajski. “Transaction level modeling: an overview”. In Proc. The International Conference on Hardware-Software Codesign and System Synthesis (CODES+ISSS) 2003, pp. 19-24, 2003.

[4] W. Lam. “Hardware design verification: simulation and formal method-based approaches”. Prentice Hall, 2005.

[5] C.-M.R. Ho. “Validation tools for complex digital designs”. PhD thesis, Stanford University, 1996.

[6] H.D. Foster, A.C. Krolnik, D.J. Lacey. “Assertion-based design”. Kluwer Academic Publishers, 2004.

[7] OVM User Guide – http://www.ovmworld.org.

[8] Я.С. Губенко, А.С. Камкин, М.М. Чупилко. “Сравнительный анализ современных технологий разработки тестов для моделей аппаратного обеспечения”. Труды Института системного программирования РАН, т. 17, с. 133-143, 2009.

[9] А.С. Камкин. “Метод формальной спецификации аппаратуры с конвейерной организацией и его приложение к задачам функционального тестирования”. Труды Института системного программирования РАН, т. 16, с. 107-128, 2009.

[10] M. Chupilko, A. Kamkin. “Developing cycle-accurate contract specifications for synchronous parallel-pipeline hardware: application to verification”. In Proc. The Baltic Electronic Conference (BEC) 2010, pp. 185-188, 2010.

[11] M. Chupilko, A. Kamkin, D. Vorobyev. “Methodology and experience of simulation-based verification of microprocessor units based on cycle-accurate contract specifications”. In Proc. The Spring Young Researchers’ Colloquium on Software Engineering (SYRCoSE) 2008, vol. 2, pp. 25-31, 2008.

# Mechanisms for functional testing of hardware models at different levels of abstraction

*A.S. Kamkin, M.M. Chupilko*
*{kamkin,chupilko}@ispras.ru*

**Abstract.** It is known that at different design stages different representations of a target system are used (a general description of the system's architecture is step by step concretized up to a physical layout). Depending on the project maturity engineers apply different verification methods and, in particular, methods for developing reference models, which are used to check the design correctness (at the beginning of the design process only abstract models are applied; when the process is close to the end, models accuracy is increased). Distinction in abstraction makes it difficult to reuse testbenches having been created at the early stages for verifying the same components some time later. In this paper, we suggest an approach to construct reference models and test oracles that eases testbench reuse thereby reducing verification costs.

**Keywords:** hardware design, functional verification, simulation-based verification, transaction level modeling

# Архитектура Linux Driver Verification

*Мутилин В.С., Новиков Е.М., Страх А.В., Хорошилов А.В., Швед П.Е.*
*mutilin@ispras.ru, joker@ispras.ru, strakh@ispras.ru, khoroshilov@ispras.ru, shved@ispras.ru*

**Аннотация.** В настоящей статье исследуются требования к построению архитектуры открытой системы верификации, которая могла бы предоставить площадку для экспериментов с различными методами статического анализа кода на реальном программном обеспечении и в то же время являлась полноценной системой верификации, готовой к индустриальному применению. По результатам обсуждения требований предлагается архитектура такой системы верификации и детально рассматриваются ее компоненты. В заключении описывается имеющийся опыт работы с предложенной архитектурой на практике и предлагаются пути дальнейшего развития.

**Ключевые слова:** статический анализ кода; правила корректности; верификация драйверов устройств; моделирование окружения драйвера; аспектно-ориентированное программирование; верификатор достижимости; визуализация трассы ошибки.

## *1. Введение*

Статический анализ кода позволяет проверять выполнимость определенных свойств программ на основе некоторого представления их исходного кода без необходимости реального выполнения программ. Основными преимуществами данного подхода являются то, что, во-первых, статический анализ не требует подготовки специального тестового окружения и тестовых данных и может осуществляться сразу после написания исходного кода программы; во-вторых, статический анализ позволяет рассмотреть сразу все пути выполнения программы, в том числе, редко встречающиеся и сложно воспроизводимые при динамическом тестировании.

Однако практическое применение статического анализа кода имеет некоторые ограничения. Наиболее существенным ограничением при использовании статического анализа кода является время проведения анализа. Дело в том, что современные программы являются очень большими и сложными. В свою очередь это приводит к тому, что выполнение полного статического анализа реальных приложений за разумное время практически невозможно, также как, впрочем, и проведение любого вида исчерпывающего тестирования. Поэтому при практическом применении статического анализа используются различные методы упрощения анализируемых моделей программ и эвристики, которые

позволяют получить результат за приемлемое время за счет снижения качества статического анализа. Ключевыми характеристиками качества анализа являются число ложных предупреждений и число пропущенных ошибок искомого вида. В зависимости от степени упрощения и целевого времени работы в рамках статического анализа кода можно условно выделить легковесные и тяжеловесные подходы.

Легковесные подходы нацелены на то, чтобы получать результаты быстро, сравнимо по порядку величины со временем компиляции анализируемого приложения. Для достижения такой высокой скорости данные подходы обычно используют анализ графа потока данных в сопровождении с множеством различных эвристик, что в конечном итоге отрицательно сказывается на качестве анализа, как в плане количества пропущенных ошибок, так и в плане количества ложных предупреждений. Причем часто уменьшение числа ложных предупреждений имеет даже больший приоритет, чем обнаружение всех ошибок, так как опыт использования инструментов статического анализа показывает, что при большом проценте ложных предупреждений общая эффективность их применения значительно падает. Несмотря на это, на сегодняшний день легковесные подходы развиты достаточно хорошо. Существует большое количество различных инструментов, которые их реализуют и широко применяются в индустриальной разработке программ. К наиболее успешным коммерческим инструментам относятся Coverity [1] и Klocwork Insight [2], академическим — Svace [3], [4], [5], Saturn [6], FindBugs [7], Splint [8] и др.

При использовании тяжеловесных подходов ограничению по времени работы придается существенно меньшее значение, хотя, тем не менее, время проверки должно оставаться в разумных пределах. Это позволяет использовать значительно меньше эвристик при интерпретации исходного кода программ и, соответственно, применять более качественные методы статического анализа кода, что в свою очередь приводит как к уменьшению числа ложных срабатываний, так и к увеличению числа обнаруживаемых ошибок. Однако на сегодняшний день тяжеловесные подходы мало используются при анализе реальных приложений. Существует большое количество академических проектов, которые предлагают различные реализации тяжеловесных подходов, например, SLAM [9], BLAST [10], CPAchecker [11], CBMC [12], ARMC [13] и др. Но в индустрии тяжеловесные методы нашли свое применение только в проекте Microsoft SDV [14]. Этот проект следует выделить особо, так как он предоставляет полноценный набор инструментов, позволяющих проводить тяжеловесный статический анализ кода драйверов операционной системы (ОС) Microsoft Windows. Предлагаемые инструменты используются в процессе сертификации драйверов и включены в состав Microsoft Windows Driver Developer Kit, начиная с 2006 года. Проект Microsoft SDV наглядно демонстрирует возможность применения тяжеловесного подхода для верификации реальных программ. Однако, Microsoft SDV является, во-первых, узкоспециализированным, так как по сути нацелен на

применение только для драйверов ОС Microsoft Windows, а во-вторых, закрытым, что не позволяет ни расширять область его применения, ни использовать для экспериментов в области алгоритмов статического анализа.

Применение других существующих тяжеловесных инструментов статического анализа на практике носит фрагментарный характер. По сути, не существует площадки, на которой можно было бы сравнить характеристики различных инструментов анализа на исходном коде программ, активно используемых в промышленности. В проекте *Linux Driver Verification* [15], [16], [17] сделана попытка построить такую площадку для инструментов (в первую очередь, тяжеловесных), предназначенных для анализа программ на языке Си, на примере драйверов устройств ОС Linux. В настоящей статье исследуются требования и предлагается архитектура такой площадки, которая одновременно должна являться полноценной системой верификации, готовой к индустриальному применению.

Настоящая статья построена следующим образом. В разделе 2 сформулированы требования к открытой системе верификации драйверов устройств ОС Linux и проведен анализ существующих инструментов статического анализа драйверов для ОС Linux и Microsoft Windows. Раздел 3 содержит подробное описание предложенной архитектуры системы верификации *Linux Driver Verification* и ее компонентов. В разделе 4 описан накопленный опыт использования системы верификации, в том числе с точки зрения предложенных архитектурных решений. В заключении подведен итог и представлены направления дальнейшего развития проекта.

## 2. Требования к открытой системе верификации и существующие решения

Одной из основных целей проекта *Linux Driver Verification* — является построение открытой площадки для экспериментов с различными методами статического анализа кода, в первую очередь, тяжеловесными, при верификации реальных программ. Для достижения этой цели система верификации должна предоставлять удобные средства для интеграции новых инструментов и сравнительного анализа их работы с различными настройками. В качестве целевых программ в проекте рассматриваются драйвера ОС Linux, тем не менее, архитектура инструментария должна предусматривать возможность последующего расширения и на другие приложения.

С точки зрения применения статического анализа кода драйверы ядра ОС Linux являются весьма привлекательной целью по следующим причинам:

- Драйверов устройств ОС Linux достаточно много и скорость их появления только возрастает с непрерывно растущей популярностью ОС Linux.

- Большинство драйверов публикуются вместе с исходным кодом, который является необходимым для статического анализа.
- Корректность драйверов является важной составляющей безопасности систем [18], так как драйверы работают с тем же уровнем привилегий, что и остальное ядро.
- Исходный код драйверов относительно простой, а потому может быть проанализирован с высоким уровнем качества.
- Драйверы достаточно небольшие по размеру, а потому можно предположить, что время проверки одного драйвера будет сравнительно невелико.

В то же время ОС Linux имеет ряд особенностей, которые должны учитываться при разработке системы верификации. Ядро ОС Linux является одним из самых динамично развивающихся проектов в мире. В среднем, начиная с ядра версии 2.6.30, которое было выпущено в середине 2009 года, каждый день добавляется порядка 9000 новых строк кода, удаляется порядка 4500 и модифицируется порядка 2000 строк кода [19]. Драйверы устройств занимают наибольшую часть (до 70%) ядра, кроме того именно в исходном коде драйверов содержится наибольшее число (более 85%) различных ошибок, приводящих к некорректной работе всей ОС, зависаниям и падениям [20], [21]. Обеспечивать надежность драйверов Linux вручную, даже несмотря на большое количество разработчиков (более 1000 человек на сегодняшний день [19]), весьма затруднительно ввиду огромного количества достаточно сложного исходного кода (более 13 млн. строк кода [19]), который должен удовлетворять достаточно большому числу разнообразных правил корректности, начиная от общих правил, которым должны подчиняться все программы на Си (ядро и драйверы ОС Linux разрабатываются на языке программирования Си) и заканчивая специфичными правилами, которые говорят о том, как драйверы должны использовать интерфейс ядра. При этом важным отличием Linux от многих других ОС является то, что интерфейс ядра с драйверами постоянно расширяется и не является стабильным, поэтому со временем появляются новые правила, а старые частично модифицируются. Как следствие система верификации драйверов ОС Linux должны быть готова развиваться одновременно с развитием ядра и предоставлять удобные возможности для настройки существующих и добавления новых правил корректности.

Второй целью проекта *Linux Driver Verification* является разработка системы верификации, готовой к индустриальному применению, что требует минимизации участия человека в настройке инструментов и максимально удобный интерфейс для их использования. С точки зрения минимизации участия человека, в первую очередь, необходимо автоматизировать извлечение информации о составе драйвера и настройках его компиляции из уже имеющихся данных, предназначенных для сборки драйвера. При этом

важно учитывать, что с одной стороны у драйверов есть много различных зависимостей, а с другой — что для эффективного применения статического анализа количество строк кода должно быть не очень большим.

Вторая потребность в автоматизации связана с отсутствием традиционной точки входа (иными словами, функции *main*) у драйверов устройств. Для большинства тяжеловесных подходов статического анализа кода наличие точки входа является необходимым условием в виду того, что они исследуют пути выполнения в программе, начиная от данной точки. Поэтому для проведения верификации драйверов требуется генерация модельного окружения драйвера в виде искусственной точки входа, где должны вызываться функции обработчики драйверов (например, функция инициализации драйвера, чтения с устройства и т.п.) на манер того, как это делается при реальном взаимодействии драйверов, ядра и оборудования.

Удобство использования инструментов включает в себя удобство запуска верификации и удобство анализа ее результатов. Причем последнее является наиболее значимым, так как анализ выявленных ошибок может занимать немалое время и требовать привлечения высококвалифицированных, а значит и дорогостоящих, специалистов.

Рассмотрим существующие проекты по верификации драйверов, основывающиеся на тяжеловесных методах статического анализа кода, в первую очередь, с точки зрения сформулированных требований к построению открытой системы верификации, подходящей для индустриального использования.

Наиболее полноценным образом подход реализуется в уже упоминавшемся ранее проекте Microsoft SDV [14]. Данный проект предоставляет широкие возможности по верификации драйверов ОС Microsoft Windows и используется как для анализа драйверов, входящих в состав ОС Windows, так и в существующей программе сертификации драйверов сторонних разработчиков. К особенностям подхода относятся следующие:

- Для создания окружения от пользователя требуется вручную аннотировать исходный код драйверов, указав в нем роли каждой из функций обработчиков.
- Проверяемые правила корректности формализуются с помощью языка SLIC [22], в котором связь с исходным кодом драйвера задается с помощью аспектно-ориентированных конструкций, перехватывающих вызовы функций ядра. В настоящее время уже выделен набор из примерно 200 правил, а в исследовательской версии была добавлена возможность добавления новых правил. Следует отметить, что в отличие от ядра Linux в ядре Microsoft Windows интерфейс меняется гораздо реже, поэтому проблема подстраивания под изменения ядра для Microsoft SDV не столь актуальна.

- Известно, что в собственных исследовательских целях разработчики Microsoft SDV могут подключать два статических анализатора кода SLAM и Yogi [23]. Подключение верификаторов сторонними разработчиками не предусмотрено.
- По результатам проведения анализа генерируются статистические данные о запуске и детальные трассы ошибок, для анализа которых разработаны специализированные графические инструменты.

Существует также несколько инструментов, использующих тяжеловесные методы статического анализа кода для верификации драйверов ядра ОС Linux: Avinux [24], разработка университета города Тюбинген (Германия) и DDVerify [25], разработка университета Карнеги-Меллон (США).

Особенности инструмента Avinux таковы:

- Получение исходного кода драйвера для последующей верификации происходит на основе встраивания в процесс сборки ядра путем модификации файлов, описывающих сборку. Однако, Avinux предоставляет возможность автоматической работы только с единичными препроцессированными файлами. Поэтому, например, верификация драйверов, состоящих из нескольких файлов возможна только вручную, так как информация о зависимостях теряется.
- Так же, как и в Microsoft SDV, для создания окружения драйвера требуется вручную задать функции инициализации, выхода и обработчиков. При этом код инициализации состояния входных параметров генерируется автоматически.
- Для задания правил используются аспектно-ориентированные конструкции похожие на конструкции SLIC.
- Инструмент интегрирован с единственным статическим верификатором CBMC [12].

Инструмент DDVerify, который также предназначен для верификации драйверов ядра ОС Linux, обладает следующими характерными особенностями:

- Информацию об исходном коде драйверов для проверки DDVerify получает, используя собственные файлы сборки без учета файлов сборки ядра. Поэтому инструмент не учитывает специфику компиляции ядра в полной мере.
- Для создания окружения используется модель ядра для некоторых типов драйверов. Разработчиками были написаны модели для трех типов, а всего их несколько десятков.
- Правила корректности задаются как часть модели ядра. Код ограничений, накладываемых правилом, задается вместе с кодом, описывающим семантику функции. В данном подходе можно проверять только те функции, которые привязываются к драйверу с

помощью линковки. Поэтому для использования DDVerify требуется существенно изменять заголовочные файлы ядра.

- Инструмент позволяет подключать два инструмента статического анализа кода: CBMC [12] и SATABS [26].

| Требования | Microsoft SDV | Avinux | DDVerify |
|---|---|---|---|
| Поддержка интеграции новых инструментов верификации | – | – | – |
| Переиспользование информации о параметрах сборки | + | Только для одного файла | – |
| Генерация модели окружения | По аннотации | Ручная | Для 3-х типов драйверов |
| Сопровождаемость в условиях непрерывного развитию ядра | Не требуется | + | – |
| Поддержка добавления новых правил корректности | + | + | ± |
| Автоматизация анализа результатов | + | – | – |

*Табл. 1. Сравнение существующих инструментов верификации драйверо.*

Таким образом, ни один из рассмотренных инструментов не позволяет полностью достигнуть заданных целей. Инструмент Microsoft SDV является закрытым и предназначен только для верификации драйверов ОС Microsoft Windows. Инструменты Avinux и DDVerify не подходят для широкомасштабного использования. Avinux требует описания функций обработчиков драйвера и не поддерживает драйверы, состоящие из нескольких файлов. DDVerify требует серьезной переработки собственного процесса сборки, заголовочных файлов и модели ядра для каждой новой версии ядра, что заметно усложняет сопровождение инструмента в условиях большого количества изменений в ядре ОС Linux. В случае обнаружения ошибки и Avinux, и DDVerify выдают пользователю текстовый вывод инструмента верификации, в то время как SDV обладает специализированными инструментами визуализации пути в программе, приводящей к обнаруженному нарушению правила корректности. Ни один из рассмотренных инструментов не поддерживает интеграцию сторонних инструментов верификации. В табл. 1 приведена сравнительная информация по рассмотренным инструментам с точки зрения выявленных требований к открытой системе верификации.

## 3. Архитектура системы верификации Linux Driver Verification

Архитектура *Linux Driver Verification* разрабатывалась для достижения описанных ранее целей: предоставить инфраструктуру для проверки драйверов ядра ОС Linux с высокой степенью автоматизации и возможность подключения различных инструментов, реализующих тяжеловесные подходы статического анализа кода.

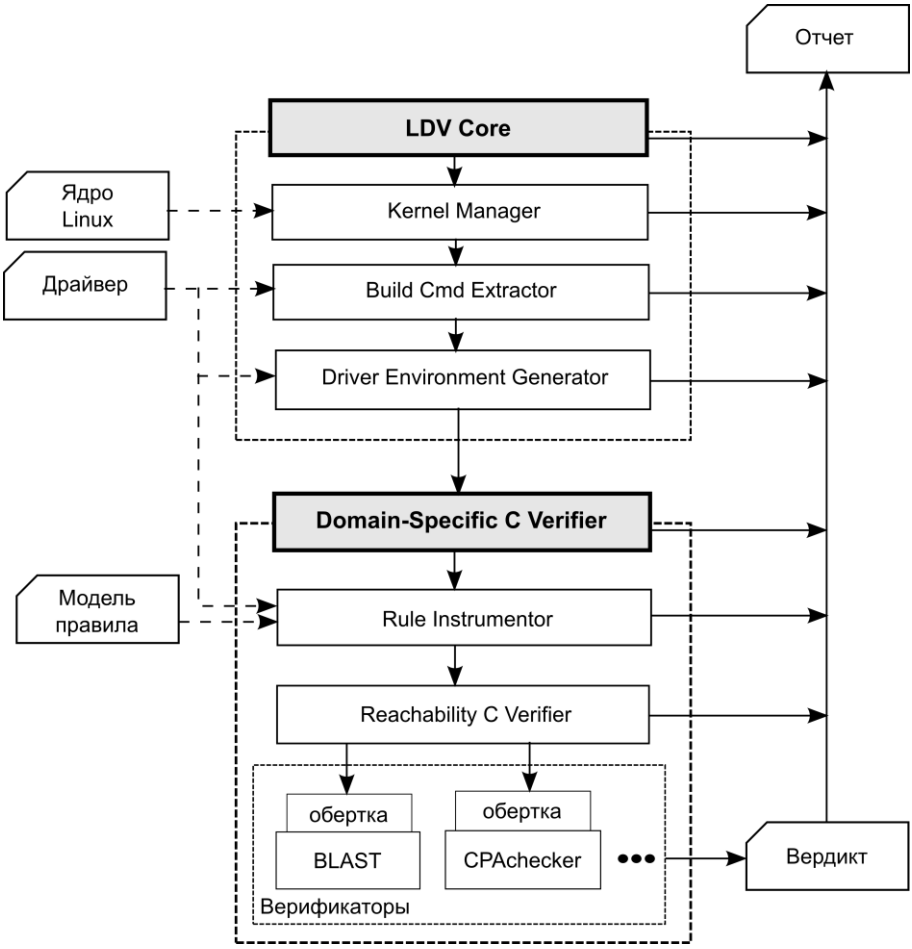



*Рис. 1. Компоненты архитектуры Linux Driver Verification*

Схематично архитектура изображена на рис. 1. Компоненты архитектуры изображены в центре. Стрелки указывают порядок, в котором соответствующие компоненты обрабатывают входные данные. Слева

изображены входные данные, которые предоставляются пользователем, однако, некоторые из них частично подготовлены разработчиками *Linux Driver Verification* (например, модели некоторых правил). Справа показан порядок формирования отчета по результатам верификации.

В общих чертах, процесс верификации драйверов происходит следующим образом. Запуск инструментария происходит через компонент *LDV-Core*. Данный компонент в начале вызывает компонент *Kernel Manager*, который на основе архива с исходным кодом ядра Linux создает на диске копию этого ядра со специально модифицированной подсистемой сборки (при последующих запусках эта копия переиспользуется).

Далее *LDV-Core* запускает процесс компиляции драйверов (внешних для данного ядра или входящих в его состав), и одновременно с этим процессом на основе модификаций сборки *Build Cmd Extractor* читает поток команд компиляции и линковки, выделяя из них те, которые относятся к верифицируемым драйверам.

Затем *LDV-Core* запускает *Driver Environment Generator*, который автоматически создает для каждого типа драйвера одну или несколько моделей окружения, добавляя код моделей к соответствующим файлам драйвера.

*Domain Specific C Verifier* уже не имеет представления о том, что поданный ему на вход код является драйверами ядра ОС Linux и может быть использован в неизменном виде для верификации произвольных программ на языке программирования Си. Данный компонент принимает абстрактную программу на языке программирования Си и передает ее компоненту *Rule Instrumentor*, чтобы тот встроил в нее код проверок указанных правил корректности.

Затем *Domain Specific C Verifier* передает модифицированный код конкретному инструменту верификации. Для этого он использует специальные обертки, реализации интерфейса взаимодействия конкретных верификаторов и инфраструктуры *Linux Driver Verification*. Вообще говоря, такие обертки предоставляются пользователем, но несколько из них входят непосредственно в состав инструментария. Обертки, получив окончательный список опций для вызова верификатора, непосредственно вызывают его.

На сегодняшний день в проекте *Linux Driver Verification* используется только один вид инструментов верификации, так называемые *верификаторы достижимости*, то есть инструменты, предназначенные для выявления нарушений правил корректности, выраженных в виде достижимости ошибочной точки в программе. В дальнейшем планируется реализовать поддержку и других классов верификаторов, например, решающих задачу завершаемости, и это не потребует внесения изменений в архитектуру системы верификации. Тем не менее, в рамках настоящей статьи мы будем рассматривать только верификаторы достижимости.

Инструмент верификации выносит вердикт, который может принимать одно из трех значений: SAFE, UNSAFE и UNKNOWN. Вердикт SAFE означает, что инструмент убедился в отсутствии нарушений проверяемого правила корректности при условии выполнения предположений, специфичных для данного инструмента. Вердикт UNSAFE говорит об обнаружении нарушения правила и сопровождается более детальной информацией о проблеме. В случае верификаторов достижимости, такой информацией является путь выполнения программы, который приводит к ошибочному состоянию. Вердикт UNKNOWN означает, что инструмент по тем или иным причинам не смог найти однозначного ответа на поставленный вопрос.

После этого полученный вердикт проходит все описанные выше стадии в обратном порядке. По ходу этого процесса вердикт дополняется отчетами о работе других компонентов, после чего формируется финальный отчет о проверке всего задания.

Компоненты между собой общаются с помощью потока команд. Изначально он является некоторым представлением команд сборки (компиляции и линковки), но, по мере обработки, модифицируется каждым компонентом. Компоненты могут изменять опции препроцессора, дописывать мета-информацию и даже подменять пути к файлам теми, по которым они располагают модифицированные файлы.

Все компоненты имеют определенный интерфейс и могут быть, при необходимости, заменены и/или модифицированы. Например, это может быть сделано при частичном изменении постановки задачи (см. *LDV-Git*).

Далее приведено подробное описание компонентов и интерфейсов *Linux Driver Verification*.

## 3.1. LDV-Core

На вход LDV-core поступает список драйверов для верификации, список ядер и список идентификаторов моделей правил. Данный компонент последовательно вызывает Kernel Manager, Build Cmd Extractor и Driver Environment Generator, описанные ниже.

## 3.2. Kernel Manager

В начале *LDV-core* для каждого из ядер вызывает *Kernel Manager*. Данный компонент в свою очередь создает переиспользуемую копию ядра, поданного ему на вход, одновременно дополняя процесс сборки соответствующего ядра вспомогательными командами. Последнее нужно с целью последующего сохранения информации о выполнении команд компиляции. В дальнейшем подготовленные ядра могут либо предоставлять свои внутренние драйвера для верификации, либо быть ядрами, с которыми собираются какие-либо внешние, не включенные в состав ядра, драйвера.

## 3.3. Build Cmd Extractor

После подготовки ядер *LDV-core* вызывает *Build Cmd Extractor*. Задача *Build Cmd Extractor* — выделить из процесса сборки ядра поток команд *cmdstream* и необходимые файлы с зависимостями для последующей верификации. Для внешних драйверов *Build Cmd Extractor* определяет наличие *Makefile* и *Kbuild* файлов, которые необходимы для сборки драйвера, и наличие соответствующих целей сборки.

Подкомпонент *Command Stream Divider* разделяет полученный после сборки *cmdstream* на множество небольших фрагментов (небольших файлов *cmdstream),* в каждом из которых собирается только один модуль. Этот подкомпонент особенно важен для анализа драйверов, включенных в ядро, поскольку подсистема сборки ядра компилирует их все вместе, не делая такого разделения.

## 3.4. Driver Environment Generator

Драйвер ядра ОС Linux состоит из одного или нескольких модулей. Эти модули не исполняются непосредственно как программы, а взаимодействует с ядром через специальный интерфейс.

Каждый модуль определяет две специальные функции: *module_init* и *module_exit*. Функция *module_init* вызывается при загрузке драйвера в ядро. В задачи функции входит инициализация состояния драйвера, а также регистрация специальных функций-обработчиков, предоставляющих интерфейс для работы с устройством. При наступлении определенных событий ядро вызывает функции-обработчики драйвера, которые обращаются непосредственно к физическому устройству для выполнения требуемых действий. Кроме того драйвер может регистрировать обработчики таймеров, прерываний, очередей ожидания и другие, которые также вызываются ядром при наступлении определенных событий.

Задача *Driver Environment Generator* — смоделировать окружение драйвера, включающее загрузку и выгрузку драйвера, а также смоделировать вызовы ядром ОС функций-обработчиков драйвера. Сгенерированная модель окружения выводится на языке Си в форме функции *main*, в которой реализованы вызовы функций драйвера. В качестве примера, для следующего драйвера (исходный код приведен частично):

```
 static struct file_operations fops = {
      .open = sample_open,
 };
 static int sample_open(struct inode *inode, struct file
*file);
 static int __init sample_init(void) { … }
 static void __exit sample_exit(void) { … }
 module_init(sample_init);
 module_exit(sample_exit);
```

компонент *Driver Environment Generator* генерирует следующую модель окружения (выделены вызовы функций драйвера):

```
#ifdef LDV_MAIN0_plain_sorted_withcheck
void ldv_main0_plain_sorted_withcheck(void) {
     struct inode * var_sample_open_0_p0;
     struct file * var_sample_open_0_p1;
     static int res_sample_open_0;
     if(sample_init() != 0)
          goto ldv_final;
res_sample_open_0=sample_open(var_sample_open_0_p0,
                 var_sample_open_0_p1);
     if(res_sample_open_0)
          goto ldv_module_exit;
ldv_module_exit:
     sample_exit();
ldv_final:
     check_final_state();
}
#endif
```

В данной модели продемонстрировано, как в начале объявляются переменные для использования в качестве аргументов функций (важно, чтобы получившийся код был корректной программой на Си), а затем происходит вызов инициализирующей функции *sample_init*. В том случае, если произойдет ошибка, дальнейшая работа модели бессмысленна, а потому в случае ошибки пропускаются вызовы остальных функций драйвера. В случае успеха инициализации, вызываются функции из полей стандартной структуры *fops* в некоторой заранее определенной последовательности, задающейся в *Driver Environment Generator*.

На данный момент *Driver Environment Generator* позволяет генерировать следующие модели окружения:

1. Фиксированная последовательность вызовов функций-обработчиков.
2. Произвольная последовательность вызовов функций-обработчиков ограниченной длины.
3. Произвольная последовательность вызовов функций-обработчиков неограниченной длины.

## 3.5. Domain Specific C Verifier

Проблемно ориентированный верификатор Си (*Domain Specific C Verifier*) предоставляет интерфейс верификации Си программ, не зависящий от способа описания моделей правил и от используемого верификатора. На вход ему поступает поток команд *cmdstream* и список идентификаторов моделей правил для проверки. Для каждого идентификатора модели *Domain Specific C Verifier*

вызывает *Rule Instrumentor*, который преобразует *cmdstream* в задание верификатору (подробнее о данном компоненте будет сказано в соответствующем разделе). Взаимодействие с верификатором происходит через специальную обертку *Reachibility C Verifier*, специфичную для каждого верификатора достижимости. Таким образом, обеспечивается достаточно простая взаимозаменяемость верификаторов.

Отметим, что информация, специфичная для ядер Linux, полностью скрывается в описании модели правила, поэтому компонент *Domain Specific C Verifier* (как впрочем и все нижележащие компоненты), вообще говоря, может быть применен и для других предметных областей в том случае, если предоставлены соответствующие описания моделей правил корректности и поток команд. Для примера, уже на сегодняшний день в базе моделей правил описано правило проверяющее, что в программе не нарушается ни один *assert*. Для проверки этого правила *Domain Specific C Verifier* может быть запущен без *LDV-core* и *Driver Environment Generator* на исходном коде любой программы на языке Си.

## 3.6. Rule Instrumentor

*Rule Instrumentor* — компонент, основное назначение которого — связывать формализованные в виде моделей правила корректности с исходным кодом проверяемой программы для последующей верификации с помощью некоторого инструмента статического анализа кода.

Следует обратить внимание на два важных момента. Во-первых, в настоящий момент в рамках проекта *Linux Driver Verification* на основе различных источников для драйверов уже выделено достаточно много правил (более 70), а в будущем будут постоянно появляться новые правила [17]. Обычно правило корректности описывает, каким образом драйверам следует использовать интерфейс ядра Linux для того, чтобы они работали корректно. Это описание в конечном итоге сводится к тому, как необходимо использовать некоторые специфичные конструкции языка Си, например, вызов функции блокировки или выделения памяти. Принимая во внимание еще тот факт, что процесс верификации должен быть очень хорошо автоматизирован, получается, что необходимо иметь возможности для достаточно легкой формализации правил корректности, а также последующей связи формализованного представления с исходным кодом программ. Близкая по виду задача решается в аспектно-ориентированном программировании (АОП). Также стоит отметить, что практически все инструменты, которые используют реализации тяжеловесных подходов статического анализа кода, в той или иной степени применяют АОП для записи правил.

*Rule Instrumentor* на вход поступает идентификатор модели и командный файл *cmdstream*. Информация о моделях хранится в базе данных моделей. Используя идентификатор, компонент получает описание необходимой модели, которое, по сути, состоит из путей к аспектным файлам и информации

для верификатора. Аспектные файлы пишутся на языке, подобном аспектно-ориентированному расширению языка программирования Си.

Для выполнения всего процесса инструментирования в целом в настоящее время используется набор инструментов LLVM [27]. Непосредственно для препроцессирования, разбора файлов с исходным кодом и одновременно выполняемого инструментирования на основе аспектных файлов используется LLVM GCC Front End, ввиду того, что GCC является основным для сборки ядра под ОС Linux.

В качестве примера далее приведено упрощенное формализованное описание правила корректности для функции блокировки и общая схема инструментирования исходного кода.

- «*Аспект 1*» – модельные состояние и функции:

```
int islocked = UNLOCKED; // Модельное состояние.
void own_mutex_lock() {   // Модельная функция.
  if (islocked == LOCKED) // Проверка правила.
    ERROR: abort();       // Ошибочная точка.
  islocked = LOCKED;        // Моделирование поведения функции.
}
```

- «*Аспект 2*» – связь конструкций исходного кода с «*аспектом 1*»:

```
before:call($mutex_lock(..)){// Перед вызовом mutex_lock
  own_mutex_lock();       // вызвать модельную функцию.
}
```

- Исходный код (drivers/pcmcia/cs.c):

```
…
// Вызов функции extern void mutex_lock(struct mutex *).
mutex_lock(&socket_mutex);
…
```

- Инструментированный исходный код:

```
…
// Вызов вспомогательной функции.
ldv_mutex_lock(&socket_mutex);
…
// Определение вспомогательной функции.
void ldv_mutex_lock(struct mutex *arg) {
  own_mutex_lock(); // Вызов модельной функции
  mutex_lock(arg);  // Вызов функции mutex_lock
}
```

## 3.7. Reachibility C Verifier

Компонент Reachibility C Verifier решает задачу преобразования задачи верификации из внутреннего представления в виде потока команд cmdstream в представление конкретного верификатора. Подающийся на вход поток команд уже содержит файлы для проверки со сгенерированными точками входа и ошибочными метками. Для каждого верификатора, который используется в LDV, пользователь должен написать обёртку, которая содержит:

1. Описание способа вызова верификатора.
2. Указание необходимости препроцессирования, упрощения, линковки входных файлов.
3. Интерпретацию ошибочных точек и точек входа и перевод этой информации на язык, понятный верификатору.
4. Интерпретацию специальных настроек верификатора.
5. Интерпретатор результатов анализа.

В настоящее время для целей тестирования и демонстрации возможностей реализованы 2 обертки соответственно для верификаторов достижимости BLAST и CPAchecker.

## 3.8. Пользовательский интерфейс Linux Driver Verification

### *3.8.1. LDV-manager*

LDV-manager предоставляет наиболее высокоуровневый интерфейс использования всего инструментария из командной строки. Данный компонент позволяет проверить некоторый набор драйверов (внутренних или внешних) для некоторого набора ядер по одному или нескольким правилам корректности. На выходе, в случае успешной работы, получается архив, содержащий результаты анализа, включая трассы ошибок и необходимые для их визуализации файлы с исходным кодом. Далее данный архив может быть загружен в базу данных и использован для анализа, например, с помощью сервера статистики или LDV-online, которые будут рассмотрены ниже.

### *3.8.2. LDV-online*

*LDV-online* предоставляет веб-интерфейс для проведения верификации драйверов и последующего анализа получаемых результатов по некоторому набору ядер и правил. *LDV-online* нацелен на разработчиков драйверов, которые в наименьшей степени заинтересованы во внутренностях проводимого процесса верификации. Поэтому данный компонент в наибольшей степени упрощает процедуру запуска верификации и анализа результатов.

Для верификации драйвера с помощью LDV-online достаточно загрузить архив с драйвером с помощью веб-интерфейса на специальный сервер, а затем дождаться результатов. По мере появления промежуточных результатов они поступают пользователю, и он может сразу же приступить к их анализу. Представление результатов для их анализа тоже предлагается в упрощённом виде; трассы ошибок и текстовые описания правил, тем не менее, будут предложены пользователю.

Для верификации драйвера с помощью *LDV-online* в настоящее время, по сути, нужно только загрузить архив с драйвером с помощью веб-интерфейса, а затем дождаться результатов. Набор ядер Linux и правил является предопределенным. По мере появления промежуточных результатов они поступают пользователю, и он может сразу же приступить к их анализу.

LDV-online поддерживает авторизацию и хранение истории запусков. История запусков авторизованных пользователей показывается только им, а неавторизованных видна всем.

### *3.8.3. LDV-Git*

LDV-Git — инструмент для непрерывного отслеживания изменений в драйверах ядра Linux, которое хранится в репозитории под управлением системы контроля версий Git. Данная система выбрана ввиду своей популярности при разработке ядра ОС Linux. Компонент LDV-Git, на основе истории изменений, вносимых в ядро Linux, формулирует и запускает задания для других компонентов Linux Driver Verification. Он автоматически определяет, какие драйвера следует перепроверить, а для каких результат верификации не изменится. LDV-Git проделывает полную проверку всех драйверов ядра один раз, а после внесения изменений в ядро запускает анализ только для тех драйверов, поведение которых могло измениться. Для изменившихся драйверов LDV-Git создает поток команд cmdstream, который сразу же передается генератору моделей окружения, а затем — Domain Specific C Verifier.

Определение драйверов для перепроверки делается на основе исходного кода и истории изменений, получаемой от Git; возможность этого обусловлена тем, что дальнейшая верификация также будет осуществляться по исходному коду драйверов.

*LDV-Git* предоставляет удобный интерфейс для пользователя, без его участия обрабатывая и сохраняя данные о предыдущих проверках. Основным способом использования *LDV-Git* предполагается периодическая проверка какой-либо ветви разработки ядра, например, при интенсивном внесении изменений в драйверы беспроводных устройств. В начале разработки ветви *LDV-Git* позволяет пометить некоторую ревизию как базисную; после этого влиять на выбор драйверов для проверки будут только изменения, внесенные в последующих версиях.

*LDV-Git* является примером того, как можно заменить несколько компонентов *Linux Driver Verification*, придав инструментарию не предусмотренную в нем изначально функциональность.

### *3.8.4. Сервер статистики*

Сервер статистики – это компонент, который предоставляет интерфейс для статистического анализа результатов и их изменений, происходящих с течением времени. Целесообразность использования сервера статистики происходит, прежде всего, из необходимости быстрого анализа огромного количества данных и оценки динамики изменения данных по мере развития ядра, драйверов и компонентов описываемой архитектуры, в том числе, и внешних (например, инструментов статического анализа кода).

В основе составления статистики и сравнения результатов лежит упорядочивание и группировка данных по некоторому ключу, состоящему из одного или нескольких полей таких, как, например, имя ядра, идентификатор модели, имя драйвера и т.д. Это позволяет кластеризовать данные и существенно уменьшить размер их представления вплоть до характерных размеров экрана компьютера.

Для визуального представления данных сервер статистики использует веб-интерфейс. Благодаря использованию гиперссылок сервер статистики позволяет также анализировать полные списки, отвечающие некоторому набору ограничений таких, как ключ и, например, ошибки драйвера или некоторая проблема в работе компонента. В результате имеется как возможность быстрой оценки большого количества данных, так и их детального анализа в зависимости от текущих задач (например, оценка качества работы верификатора или исследование ошибок в компоненте Rule Instrumentor).

*Linux Driver Verification* может быть использован различной целевой аудиторией такой, как разработчики компонентов, разработчики ядра, разработчики верификаторов достижимости и т.д. Как правило, запросы к представлению статистики у этих групп отличаются, поэтому сервер статистики предлагает различные заранее подготовленные профили представления.

### *3.8.5. Визуализатор трассы ошибки*

Когда инструмент верификации выдает вердикт UNSAFE, то есть находит в коде ошибку, её нужно показать пользователю. Причём пользователь не только должен понять, как эта ошибка проявляется, но и проверить, не является ли она ложным срабатыванием. Поэтому необходима визуализация трассы ошибки и связь её с контекстом — исходным кодом самого проверяемого драйвера.

| Error trace | | | Source code | | | |
|---|---|---|---|---|---|---|
| ☑ Function bodies | ☑ Blocks | Others. | doublelock.c. common.c | model0032.c | fs.h | slub_def.h |

```
    - entry_point();
    {
113   + tmp___1 =
my_init();
113   assert(tmp___1 == 0);
119   - rtmp0 =
misc_open(var0 /* inode */,
      {
 22     +
mutex_lock(&(my_lock) /*
lock */)
 23     - alock();
        {
 16       +
mutex_lock(&(my_lock) /*
lock *
        }
      }
    }
◄
►
```

```
 1 /* Check that double mutex
      lock is detected */ ▲
 2 #include <linux/kernel.h>
 3 #include <linux/module.h>
 4 #include <linux/mutex.h>
 5 #include <linux/major.h>
 6 #include <linux/fs.h>
 7
 8 static DEFINE_MUTEX(my_lock);
 9
10 static const struct
file_operations fops = {
11   .owner = THIS_MODULE,
12   .open  = misc_open
13 };
14
15 static void alock(void) {
16   mutex_lock(&my_lock);
17   mutex_unlock(&my_lock);
18 }
19
20 static int misc_open(struct
      inode * inode, stru
21 {
22   mutex_lock(&my_lock);
23   alock();
24   mutex_unlock(&my_lock);
25   return 0;
26 }
27
28 static int __init
      my_init(void)
34 {
▼
◄
►
```

*Табл. 2. Визуализация трассы ошибки*

Визуализатор трассы, таким образом, интерпретирует трассу, полученную от верификатора, преобразует её и связывает её с исходным кодом. Для представления результата используются средства разметки HTML и Javascript.

Для различных конструкций трассы ошибки (например, вызов модельных функций, проверка условий и т.д.) используются различные стили и цвета. Исходный код синтаксически подсвечивается. Та часть конструкций трассы ошибки, которая представляет небольшой интерес для ее анализа, автоматически скрывается.

Пример работы визуализатора трассы ошибки представлен в табл. 2 (представление значительно упрощено в виду ограниченности средств печати).

### 3.9. Тестовые наборы

По мере развития и постепенной стабилизации интерфейсов компонентов, возникла необходимость в различных регрессионных тестовых наборах. Основными поставщиками данных для тестовых наборов являются драйверы (как внешние, так и внутренние драйверы ядра). Еще один вариант использования тестовых наборов заключается в слежении за изменениями во времени работы различных компонентов, причем, как разрабатываемых, так и внешних. Здесь наибольший интерес представляют различные статические верификаторы (на практике, их работа занимает основную часть времени). В настоящий момент тестовые наборы уже используются для контроля времени работы компонентов, а также для сравнения различных верификаторов.

## *4. Апробация*

Представленная архитектура *Linux Driver Verification* была реализована и для реализации была произведена апробация на ядрах Linux с версии 2.6.30 по 2.6.37. При этом использовались как драйверы, входящие в состав ядра и компилируемые как модули (порядка двух тысяч), так и внешние драйвера (несколько десятков). Для прогонов были выбраны девять формализованных правил.

Результаты показывают, что верификация достаточно большого количества драйверов возможна за приемлемое время. Так, например, общее время работы при проверке всех драйверов ядра 2.6.31.6, сконфигурированного для сборки всех модулей (*allmodconfig*), на одном правиле составляет около 17 часов. Зависимость среднего времени ожидания ответа от размера верифицируемого драйвера представлена на рис. 2. Среднее время считается по группам драйверов. В первую группу попадают драйверы размером до 2 тыс. строк кода, во вторую — с размером от 2 до 4 тыс. строк кода, в третью — от 4 до 6 тыс. и т.д. Для прогонов был установлен лимит по времени в 15 минут (900 секунд).

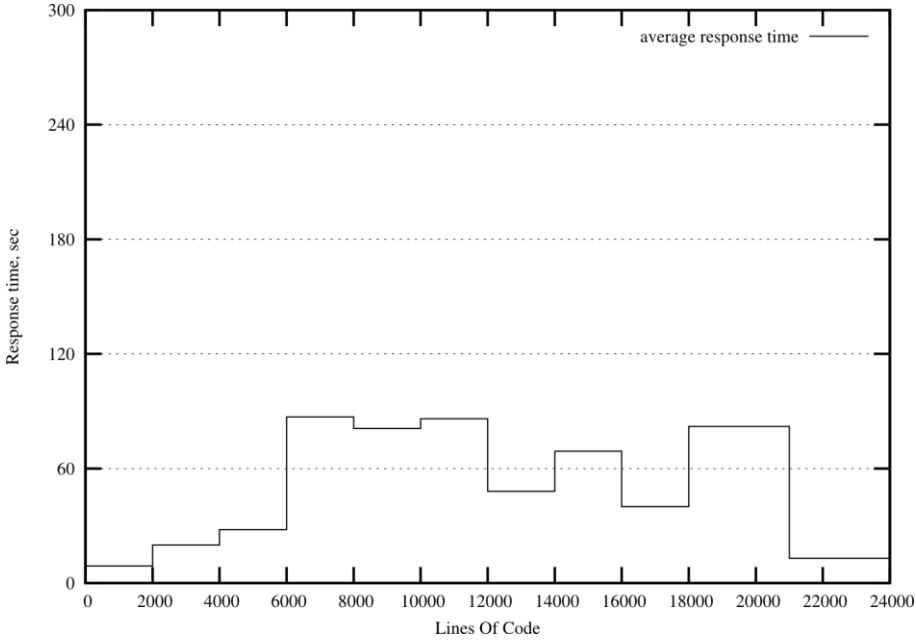



*Рис. 2. Зависимость среднего времени ответа от размера исходного кода драйверов*

Количество успешных результатов верификации составляет 86%, соответственно для 14% не удалось установить выполнено ли правило корректности из-за внутренних ошибок компонентов и наложенного ограничения по времени. Кроме того, прогоны показывают, что наиболее ресурсоемким компонентом является *Reachibility C Verifier*, на него приходится 90-95% процентов временных затрат.

Характерный размер драйвера, который успешно может быть проверен с помощью *Linux Driver Verification* можно оценить на основе графика, представленного на рис. 3. На нем показано распределение доли успешно завершившихся запусков по размеру верифицируемого драйвера в строках кода. Драйверы сгруппированы через каждые 2 тыс. строк кода; так же, как и на рис. 2. Сплошной линией показано общее количество драйверов, лежащих в данной группе. Пунктирная линия показывает долю (в процентах) успешно завершившихся запусков среди драйверов данной группы. Из этого графика видно, что для выбранного правила (проверка двойных блокировок) можно ожидать, что драйвер размером менее чем 12 тыс. строк кода, будет успешно проверен с не менее, чем 50% вероятностью. При этом ограничения на ресурсы составили 15 минут времени и 2 Гб памяти. Аномально высокие

показатели при размерах, превышающих 24 тыс. строк кода, обусловлены тем, что таких драйверов очень мало, а статическому верификатору было достаточно самой грубой абстракции для доказательства отсутствия достижимых ошибочных точек.

## 4.1. Найденные ошибки в драйверах

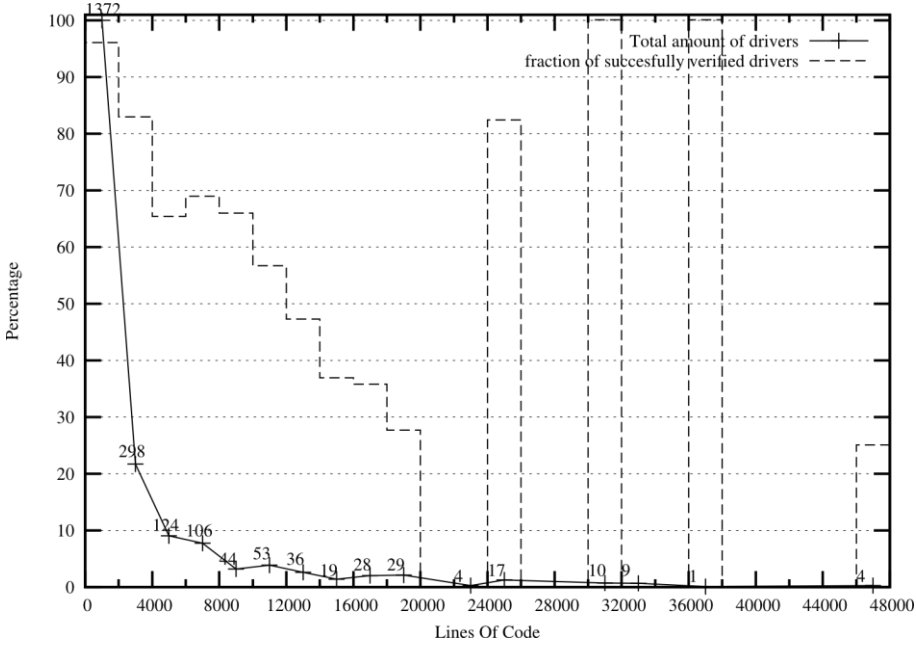



*Рис. 3. Зависимость доли успешных результатов верификации от размера верифицируемого драйвера в строках кода*

Самые простые из найденных ошибок - разыменование нулевых указателей, потенциальные возможности переполнения буфера и утечки памяти в драйверах fs/cifs/cifsencrypt.c и security/selinux/hooks.c. По последнему разработчикам было отправлено исправление (патч) и ошибка была исправлена. Некоторые найденные ошибки на практике встречаются редко и их трудно обнаруживать. Например, в драйвере drivers/media/video/cafe_ccic.c после неудачного запроса прерывания не освобождался mutex, а в драйвере drivers/media/video/hdpvr/hdpvr-core.c — mutex не освобождался после неудачной отсылки управляющего сообщения подсистемы USB. По этим двум драйверам были направлены исправления, которые были включены в последние версии ядра. Также были найдены ошибки, связанные с вызовами некоторых функций в запрещенном контексте (drivers/char/isicom.c и drivers/net/znet.c), двойная блокировка (drivers/hid/hidraw.c), разблокировка не

захваченного mutex (drivers/net/wireless/iwlwifi/iwl3945-base.c, drivers/input/input.c); неправильное использование функции module_put в drivers/mtd/mtd_blkdevs.c.

В общей сложности, из 25 выявленных нарушений, 21 нарушение было признано разработчиками ядра как ошибки, требующие исправления. Важно отметить то, что эти ошибки были обнаружены на достаточно малом наборе правил (5 правилах) при тестовых прогонах, в то время как потенциал использования *Linux Driver Verification* гораздо шире.

## 4.2. Гибкость и расширяемость

За время разработки ядер с версии 2.6.30 по 2.6.37 был изменен интерфейс ядра, используемый драйверами. Например, макрос *spin_lock* был заменен на *static inline* функцию, при этом его семантика не менялась. Кроме того, были добавлены специальные конструкции, поддерживаемые только новыми компиляторами *gcc*. Несмотря на эти изменения, трудозатраты на поддержание новых версий ядра в представленной архитектуре *Linux Driver Verification* оставались минимальными. Например, для того, чтобы учесть изменения в интерфейсе ядра, потребовалось добавить привязку к новому интерфейсу в связующий аспектный файл соответствующей модели.

Архитектура обладает высокой степенью расширяемости. Добавление новых правил осуществляется с помощью написания аспектных файлов на языке подобном аспектно-ориентированному расширению языка программирования Си. Использование аспектно-подобных описаний хорошо зарекомендовало себя в инструментах Microsoft SDV и Avinux. Наш опыт также свидетельствует о том, что это сравнительно простой интуитивный язык, с помощью, которого удобно поддерживать актуальность моделей для стремительно меняющегося ядра Linux.

Добавление верификаторов достижимости осуществляется заданием специальной обертки, описывающей связь соответствующего верификатора с компонентами *Linux Driver Verification*. На данный момент обертки были написаны для двух верификаторов достижимости BLAST и CPAchecker. В ближайшем будущем планируется расширение списка поддерживаемых верификаторов.

Коллектив разработчиков, которые также являются и авторами данной статьи, продолжает поддерживать и развивать архитектуру *Linux Driver Verification* по различным направлениям, а также приглашает желающих принять участие в данном процессе. Текущее состояние архитектуры можно увидеть на странице проекта [28]. Также с данной страницы возможно скачать полный набор компонентов, прочитать инструкции, установить требуемые компоненты и использовать их для решения своих задач.

## 5. Направления дальнейшего развития

Развитие архитектуры *Linux Driver Verification* в краткосрочной перспективе планируется по следующим направлениям:

1. Распараллеливание многих тяжеловесных подпроцессов позволит значительно повысить скорость верификации при более полном использовании современных вычислительных мощностей. Реализованная на сегодняшний день архитектура во многом способствуют проведению распараллеливания за счет аккуратной функциональной декомпозиции компонентов.

2. Расширение базы данных правил и увеличение числа формализованных правил с целью повышения практической ценности результатов работы системы верификации для разработчиков драйверов.

3. Дальнейшее исследование и улучшение используемых инструментов статического анализа кода для достижения большего уровня качества и увеличения скорости анализа. Помимо этого планируется проведение исследований возможностей других инструментов верификации.

4. Повышение конфигурируемости за счет автоматического анализа различных конфигураций ядра, в том числе, различных архитектур; автоматического подбора более подходящего окружения драйверов и статического анализатора с опциями для получения более точных результатов верификации за меньшее время.

## 6. Заключение

В статье представлена архитектура открытой системы верификации *Linux Driver Verification*, которая разработана с тем, чтобы предоставить площадку для экспериментов с различными методами статического анализа кода на реальном программном обеспечении и в то же время стать полноценной системой верификации, готовой к индустриальному применению в области верификации драйверов ОС Linux. Первоначальный опыт использования системы верификации *Linux Driver Verification* показывает, что предложенная архитектура позволяет решать поставленные задачи и, кроме того, является достаточно гибкой для безболезненной реализации заранее не предусмотренных вариантов использования. Тем не менее, еще предстоит сделать немало шагов для внедрения системы верификации в процессы разработки драйверов ОС Linux и для привлечения сторонних исследователей к экспериментам с новыми алгоритмами верификации на основе *Linux Driver Verification*, чтобы качество предложенной архитектуры получило объективную оценку, а цели проекта оказались достигнутыми.

**Литература**


[1] Инструмент *Coverity*. http://www.coverity.com/products/static-analysis.html.
[2] Инструмент *Klocwork Insight*. http://www.klocwork.com/products/insight/.
[3] В.С. Несов, О.Р. Маликов. Использование информации о линейных зависимостях для обнаружения уязвимостей в исходном коде программ. Труды Института системного программирования РАН, том 9, стр. 51-56, 2006.
[4] В.С. Несов, С.С. Гайсарян. Автоматическое обнаружение дефектов в исходном коде программ. Методы и технические средства обеспечения безопасности информации: Материалы XVII Общероссийской научно-технической конференции. СПб.: Изд-во Политехн. Ун-та, с. 107, 2008.
[5] V. Nesov. Automatically Finding Bugs in Open Source Programs. Proceedings of the Third International Workshop on Foundations and Techniques for Open Source Software Certification, Volume 20, pages 19-29, 2009.
[6] Инструмент *Saturn*. http://saturn.stanford.edu/.
[7] Инструмент *FindBugs*. http://findbugs.sourceforge.net/.
[8] Инструмент *Splint*. http://www.splint.org/.
[9] T. Ball, E. Bounimova, R. Kumar, V. Levin. SLAM2: Static Driver Verification with Under 4% False Alarms. FMCAD, 2010.
[10] D. Beyer, T. Henzinger, R. Jhala, R. Majumdar. The Software Model Checker Blast: Applications to Software Engineering. Int. Journal on Software Tools for Technology Transfer, 9(5-6):505-525, 2007.
[11] D. Beyer, M. Keremoglu. CPAchecker: A Tool for Configurable Software Verification. Technical report SFU-CS-2009-02, School of Computing Science (CMPT), Simon Fraser University (SFU), January 2009.
[12] E. Clarke, D. Kroening, F. Lerda. A Tool for Checking ANSI-C Programs. In Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, LNCS, Vol. 2988/168-176, 2004.
[13] Инструмент *ARMC*. http://www.mpi-sws.org/~rybal/armc/.
[14] T. Ball, E. Bounimova, V. Levin, R. Kumar, J. Lichtenberg. The Static Driver Verifier Research Platform. CAV 2010, 2010.
[15] A. Khoroshilov, V. Mutilin. Formal Methods for Open Source Components Certification. OpenCert 2008 2nd International Workshop on Foundations and Techniques for Open Source Software Certification 52-63 Milan, 2008.
[16] A. Khoroshilov, V. Mutilin, V. Shcherbina, O. Strikov, S. Vinogradov, V. Zakharov. How to Cook an Automated System for Linux Driver Verification. SYRCoSE'2008 2nd Spring Young Researchers' Colloquium on Software Engineering, Volume 2 11-14 St. Petersburg May 29-30, 2008.
[17] A. Khoroshilov, V. Mutilin, A. Petrenko, V. Zakharov. Establishing Linux Driver Verification Process. PSI 2009 Perspectives of System informatics 2009 LNCS, Vol. 5947/ 165-176, 2009.
[18] В.П.Иванников, А.К. Петренко. Задачи верификации ОС Linux в контексте ее использования в государственном секторе. Труды Института системного программирования РАН, том 10, стр. 9-14, 2006.
[19] G. Kroah-Hartman, J. Corbet, A. McPherson. Linux kernel development. http://www.linux-foundation.org/publications/linuxkerneldevelopment.php, 2008.
[20] A. Chou. An Empirical Study of Operating System Errors. Proc. 18th ACM Symp. Operating System Principles, ACM Press, 2001.

[21] M. Swift, B. Bershad, H. Levy. Improving the reliability of commodity operating systems. In: SOSP ’03: Proceedings of the nineteenth ACM symposium on Operating systems principles, New York, NY, USA, ACM 207–222, 2003.
[22] T. Ball, S. K. Rajamani. Slic: A specification language for interface checking of C. Technical Report MSR-TR-2001-21, Microsoft Research, 2001.
[23] N. Beckman, A. Nori, S. Rajamani, R. Simmons. Proofs from tests. In ISSTA’08: International Symposium on Software Testing and Analysis, pages 3–14, 2008.
[24] H. Post, W. Küchlin. Integration of static analysis for linux device driver verification. The 6th Intl. Conf. on Integrated Formal Methods, IFM 2007, 2007.
[25] T. Witkowski, N. Blanc, D. Kroening, G. Weissenbacher. Model checking concurrent linux device drivers. In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE '07), pages 501-504, 2007.
[26] E. Clarke, D. Kroening, N. Sharygina, K. Yorav. SATABS: SAT-based Predicate Abstraction for ANSI-C. In Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, LNCS, Vol. 3440/570-574, 2005.
[27] Набор инструментов *LLVM*. http://llvm.org/.
[28] Инструмент *Linux Driver Verifier*. http://forge.ispras.ru/projects/ldv.

## Linux Driver Verification Architecture

*A.V. Khoroshilov, V.S. Mutilin, E.M. Novikov, P.E. Shved, A.V. Strakh*
*khoroshilov@ispras.ru, mutilin@ispras.ru, joker@ispras.ru, shved@ispras.ru, strakh@ispras.ru*

**Abstract.** The paper discusses requirements to a twofold verification system that should be an open platform for experimentation with various verification techniques as well as an industrial-ready domain specific verification tool for Linux device drivers. An architecture of a verification system implementing the requirements is presented and its components are described in details. Finally, we discuss the first experience of usage of the system and evaluate directions for its further development.

**Keywords:** static analysis; safety rule; device driver verification; driver environment model; aspect oriented programming; domain specific verification; reachability verifier; error trace visualizer.

# Транзакционные параллельные СУБД: новая волна

*С.Д. Кузнецов*
*kuzloc@ispras.ru*

**Аннотация.** Возможность построения неограниченно масштабируемых кластерных систем привела к резкой активизации исследований и разработок архитектур систем управления данными без совместного использования ресурсов. Образовались два основных фронта: "NoSQL", где отрицаются основные принципы, свойственные СУБД, и "один размер непригоден для всех", где упор делается на специализацию систем при сохранении важнейших свойств СУБД. Особенно интересным является противостояние этих фронтов в области "транзакционных" систем управления данными. Опираясь на "теорему" CAP Эрика Брювера (Eric Bruwer), представители лагеря NoSQL отказываются от обеспечения в своих системах традиционных свойств ACID в транзакциях баз данных. В этой статье обсуждается суть "теоремы" Брювера и обосновывается, что она не имеет отношения к свойствам ACID. Рассматриваются наиболее интересные современные исследовательские работы, обеспечивающие классические ACID-транзакции в параллельных средах без общих ресурсов, а также наиболее здравые подходы, в которых из чисто прагматических соображений свойства ACID частично ослабляются (но совсем не в связи с "теоремой" CAP).

**Ключевые слова:** транзакционные массивно-параллельные СУБД, "один размер непригоден для всех", NoSQL, ACID-транзакции.

## *1. Введение*

Происходящие важные изменения в области компьютерных аппаратных средств, а именно, возможность сравнительно дешевого построения неограниченно горизонтально масштабируемых кластерных систем (будь то системы, основанные на использовании публичных или частных облачных инфраструктур, или кластеры, конфигурируемые традиционным образом) привели к резкой активизации исследований и разработок пригодных для использования в таких средах архитектур систем управления данных без совместного использования ресурсов (shared nothing). Работы ведутся на двух основных фронтах. (Я не буду здесь говорить про компании-производители основных SQL-ориентированных СУБД, которые всегда стараются решать все проблемы за счет своих собственных, накопленных в течение десятилетий возможностей.)

Первый фронт составляют основные поставщики Internet-услуг, такие как компании Google, Yahoo!, Facebook, Amazon и т.д., которые для своих собственных нужд и для обеспечения публично доступных "облачных" служб разрабатывают средства управления данными, идеологически и архитектурно отходящие от традиционных канонов сообщества баз данных. Во многом именно с деятельностью этих компаний связано возникновения понятия "NoSQL", т.е. (в исходном своем смысле) отказа от существующих решений.

На втором фронте, с моей точки зрения, находятся последователи концепции Майкла Стоунбрейкера (Michael Stonebraker) "один размер непригоден для всех" [1], в которой, по сути, основными являются два соображения: (a) прошло время универсальных систем управления базами данных, пригодных для поддержки приложений всех возможных разновидностей, и (b) при разработке новых систем необходимо пользоваться всеми полезными технологиями и идеями, накопленными в сообществе баз данных. К этому лагерю относятся исследовательские группы ряда университетов США, компании VoltDB, Vertica, Asterdata и т.д.

Следует обратить внимание на то, что представители обоих фронтов сходятся по первому пункту, т.е. их объединяет желание создать новые средства управления данными, специализированные для некоторых классов приложений и поддерживающие эти приложения более эффективно, чем универсальные СУБД. Второй пункт их разделяет: люди из первого лагеря считают (немного утрированно), что технологии баз данных являются тяжким грузом прошлого, а второй лагерь относится к ним, как к ценному наследию, жертвовать которым нельзя.

На обоих фронтах ведутся работы в двух наиболее важных направлениях управления данными – аналитические системы управления данными (т.е. системы, пригодные для построения над ними приложений категории OLAP (online analytical processing, оперативная аналитическая обработка данных)) и транзакционные системы управления данными (т.е. системы, пригодные для построения приложений категории OLTP (online transaction processing, оперативная обработка транзакций)). В первом направлении представителей двух рассматриваемых лагерей в прошлые годы разделяло, прежде всего, отношение к NoSQL-технологии анализа данных *MapReduce* [2]. Не так давно представители лагеря NoSQL считали, что MapReduce заменит в динамических кластерных архитектурах параллельные аналитические системы баз данных, а исследователи из второго лагеря обвиняли создателей MapReduce к возврату в дремучее прошлое, когда технология баз данных не существовала [3-4]. Однако это время, как кажется, миновало. По крайней мере, сообщество баз данных приняло технологию MapReduce и научилось ее использовать [5], и я (может быть, временно) считаю эту тему закрытой (в [5] можно найти много полезных ссылок на эту тему).

## 1.1. Цель статьи

Цель этой статьи состоит в том, чтобы разобраться в происходящем в области средств управления данными категории OLTP. В общих словах, ситуацию можно охарактеризовать следующим образом. Значительную часть современных Internet-приложений составляют *транзакционные* приложения (онлайновые магазины, аукционы, системы резервирования и т.д.).

Успешные Internet-компании быстро растут, и им просто необходимо уметь легко, быстро и эффективно масштабировать свои системы управления данными, причем, как правило, речь идет именно о *горизонтальном* масштабировании, при котором при увеличении числа узлов в используемом кластере линейно возрастает пропускная способность системы. Другими словами, способность *горизонтального масштабирования (scaling out)* системы управления данными является необходимым условием развития бизнеса. Транзакции в таких приложениях обычно являются очень короткими и состоят из простых операций, так что обычно пользователей вполне устраивает время реакции неперегруженного запросами приложения.

Однако временами при взаимодействии пользователя с онлайновым приложением могут возникать задержки из-за недоступности каких-либо ресурсов на уровне управления данными. В условиях жесткой конкуренции на рынке онлайновых приложений такие задержки могут оказаться губительными для бизнеса. Поэтому не менее важным требованием к системе управления данными является *доступность (availability)*, гарантирующая отсутствие (или, как минимум, по возможности снижающая вероятность возникновения) таких задержек. Хуже всего на отношение клиентов к Internet-компании действуют сообщения типа *service unavailable*. Поэтому многие онлайновые компании готовы согласиться с тем, что в некоторых случаях результаты выполнения транзакций системой управления данными будут некорректными, если при этом доступность системы будет максимально возможной.

В [6] даже выстраивается некоторая "теоретическая" база, оправдывающая перед пользователями возможное некорректное поведение приложений тем, что "потребность в извинениях возникает в любом бизнесе". Другими словами, поскольку в любом бизнесе могут возникать ошибки, за которые придется извиняться перед клиентами, то почему бы не отнести к числу таких ошибок и подобное поведение приложений (рано или поздно наличие некоррректности будет установлено, и соответствующий пользователь получит моральную (или даже материальную) компенсацию).

Более серьезным "теоретическим" основанием NoSQL-разработок, в которых общепринятые полезные свойства систем управления данными приносятся в жертву доступности этих систем, является так называемая "теорема CAP", впервые сформулированная Эриком Брювером (Eric Brewer) [7]. (Здесь и далее в тесте я заключаю слово *теорема* в кавычки, поскольку утверждение, названное Брювером теоремой, таковым я признать не могу из-за отсутствия

какой-либо четкой и хотя бы немного формальной постановки задачи.) Как мне представляется, мало кто из людей из сообщества NoSQL (да и из традиционного сообщества баз данных) серьезно разбирался в сути этой "теоремы", но широко распространено мнение, что она означает невозможность поддержки в одной распределенной системе управления данных свойств согласованности данных (*Consistency*), доступности (*Availability*) и устойчивости к разделению сети (*Partitioning*). Обобщая *consistency* в смысле Брювера до полного набора свойств *ACID (Atomicy, Consistency, Isolation, Durability)* классических транзакций баз данных, сообщество NoSQL с готовностью отказывается от реальной поддержки транзакций в своих системах (поэтому, например, в [8] предлагается переименовать *NoSQL* в *NoACID*).

В этой статье я не буду касаться "экстремистских" решений для поддержки оналайновых "OLTP"-приложений (я заключил OLTP в кавычки, поскольку не понимаю, о каких "транзакциях" можно в этом случае говорить). Не то чтобы я недооцениваю эту ветвь разработок – с практической точки зрения они очень важны, но, во-первых, они просто не вписываются в контекст статьи, а во-вторых, я не вижу у разработок этой категории какой-либо общей идеологии, кроме отрицания SQL и ACID. Но имеется и другая ветвь, наиболее ярким представителями которой мне кажутся исследователи из Федерального швейцарского технологического института (ETH) Цюриха и среди них, прежде всего, Дональд Коссманн (Donald Kossmann).

Одна из основных (не технических) идей, на которых основываются исследования и разработки этой группы в последние годы, состоит в следующем (см., например, [9]). Облачная инфраструктура, в которой все чаще развертываются транзакционные приложения, основана на предоставлении пользователям разнообразных сервисов, в том числе, и сервисов управления данными. В этом случае компании-пользователи платят именно за услуги, и чем сложнее услуга, чем дороже обходится ее поддержка, тем больше за нее приходится платить. По мнению Коссманна, для поддержки ACID-транзакций требуются дополнительные расходы, оплачивать которые пользователи должны только в тех случаях, когда это качество службы управления данными им действительно требуется. Вокруг этих идей выполняются интересные исследования и разработки, заслуживающие, на мой взгляд, анализа.

Среди работ, в которых ни в коей мере не затрагиваются фундаментальные свойства ACID и при этом обеспечивается горизонтальное масштабирование параллельных систем баз данных, мне больше всего нравится проект H-Store [10], в котором участвуют исследователи Массачусетского технологического института, Йельского университета и университета Браун при участии таких "грандов" в области баз данных, как Майкл Стоунбрейкер и Стенли Здоник (Stanley Zdonik). На основе предварительных результатов этого проекта была основана компания VoltDB [11], выпустившая в середине 2010 г. свой первый

коммерческий программный продукт (кстати, с открытыми исходными текстами и лицензией GPL3).

Основной упор в этом проекте делается на достижение максимально возможной эффективности и обеспечение линейной горизонтальной масштабируемости при полной поддержке ACID-транзакций. Причем в последнее время во многих своих публикациях (см., например, [12-13]) Майкл Стоунбрейкер, на мой взгляд, вполне убедительно доказывает, что отказ от свойств ACID никоим образом не способствует повышению уровня доступности распределенных систем управления данными. Высокой же производительности и горизонтальной масштабируемости массивно-параллельных систем баз данных в наибольшей степени мешают распределенные транзакции и, в особенности, их двухфазная фиксация.

Устранению отрицательного влияния распределенных транзакций и посвящается бóльшая часть исследований проекта H-Store. Кроме того, внимания заслуживает работа [14], стоящая несколько особняком от основного направления проекта, но, несомненно, способствующая его успешному выполнению. В параллельных СУБД без совместного использования ресурсов база данных должна быть разделена на части, каждая из которых управляется локальным компонентом СУБД в отдельном узле кластера. В транзакционных системах важно научиться так разделять базу данных, чтобы в рабочих нагрузках появлялось как можно меньше распределенных транзакций. Авторы [14] предлагают интересный подход к обнаружению методов таких разделений.

Наконец, параллельным транзакционным СУБД свойственна еще одна проблема, к которой, на мой взгляд, недостаточно внимательно относятся участники проекта H-Store. Они отказываются от использования общих ресурсов даже внутри одного узла, в котором установлен компьютер с многоядерным процессором. Такой компьютер используется как набор виртуальных изолированных узлов, каждому из которых соответствует ядро процессора. В [15] обосновывается неэффективность такого подхода и предлагается оригинальная архитектура параллельной "одноузловой" СУБД, работающей на машине с многоядерным процессором. В этой архитектуре на физическом уровне все основные ресурсы процессора (основная и дисковая память) являются общими для всех потоков управления СУБД, а на логическом уровне данные разделяются между потоками управления. Демонстрируется, что такая организация СУБД позволяет резко снизить нагрузку на центральный менеджер блокировок и обеспечить хорошее масштабирование системы при возрастании числа ядер.

## 1.2. Структура статьи

Оставшаяся часть статьи организована следующим образом. В разд. 2 напоминается исходный смысл свойств ACID транзакций баз данных. Затем анализируется, как связано свойство "consustency" в смысле "теоремы" CAP с

соответствующим свойством ACID. Разд. 3 посвящен рассмотрению наиболее интересных архитектур и методов построения параллельных СУБД, в обязательном порядке поддерживающих ACID-транзакции. В разд. 4 обсуждаются подходы к обоснованному ослаблению свойства согласованности. Наконец, разд. 5 содержит заключение.

## 2. Классические свойства транзакций и "теорема" CAP

Во введении отмечалось, что многочисленные обсуждения следствий "теоремы" CAP на возможность поддержки ACID-транзакций в распределенных СУБД без совместно используемых ресурсов часто демонстрируют непонимание авторами сути свойств ACID и/или смысла "теоремы" CAP. В этом разделе, прежде всего, напоминается исходный смысл свойств ACID, который имелся в виду изобретателями этой аббревиатуры. Затем я постараюсь прояснить смысл "теоремы" Брювера и подвести читателей к мысли, что *consistensy* в этой теореме имеет мало общего с *consistensy*, входящей в число свойств ACID.

### 2.1. ACID: вернемся к истокам

Впервые аббревиатура ACID появилась в 1983 г. в статье Тео Хаердера (Theo Haerder) и Адреаса Рейтера (Andreas Reuter) [16]. Для упрощения текста и пущей убедительности я приведу перевод фрагмента этой статьи (с небольшими сокращениями). В этом фрагменте используется пример банковской транзакции из [17], в которой деньги переводятся с одного счета на другой (рис. 1).

Концепция транзакции, включающей в приведенном примере все взаимодействия с базой данных между `$BEGIN_TRANSACTION` и `$COMMIT_TRANSACTION`, требует, чтобы все действия выполнялись *нераздельно (indivisibly)*: либо все действия должным образом отражаются в состоянии базы данных, либо ничего не происходит. Если в какой-либо момент времени до достижения `$COMMIT_TRANSACTION` пользователь вводит оператор `ERROR`, содержащий `$RESTORE_TRANSACTION`, то в базе данных не отражаются никакие изменения. Для достижения такой неделимости транзакция должна обладать следующими четырьмя свойствами:

> *Атомарность (Atomicity).* Транзакция должна иметь описанный выше тип "все или ничего", и, что бы ни произошло, пользователь должен знать, в каком состоянии находится его транзакция.
>
> *Согласованность (*Consistency*).* Транзакция, достигающая своего нормального завершения (EOT – end of transaction, завершение транзакции) и, тем самым, фиксирующая свои результаты, сохраняет согласованность базы данных. Другими словами, каждая успешная транзакция по определению фиксирует только допустимые результаты.

Это условие является необходимым для поддержки четвертого свойства – долговечности.

*Изоляция (Isolation).* События, происходящие внутри транзакции, должны быть скрыты от других одновременно выполняемых транзакций. Если бы это условие не выполнялось, то по причинам, упомянутым выше, транзакцию было бы невозможно вернуть к своему началу. Для достижения изоляции используются методы, называемые *синхронизацией*...

*Долговечность (Durability).* После того, как транзакция завершилась и зафиксировала свои результаты в базе данных, система должна гарантировать, что эти результаты переживут любые последующие сбои. Поскольку отсутствует какая-либо область управления, охватывающая наборы транзакций, у системы управления базами данных (СУБД) нет никакого контроля вне пределов границ транзакций. Поэтому пользователю должно гарантироваться, что если система сообщает ему о том, что нечто произошло, то это "нечто" действительно произошло. Поскольку по определению любая (*успешно завершенная – С.К.*) транзакция является корректной, результаты неизбежно появляющихся некорректных транзакций (т.е. транзакций, содержащих ошибочные данные), могут быть устранены только соответствующей "контр"-транзакцией (countertransaction).

Эти четыре свойства – атомарность, согласованность, изоляция и долговечность (ACID) – описывают основные черты парадигмы транзакций, которые влияют на многие аспекты разработки систем баз данных. Поэтому мы считаем, что способность какой-либо системы к поддержке транзакций является пробным камнем (ACID test) качества этой системы.

```
FUNDS_TRANSFER. PROCEDURE,
 $BEGIN_TRANSACTION;
 ON ERROR DO;                          /* in case of error */
       $RESTORE_TRANSACTION,              /* undo all work */
       GET INPUT MESSAGE;               /* reacquire input */
       PUT MESSAGE ('TRANSFER FAILED');   /* report failure */
       GO TO COMMIT;
       END;
 GET INPUT MESSAGE;                     /* get and parse input */
 EXTRACT ACCOUNT_EBIT, ACCOUNT_CREDIT,
  AMOUNT FROM MESSAGE,
 $UPDATE ACCOUNTS                       /* do debit */
          SET BALANCE ffi BALANCE— AMOUNT
    WHERE ACCOUNTS.NUMBER = ACCOUNT_DEBIT;
 $UPDATE ACCOUNTS                       /* do credit */
          SET BALANCE = BALANCE + AMOUNT
```

```
   WHERE ACCOUNTS.NUMBER = ACCOUNT_CREDIT;
$INSERT INTO HISTORY                    /* keep audit trail */
  <DATE, MESSAGE>;
PUT MESSAGE ('TRANSFER DONE');          /* report success */
COMMIT:                                 /* commit updates */
$COMMIT_TRANSACTION;
END;                                    /* end of program */
```

*Рис. 1. Простая программа на языке PL/1-SQL, переводящая средства с одного счета на другой.*

Я привел эту длинную цитату, чтобы напомнить, что, по сути, свойства ACID, с одной стороны, можно рассматривать как требования к любой СУБД, претендующей на поддержку транзакций, а с другой стороны, – как *определение* транзакции в системе баз данных. Это определение полностью соответствует житейской практике. Трудно представить, например, чтобы клиент, выполняющий банковскую транзакцию (неважно, при содействии живого человека-операциониста, или с использованием Internet-банкинга), не рассчитывал на удовлетворение банком всех свойств ACID. Банк, не поддерживающий свойства ACID для своих транзакций, в лучшем случае потеряет клиентов, а в худшем – обанкротится.

Очень важно, что свойства ACID являются нераздельными, отбрасывание любого из них делает оставшуюся комбинацию бессмысленной. В частности, если отбросить свойство *согласованности* (в том смысле, в котором оно использовалось в приведенной цитате), то мы потеряем критерий корректности транзакции. Система баз данных не сможет каким-либо осмысленным образом принимать решение о допустимости или недопустимости фиксации транзакций, и все проверки корректности выполнения операций при текущем состоянии базы данных придется выполнять в коде приложений.

Нужно понимать, что в данном случае речь идет о *логической* согласованности. Клиенту банка нужно, чтобы банк работал по установленным им и известным клиентам правилам, чтобы нельзя было выполнять какую-либо транзакцию, нарушающую эти правила, чтобы следующая транзакция того же клиента выполнялась в среде, согласованной в соответствии с этими правилами. Клиенту онлайнового магазина нужно, чтобы заказанный и оплаченный им товар был своевременно ему доставлен (в соответствии с установленными и известными клиенту правилами). Иначе он не будет доверять этому магазину. При этом ни клиенту банка, ни клиенту Internet-магазина нет никакого дела до внутренней кухни предприятия, до того, какие внутренние действия предпринимаются для выполнения его транзакции. Клиенту нет дела до того, каким образом поддерживается *физическая* согласованность этого предприятия, каким образом выполняются операции на физическом уровне.

Если заботу о поддержке логической согласованности транзакций (и базы данных) берет на себя СУБД, то приложения становятся более простыми, понятными и надежными. Вся логика прикладной области (банка, магазина, склада и т.д.), касающаяся транзакций и допустимого состояния данных уходит в систему баз данных. И требования к этой системе очень просты: поддержка ACID-транзакций с учетом правил согласованности, обеспеченной в базе данных приложением. С моей точки зрения, отказ от ACID-транзакций создает немереные трудности для разработчиков приложений, которым, хочешь - не хочешь, придется самим реализовывать нечто похожее, чтобы удовлетворить естественные потребности своих клиентов.

И еще раз замечу, что свойства ACID, фактически, определяют понятие транзакции. На мой взгляд, чтобы иметь хотя бы какую-нибудь возможность говорить о транзакционной системе управления данными, в которой не поддерживается свойство согласованности транзакций, совершенно необходимо определить, что в этом случае понимается под термином *транзакция*. К сожалению, сегодня во многих случаях (в особенности, это свойственно направлению NoSQL) люди говорят о поддержке OLTP-приложений, совершенно не уточняя, что за транзакции имеются в виду. Поэтому в данной статье я буду использовать сочетание *ACID-транзакции* для обозначения настоящих транзакций, а неуточняемый термин *транзакция* будет использоваться в неформальном смысле, разном в разных контекстах.

Займемся теперь "теоремой" CAP и постараемся разобраться, что же означает *согласованность* в смысле Брювера.

## 2.2. Согласованность по Брюверу

Начнем с того, что Эрик Брювер не является и никогда не объявлял себя специалистом в области баз данных. Он относится к сообществу распределенных систем, и его знаменитый доклад [7], в котором появилась "теорема" CAP, был сделан на конференции "Принципы распределенных вычислений". (Кстати, десять лет спустя, в 2010 г. он еще раз выступил с приглашенным докладом [18] на той же конференции, и в этом докладе привел, в частности, ряд примеров распределенных систем, при разработке которых учитывалась "теорема" CAP.) В этой области имеется свое толкование терминов, используемых в области баз данных.

В частности, термин *мгновенная согласованность (immediate consistency)* означает, что после того как пользователь получает от системы извещение об успешном выполнении некоторой операции обновления данных, результат этой операции становится мгновенно видимым для всех наблюдателей. *Согласованность в конечном счете (eventual consistency)* означает, что если в течение достаточно долгого периода времени в систему не поступают новые операции обновления данных, то можно ожидать, что результаты всех предыдущих операций обновления данных, в конце концов, распространятся по всем узлам системы, и все реплики данных *согласуются* (*по всей*

*видимости, это нужно понимать как "у всех реплик будет одно и то же состояние" – С.К.*). Скорее всего, в [7] под *согласованностью* понимается именно *мгновенная согласованность* данных.

Имея в виду этот смысл понятия согласованность, можно считать "теорему" Брювера вполне понятной и очевидной: в любой распределенной системе с разделенными данными можно одновременно обеспечить только любые два свойства из *согласованности*, *доступности* и *устойчивости к разделению сети*. В связи с этим Брювер даже противопоставляет набор свойств ACID предлагаемому им набору свойств BASE (Basically Available, Soft-state, Eventual consistency – доступность в большинстве случаев; неустойчивое состояние; согласованность в конечном счете). Но это противопоставление, по моему мнению, неправомерно, поскольку в первом случае речь идет о логических характеристиках транзакций, а во втором – о физических свойствах распределенных систем.

Многие считают, что "теорема" Брювера формально доказана. Действительно, в статье Сета Гильберта (Seth Gilbert) и Нэнси Линч (Nancy Lynch) [19] вводятся некоторые (почти) формальные определения, в контексте которых "теорема" действительно становится теоремой и доказывается. Однако давайте разберемся, как же определяются те три свойства распределенной системы, из числа которых по "теореме" Брювера можно одновременно обеспечить поддержку только двух свойств.

Согласованностью в [19] называется *атомарная, или линеаризуемая согласованность (atomic, or linearizable consistency)*, являющаяся свойством системы, все индивидуальные объекты данных которой являются атомарными (линеаризуемыми). В свою очередь, *атомарным объектом* называется объект с несколькими операциями, такими что вызов операции и получение ответных данных происходят как бы мгновенно, т.е. объект не принимает вызов следующей операции до полного завершения предыдущей операции. При этом порядок приема операций должен быть таким, что если операция типа чтения поступает после выполнения некоторой операции типа записи, то операция чтения должна вернуть значение, записанное этой или какой-либо более поздней операцией записи.

Распределенная система является *постоянно доступной*, если на каждый запрос, полученный не отказавшим узлом, должен быть получен ответ. *Устойчивость* системы *к разделению сети* в [19] моделируется как сохранение жизнеспособности системы при потере произвольного числа сообщений, посылаемых из одного узла в другой.

На основе этих определений Гильберт и Линч формулируют следующую теорему (в асинхронной модели сети отсутствуют часы, и в узлах должны приниматься решения только на основе получаемых сообщений и локальных вычислений):

*В асинхронной модели сети невозможно реализовать объект данных с операциями чтения и записи, гарантирующий обеспечение свойств доступности и атомарной согласованности для всех допустимых выполнений (включая те, в которых теряются сообщения).*

Эта теорема действительно достаточно просто формально доказывается методом "от противного". Далее в [19] выводится следствие, заключающееся в том, что:

> В асинхронной модели сети невозможно реализовать объект данных с операциями чтения и записи, гарантирующий обеспечение свойств доступности для всех допустимых выполнений и атомарной согласованности для допустимых выполнений, в которых сообщения не теряются.

Кроме того, доказывается истинность основной теоремы для частично синхронной модели сети, в которой в каждом узле присутствуют часы, время, показываемое которыми, увеличивается с одной и той же скоростью, но которые не синхронизованы, т.е. могут показывать разное время в один и тот же реальный момент. Показано, что для этого случая аналогичное следствие не выводится, и, значит, для частично синхронных сетей имеется больше возможностей организации распределенных систем с "хорошими" свойствами.

Да, можно считать, что в некотором смысле (не обязательно совпадающем со смыслом, который имелся в виду Брювером) Гильберт и Линч доказали невозможность одновременного обеспечения в одной распределенной системе свойств атомарной согласованности, доступности и устойчивости к разделению сети. Но какое отношение это имеет к транзакциям баз данных вообще и к ACID-транзакциям в частности?

Вот что пишет по этому поводу в своей заметке [20], посвященной обсуждению "теоремы" CAP и статьи [19], Джулиан Браун (Julian Browne):

> В своем доказательстве Гильберт и Линч используют вместо термина *согласованность* термин *атомарность*, что с технической точки зрения более осмысленно, потому что, строго говоря, *согласованность* в смысле ACID относится к идеальным свойствам транзакций баз данных и означает, что никакие данные не станут долговременно хранимыми, если они нарушают некоторые заранее установленные ограничения. Но если полагать, что заранее установленным ограничением распределенных систем является запрет наличия нескольких разных значений у одного и того же элемента данных, то, по моему мнению, этот изъян в абстракции *согласованности* можно считать несущественным (кроме того, если бы Брювер использовал термин *атомарность*, то появилась бы теорема AAP, название которой было бы чрезвычайно неудобно произносить).

Это написано не очень серьезно, но честно. И, на самом деле, требование атомарной согласованности нельзя перемешивать с требованиями согласованности транзакций в смысле ACID. Ограничения целостности базы данных – это логические, если угодно, бизнес-требования. Они происходят из логики прикладной области. Требование атомарной согласованности совсем другого рода. Это реализационное требование, относящееся к той категории, которую традиционно в области баз данных называли *физической* согласованностью (например, при выполнении любой операции изменения индекса все блоки соответствующего B+-дерева должны содержать корректные значения и быть связаны корректными ссылками).

А вот что уже совсем серьезно пишут в своей заметке [8] представители сообщества баз данных Дэниэль Абади (Daniel Abadi) и Александер Томсон (Alexander Thomson):

> ... все более критичным становится требование к доступности масштабируемых транзакционных систем, и обычно оно удовлетворяется за счет репликации и автоматического перенаправления запросов в случае сбоя одного из узлов. Поэтому разработчики приложений ожидают, что гарантии согласованности (consistency) ACID-систем (первоначально заключавшиеся в локальной поддержке определенных пользователями инвариантов) будут распространены на обеспечение строгой согласованности (того, что все реплики одних и тех же данных в любой момент времени будут являться идентичными копиями, т.е. в этом случае согласованность подразумевается в смысле CAP/PACELC (*про PACELC см. в [21] – С.К.*)).

Другими словами, согласованность по Брюверу не имеет ничего общего с согласованностью в смысле ACID, но именно в системах, ориентированных на обеспечение высокого уровня доступности за счет репликации данных, желательно поддерживать строгую согласованность реплик. Это не свойство ACID, а техническая (физическая) особенность массивно-параллельных СУБД, облегчающая разработку приложений.

Как считает Майкл Стоунбрейкер [12-13], залогом построения качественной современной СУБД является правильный выбор технических компромиссов. При выборе конкретного инженерного решения нужно учитывать множество факторов – требования будущих пользователей, вероятности возникновения различных сбойных ситуаций и т.д., а не руководствоваться догматическим образом каким-либо общими теоретическими указаниями (в том числе, и "теоремой" CAP).

Стоунбрейкер полагает, что в области транзакционных параллельных систем баз данных отказ от согласованности по Брюверу в пользу поддержки высокой доступности и устойчивости к разделению сети является плохим компромиссом, поскольку (a) согласованность реплик является очень полезным свойством системы; (b) транзакционные массивно-параллельные

СУБД не нуждаются в кластерах с очень большим числом узлов, так что ситуации разделения сети маловероятны; (c) система может легко стать недоступной не из-за разделения сети, а, например, из-за наличия регулярно проявляющихся программных ошибок.

Таким образом, высокая активность представителей лагеря NoSQL (читай NoACID), которые часто ссылаются на "теорему" Брювера, связана не с теоретической невозможностью построения массивно-параллельных транзакционных СУБД, поддерживающих ACID-транзакции, а с тем, что упрощенные системы, не поддерживающие не только ACID-транзакции, но и согласованность реплик, создаются проще и быстрее. Из-за своей упрощенной организации они способны обеспечивать очень быструю обработку данных, и для ряда приложений это оказывается более важным, чем все удобства, свойственные технологии баз данных.

Посмотрим, как отвечает на этот вызов сообщество баз данных.

## 3. Новые транзакционные архитектуры, поддерживающие классические свойства транзакций

Понятно, что конкурировать на рынке OLTP с системами основных поставщиков SQL-ориентированных СУБД (IBM, Oracle и Microsoft), которые в течение многих лет оптимизировались именно для обработки ACID-транзакций, может только СУБД, обладающая некоторыми принципиально отличными качествами. В настоящее время такими потенциально достижимыми новыми качествами являются существенно (в десятки раз) б*о*льшая пропускная способность транзакций и горизонтальная масштабируемость (т.е. возможность линейного повышения пропускной способности при наращивании аппаратных ресурсов). Все более распространенным (и подтверждаемым практикой) мнением является то, что такие качества можно обеспечить только при использовании архитектур без совместного использования ресурсов (shared-nothing). Одним из наиболее ярких свидетельств правильности этого мнения являются проект H-Store [10] и достижения компании VoltDB [11], которым посвящается подраздел 3.1.

С другой стороны, нельзя не учитывать, что продолжает действовать закон Мура, в соответствии с новой трактовкой которого экспоненциально возрастает число ядер в микропроцессорах. В действительности в узлах массивно-параллельной СУБД (даже основанной на использовании компьютеров категории массового спроса) применяются мощные компьютеры, позволяющие эффективно использовать локальные СУБД с совместным использованием ресурсов. Другими словами, для построения предельно эффективной транзакционной массивно-параллельной СУБД нельзя не позаботиться об эффективности локальных СУБД, работающих на многоядерных процессорах. В этом направлении исследований мне хочется выделить проект DORA [15], в котором оригинальным образом сочетаются

подходы *shared-everything* на физическом уровне архитектуры СУБД и *shared-nothing* на логическом уровне. Этот проект обсуждается в подразделе 3.2.

## 3.1. H-Store: ничего лишнего

Впервые краткое описание исходных идей проекта H-Store появилось в 2007 г. в [22]. Эта статья была последней в цикле "один размер непригоден для всех" (см. также [1, 23]), в котором доказывалось, что прошло время универсальных, пригодных для поддержки любых приложений баз данных SQL-ориентированных СУБД, и обосновывались преимущества специализированных архитектур. В [22] речь идет исключительно о специализированных транзакционных системах, основанных на следующих пяти основных соображениях.

i. В основной памяти недорогой массивно-параллельной системы уже сейчас можно разместить базу данных объемом до одного терабайта. Этого достаточно для большинства приложений OLTP. Поэтому будущее за системами транзакционных баз данных, полностью размещаемых в основной памяти.

ii. В системах OLTP транзакции являются очень легковесными. При работе с базой данных в основной памяти время выполнения наиболее тяжелой транзакции из тестового набора TPC-C составляет менее одной миллисекунды. В большинстве приложений OLTP при выполнении транзакций отсутствуют задержки по вине пользователей. Поэтому имеет смысл выполнять все операции каждой транзакции последовательно в одном потоке управления (*если транзакция не затрагивает данные нескольких узлов – С.К.*).

iii. Кажется правдоподобным, что в следующем десятилетии будут доминировать компьютерные системы без общих ресурсов, и все СУБД следует оптимизировать в расчете на использование такой архитектуры. Если система с *N* узлами не обеспечивает достаточной мощности, должна иметься возможность добавления к ней дополнительных *K* узлов без потребности в каких-либо сложных действиях над используемой СУБД (*то самое горизонтальное масштабирование – С.К.*).

iv. В будущем высокий уровень доступности и встроенные средства восстановления после отказов станут важными чертами рынка OLTP. Из этого следует несколько выводов.

   a. В любой СУБД, ориентированной на поддержку OLTP, потребуется согласованная репликация данных.

   b. Наиболее эффективной является поддержка архитектуры *shared-nothing* на всех уровнях системы (*как я уже говорил, это неочевидно, см. следующий подраздел – С.К.*)

   c. Наилучший способ поддержки архитектуры без общих ресурсов состоит в использовании нескольких машин в одноранговой (peer-to-peer) конфигурации. Тогда нагрузка

OLTP может быть распределена между несколькими машинами, а межмашинную репликацию можно использовать для обеспечения отказоустойчивости.

d. В мире высокой доступности не требуется поддержка журнала повторного выполнения операций, а нужен только временный, сохраняемый в основной памяти журнал откатов.

e. Основные расходы IT-подразделений уходят на содержание персонала. Единственным выходом из этого положения является перевод систем на «самообслуживание» (самовосстановление, автоматическое техническое обслуживание, автоматическую настройку и т.д.). Требуется полный пересмотр процесса настройки системы без явных ручек управления.

Эти соображения приводят к следующим выводам.

i. Основным препятствием для достижения высокой производительности системы почти наверняка станет журнал повторного выполнения операций, сохраняемый в дисковой памяти. Без него можно обойтись за счет подсистемы поддержки высокого уровня доступности и обработки отказов.

ii. Следующим по значимости узким местом системы является вызов в ней операций и возврат результатов в приложение. Наиболее эффективным способом решения этой проблемы является выполнение логики приложений в виде хранимых процедур внутри системы баз данных.

iii. Во всех возможных случаях следует отказаться от поддержки и журнала откатов транзакций, поскольку он тоже будет сдерживать производительность.

iv. Следует приложить все усилия, чтобы максимально освободиться от затрат на синхронизационные блокировки.

v. Желательно освободиться и от синхронизации на основе "защелок" при доступе к одним и тем же структурам данных из нескольких потоков управления. С учетом кратковременности транзакций эти накладные расходы можно устранить путем перехода к однопотоковой модели выполнения транзакций.

vi. По мере возможности следует избегать применения двухфазного протокола фиксации распределенных транзакций.

### *3.1.1. Свойства схем транзакционных баз данных и типичных транзакций*

В H-Store требуется наличие заранее специфицированного набора классов транзакций, которые могут входить в рабочую нагрузку системы. Каждый класс характеризует транзакции с одними и теми же операторами SQL и логикой приложения, различающиеся только значениями констант времени выполнения. Это требование не является неестественным, поскольку для

транзакционных приложений нехарактерно наличие непредвиденных запросов, явно вводимых пользователями во время выполнения транзакции. Таким образом, задержки выполнения транзакций по вине пользователей невозможны.

Аналогичным образом, считается, что заранее известна логическая схема базы данных, над которой будут выполняться эти транзакции. Авторы [22] обнаружили, что многие транзакционные базы данных обладают древовидной логической схемой, в которой каждая таблица (кроме одной – корневой) имеет связь *n*:1 ровно с одной таблицей-предком (т.е. она естественным образом соединяется только с одной таблицей). Для баз данных с древовидной схемой имеется очевидный метод горизонтального разделения данных: корневая таблица разделяется по диапазонам значений первичного ключа (или на основе хэширования этих значений); каждая таблица-потомок разделяется таким образом, чтобы при естественном соединении с каждым разделом таблицы-предка потребовались бы строки только одного раздела таблицы-потомка, причем этот раздел размещается в том же узле, что и соответствующий раздел таблицы-предка.

Если в каждом операторе SQL каждой транзакции содержится условие, выделяющее ровно одну строку корневой таблицы (например, любая операция относится к некоторому клиенту онлайнового магазина, и таблица клиентов является корневой), то при таком разделении каждый оператор будет выполняться ровно в одном узле (будет являться *локальным* для этого узла). Если все операторы каждой транзакции локальны для одного и того же узла, то соответствующее приложение называется *приложением над ограниченным деревом (constrained tree application, CTA)*. Ценное свойство CTA-приложений состоит в том, что все его транзакции могут быть полностью выполнены в одном узле, т.е. являются *одноузловыми*. Такие транзакции выполняются без каких-либо задержек из-за коммуникации с другими узлами (кроме возможных задержек из-за синхронизации обновления реплик).

По опыту авторов, многие приложения OLTP сразу разрабатываются в стиле CTA, а во многих других случаях их можно преобразовать в CTA-приложения. (Заметим, что здесь и далее очень заметно влияние на ранней стадии проекта H-Store Пэта Хелланда (Pat Helland), который входит в число авторов [22], а ранее, будучи сотрудником Amazon, написал статью [24], где высказывал схожие соображения. Интересно также отметить, что в дальнейших работах, посвященных H-Store, влияние идей Хелланда почти незаметно.)

Для преобразования к виду CTA приложений, изначально таковыми не являющихся, в [22] предлагалось использовать два подхода. Во-первых, можно выделить все таблицы, которые во всех транзакциях только читаются. Такие таблицы можно реплицировать во всех узлах. Если некоторое приложение обладает свойством CTA по отношению ко всем остальным таблицам, то после такой репликации оно станет CTA-приложением. Во-

вторых, имеется еще один важный класс OLTP-приложений, части транзакций которых можно выполнять параллельно без потребности в передаче между узлами промежуточных результатов, причем результаты операций SQL никогда не требуются при выполнении последующих операций. Транзакции таких приложений *с одноразовым использованием результатов (one-shot)* можно преобразовать в набор одноузловых планов, каждый из которых выполняется только в одном узле. Часто такие преобразования можно произвести за счет *вертикального* разделения таблиц между узлами (только читаемые вертикальные разделы реплицируются).

Некоторые классы транзакций обладают свойством *двухфазности (two-phase)* или могут быть преобразованы к двухфазным транзакциям. На первой фазе такой транзакции выполняются только операции чтения, и только на этой фазе допускается аварийное завершение транзакции по ее собственной инициативе (т.е. в это время производятся все возможные проверки, вследствие которых может потребоваться аварийное завершение транзакции). Важным свойством двухфазных транзакций является то, что при их выполнении можно обойтись без журналов откатов транзакций. Класс двухфазных транзакций называется *строго двухфазным (strongly two-phase)*, если на первой фазе выполнения транзакции во всех узлах, участвующих в ее выполнении, принимается одно и то же решение относительно ее продолжения или аварийного завершения.

Две параллельно выполняемые транзакции (из одного или разных классов) называются *коммутативными*, если при любом чередовании их одноузловых планов производится одно и то же окончательное состояние базы данных (если обе транзакции фиксируются). Класс транзакций, коммутативных со всеми транзакциями, называется *стерильным (sterile)*.

### *3.1.2. Как был устроен и как работал начальный вариант H-Store*

H-Store выполняется в кластере компьютеров (*почему-то в [22] эта аппаратная среда упорно называется grid'ом – С.К.*). При конфигурировании системы можно указать желаемый уровень ее надежности – число узлов, при выходе из строя которых система может восстановить работоспособность без потери выполняемых транзакций в течение заданного времени. (*Поскольку восстановление системы основано на использовании реплик, то, очевидно, уровень надежности коррелирует с числом поддерживаемых реплик данных – С.К.*).

В каждом узле строки разделов таблиц размещаются вплотную одна к другой, и доступ к ним производится на основе B-деревьев (т.е. строки размещаются в порядке сортировки по значениям ключа B-дерева). Размер блока B-дерева соответствует размеру блока кэша второго уровня используемого процессора. (*Сравнительно ясно, что является ключом B-дерева для баз данных с древовидной схемой – первичный ключ для корневой таблицы и внешний ключ*

*для любой таблицы-потомка. Что выбирается в качестве ключа B-дерева раздела таблицы при наличии других схем, неясно – С.К.*).

В каждом узле H-Store поддерживает ровно один поток управления, в котором полностью, без каких-либо задержек выполняется каждая поступающая операция SQL. Узлы, в процессорах которых имеется несколько ядер, разбиваются на соответствующее число логических узлов. В H-Store каждый логический узел трактуется так же, как и любой физически независимый узел, и основная память многоядерного компьютера разделяется между логическими узлами.

Транзакции представляются в виде хранимых процедур базы данных, и в системе поддерживается только одна внешняя операция `Execute transaction (parameter_list)`, позволяющая в любом узле инициировать выполнение любой предопределенной транзакции с передачей ей значений параметров. Внутри таких хранимых процедур (для написания которых в исходном прототипе использовался язык C++) сочетается логика приложений и операции манипулирования базами данных, причем вызовы SQL производятся как локальные вызовы. Журнал повторного выполнения операций не поддерживается, а журнал отката (сохраняемый в основной памяти и освобождаемый при завершении транзакции) ведется только для транзакций, не являющихся двухфазными.

В исходном прототипе H-Store отсутствовал компилятор SQL, и планы всех операций SQL генерировались и оптимизировались вручную. Однако в [22] отмечалось, что планируется разработка компилятора SQL с оценочной (cost-based) оптимизацией, и что этот компилятор-оптимизатор должен быть сравнительно простым, поскольку в типичном OLTP-запросе всегда идентифицируется некоторый *опорный кортеж (anchor tuple)*, с которым соединяются несколько (немного) таблиц. И некоторый компилятор SQL действительно появился в коммерческом варианте H-Store – VoltDB (см., например, [25]), хотя по доступной документации системы трудно судить, какие возможности оптимизации в нем реализованы.

Для обеспечения возможности использования H-Store без потребности в "ручках управления" планировалось создание средства автоматического проектирования физических схем баз данных (*дизайнера баз данных*), определяющего горизонтальное разделение, репликацию и выбор ключей индексов. Цель дизайнера состоит в том, чтобы сделать как можно больше транзакций *одноузловыми* (т.е. избежать появления распределенных транзакций). (*И, кроме того, насколько я понимаю, добиться выявления двухфазных и стерильных транзакций – С.К.*).

Мне с самого начала возможность создания такого средства казалась сомнительной. Уж очень трудна задача статического анализа многочисленных хранимых процедур с многочисленными вызовами операций SQL. К настоящему времени (конец 2010 г.) эта задача, по всей видимости, не решена. В документации VoltDB [25] разработчикам приложений предлагается лишь

методика физического проектирования баз данных, да и то подчеркивается необходимость многократного выполнения тестовых испытаний (benchmarking, benchmarking, benchmarking!) на реальных данных до вывода приложения в производственный режим. С другой стороны, некоторую надежду на продвижение в этом направлении дает работа [14], хотя она основывается уже не на статическом анализе, а на анализе трасс выполнения рабочей нагрузки (см. ниже).

Выполнение транзакций в исходном прототипе H-Store происходило по следующей схеме. На входе в систему каждой транзакции назначалась *временная метка (timestamp)* в формате (`site_id, local_unique_time-stamp`). Если поддерживается порядок на множестве узлов кластера, то все метки являются уникальными и полностью упорядоченными. Предполагалось, что локальные часы в каждом узле некоторым образом синхронизируются.

Если все классы транзакций являются *одноузловыми*, то каждая транзакция может быть направлена в узел с требуемыми копиями данных и полностью в нем выполнена. Если не все классы транзакций являются *стерильными*, то узел, в котором завершилось выполнение некоторой транзакции, в течение небольшого времени (времени сетевых задержек – для локальной сети в пределах нескольких миллисекунд) ожидает поступления транзакций от других узлов-инициаторов, чтобы выполнение транзакций происходило в порядке временны*х* меток. За счет этого все реплики будут обновляться в одном и том же порядке, и их состояние после конца любой транзакции будет идентично. Следовательно, для выполнения одноузловых транзакций не требуется журнал повторного выполнения операций, какое-либо управление параллелизмом и распределенная фиксация транзакций. Если же, в придачу ко всему остальному, транзакция является двухфазной, то для нее не требуется и журнал отката.

Если все транзакции являются *стерильными*, то обычно для их выполнения не требуется какое-либо управление параллелизмом. Более того, в этом случае не требуется назначение транзакциям временны*х* меток и их выполнение в одном и том же порядке над всеми репликами. Но если транзакция распространяется на несколько узлов, то отсутствует гарантия, что она будет во всех узлах успешно выполнена или аварийно завершена. Поэтому каждый исполнитель должен послать диспетчеру выполнения транзакции (в том узле, в котором она была инициирована) сообщение "аварийное завершение" или "нормальное завершение" в той точке своей части транзакции, после которой соответствующее решение изменить уже нельзя (для *двухфазных* транзакций – в конце первой фазы). Диспетчер, в свою очередь, должен рассылать эти сообщения в другие узлы-исполнители. Другими словами, в этом случае приходится выполнять обычную процедуру фиксации распределенной транзакции. Если транзакция является *строго двухфазной*, этих накладных расходов можно избежать.

В общем случае (когда транзакция не является *одноузловой* или *стерильной*), приходится применять средства управления параллелизмом. При реализации исходного прототипа H-Store было принято решение отказаться от традиционной для SQL-ориентированных СУБД пессимистической схемы синхронизационных блокировок в пользу более оптимистических методов (*как показывают более поздние статьи, посвященные H-Store в целом [26] и управлению транзакциями в H-Store [27-28], это решение не является стратегическим – поиск методов продолжается; кстати, совершенно неясно, какая схема управления параллелизмом применяется в VoltDB – С.К.*).

В схеме управления параллелизмом, описываемой в [22], применяются три стратегии – *основная*, *промежуточная* и *усложненная*. Для каждого класса транзакций определяются классы транзакций, с которыми транзакции данного класса могут конфликтовать (*по-видимому, выявляются стандартные конфликты "запись-запись", "чтение-запись" и "запись-чтение" на уровне таблиц – С.К.*). Каждая транзакция инициируется в некотором узле системы, и ее выполнение отслеживается *координатором транзакций* в этом узле. Координатор играет роль *диспетчера транзакций* в узле инициации и рассылает фрагменты транзакции в соответствующие узлы.

При применении *основной* стратегии сайт-*исполнитель* получает фрагмент и выжидает в течение времени сетевых задержек до момента, когда возможно поступление транзакции с *большим* значением временной метки. Если после этого в узле обнаруживаются транзакции с более поздними временными метками из классов, потенциально конфликтующих с данной транзакцией, то исполнитель не выполняет ее фрагмент и посылает координатору сообщение "аварийное завершение". В противном случае фрагмент выполняется, и результирующие данные возвращаются в узел, из которого этот фрагмент был получен. Если координатор получает сообщение о нормальном выполнении фрагментов от всех узлов, он продолжает выполнение транзакции при наличии следующего набора ее фрагментов или же фиксирует транзакцию. В противном случае транзакция завершается аварийным образом.

Если при использовании *основной* стратегии возникает слишком много аварийных завершений транзакций (*т.е. в данной рабочей нагрузке имеется высокий уровень потенциальной конфликтности транзакций – С.К.*), применяется более сложная *промежуточная* стратегия. В этом случае каждый исполнитель до принятия решения о выполнении или аварийном завершении полученного фрагмента выжидает интервал времени величиной `MaxD * среднее_время_обмена_сетевыми_сообщениями` (где `MaxD` – максимальное число межузловых сообщений, которое потребуется для выполнения любой потенциально конфликтующей транзакции), чтобы установить, не появится ли фрагмент транзакции с *меньшим* значением временной метки. В этом случае исполнитель получает возможность корректного упорядочивания фрагментов, что уменьшает вероятность аварийного завершения транзакций.

Наконец, *усложненная* стратегия, к которой планировалось прибегать в тех случаях, когда ни *основная*, ни *промежуточная* стратегии не позволяют в достаточной степени сократить число аварийных завершений транзакций, – это достаточно традиционная стратегия оптимистического управления параллелизмом. В этом случае конфликты распознаются во время выполнения, для чего в каждом узле отслеживаются *наборы прочитанных данных (read set)* и *наборы измененных данных (write set)* каждой транзакции. Любой исполнитель запускает выполнение любого фрагмента и аварийно его завершает, если это требуется для разрешения динамически обнаруженной конфликтной ситуации.

Авторы [22] сравнивали производительность начального прототипа H-Store с производительностью неназываемой коммерческой SQL-ориентированной СУБД на эталонном тестовом наборе TPC-C. В обоих случаях транзакции реализовывались в виде хранимых процедур и запускались внешней командой, передаваемой по сети. За счет специально подобранного метода разделения базы данных TPC-C и тщательного анализа транзакций удалось преобразовать все классы транзакций к стерильному и строго двухфазному виду. В результате на одной и той же аппаратной конфигурации (один компьютер с двухъядерным процессором) H-Store показала производительность, в 82 раза превышающую производительность традиционной СУБД. По наблюдениям авторов, главным тормозом традиционной СУБД стала система журнализации изменений в дисковой памяти. На втором месте – накладные расходы на управление параллелизмом.

Кстати, в 2008 г. основные авторы [22] опубликовали результаты более глубокого исследования влияния на производительность традиционных механизмов транзакционных СУБД [29]. Это исследование, фактически, объясняет, за счет чего в H-Store удалось добиться такой производительности. Авторы взяли не очень известную систему Shore [30] с открытыми исходными текстами, сконфигурировали ее таким образом, чтобы требуемая для их экспериментов база данных полностью помещалась в основной памяти, и измерили производительность полученной системы базы данных на смеси двух транзакций из тестового набора TPC-C. Затем они последовательно стали удалять из состава Shore компоненты журнализации, синхронизации и управления буферным пулом, и в результате получили вариант системы с ограниченной функциональностью, которая показала на том же тестовом наборе производительность, в 20 раз большую, чем у исходной Shore.

### *3.1.3. Спекулятивное выполнение транзакций*

Как отмечалось выше, оптимистическая схема управления распределенными (не одноразделными) транзакциями, разработанная в начале проекта H-Store, не стала стратегической схемой этого проекта. Продолжались (и, по-видимому, продолжаются) исследования, направленные на поиск более эффективных схем. Результаты одного из перспективных исследований

описываются в [27]. Идея заключается в том, чтобы постараться избежать простаивания потоков управления, когда выполняемые в них фрагменты распределенных транзакций вынуждаются ожидать поступления сетевых сообщений.

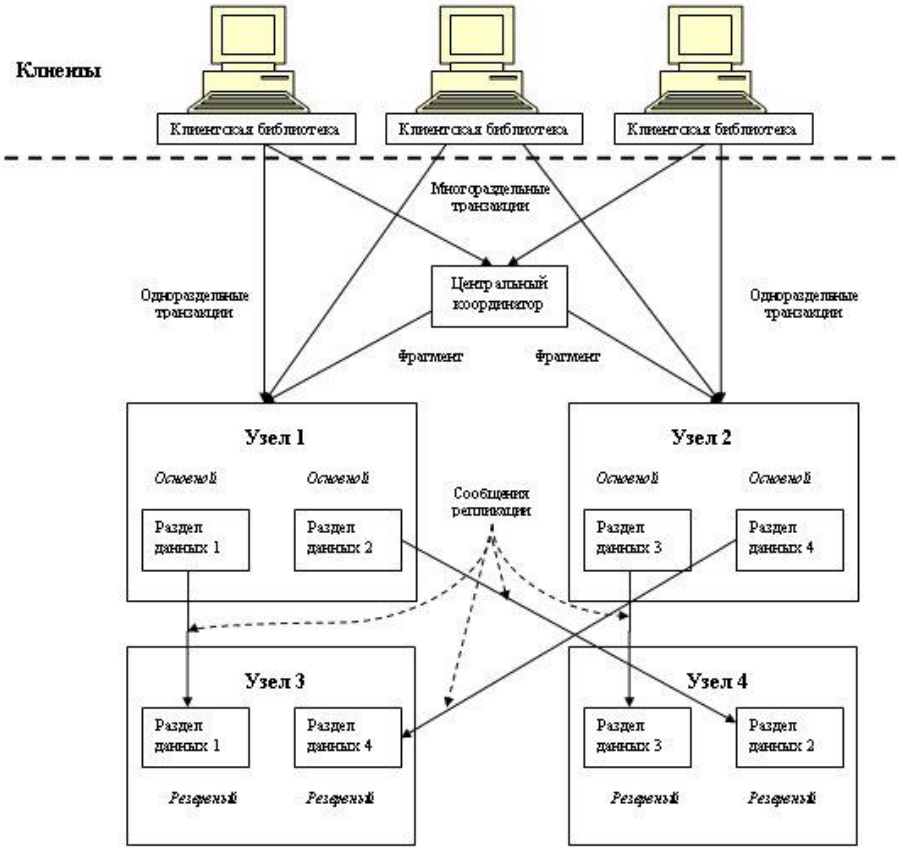



*Рис. 2. Архитектура системы*

В этом исследовании использовался прототип H-Store, архитектура которого показана на рис. 2. Для каждого раздела данных имеются один основной поток управления и $k$-1 резервных потоков управления (где $k$ – это упоминавшийся в предыдущем пункте уровень надежности системы; у системы на рис. 2 $k$=2). В отличие от [22], в архитектуре на рис. 2 используется только один координатор распределенных транзакций (авторы объясняют это тем, что исследования способов полного упорядочивания транзакций при наличии нескольких координаторов, т.е. способов обеспечения уникальных и

полностью упорядоченных временны*х* меток транзакций, пока еще не завершены). Наличие единственного центрального координатора ограничивает число одновременно выполняемых распределенных транзакций, но этот вариант конфигурации системы является временным.

Каждая транзакция разбивается на *фрагменты* – части работы, каждую из которых можно выполнить над данными только одного раздела. Транзакции делятся на *одноразделные* и *многоразделные* (*в [22] они назывались* одноузловыми *и* многоузловыми *соответственно; с моей точки зрения, новые термины являются более правильными, поскольку более точно характеризуют природу транзакций; тем не менее, в предыдущем пункте я решил следовать терминологии [22] – К.С.*).

Одноразделной транзакцией называется такая транзакция, для полного выполнения которой достаточно данных некоторого одного раздела (т.е. одноразделная транзакция состоит из одного фрагмента). Как показывает рис. 2, одноразделные транзакции запускаются сразу в потоках управления соответствующих разделов, минуя центральный координатор. (В прототипе H-Store приписывание фрагментов транзакций к разделам делается вручную, хотя в перспективе это должен делать автоматический планировщик). Для обеспечения долговечности результатов транзакций используется протокол репликации. Поток управления основного раздела, получив заявку на выполнение транзакции, пересылает ее всем потокам управления резервных разделов. После этого в основном разделе выполняется транзакция, в конце которой поток управления убеждается в получении подтверждения от потоков резервных разделов. Фактически, если получено подтверждение хотя бы от одной реплики, транзакцию можно считать зафиксированной.

В ходе выполнения одноразделной транзакции в соответствующем потоке управления отсутствует какой-либо параллелизм, поэтому не требуется синхронизация. Если в транзакции невозможно аварийное завершение по инициативе пользователя, то в большинстве случаев не требуется поддерживать журнал отката транзакции. (*В используемом прототипе H-Store, а также и в текущей версии VoltDB, пользователи могут сопровождать транзакции аннотациями, сообщающими системе, что в транзакции нет инициируемых пользователями откатов, – своеобразное приближение к двухфазным транзакциям, обсуждавшимся выше. – С.К.*). При наличии потребности журнал отката поддерживается в основной памяти и освобождается при завершении транзакции. (*Кстати, похоже, что если в транзакции возникает откат по инициативе пользователя, то он должен возникнуть и во всех репликах, поскольку во всех них выполняется один и тот же код над одними и теми же данными. – С.К.*).

Многораздельные транзакции поступают в систему через центральный координатор, устанавливающий их глобальный порядок. Координатор разбивает транзакцию на фрагменты и посылает их потокам управления соответствующих разделов. После получения ответов от этих потоков

управления в координаторе выполняется код приложения, определяющий дальнейшее выполнение транзакции, для которого может потребоваться рассылка по потокам управления разделов дополнительных фрагментов. В каждом разделе фрагменты выполняются последовательно.

При выполнении всех фрагментов многораздельных транзакций поддерживаются журналы откатов. Они требуются для поддержки двухфазного протокола фиксации. Координатор посылает сообщение "подготовиться к фиксации" в каждый поток управления основного раздела, участвующий в выполнении данной распределенной транзакции, вместе с последним предназначенным этому потоку фрагментом транзакции. Получив это сообщение, поток основного раздела рассылает все полученные им ранее фрагменты этой транзакции в потоки резервных разделов, выполняет последний фрагмент транзакции, дожидается подтверждений от потоков резервных разделов и отправляет свои окончательные результаты координатору.

Если координатор получил все подтверждения от всех участников транзакции, он завершает транзакцию, посылая сообщение "фиксация" всем участникам и возвращая результаты транзакции приложению. Если же хотя бы от одного участника распределенной транзакции подтверждение о готовности к фиксации не получено, координатор посылает всем остальным участникам сообщение "аварийное завершение", принуждая их откатить все свои фрагменты транзакции. В результате, при выходе из строя некоторых узлов кластера или потере связности сети можно продолжать выполнять транзакции над разделами, оставшимися доступными клиентам (для одноразделных транзакций) и/или координатору (для многораздельных транзакций).

Тем самым, при выполнении многораздельных транзакций в потоках управления основных разделов могут возникать задержки из-за ожидания получения по сети сообщений от других потоков управления. Причем при использовании гигабитного Ethernet за время обмена сообщениями между двумя машинами по сети (40 миллисекунд) в потоке управления можно выполнить почти две одноузловые транзакции из тестового набора TPC-C. В [27] предлагаются и сравниваются методы, позволяющие системе выполнять полезную работу в то время, в которое она в противном случае простаивала бы.

Самый простой способ управления многоузловыми транзакциями заключается в том, что после получения первого фрагмента многораздельной транзакции поток управления основного раздела не обрабатывает фрагменты других многораздельных транзакций до тех пор, пока не обработает последний фрагмент данной транзакции. Фрагменты всех остальных транзакций ставятся в очередь, т.е. блокируются. Можно считать, что система расценивает все транзакции, затрагивающие один и тот же раздел, как *конфликтующие*, и выполняет их по очереди. Естественно, метод *блокирования транзакций* не позволяет системе выполнять полезную работу во время простоев из-за

ожидания сетевых сообщений. В [27] этот метод используется в качестве отправной точки для выявления преимуществ и сравнения более развитых методов.

Оригинальным методом выполнения транзакций, предложенным в [27], является метод *спекулятивного* выполнения. Основная идея состоит в том, что когда в некотором потоке управления основного раздела возникает задержка из-за выполнения двухфазного протокола распределенной фиксации транзакций, этот поток не простаивает, а выполняет другие фрагменты транзакций из своей входной очереди. Это выполнение на фоне фиксации транзакции является спекулятивным (*рискованным*), потому что результаты выполненных таким образом транзакций будут некорректными, если транзакция, на фоне фиксации которой эти транзакции выполняются, не зафиксируется, а откатится. Поэтому все спекулятивно выполненные транзакции не фиксируются до тех пор, пока не будет зафиксирована первая транзакция, в случае ее отката – все тоже откатываются (тем самым, для всех спекулятивно выполняемых транзакций необходимо поддерживать журналы откатов). Очевидно, что спекулятивная схема обеспечивает сериальный план выполнения транзакций.

Достаточно проста схема спекулятивного выполнения одноразделных транзакций. Для каждого потока управления основного раздела поддерживаются две очереди: очередь *невыполненных* транзакций и очередь *незафиксированных* транзакций (при спекулятивном выполнении первой в этой очереди всегда будет многораздельная транзакция, на фоне фиксации которой происходит спекулятивное выполнение). Если многораздельная транзакция откатывается, то по очереди откатываются все спекулятивно выполненные одноразделные транзакции и ставятся в начало очереди невыполненных транзакций для повторного выполнения (*в [27] по этому поводу ничего не сказано, но похоже, что в этом случае перед откатом каждого фрагмента требуется послать требования отката во все потоки управления соответствующих резервных разделов; нельзя ставить одноразделную транзакцию в очередь для повторного выполнения, пока не выполнен откат резервных разделов. – С.К.*). Если многораздельная транзакция фиксируется, то окончательно фиксируются и все транзакции из очереди незафиксированных транзакций, т.е. они удаляются из этой очереди, и их результаты посылаются в приложение. После обработки всех спекулятивно выполненных незафиксированных транзакций система возобновляет выполнение транзакций в не спекулятивном режиме.

Что касается спекулятивного выполнения многораздельных транзакций, то в [27] предлагается поддержка такого выполнения только для *простых многораздельных транзакций*, т.е. для транзакций, в которых с каждым разделом базы данных работает не более чем один фрагмент. Отмечается распространенность таких транзакций в транзакционных рабочих нагрузках

(например, все транзакции эталонного тестового набора TPC-C относятся к этой категории).

Пусть, например, в некотором потоке управления основного раздела на фоне выполнения двухфазного протокола фиксации многораздельной транзакции $T_1$ после спекулятивного выполнения транзакций $T_2$, ..., $T_{n-1}$ спекулятивно выполняется фрагмент другой многораздельной транзакции $T_n$. Тогда после завершения выполнения этого фрагмента координатору посылаются его результаты (вместе с подтверждением готовности к фиксации $T_n$), а также указывается, что эти результаты зависят от $T_1$.

Если координатор принимает решение зафиксировать $T_1$, то спекулятивные результаты $T_n$ являются корректными, и координатор может учесть это при фиксации $T_n$. В этом случае окончательно фиксируются транзакции $T_1$, ..., $T_{n-1}$, а на фоне ожидания решения координатора о фиксации транзакции $T_n$ продолжается спекулятивное выполнение транзакций из очереди необработанных транзакций потока управления.

Если же координатор принимает решение об откате $T_1$, то он просто отбрасывает результаты транзакции $T_n$, полученные из данного потока управления. При обработке аварийного завершения $T_1$ откатываются и заново ставятся в очередь необработанных фрагментов все спекулятивно выполненные транзакции, включая $T_1$.

Достоинством метода спекуляций является возможность избежать простоев в работе потоков управления основных разделов без использования синхронизационных блокировок и отслеживания наборов чтения и записи транзакций. Недостатками и ограничениями являются:

- возможность перехода в спекулятивный режим выполнения только после обработки последнего фрагмента многораздельной транзакции (т.е. простои между выполнением последовательных фрагментов одной и той же транзакции возможны);
- потребность в едином координаторе для все многораздельных транзакций, чтобы можно было допустить их спекулятивное выполнение (как отмечалось ранее, центральный координатор может ограничивать потенциально возможную пропускную способность системы);
- потенциальная возможность лишних откатов, поскольку неявно предполагается, что все транзакции, работающие с одним и тем же разделом базы данных, конфликтуют.

Наконец, последняя схема, рассматриваемая в [27], основывается на *синхронизационных блокировках*. Идейно эта схема очень близка к схеме применяемой в [15] (см. подраздел 3.2). Поскольку все фрагменты транзакций, адресуемые одному разделу базы данных, выполняются в одном потоке управления, нет потребности в менеджере блокировок, доступ к которому производился бы из разных потоков управления. Тем самым, нет потребности

в какой-либо синхронизации доступа к структурам данных, разделяемым между несколькими потоками управления. Более того, если в некотором потоке управления отсутствуют активные (незафиксированные) многораздельные транзакции, то поступающие в него одноразделные транзакции могут выполняться без блокировок (и без журналов откатов, если в них невозможно аварийное завершение по инициативе пользователей). Но как только в этом потоке начинает обрабатываться многораздельная транзакция, все остальные транзакции должны выполняться с блокировками и с поддержкой журналов откатов (для разрешения возможных тупиковых ситуаций).

В этом случае перед выполнением любой операции чтения или изменения записи в некотором фрагменте транзакции производится попытка установки синхронизационной блокировки этой записи в соответствующем режиме. Если эта попытка удается, выполнение фрагмента продолжается, иначе выполнение приостанавливается, фрагмент заносится в список блокированных фрагментов, и выбирается следующий фрагмент из очереди необработанных фрагментов. Аналогично, следующий фрагмент выбирается на обработку и при завершении выполнения очередного фрагмента.

Если завершается последний фрагмент многоузловой транзакции, то все ее выполненные фрагменты посылаются в потоки управления резервных разделов. Там они выполняются последовательно без всякой синхронизации, поскольку синхронизационные блокировки продолжают удерживаться в потоке основного раздела. После выполнения фрагментов транзакции в резервных узлах она становится готовой к фиксации в данном потоке управления.

Как точно выполняется фиксация многораздельной транзакции при использовании синхронизационных блокировок, в [27] не описывается. Говорится только, что в этом случае не требуется центральный координатор, поскольку сериализацию транзакций гарантирует поддержка строго двухфазного протокола фиксации в потоках управления каждого основного раздела, и фрагменты многораздельной транзакции рассылаются по потокам управления прямо из клиентов. Думаю, что имеется в виду, что каждый клиент координирует свою распределенную транзакцию, поскольку непонятно, как обойтись без использования двухфазной фиксации.

По всей видимости, именно этому "частному" координатору после завершения последнего фрагмента транзакции в некотором потоке управления посылается сообщение, подтверждающее готовность к фиксации. Во время ожидания последнего сообщения от координатора в этом потоке управления могут выполняться другие фрагменты. Если от координатора поступает сообщение "фиксация", транзакция может освободить свои блокировки данных в соответствующем разделе и, тем самым, обеспечить возможность продолжения выполнения некоторых ранее блокированных фрагментов. Если

же от координатора приходит сообщение "аварийное завершение", то транзакция должна послать всем потокам управления резервных разделов сообщение, требующее отката всех фрагментов, дождаться подтверждения (в это время могут выполняться фрагменты других транзакций), выполнить собственный откат и, в конце концов, освободить блокировки. Еще раз замечу, что это мои собственные домыслы, поскольку в [27] об этом ничего не говорится.

Естественно, при использовании схемы с синхронизационными блокировками возможно возникновение как локальных (в пределах одного потока управления), так и распределенных (затрагивающих блокировки объектов разных разделов в разных потоках управления) синхронизационных тупиков. Локальные тупики выявляются путем обнаружения циклов в графах ожидания, а распределенные – за счет таймаутов. При обнаружении тупика, для его разрешения, система старается аварийно завершать одноразделные транзакции, поскольку их дешевле выполнить повторно (*кроме того, как мне представляется, само по себе аварийное принудительное завершение распределенной транзакции тоже стоит недешево – С.К.*).

Авторы [27] выполнили ряд интересных и содержательных экспериментов, которые, в частности, показали, что:

- блокировочная схема почти всегда существенно уступает и спекулятивной схеме, и схеме с синхронизационными блокировками;

- спекулятивная схема работает значительно лучше двух других схем при наличии многораздельных транзакций с одним циклом коммуникаций транзакции с координатором (т.е. для *простых многораздельных транзакций*, в которых для каждого используемого раздела имеется один фрагмент) и небольшой доли аварийно завершающихся транзакций;

- схема с синхронизационными блокировками обеспечивает наилучшие результаты при наличии многих транзакций с несколькими циклами коммуникаций (*для справедливости замечу, что спекулятивный способ выполнения таких транзакций настолько сложен и непонятен, что авторы его даже и не описывают – С.К.*).

Общий вывод состоит в том, что в системе стоило бы вести статистику разновидностей транзакций во время выполнения и выбирать метод, следуя, например, модели, показанной в табл. 1.

| | | Редкие аварийные завершения | | Частые аварийные завершения | |
|---|---|---|---|---|---|
| | | Мало конфликтов | Много конфликтов | Мало конфликтов | Много конфликтов |
| Мало транзакций с несколькими циклами коммуникации | Много многораздных транзакций. | Спекулятивное выполнение | Спекулятивное выполнение | Синхрятивное блокирование | Синхрятивное блокирование и спекулятивное выполнение |
| | Мало многораздных транзакций. | Спекулятивное выполнение | Спекулятивное выполнение | Блокировка выполнения или синхронизация блокировки | Блокировка выполнения |
| Много транзакций с несколькими циклами коммуникации | | Синхронизация блокировки | Синхронизация блокировки | Синхронизация блокировки | Синхронизация блокировки |

*Табл. 1. Наилучшие схемы управления параллелизмом для разных ситуаций.*

### *3.1.4. Детерминированное выполнение транзакций*

Как видно из материала предыдущих пунктов, в H-Store имеются две основные проблемы, обе связанные с поддержкой многораздельных транзакций: отображение результатов транзакции в репликах базы данных и двухфазная фиксация. В [28] полагается, что эти проблемы можно решить путем перехода к полностью детерминированной схеме выполнения транзакций. (*Замечу, что при этом описываемое исследование выполнено не в контексте общей архитектуры H-Store, хотя имеет явное отношение к этому проекту. – С.К.*) Связанное с поддержкой свойств ACID требование *сериализации* транзакций, по мнению авторов [28], традиционно формулируется слишком слабо, поскольку требуется эквивалентность плана выполнения смеси транзакций *какому-либо* плану их последовательного выполнения, что оставляет простор для недетерминизма. При *детерминированной* сериализации смеси транзакций $\{T_1, T_2, ..., T_n\}$ требуется эвивалентность плана выполнения этих транзакций некоторому *предопределенному* последовательному плану их выполнения ($T_{i1}, T_{i2}, ..., T_{in}$).

Простейшим способом детерминированного выполнения смеси транзакций, гарантирующим эквивалентность предопределенному последовательному плану было бы последовательное выполнение транзакций в порядке этого

плана без какого-либо параллелизма. Однако в большинстве случаев это привело бы к неоптимальному использованию компьютерных ресурсов и, тем самым, к плохой производительности системы. Поэтому авторы предлагают использовать синхронизационные блокировки, но при соблюдении следующих ограничений, гарантирующих эквивалентность получаемого сериального плана предопределенному плану:

- Если двум транзакции $T_i$ и $T_j$ требуются блокировки одной и той же записи $r$, и в предопределенном плане $T_i$ находится раньше, чем $T_j$, то $T_i$ должна запросить блокировку $r$ *раньше*, чем $T_j$ (т.е. должна использоваться схема *упорядоченного запроса блокировок* – помимо прочего, легко видеть, что при использовании такой схемы невозможно возникновение синхронизационных тупиков).

- Каждая транзакция, не ожидающая удовлетворения запроса блокировки, должна продолжать выполняться до тех пор, пока не зафиксируется или не будет аварийно завершена *детерминированным образом* (т.е. в соответствии с логикой приложения). Если выполнение какой-либо транзакции задерживается (например, из-за какого-то сбоя в системе), то система должна поддерживать эту транзакцию активной, пока она не завершится, или пока не будет ликвидирована сама система.

Эксперименты авторов показывают, что если возможны длительные задержки при выполнении какой-либо транзакции, то детерминированная схема приводит к быстрому "загромождению" системы блокированными транзакциями и резкому падению производительности. Поэтому, в частности, детерминированная схема непригодна для СУБД, работающих с базами данных в дисковой памяти. Однако в системах, обрабатывающих данные исключительно в основной памяти, ситуация меняется. В ряде случаев детерминизм позволяет повысить производительность систем, упрощая при этом репликацию и избавляя от потребности в двухфазном протоколе фиксации распределенных транзакций.

Возможная архитектура детерминированной реплицированной СУБД показана на рис. 3. Запросы на образование транзакций (как и раньше, транзакции являются заранее определенными и сохраняемыми в виде хранимых процедур) от пользовательских приложений поступают в *препроцессор*, являющий границей детерминированной системы. Препроцессор выполняет всю необходимую недерминированную подготовительную работу (например, параметризует транзакции в соответствии с указаниями пользователей) и упорядочивает транзакции. После этого транзакции объединяются в пакет и надежно сохраняются. С этого момента система обязуется выполнить все поступившие транзакции, и все выполнение далее производится в соответствии с установленным в пакете

порядком выполнения транзакций. Наконец, пакет транзакций надежным образом отсылается во все системы, содержащие реплики базы данных (в этой работе неявно полагается, что базы данных реплицируются целиком, так что любую транзакцию можно полностью выполнить в любой реплике).


*Рис. 3. Детерминированная система.*

От каждой системы-реплики требуется только то, чтобы в ней была реализована некоторая модель выполнения, гарантирующая отсутствие синхронизационных тупиков и эквивалентность порядку выполнения транзакций, установленному препроцессором. Все системы реплики работают полностью независимо (*по всей видимости, сообщая препроцессору о завершении обработки полученного пакета транзакций – С.К.*). В случае отказа какой-либо системы реплики ее восстановление производится на основе реплик, сохранивших работоспособность (*кстати, в контексте [28] это не очень важно, поскольку эксперименты производились на прототипе, не поддерживающем репликацию, – С.К.*).

Для поддержки упорядоченности запросов блокировок в [28] предлагается запрашивать все блокировки, требуемые для каждой транзакции, перед началом ее выполнения (*заметим, что если, как в предыдущем пункте, для*

*каждого раздела базы данных используется только один поток управления, то, как и раньше, до появления первой многораздельной транзакции все предшествующие одноразделные транзакции можно выполнять без синхронизационных блокировок – С.К.*). Если это возможно, то после этого транзакция выполняется вплоть до своего завершения (над данным разделом), не освобождая блокировок. Однако не для всех транзакций заранее известно, какие записи в них будут читаться и изменяться. Например, возможна следующая транзакция:

```
T(x):
y := read(x)
write(y)
```

Здесь параметр *x* указывает на некоторую запись, содержащую значение первичного ключа той записи, которую требуется обновить. В этом случае невозможно сразу запросить синхронизационную блокировку второй записи, поскольку значение ее первичного ключа неизвестно до выполнения первой операции чтения. Такие транзакции в [28] называются *зависимыми*. В предлагаемой схеме зависимые транзакции разбиваются на несколько транзакций, из которых все транзакции, кроме последней, занимаются исключительно выяснением состава наборов чтения и записи, а последняя транзакция начинает свое выполнение при наличии полного знания наборов записей, которые она будет читать и изменять. Например, транзакцию $T$ можно разбить на следующие транзакции $T_1$ и $T_2$:

```
T1(x):
    y := read(x)
    запрос_следующей_транзакции(T2(x, y))
T2(x, y):
    y´ := read(x)
    if (y´ ≠ y)
        запрос_следующей_транзакции(T2(x, y))
        abort()
    else
        write(y)
```

Препроцессор не включает транзакцию $T_2$ в пакеты транзакций, пока не получит результат транзакции $T_1$. При начале выполнения $T_2$ блокируются записи с ключами, содержащимися в *x* и *y*, а затем проверяется, что за время, прошедшее между завершением $T_1$ и началом $T_2$ не изменилось содержимое записи, на которую указывает *x*. Если эта проверка оказывается успешной, выполнение $T_2$ продолжается. В противном случае $T_2$ аварийно завершается с освобождением своих блокировок. Об этом оповещается препроцессор, который снова включает $T_2$ в следующий пакет транзакций. Все действия по аварийному завершению транзакций и выполнению новых попыток детерминированы; они одинаковым образом выполняются во всех системах-репликах.

В этом примере для разбиения транзакции *T* требуется только одна дополнительная транзакция (в *T* имеется зависимость *первого порядка)*. По наблюдениям авторов [28], такие транзакции часто встречаются в реальных рабочих нагрузках OLTP. Транзакции с зависимостями более высокого порядка встречаются реже, но теоретически с ними можно справиться с помощью того же приема. Эксперименты и аналитическое моделирование показали, что наличие в рабочей нагрузке OLTP транзакций с зависимостями первого порядка не может служить основанием для отказа от детерминированной схемы выполнения транзакций.

Наиболее интересен детерминизм при выполнении многораздельных транзакций. Общая схема выполнения многораздельной транзакции в [28] не описывается, но идея состоит в том, что препроцессор разбивает каждую многораздельную транзакцию на фрагменты, каждый из которых затрагивает данные только одного раздела. Для каждого фрагмента обеспечивается информация о том, какие сообщения могут поступить от других фрагментов (включая сообщения, содержащие данные, и сообщения о детерминированном аварийном завершении). Фрагмент, получивший сообщение об аварийном завершении, освобождает свои синхронизационные блокировки и завершается. Фрагмент, получивший от других фрагментов все требуемые данные, выполняется до своего логического конца и освобождает синхронизационные блокировки. Не требуется использование какого-либо протокола фиксации транзакции, поскольку отказ любого узла означает отказ данной системы-реплики. Транзакция зафиксируется в какой-либо другой реплике, и на ее основе будет восстановлена отказавшая система.

Еще раз подчеркну, что это только идея. Думаю, что авторы [28] сами до конца не продумали эту схему, которая в общем случае может оказаться очень нетривиальной. В своих экспериментах, демонстрирующих преимущество детерминированного выполнения многораздельных транзакций, они использовали очень простые транзакции из тестового набора TPC-C, а на общий случай пока не замахивались. В этой статье я не берусь разобраться со всеми возникающими сложностями, но нельзя не согласиться, что перспектива обойтись без двухфазного протокола фиксаций за счет детерминированного выполнения транзакций кажется очень привлекательной и потенциально достижимой. Как бы только не оказалось, что для этого требуется еще более сложный статический анализ транзакций, чем тот, который упоминался ранее в этом подразделе.

### *3.1.5. Автоматизация методов разделения и реплицирования баз данных*

Понятно, что как не борись с двухфазной фиксацией распределенных транзакций, многораздельные транзакции останутся более дорогостоящими, чем одноразделные. И хотя в общем случае от многораздельных транзакций никуда не денешься, нужно стремиться к тому, что в любой рабочей нагрузке

OLTP их было как можно меньше. Понятно, что поскольку для любой системы баз данных рабочая нагрузка является независимым внешним фактором, можно лишь стремиться физическим образом организовать разделенную и реплицированную базу данных таким образом, чтобы среди транзакций именно этой рабочей нагрузки было по возможности меньше многораздельных транзакций.

Методы циклического разделения (каждый следующий кортеж направляется в следующий раздел) и хэш-разделения (кортежу назначается раздел в соответствии со значением некоторой хэш-функции от значения его атрибута разделения), часто с успехом применяемые в аналитических массивно-параллельных системах баз данных, как правило, не подходят для транзакционных массивно-параллельных систем баз данных, поскольку способствуют появлению большого числа многораздельных транзакций. Хорошие результаты может обеспечить разделение по диапазонам значений кортежей (выбор раздела для данного кортежа основывается на вычислении логического выражения, построенного на основе вхождений значений атрибутов кортежа в заданные диапазоны значений), но выбор соответствующих диапазонов с учетом заданной рабочей нагрузки вручную производить очень трудно.

Подход к решению этой проблемы представлен в [14]. В общих словах, на основе одноразделного представления базы данных и заданной рабочей нагрузки производится разделение базы данных на заданное число сбалансированных разделов с целью минимизации числа многораздельных транзакций в рабочей нагрузке. Далее система пытается аппроксимировать полученное разделение разделением по диапазонам значений на основе автоматически производимого условного выражения. Наконец, производится сравнение числа распределенных транзакций в исходной рабочей нагрузке, которые образуются при применении построенного метода разделения, с числом распределенных транзакций, которые возникают при использовании хэш-разделения и разделения на уровне таблиц, и для реального использования выбирается метод, обеспечивающий наилучший результат.

На первом этапе строится граф, в котором вершины соответствуют кортежам всех таблиц базы данных, и дуги связывают все кортежи, используемые в одной и той же транзакции. Вес каждой дуги – число транзакций, обращающихся к данной паре кортежей. На рис. 5 изображен граф для базы данных, состоящей из одной таблицы `ACCOUNT` с пятью кортежами, и рабочей нагрузки из четырех транзакций, показанных на рис. 4. На рис. 5 показаны четыре части графа, соответствующие четырем транзакциям с рис. 4.

**База данных**

ACCOUNT

| id | name | balance |
|---|---|---|
| 1 | Carlo | 80K |
| 2 | Evan | 100K |
| 3 | Sam | 129K |
| 4 | Eugene | 29K |
| 5 | Yang | 12K |

**Рабочая нагрузка**

**I**

```
BEGIN
UPDATE account SET bal=bal-1k
  WHERE name="carlo";
UPDATE account SET bal=bal+1k
  WHERE name="evan";
COMMIT
```

**II**

```
BEGIN
UPDATE account SET bal=60k
  WHERE id=2;
SELECT * FROM account
  WHERE id=5;
COMMIT
```

**III**

```
BEGIN
SELECT * FROM account
  WHERE id IN {1,3};
ABORT
```

**IV**

```
BEGIN
UPDATE account SET bal=bal+1k
  WHERE bal < 100k;
COMMIT
```

*Рис. 4. Примерные база данных и рабочая нагрузка*

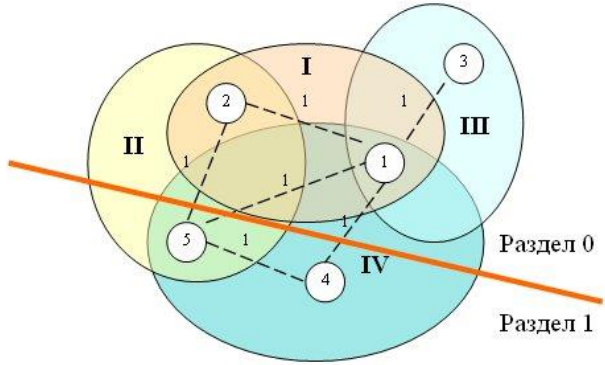



*Рис. 5. Общая идея графа, используемого для разделения базы данных.*

На рис. 6 показано расширенное представление графа с учетом возможности репликации на уровне кортежей. Для этого каждая вершина, соответствующая кортежу, к которому обращается $n > 1$ транзакций, заменяется "звездообразным" подграфом из $n+1$ вершин. Веса дуг, соединяющих вершины-реплики с центральной вершиной, характеризуют стоимость репликации данного кортежа и определяются как число транзакций в данной рабочей нагрузке, обновляющих данный кортеж. Например, на рис. 6 показано, что кортеж с `id`=1 представлен четырьмя вершинами, поскольку к нему обращаются три транзакции (I, III и IV). Веса дуг у соответствующего подграфа равны 2, поскольку только транзакции I и IV обновляют этот кортеж. Такая графовая структура позволяет алгоритму разделения соблюдать баланс между стоимостью репликации и выигрышем от ее применения.

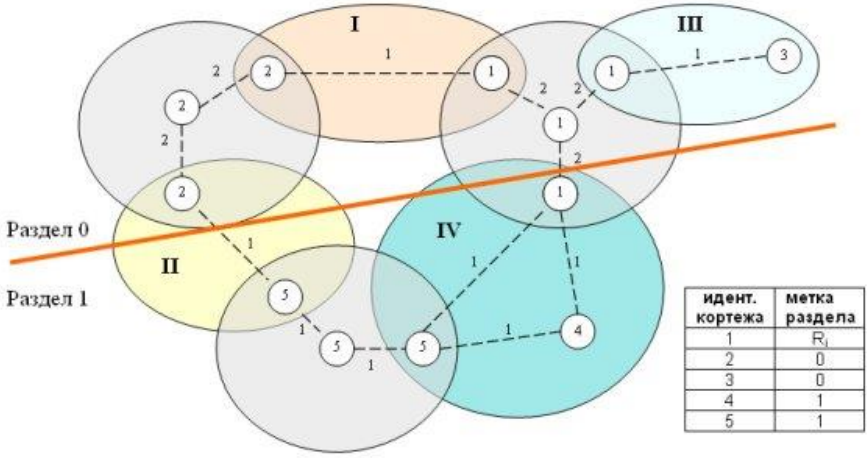



*Рис. 6. Граф с учетом возможности репликации*

Далее этот граф расщепляется на $k$ разделов без общих вершин с минимизацией общей стоимости разрезания дуг (суммы весов дуг, концы которых оказываются в разных разделах графа). Другим ограничением является примерная балансировка весов разделов (допустимое отклонение является параметром системы). Весом раздела считается сумма весов вершин, входящих в этот раздел. Вес узла можно определять по-разному, и авторы [14] экспериментировали со случаями, когда вес вершины равен числу байт в соответствующем кортеже (в этом случае выполняется балансировка по размеру базы данных) и когда вес раздела равен числу обращений к кортежу (балансировка по рабочей нагрузке).

Расщепление графа на *k* частей при наличии подобных ограничений – это *NP*-полная проблема. Однако оказывается, что подобные задачи часто приходится решать в области автоматизации проектирования сверхбольших интегральных схем. За последние десятилетия были найдены сложные эвристические правила расщепления графов и созданы хорошо оптимизированные свободно доступные библиотеки программного обеспечения, позволяющие обрабатывать графы с сотнями миллионов дуг. В [14] использовались программные средства *METIS* [31]. Утверждается, что расщепление графа "производится быстро" (хотя абсолютные цифры не приводятся). Вместе с тем, отмечается, что при росте графа скорость обработки быстро возрастает, из-за чего авторы выработали ряд эвристик, позволяющих сдерживать размер графов (см. ниже).

В результате расщепления графа порождается отображение вершин-кортежей на набор меток разделов. Одним из способов использования этого результата является сохранение отображения в некоторой таблице типа той, которая показана в правом нижнем углу рис. 6. Как видно из этой таблицы, для нереплицируемых кортежей можно прямо указать номер раздела. Реплицируемые кортежи помечаются специальным образом, позволяющим понять, в каких разделах должны размещаться реплики.

В распространенном случае, когда в разделах `WHERE` операторов SQL содержатся условия сравнения на равенство с константой или вхождения в диапазон заданных значений, поисковые таблицы можно непосредственно использовать для направления операции в соответствующий(ие) раздел(ы). При наличии плотного множества идентификаторов кортежей и не более 256 разделов в 16 гигабайтной основной памяти можно хранить таблицу о разделении 15 миллиардов кортежей. Кортежи, заново вставляемые в базу данных, сначала могут помещаться в произвольные разделы, а после пересчета разделения графа их можно переместить в нужные разделы. Однако для очень крупных систем баз данных при наличии рабочей нагрузки с интенсивной вставкой кортежей этот подход может оказаться неудовлетворительным. Поэтому авторы [14] разработали дополнительное инструментальное средство, позволяющее аппроксимировать разделение, получаемое при обработке графа, методом разделения по диапазонам значений.

Это инструментальное средство основывается на методах машинного обучения, и в нем активно используются возможности свободно доступного пакета программных средств интеллектуального анализа данных WEKA [32]. Сначала на основе трассы рабочей нагрузки создается обучающая выборка. Для сокращения времени работы из трассы выделяется представительные образцы кортежей, которые помечаются метками разделов, полученными при расщеплении графа.

Затем разбираются операторы SQL, присутствующие в трассе рабочей нагрузки, и выделяются атрибуты кортежей, наиболее часто присутствующие

в условиях разделов `WHERE`. Выбранные атрибуты обрабатываются компонентом *отбора признаков (feature selection)* пакета WEKA, которые отбирает атрибуты, коррелирующие с метками разделов.

Наконец, на основе обучающей выборки и отобранного набора атрибутов строится классификатор в виде дерева решений (используется реализация J48 из пакета WEKA). На выходе классификатора получается набор условий, аппроксимирующих разделение на уровне кортежей, которое было произведено при расщеплении графа.

Авторы называют процедуру получения аппроксимирующих условий *толкованием (explanation)* разделения графа. Отмечается, что получить разумное толкование не всегда возможно, и толкование является полезным только при выполнении следующих условий: (1) оно основывается на атрибутах, часто используемых в запросах; (2) не слишком снижает качество разделения за счет неправильной классификации кортежей; (3) успешно работает для операторов SQL, не использованных для построения обучающей выборки.

Последним шагом подхода Schism при выборе метода разделения является сравнение числа распределенных транзакций, которые обеспечиваются при заданной рабочей нагрузке схемой поисковой таблицы, полученной при разделении графа; схемой разделения по условиям вхождения в диапазоны значений, сгенерированной при толковании разделения графа; схемой хэш-разделения по наиболее часто используемым атрибутам и схемой репликации базы данных на уровне таблиц. Выбирается схема, приводящая к наименьшему числу распределенных транзакций. Если несколько схем обеспечивают близкие результаты, то выбирается наименее сложная схема.

Поскольку с ростом графа, связывающего рабочую нагрузку с кортежами базы данных, значительно возрастает время его расщепления, в [14] предлагается ряд эвристик, позволяющих сократить размер графа без существенного влияния на результаты разделения, толкования и т.д. К числу этих эвристик относятся взятие образцов на уровне транзакций, взятие образцов на уровне кортежей, отбрасывание операций SQL, приводящих к сканированию больших частей таблиц и т.д. Эксперименты, описанные в [14], показывают, что подход Schism позволяет добиться качества разделения баз данных, соизмеримого с качеством наилучших схем, получаемых вручную.

Подводя итог обсуждению разных аспектов проекта H-Store, еще раз отмечу его основные черты:

- бескомпромиссное использование подхода shared-nothing – каждому разделу базы данных (основному или резервному) соответствует в точности один поток управления, и в потоках управления не используются общие ресурсы, даже если они реализуются ядрами одного и того же процессора;

- поддержка баз данных в основной памяти; долговременность хранения обеспечивается только за счет репликации данных в разных узлах кластера;
- выполнение транзакций поблизости от данных без потребности в передаче по сети операций SQL и их результатов;
- стремление к минимизации числа распределенных транзакций за счет статического анализа и преобразований транзакций, а также за счет разделения данных с учетом рабочей нагрузки.

Последнее желание понятно, поскольку подход H-Store показывает чудеса производительности именно при наличии только одноразделных транзакций. К сожалению, даже если удастся реализовать все мыслимые и немыслимые преобразования транзакций, появление распределенных транзакций полностью исключить не удастся, и поэтому основной текущей задачей проекта H-Store является нахождение способов выполнения распределенных транзакций, которые позволили бы свести к минимуму накладные расходы на их фиксацию.

Несмотря на наличие ряда нерешенных проблем, на основе промежуточных результатов проекта H-Store успешно стартовала компания VoltDB. А это означает, что имеются пользователи, для которых новый уровень производительности транзакционных систем важнее технологической завершенности предлагаемых решений.

## 3.2. DORA: почти «shared-nothing» в среде «shared-everything»

Подход H-Store с отказом от совместного использования ресурсов лишен компромиссов: в этой архитектуре любой аппаратно поддерживаемый поток управления рассматривается как отдельный узел системы со своими собственными ресурсами. В этой бескомпромиссности кроется много плюсов: в частности, параллельная СУБД строится в виде набора полностью идентичных компонентов, каждый из которых обладает единственной активностью. Но, с другой стороны, при этом приносятся в жертву аппаратные возможности многоядерных процессоров, позволяющие *на физическом уровне* использовать все ресурсы компьютера в потоках управления всех ядер.

В связи с этим мне кажется очень поучительным проект DORA [15], в котором разрабатывается архитектура СУБД, обладающая свойствами shared-nothing на логическом уровне, но использующая аппаратные возможности shared-everything на физическом уровне. В качестве экспериментальной аппаратной платформы в DORA использовался компьютер Sun T5220 "Niagara II". В микропроцессоре Niagara II имеется 8 ядер, в каждом из которых на аппаратном уровне поддерживается 8 потоков управления, т.е. на уровне операционной системы в компьютере имеется 64 процессора, каждому из которых доступны все остальные ресурсы системы.

Исследование [15] выполнялось не в том же контексте, что H-Store; в DORA основной упор делается на сокращение взаимодействий с центральным менеджером блокировок. Поэтому в большей части этого раздела я опишу основные идеи DORA в том виде, как они подаются авторами [15]. Однако, с моей точки зрения, между проектами H-Store и DORA имеется более глубокая связь, чем это отмечается авторами, например, [14] и [15], и на этой связи я остановлюсь в заключение подраздела.

### *3.2.1. Проблемы блокировок в традиционных многопоточных СУБД*

Авторы [15] использовали в своем проекте транзакционную систему управления базами данных Shore-NT [33], являющуюся модифицированным вариантом системы Shore [30] с многопотоковым ядром. В Shore-NT (как и в Shore) поддерживаются все основные возможности традиционных транзакционных СУБД: ACID-транзакции с обеспечением полной изоляции на основе иерархических блокировок, управление буферным пулом, индексы на основе B-деревьев, классическое управление журнализацией и восстановлением баз данных. Каждой транзакции назначается отдельный поток управления. Как утверждается в [15], выбор пал на Shore-NT, поскольку среди всех реализаций СУБД с открытыми исходными текстами эта система лучше всего масштабируется при росте числа ядер в процессоре [34]. Однако я думаю, что не менее важную роль при этом выборе сыграло и то, что в разработке Shore-NT активно участвовали именно участники проекта DORA.

С использованием Shore-NT были выполнены эксперименты, полностью подтвердившие выводы авторов [29] (которые экспериментировали с однопотоковым вариантом Shore) об отрицательном влиянии традиционных блокировок на производительность транзакционных СУБД. При традиционном назначении каждой транзакции отдельного потока управления в них приходится использовать большое число критических участков для координации доступа к совместно используемым ресурсам. Для организации критических участков в многоядерных процессорах приходится использовать *защелки (latch)*, основанные на применении *спинлоков* [35], что в ряде случаев вынуждает аппаратный поток управления "зависать" при входе в критический участок.

В Shore-NT (как и большинстве других многопотоковых систем с использованием общих ресурсов) каждой логической блокировке соответствует структура данных (описатель блокировки), содержащая режим блокировки, указатель на списки удовлетворенных или ожидающих удовлетворения запросов блокировки, а также *защелку*. Когда некоторая транзакция пытается получить некоторую блокировку, менеджер блокировок сначала проверяет, что для этой транзакции уже удерживаются блокировки намерений для объектов более высокого уровня, и в случае потребности

устанавливает требуемые блокировки (как уже упоминалось, в Shore-NT используется иерархическая схема блокировок, подробности об этой схеме см., например, в [36]). Если оказывается, что запрашиваемая блокировка транзакции не требуется, поскольку для нее уже удерживается подходящая блокировка более высокого уровня, текущий запрос сразу удовлетворяется.

В противном случае менеджер блокировок через хэш-таблицу ищет описатель требуемой блокировки (образуя его в случае отсутствия). Описатель блокировки "защелкивается", и запрос блокировки добавляется к списку запросов. Если запрос блокировки можно удовлетворить (требуемый объект не заблокирован или текущий режим его блокировки совместим с режимом запрашиваемой блокировки), то запрос помечается как удовлетворенный, защелка освобождается, и продолжается выполнение транзакции. Иначе запрос блокировки не удовлетворяется, и транзакция блокируется (защелка описателя блокировки при этом освобождается).

В каждой транзакции поддерживается список полученных ей блокировок. При завершении транзакции они освобождаются в хронологическом порядке. Для освобождения очередной блокировки менеджер блокировок защелкивает ее описатель, удаляет из списка запрос освобождаемой блокировки, устанавливает новый режим блокировки и удовлетворяет все отложенные запросы блокировки, которые можно удовлетворить (активизируя соответствующие транзакции). После этого защелка описателя блокировки освобождается.

По мере роста числа активных транзакций (т.е. числа доступных потоков управления, которое, очевидно, коррелирует с числом ядер процессора) растет объем работы, требуемой для удовлетворения запросов на получение или освобождение блокировок, поскольку удлиняются списки запросов блокировок. Дополнительные обходы списков блокировок нужны для выявления синхронизационных тупиков (а все это увеличивает размер критических участков). Возникающие последствия губительны для производительности системы.

На рис. 7 показана разбивка по времени работы менеджера блокировок Shore-NT при возрастании коэффициента загрузки процессора. Как видно, при слабой загрузке системы 85% времени менеджер блокировок выполняет полезную работу. Однако по мере роста загруженности процессора растут расходы на обслуживание конкуренции потоков управления. При стопроцентной загрузке процессора 85% времени работы менеджера блокировок уходит на выполнение операций над защелками.

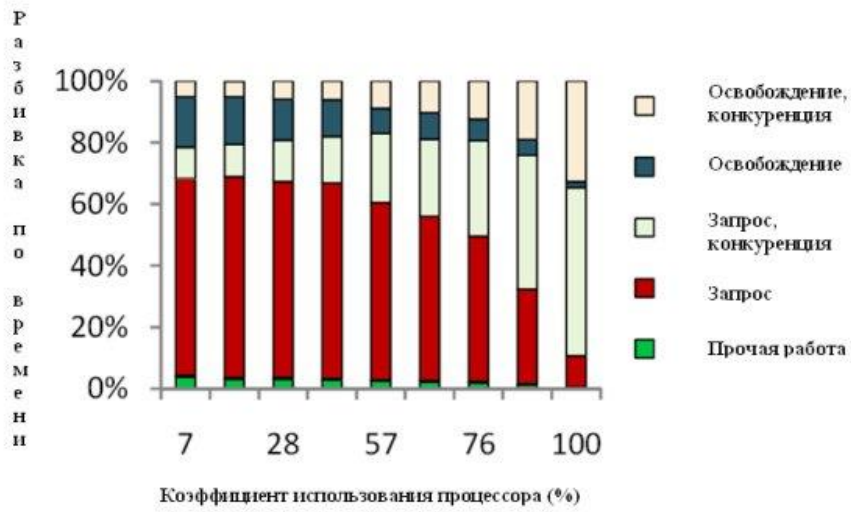



*Рис. 7. Разбивка по времени работы менеджера блокировок Shore-MT при выполнении тестового набора TPC-B.*

### *3.2.2. Архитектура DORA*

Основные идеи DORA (Data-ORiented Architecture) состоят в следующем:

- потоки управления связываются не с транзакциями, а с отдельными частями базы данных;
- система распределяет работу каждой транзакции по потокам управления в соответствии с тем, к каким данным обращается транзакция;
- при обработке запросов система по мере возможности избегает взаимодействий с централизованным менеджером блокировок.

Потоки управления (*исполнители*), в которых выполняются операции транзакций, связываются с данными за счет установки для каждой таблицы базы данных *правила маршрутизации*. Правило маршрутизации разделяет соответствующую таблицу на наборы записей (фактически, разделы) так, что каждая запись относится к одному и только одному набору. Каждый набор записей приписывается одному исполнителю, и одному исполнителю может быть приписано несколько наборов записей одной таблицы. Физический доступ к данным производится через общесистемный буферный пул, и правила маршрутизации не вызывают какого-либо физического разделения или перемещения данных. Правила маршрутизации поддерживаются во время работы системы менеджером ресурсов DORA и периодически им обновляются для балансировки нагрузки. Столбцы таблицы, используемые в

соответствующем правиле маршрутизации, называются *полями маршрутизации*.

Для распределения работы каждой транзакции по исполнителям в DORA для каждой транзакции образуется *граф потока транзакции*, в котором фиксируются зависимости действий транзакции и их связь с наборами записей. Действие – это часть транзакции, заключающаяся в обращении к одной записи или небольшому набору записей одной таблицы. С каждым действием связывается *идентификатор действия*, который может содержать значения полей маршрутизации или быть пустым. Последовательные действия с одним идентификатором можно слить.

На рис. 8 показан граф потока транзакции `Payment` из тестового набора TPC-C. В соответствии со спецификацией TPC-C [37], транзакция `Payment` изменяет отстаток на счете *клиента* (изменяет некоторую строку таблицы `Customer`), отражает данные о платеже в статистике продаж соответствующих *склада* и *округа* (изменяет по одной строке в таблицах `Warehouse` и `District`) и сохраняет данные о платеже в журнале *истории* (вставляет одну строку в таблицу `History`). Первичными ключами таблиц `Warehouse`, `District` и `Customer` являются `W_ID`, `(D_W_ID, D_ID)` и `(C_W_ID, C_D_ID, C_ID)` соответственно, и если полем маршрутизации для всех трех таблиц является идентификатор склада, то этот атрибут и становится идентификатором действий в группе операций в верхней части рис. 8. Поскольку пары действий чтения и изменения строки для каждой из таблиц `Warehouse`, `District` и `Customer` обладают одинаковыми идентификаторами, их можно слить.

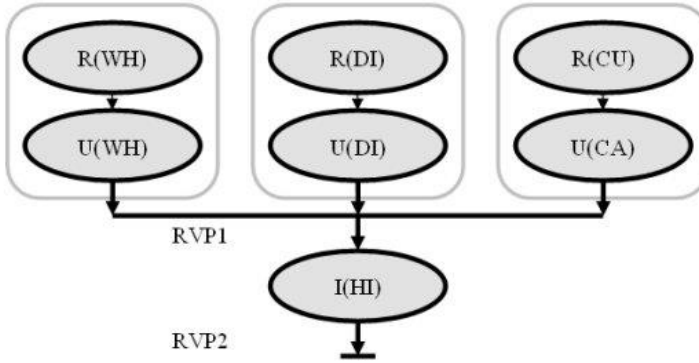



*Рис. 8. Граф потока транзакции `Payment` из тестового набора TPC-C.*

Пары действий над таблицами складов, округов и клиентов, вообще говоря, можно выполнять параллельно, поскольку между ними нет зависимостей по данным. Последнее действие транзакции `Payment` (вставку кортежа в таблицу `History`) нельзя выполнять раньше, чем завершатся все три объединенные предыдущие действия. Для управления распределенными транзакциями и передачи данных между действиями, зависимыми по данным, в DORA используются разделяемые между потоками управления объекты, называемые *точками рандеву (rendezvous point)* (*RVP*). Если между действиями транзакции имеются зависимости по данным, между ними помещается RVP.

RVP разделяют транзакции на фазы: система не может одновременно выполнять действия, относящиеся к разным фазам. В каждом объекте-RVP имеется счетчик, изначально содержащий число действий, которые должны "доложить" о своем завершении. Каждый исполнитель, завершая выполнение некоторого действия, уменьшает на единицу значение счетчика соответствующей RVP. Когда значение счетчика становится нулевым, начинается новая фаза транзакции: исполнитель, обнуливший счетчик RVP, ставит действия следующей фазы в очереди соответствующим исполнителям. Исполнитель, обнуливший счетчик последней RVP в графе потока транзакции, запрашивает ее фиксацию. Аварийно завершить транзакцию и инициировать ее откат может любой исполнитель.

Итак, к одному исполнителю направляются все действия, которые должны выполняться над одним и тем же набором записей. Исполнитель отвечает за изоляцию и упорядоченность конфликтующих действий. В каждом исполнителе поддерживаются три структуры данных: очередь поступающих действий, очередь завершенных действий и локальная таблица блокировок. Действия выполняются в том порядке, в котором они поступают во входную очередь. Для устранения конфликтов между действиями в исполнителях используются локальные таблицы блокировок. В качестве объектов блокировки применяются идентификаторы действий, и блокировки могут устанавливаться только в совместном и монопольном режимах. Каждое действие, получившее требуемую локальную блокировку, продолжает выполнение без обращения к ценрализованному менеджеру блокировок.

Локальные блокировки, полученные каждым действием транзакции, удерживаются в соответствующих исполнителях до фиксации или аварийного завершения этой транзакции. В заключительной RVP транзакция сначала дожидается ответа от основного менеджера управления данными о заключительной фиксации или аварийном завершении транзакции. После этого все действия этой транзакции ставятся в очереди завершенных действий своих исполнителей (откат транзакции в DORA производит основная система). При обработке этих действий исполнители освобождают соответствующие локальные блокировки и удовлетворяют запросы блокировок ожидающих действий.

Одна из проблем DORA состоит в том, что оказывается невозможно обойтись локальными блокировками идентификаторов действий при выполнении операций удаления и вставки строк таблиц. Действительно, возможна ситуация, когда транзакция *T*1 удаляет запись в некоторой странице, а транзакция *T*2, действие которой выполняется в другом исполнителе, вставляет на ее место свою запись до завершения транзакции *T*1. Тогда, если транзакция *T*1 завершится аварийным образом, ее нельзя будет откатить, поскольку слот удаленной записи уже занят.

Для устранения возможности такого конфликта перед выполнением операций удаления и вставки записей исполнители блокируют соответствующие им идентификаторы (т.е. номер страницы и номер слота внутри страницы). Утверждается, что потребность в таких блокировках возникает сравнительно редко, но беда в том, что эта проблема совсем не техническая – это обратная сторона того, что данные разделяются между исполнителями на логическом уровне. Получается, что без централизованной синхронизации вообще нельзя вставлять в таблицы новые строки, поскольку эти операции могут выполняться параллельно разными исполнителями, и должна обеспечиваться координация распределения памяти. В DORA имеются и другие проблемы, в частности, проблема выявления тупиковых ситуаций с участием локальных и централизованных блокировок, но я не буду на них останавливаться, поскольку, на мой взгляд, эти вопросы в [15] проработаны недостаточно глубоко.

DORA реализована поверх Shore-NT с минимальными изменениями базовой системы. В частности, были блокированы средства Shore-NT собственного управления транзакциями. Транзакции программировались в виде заранее компилируемых хранимых процедур. При тестировании системы на различных тестовых наборах было установлено, что выигрыш в производительности DORA по сравнению с базовой системой достигает 4,8 раз. При неполной загрузке системы за счет внутреннего параллелизма транзакций удается добиться сокращения времени ответа на 60%.

### *3.2.3. DORA, H-Store и компромиссы*

Одной из наиболее серьезных проблем транзакционных параллельных СУБД без использования общих ресурсов является балансировка нагрузки. Чтобы сбалансировать нагрузку, требуется изменить разделение и/или репликацию базы данных. Фактически, нужно переслать часть записей одной или нескольких таблиц из одного раздела в другой. Пересылка данных вызывает ощутимые накладные расходы (даже если вся база данных поддерживается в основной памяти), и в каждом из изменяющихся разделов необходимо должным образом изменить существующие индексы. Во время выполнения этих операций трудно продолжать поддерживать выполнение запросов, адресуемых к данным изменяемых разделов.

В DORA физическая пересылка данных не требуется. Не требуется и массовое преобразование индексов, поскольку индексы поддерживаются для таблицы целиком. Менеджер ресурсов, выявив потребность в расширении раздела одного исполнителя (*E*1) за счет сокращения раздела другого исполнителя (*E*2), изменяет правило маршрутизации для данной таблицы и формирует служебную транзакцию из двух действий, разделенных RVP. Первое действие ставится в очередь входных действий *E*2 и приводит к тому, что *E*2 дожидается, пока не закончатся все транзакции, в обработке которых он участвует, принимая при этом для обработки новые действия, посылаемые в соответствии с новым правилом маршрутизации. После завершения всех действий, направленных ему в соответствии со старым правилом маршрутизации, *E*2 ставит в очередь действие *E*1, позволяющее этому исполнителю обрабатывать те действия, которые направляются ему в соответствии с новым правилом маршрутизации. (*Должен признаться, что эта процедура придумана мной, поскольку в [15] она описана невразумительно. – С.К.*)

Вторая проблема, с которой приходится сталкиваться системам, не использующих общих ресурсов, возникает при обработке запросов, условие выборки которых не позволяет выявить один или несколько (немного) разделов, содержащих требуемые данные. В таком случае соответствующее действие должно направляться всем исполнителям, лишь немногим из которых удастся найти то, что требуется в запросе. В DORA для выполнения таких запросов можно воспользоваться общим для таблицы "вторичным" индексом, который может помочь направить действия требуемым исполнителям. (*На самом деле, с этими "общими" индексами в перспективе не все понятно. Пока DORA живет поверх Shore-NT, поддержка работы с индексами происходит "в другом мире". Но если пытаться продумать полную архитектуру с назначением потоков управления логическим разделам данных, то в ней должно найтись место и работе с индексами, а как это делать – непонятно. – С.К.*)

Несмотря на наличие многих неясностей, основная идея DORA – при наличии в системе физически общих ресурсов производить разделение данных не на физическом, а на логическом уровне – кажется мне очень привлекательной. Прототип DORA рассчитан на традиционную работу с дисками, в нем поддерживаются общая система буферов и журнализация (на уровне Shore-NT) и поэтому:

- можно спокойно делить данные на логическом уровне с точностью до кортежа (все буферные страницы доступны всем потокам управления);
- в нем удается обойтись без двухфазного протокола фиксации транзакций;
- и он плохо согласуется с общими идеями H-Store.

Однако мне кажется, что возможен компромисс между архитектурами H-Store и DORA *внутри* одного многоядерного компьютера для поддержки баз данных в основной памяти без журнализации изменений. Для этого нужно добиться полного отсутствия потребности в централизованных блокировках. Пусть, например, каждая таблица хранится в основной памяти на основе некоторого B-дерева и кластеризуется в соответствии с его ключом (возможно, составным). Все потоки управления работают с общей виртуальной памятью, накрывающей все базу данных целиком, но каждый исполнитель получает для выполнения фрагменты транзакций, которым требуются только данные поддеревьев, приписанных к этому исполнителю. Другими словами, "верхушка" B-дерева каждой таблицы используется координатором транзакций в качестве "правила маршрутизации", а соответствующие поддеревья (определяемые диапазонами значений ключей) используются в исполнителях для доступа к кортежам. Если еще приписать каждому координатору некоторое число свободных страниц основной памяти (для обеспечения возможности автономного расширения таблиц), то при выполнении фрагментов транзакций им никогда не потребуется централизованная синхронизация.

Балансировать нагрузку в этом случае можно будет путем перераспределения поддеревьев между исполнителями, работающими с "соседними" поддеревьями. Фактически, можно легко потребовать от любого исполнителя отдать некоторую "левую" часть своего поддерева некоторой таблицы своему соседу "слева" или "правую" часть своего поддерева некоторой таблицы своему соседу "справа". По-моему, здесь сработает почти та же процедура, которая используется для балансировки нагрузки в DORA. И не потребуется никакого физического копирования данных и/или перестройки индексов, поскольку реально все данные остаются на месте.

Реплицировать, по всей видимости, придется целиком всю базу данных, поддерживаемую многоядерным компьютером. Если считать, что репликация производится для обеспечения долговечности данных и восстановления системы после отказов узлов, то трудно представить себе такой отказ, при котором перестали бы работать несколько ядер процессора, а остальные сохранили работоспособность.

Для управления прохождением транзакций в такой гибридной архитектуре можно было бы использовать любой вариант, обсуждаемый разработчиками H-Store. В зависимости от этого понадобится или не понадобится двухфазная фиксация транзакций.

Так что мне представляется, что будущее поколение ACID-транзакционных систем будет опираться на две основные параллельные архитектуры одноузловую и многоузловую. Одноузловая архитектура предполагает наличие мощного многоядерного компьютера и использование энергонезависимой внешней памяти, в которой, в частности, должен поддерживаться журнал повторного выполнения транзакций. И в этом

направлении хороший фундамент закладывает DORA. Но нужно учитывать, что для достижения высокой производительности в такой архитектуре потребуется мощная параллельная система ввода-вывода, стоимость которой, вполне вероятно, будет определять стоимость системы в целом (В [15] эксперименты выполнялись с использованием файловой системы в основной памяти именно из-за отсутствия у авторов такой дорогостоящей дисковой подсистемы.)

Многоузловая архитектура строится на основе достаточно большого числа, вообще говоря, недорогих компьютеров, связанных сетью. Базы данных хранятся только в основной памяти, для обеспечения отказоустойчивости (и, следовательно, долговечности данных) используется репликация. В этом направлении хорошей основой является H-Store (и VoltDB), хотя мне кажется, что следовало бы учесть отмеченные выше возможности использования физически общих ресурсах в многоядерных узлах подобных систем.

В заключение этого раздела замечу, что, к сожалению, эти две архитектуры являются взаимоисключающими. Трудно представить компанию, которая хорошо потратилась на приобретение мощного многоядерного сервера с дорогой дисковой подсистемой, а потом решается отказаться от использования дисков и перейти к использованию многоузловой архитектуры. Трудно представить себе и ситуацию, когда сначала был выбран подход с использованием дешевых кластерных архитектур почти без дисков, а потом вдруг покупается отдельный дорогой сервер, и компания переходит к использованию одноузловой архитектуры. Так что, скорее всего, будет продолжать существовать и направление shared disks, свойственное, например, Oracle, поскольку оно позволяет достаточно эффективно использовать кластеры, построенные с использованием мощных дисковых серверов, которые до этого использовались автономно.

## *4. Рационализация согласованности*

Обсуждение этого направления лучше всего начать с работы [38], в которой впервые был предложен новый подход к оптимизации систем управления базами данных (немного в другом контексте этот подход обсуждался в [39]). Отмечается наличие в мире баз данных нескольких новых тенденций.

- Первой тенденцией является изменение приоритетов в требованиях приложений к системам управления данными. Для многих Web-приложений строгая согласованность данных в соответствии с парадигмой ACID не требуется. Зато им требуется возможность масштабирования приложения до миллионов пользователей, ни один из которых не должен блокироваться другими пользователями. Все запросы этих пользователей (и простые, и сложные) должны обрабатываться за гарантированное время в пределах секунды.

- Вторая тенденция состоит в возрастании сложности приложений, насыщенных данными, и в использовании для их организации распределенной сервис-ориентированной архитектуры.
- Наконец, имеются технологические тенденции, такие как "облачные" вычисления (cloud computing) и крупные центры данных, основанные на использовании дешевых аппаратных средств. Эти тенденции влияют на архитектуру программного обеспечения.

Если в "прошлой" жизни в системе управления данными *при заданном наборе аппаратных ресурсов и полноценной поддержке ACID-транзакций требовалось минимизировать время ответов на запросы и максимизировать пропускную способность системы*, то в новых условиях *при заданных требованиях к производительности приложений (пиковая пропускная способность, максимально допустимое время ответа) требуется минимизировать требуемые аппаратные ресурсы и максимизировать согласованные данные*. Сводка различий в формулировке проблемы оптимизации приведена в табл. 2.

Основным показателем системы баз данных, нуждающемся в оптимизации, является *денежная оценка затрат*. По мнению авторов, технология баз данных может удовлетворить любые требования к производительности и пропускной способности приложения, вопрос лишь в том, сколько для этого потребуется машин, т.е. сколько за это придется платить. При использовании "облачных" инфраструктур показатель расходов на аппаратные ресурсы становится непрерывным: чем больше их потребляется, тем больше приходится платить.

| *Характеристика* | *Традиционные базы данных* | *Новые базы данных* |
|---|---|---|
| Денежные затраты | фиксированные | минимизируются |
| Производительность | оптимизируется | фиксированная |
| Масштабируемость (число машин) | максимизируется | фиксированная |
| Предсказуемость стоимости и производительности | - | фиксированная |
| Согласованность (в процентах) | фиксированная | максимизируется |
| Гибкость (число вариантов) | - | максимизируется |

*Табл. 2. Сопоставление традиционной и новой проблем оптимизации*

В большинстве случаев сегодня *производительность* является заданным ограничением приложения, а не целью оптимизации. В интерактивных приложениях для удовлетворения потребностей пользователей достаточно *время ответа* в несколько миллисекунд. Что касается *пропускной способности*, то реальным требованием является поддержка заданной

пиковой рабочей нагрузки, причем проблемой является не *возможность* этой поддержки (возможно практически все), а ее *стоимость*. Кроме того, из-за сложности современных приложений многие проблемы их производительности вообще не связаны с управлением данными.

В настоящее время принято считать, что любую проблему приложений баз данных можно решить путем соответствующего выбора аппаратных средств (т.е. за счет определенных денежных затрат). *Масштабируемость* означает, что расходы на поддержку приложений возрастают линейно по мере роста бизнеса, и этот рост не ограничивается возможностями системы. Неограниченная масштабируемость приложений необходима для любой развивающейся компании. *Мне кажется, что требование к маштабируемости нужно формулировать точнее: при росте числа аппаратных ресурсов должно соблюдаться линейное возрастание пропускной способности и стоимости с сохранением (или уменьшением) времени ответа. – С.К.*

Во многих случаях *предсказуемость* производительности и стоимости поддержки приложений оказывается не менее важной, чем их масштабируемость. Критичные для компании показатели приложений, в том числе, пиковая пропускная способность, время ответа и стоимость, должны гарантироваться.

Что касается *согласованности*, на авторов [38] сильное влияние оказали идеи [24] и [40] (замечу, что в то время оба автора этих публикаций работали в Amazon.com). Утверждается, что ACID-транзакции плохо сочетаются с сервис-ориентированной архитектурой, которая диктует автономию сервисов, участвующих в транзакции. Кроме того, по утверждениям авторитетных практиков из Amazon.com, в современной Internet-практике ACID-транзакции требуются нечасто. Наконец, более приоритетным требованием является обеспечение стопроцентной доступности данных по чтению и записи всех пользователей. Оказывается лучше разрабатывать систему, которая умеет обращаться с несогласованными данными и помогает устранять их несогласованность, чем систему, которая предотвращает несогласованность в любых ситуациях. Одним словом, в подходе [38] согласованность данных является целью оптимизации системы, а не ее фиксированным ограничением.

Замечу, что в этих рассуждениях о согласованности одновременно имеются и здравый смысл (очевидно, что поддержка ACID-транзакций в распределенной среде стоит немалых расходов, и если целью оптимизации является минимизация расходов, то по поводу транзакций нужны какие-то компромиссы), и изрядная путаница (во многом напоминающая путаницу, с которой мы разбирались в разд. 2; один раз авторы [38], хотя и очень стеснительно, ссылаются и на теорему CAP). И мне кажется, что частично причины этой путаницы указывают сами авторы, обосновывая в своей более ранней работе [41], что "при разработке крупномасштабных распределенных систем определения уровней согласованности в духе предложений [42], а не

определений, содержащихся в стандарте языка SQL и реализуемых в коммерческих системах баз данных текущего поколения".

Но если обратиться напрямую к [42], то видно, что сами Таненбаум (Andrew S. Tanenbaum) и Стеен (Maarten van Steen) используют термин *согласованность (consistensy)* как в смысле, традиционном для области баз данных ("если у системы до начала транзакции имелись некие инварианты, которые она постоянно должна хранить, они будут сохраняться и после ее завершения" – разд. 5.6 "Распределенные транзакции"), так и в смысле, затрагивающем лишь репликацию данных ("Модель согласованности (consistency model), по существу представляет собой контракт между процессами и хранилищем данных. Он гласит, что если процессы согласны соблюдать некоторые правила, хранилище соглашается работать правильно." – Разд. 6.2 "Модели непротиворечивости, ориентированные на данные"). Легко (еще раз!) видеть, что понятие согласованности второго вида не имеет прямого отношения к транзакциям вообще и к ACID-транзакциям в частности. Оба понятия согласованности, безусловно, полезны и имеют право на жизнь, но предпочитать одно другому – это все равно, что предпочитать *красное сладкому*.

Возвращаясь к перечню характеристик систем баз данных из табл. 2, *гибкость* – это возможность настройки программной системы к индивидуальным требованиям пользователей. Гибкость системы измеряется числом ее внедренных вариантов. В традиционных системах OLTP требование гибкости отсутствует. При отсутствии гибкости проще добиться высоких значений показателей производительности и масштабируемости.

Следуя новым приоритетам характеристик систем управления данными, авторы [38] предлагают новую архитектуру, которую мы кратко опишем в следующем подразделе.

## 4.1. Архитектура, удовлетворяющая новым требованиям

На рис. 9 схематически показана традиционная трехзвенная архитектура организации приложений баз данных. Запросы инициируются пользователями на уровне представлений. В настоящее время для этого обычно используются Web-браузеры. На среднем уровне поддерживается логика приложений; обычно на этом же уровне работают Web-серверы. Управление базами данных целиком сосредоточено на низшем уровне, и для этого используются СУБД.

Трехзвенная архитектура ориентирована на удовлетворение требований традиционных систем баз данных (средний столбец табл. 2). Согласованность данных поддерживается в нижнем звене сервером базы данных. Производительность обеспечивается за счет применения на всех трех уровней методов, разработанных в сообществе баз данных на протяжении десятилетий – кэширование, индексация, разделение данных и т.д. На верхних двух уровнях архитектура масштабируется почти неограниченно.


*Рис. 9. Традиционная трехзвенная архитектура организации приложений баз данных*

Трехзвенная архитектура ориентирована на удовлетворение требований традиционных систем баз данных (средний столбец табл. 2). Согласованность данных поддерживается в нижнем звене сервером базы данных. Производительность обеспечивается за счет применения на всех трех уровней методов, разработанных в сообществе баз данных на протяжении десятилетий – кэширование, индексация, разделение данных и т.д. На верхних двух уровнях архитектура масштабируется почти неограниченно.

Классическая трехзвенная архитектура (при традиционной реализации СУБД) не отвечает требованию предсказуемости. Когда с базой данных одновременно работает много клиентов, практически невозможно понять, что происходит в СУБД (*например, когда начинается активное замещение страниц в буферном пуле – С.К.*). Возможности масштабирования на нижнем уровне архитектуры ограничены.

Гибкость в традиционной архитектуре тоже не поддерживается должным образом. Одной из причин является использование разных технологий на каждом из трех уровней: SQL на нижнем уровне, объектно-ориентированные методы на уровне приложений и XML/HTML и скриптовые языки на уровне представлений. Настройку нужно вести на всех трех уровнях с использованием разных технологий, что затруднительно и ненадежно. (*Коротко можно сказать, что традиционной архитектуре присуща известная проблема* потери соответствия (impedance mismatch) – *С.К.*)

Двумя основными принципами разработки приложений баз данных в классической трехзвенной архитектуре являются *контроль* и *передача запросов (query shipping)*. Первый принцип означает, что на нижнем уровне СУБД контролирует все аппаратные ресурсы и весь доступ к данным. Передача запросов предполагает выталкивание на нижний уровень как можно больше функций приложений за счет использования хранимых процедур, типов данных определяемых пользователями и т.д. С одной стороны, эти принципы позволяют добиться согласованности данных и высокой производительности системы. С другой стороны, по мнению авторов [38], оба эти принципа стимулируются бизнес-моделью производителей СУБД и превращают системы управления данными в монолитных монстров, размеры и сложность которых непрерывно растут. Это наносит вред предсказуемости, гибкости и масштабируемости. Кроме того, они приводят к тому, что приложения баз данных становятся дорогостоящими, поскольку для поддержки таких СУБД требуется дорогая аппаратура. В предлагаемой авторами архитектуре используются противоположные принципы разработки.

Этот абзац кажется мне очень важным, поскольку, по моему мнению, отказ от указанных принципов в действительности вызывает очень важные последствия. Отказ от принципа *контроля* приводит к тому, что СУБД перестает быть *системой*, а превращается в набор утилит. Если у СУБД отсутствует контроль над обрабатываемыми ею данными, она, естественно, не сможет поддерживать ACID-транзакции. Как мы видели в предыдущем разделе, можно делать транзакционные СУБД массивно-параллельными (фактически, использовать для их разработки методы распределенных систем), оставляя их системами с точки зрения внешнего использования.

Если отнять у СУБД контроль над аппаратными средствами и данными, то для построения распределенных систем обработки данных на всех уровнях придется применять общие методы построения распределенных систем. Наверное, это хорошо с точки зрения повсеместного использования сервис-ориентированной архитектуры, но задачи, которые хорошо решаются традиционными СУБД (в том числе, обеспечение ACID-транзакций) становятся практически неразрешимыми. Кстати, такая организация напоминает мне архитектуру файл-серверных СУБД типа Informix SE (см. например, [43]), в которой данные контролировались файловым сервером, а запросы обрабатывались на клиентских рабочих станциях. В этой архитектуре

поддерживались ACID-транзакции, но очень высокой ценой – путем блокировки файлов целиком. И не просто так впоследствие компании-производители SQL-ориентированных СУБД перешли к использованию серверов баз данных, в которых данные контролируются СУБД.

Кстати, и отказ от второго принципа возвращает нас ко времени Informix SE, где для выполнения запросов на рабочие станции из файлового сервера передавались блоки данных. Вроде бы, вполне естественно было перейти на архитектуру, в которой из клиента в сервер баз данных передавались SQL-запросы, а возвращались только те данные, которые действительно нужны приложению. И механизмы хранимых процедур и определяемых пользователями типов данных во многом появились не из-за корысти производителей СУБД, а для того, чтобы еще сократить объемы данных, которыми приходится обмениваться клиенту и серверу. И как видно из разд. 3, все это совсем не обязательно наносит вред предсказуемости и масштабируемости (насчет гибкости ничего сказать не могу, поскольку отсутствуют точные критерии).

Взамен традиционной архитектуре с рис. 9 в [38] предлагается новая архитектура, показанная на рис. 10. Эта архитектура реализована компанией 28msec в продукте Sausalito [44]. В архитектуре Sausalito основные функции СУБД – обработка запросов и управление транзакциями – переносятся на прикладной уровень. (*В действительности, не очень понятно, что же такое* транзакция *в этой системе. Подробнее об этом см. следующий подраздел. – С.К.*) На нижнем уровне поддерживается только распределенное хранение данных за счет использования службы Amazon S3 [45] (хотя теоретически можно использовать и другие аналогичные средства). Согласованность данных поддерживается не на уровне хранения, а на прикладном уровне. Отсутствует какой-либо объект, контролирующий весь доступ к данным, и согласованность данных обеспечивается в духе [42] за счет применения во всех серверах приложений общих протоколов. (*Как уже отмечалось выше, в [42] эти протоколы направлены на поддержку согласованности* реплик *данных, так что здесь имеется в виду именно этот вид согласованности. – С.К.*)

На всех уровнях можно использовать дешевые аппаратные средства, и каждый уровень может масштабироваться до тысяч машин. В любой момент времени допускается отказ любого узла. На нижнем уровне отказоустойчивость достигается за счет репликации и предоставления гарантий только *согласованности в конечном счете (eventual consistency)*. На верхних уровнях все узлы работают без сохранения состояния, поэтому при отказе какого-либо узла в самом худшем случае могут потеряться некоторые активные транзакции.


*Рис. 10. Архитектура Sausalito*

В Sausalito все данные и заранее откомпилированный код приложения сохраняются в среде Amazon S3 в виде BLOB'ов. При обработке каждого HTTP-запроса (поступающего, например, по инициативе пользователя из Web-браузера) служба Amazon EC2 [46] с учетом балансировки нагрузки выбирает доступный сервер EC2. В этот сервер из S3 загружается код приложения, который затем интерпретируется подсистемой поддержки времени выполнения Sausalito с доступом, при необходимости, к объектам базы данных, хранимым в S3. Особенностью Sausalito является то, что для реализации логики приложений и доступа к базе данных в системе используется язык XQuery [47], расширенный средствами обновления данных и написания скриптов. Авторы [38] мотивируют этот выбор тем, что XQuery хорошо согласуется со стандартами Web, обладает развитыми средствами

запросов, и возможностей этого языка достаточно для создания развитых Web-приложений. Однако мне кажется, что важную роль сыграла и личная близость Даниелы Флореску (Daniela Florescu) и Дональда Коссманна к процессам стандартизации и реализации этого языка. Не уверен, что (единственная) возможность использования XQuery в качестве языка разработки приложений приводит в восторг потенциальных пользователей Sausalito.

Как отмечается в [38], у системы с архитектурой с рис. 10 мало шансов сравняться с классическими системами баз данных в отношении производительности и согласованности данных (*я уже говорил выше, что это обратная сторона отказа от принципов контроля над данными и пересылки запросов – С.К.*). Однако эта архитектура хорошо соответствует требованиям правого столбца табл. 2. Архитектура экономически эффективно реализуется с использованием дешевых аппаратных средств, масштабируемость на всех уровнях обеспечивается автоматически. Гибкость достигается за счет упрощения платформы и использования на всех уровнях единой модели программирования и данных (XML/XQuery). Для многих рабочих нагрузок обеспечивается предсказуемость расходов и производительности.

В [38] изложены основные идеи "облачной" системы управления данными, хорошо согласующейся с сервис-ориентированной архитектурой, гибкой и масштабируемой на всех уровнях. На появление новой архитектуры систем баз данных повлияли сценарии интерактивных (транзакционных) Web-приложений – онлайновых магазинов и т.д. Однако, как отмечалось выше, идеология поддержки транзакций в [38] выглядит очень туманной. Некоторую ясность привносит более свежая работа [9], которой посвящен следующий подраздел.

## 4.2. Разным данным разная согласованность: новая парадигма транзакций?

В [9], как и в [38], речь идет о возможности реализации средств управления данными поверх "облачных" сред хранения данных. Основная идея исследования состоит в том, что не все данные требуется обрабатывать на одном и том же уровне согласованности (*в этой статье делается попытка "примирить" понятия согласованности из областей баз данных и распределенных систем, хотя, на мой взгляд, это не всегда у авторов получается – С.К.*). Например, в онлайновом магазине данные о кредитных картах клиентов и состоянии их счетов должны поддерживаться на более высоком уровне согласованности, чем данные о предпочтениях покупателей.

Поскольку существующие "облачные" платформы хранения данных предоставляют только базовые гарантии согласованности данных (согласованность в конечном счете), обеспечение дополнительных уровней согласованности значительно повышает стоимость выполнения операций. С другой стороны, стоимость несогласованности данных также можно измерить,

оценив относительное число операций, которые выполняются некорректно из-за отсутствия согласованности, и переведя эту оценку в денежные расходы, которые должна понести компания, допустившая такое некорректное выполнение операций (например, для компенсации ущерба от ошибочно выполненных заказов).

Нахождение баланса между стоимостью выполнения операций, согласованностью данных и их доступностью является нетривиальной задачей. Этот баланс зависит от нескольких факторов, включая семантику приложений. Предлагается использовать динамическую стратегию поддержки согласованности данных – снижать уровень требований к согласованности, когда это не приводит к слишком большим убыткам, и повышать их, если убытки могут оказаться слишком большими. Авторы [9] называют этот подход *рационализацией согласованности (consistency rationing)* по аналогии с понятием *рационализации управления запасами (inventory rationing)*, когда запасы отслеживаются с разной точностью в зависимости от их наличного объема.

При применении подхода рационализации согласованности все управляемые данные разделяются на три категории A, B и C, и для каждой категории применяются свои методы обработки в зависимости от обеспечиваемого уровня согласованности. К *категории A* относятся данные, нарушение согласованности которых привело бы к крупным убыткам. Несогласованность данных *категории C* является приемлемой (несогласованность либо вызывает лишь небольшие убытки, либо в действительности не возникает). Наконец, *категория B* включает данные, требования к согласованности которых изменяются во времени. Для данных категории B можно добиться оптимального соотношения расходов на выполнение операций и обеспечиваемого уровня согласованности.

Гарантии согласованности в [9] обеспечиваются для данных, а не для транзакций. Как утверждается, в результате ослабляются только свойства изоляции и согласованности ACID-транзакций, а атомарность и долговечность обеспечиваются в полном объеме. На самом деле, здесь нужно было бы привести определение возникающего понятия *транзакции*, но такое определение в [9] отсутствует, и придется обойтись предыдущим расплывчатым предложением. Для данных возможны два уровня согласованности – *сессионная согласованность (session consistency)* [48] и *сериализуемость (serializability)*.

Сессионная согласованность данных подразумевает, что клиенты подключаются к системе в контексте *сессий* (*что такое "сессия", ни в [9], ни в [48] не определяется – С.К.*). Во время сессии система гарантирует *монотонность по чтению собственных записей (read-your-own-writes monotonicity)* (*согласно [42], система "обладает свойством монотонности по чтению собственных записей, если удовлетворяется то условие, что результат записи процесса в элемент данных* x *всегда виден последующим*

*операциям чтения* x *того же процесса" – С.К.*). За границы сессии гарантии монотонности не распространяются: в новой сессии того же клиента могут быть не видны записи, произведенные в предыдущей сессии; в сессии одного клиента могут быть не видны записи, выполненные в сессии другого клиента. Сессионная согласованность поддерживается для данных категории С.

Для данных категории А обеспечивается сериализуемость в традиционном транзакционном смысле. Все транзакции, модифицирующие данные категории А, изолируются и оставляют данные согласованными (здесь имеются в виду и согласованность в смысле ACID, и *строгая согласованность* в смысле распределенных систем). Для поддержки сериализуемости требуются значительные накладные расходы, и данные следует относить к категории А, только если их согласованность и актуальность являются обязательными. В [9] для поддержки сериализуемости используется двухфазный протокол синхронизационных блокировок.

Для данных категории В во время работы системы на основе разработанных авторами политик производится переключение между режимами сессионной согласованности и сериализуемости. Если одна транзакция работает с некоторой записью в режиме сериализуемости, а другая обновляет ее в режиме сессионной согласованности, то обеспечивается сессионная согласованность (*т.е., насколько я понимаю, транзакции, работающие с данными категории В в режиме сериализуемости, не изолируются от транзакций, работающих с теми же данными в режиме сессионной согласованности – С.К.*).

Если в одной транзакции одновременно обрабатываются данные категорий А и С, то при выполнении операции чтения данных категории А эта транзакция всегда получает их последнее согласованное состояние и оставляет их согласованными после своей фиксации. При чтении данных категории С транзакция может получить их в несогласованном и/или устаревшем состоянии.

Как считают авторы [9], в ряде случаев можно будет разделить процессы разработки приложений и рационализации данных. В процессе разработки можно предполагать наличие традиционной согласованности данных и следовать обычной модели программирования с явно определяемыми транзакциями. На стадии внедрения приложения можно произвести рационализацию данных в соответствии с оценками расходов на выполнение операций и ущерба от несогласованности данных. Рационализацию мог бы производить специалист не из числа разработчиков приложения.

В [9] подробно описываются разработанные авторами политики переключения режимов согласованности для данных категории В, однако в данной статье я не хочу затрагивать эту тему, поскольку для наших целей она не очень важна. Не буду также говорить о деталях реализации разнообразных протоколов поддержки транзакций и согласованности данных, потому что в

[9] эти вопросы освещены не слишком внятно, а в подробном техническом отчете [49] соответствующие описания занимают слишком много места.

Главный вывод, который, на мой взгляд, следует сделать на основе публикаций, посвященных Sausalito, состоит в том, что разработчики этой системы вполне смогли справиться с поддержкой ACID-транзакций и ослабляют требования к согласованности совсем не на основе ограничений, которые ставит теорема CAP, а с целью вполне разумной оптимизации стоимости службы управления данными.

## *5. Заключение*

Основные цели этой статьи заключались в обсуждении истинной сути теоремы CAP и рассмотрении нескольких новых архитектур транзакционных СУБД, хорошо масштабируемых в динамических кластерных средах. Я попытался показать, что теорема CAP Эрика Брювера не имеет никакого отношения к возможности или невозможности поддержки ACID-транзакций в горизонтально масштабируемых кластерных системах.

Проект H-Store показывает, что в параллельной архитектуре shared-nothing основную проблему представляют распределенные транзакции, накладные расходы на фиксацию которых могут заметно снижать пропускную способность системы. Однако полученные результаты показывают, что для ряда важных типов рабочей нагрузки этих расходов можно избежать.

Другую проблему транзакционных параллельных систем без совместно используемых ресурсов составляет потребность в массовой физической пересылке данных при балансировке нагрузки. И здесь, как мне кажется, может несколько помочь подход проекта DORA, в котором вместо физического разделения данных в многоядерном компьютере используется их логическое разделение за счет наличия общей памяти.

Наконец, подход Sausalito показывает, во-первых, что при отказе от основных принципов разработки СУБД – контроля над данными и передачи запросов – в угоду более точному следованию архитектуре SOA расходы на поддержку ACID-транзакций существенно возрастают. Во-вторых, эта поддержка по-прежнему возможна, и разумна оптимизация системы управления данными, позволяющая снизить расходы на управление транзакциями, если приложениям не требуется качественная согласованность данных.

По всей видимости, исследования и разработки массивно-параллельных транзакционных СУБД в ближайшие годы будут активно продолжаться, и нам предстоит еще увидеть и услышать много интересного.

**Литература**

[1] Michael Stonebraker, Ugur Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. Proceedings of the 21st International Conference on Data Engineering, 2005, pp. 2-11, http://www.cs.brown.edu/~ugur/fits_all.pdf.

Перевод на русский язык: Майкл Стоунбрейкер, Угур Кетинтемел. "Один размер пригоден для всех": идея, время которой пришло и ушло, 2007, http://citforum.ru/database/articles/one_size_fits_all/.
[2] Jeffrey Dean, Sanjay Ghemawat. MapReduce: Simplifed Data Processing on Large Clusters, Proceedings of the Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004, pp. 137-150, http://labs.google.com/papers/mapreduce-osdi04.pdf.
[3] Michael Stonebraker, David J. DeWitt. MapReduce: A major step backwards, Database Column, January 17, 2008, http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/.
[4] Michael Stonebraker, David J. DeWitt. MapReduce II, Database Column, January 25, 2008, http://databasecolumn.vertica.com/database-innovation/mapreduce-ii/.
[5] С.Д. Кузнецов. MapReduce: внутри, снаружи или сбоку от параллельных СУБД?, Труды Института системного программирования, т. 19, М., ИСП РАН, 2010, стр. 35-40, http://citforum.ru/database/articles/dw_appliance_and_mr/.
[6] Pat Helland, Dave Campbell. Building on Quicksand. Proceedings of the Fourth Biennial Conference on Innovative Data Systems Research (CIDR 2009), January 4-7, 2009, Asilomar, Pacific Grove, CA USA, http://www-db.cs.wisc.edu/cidr/cidr2009/Paper_133.pdf.
Перевод на русский язык: Пэт Хелланд, Дейв Кэмпбел. Дом на песке, 2010, http://citforum.ru/database/articles/quicksand/.
[7] Eric Brewer, Towards Robust Distributed Systems, Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 2000, p. 7, http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf.
[8] Daniel Abadi, Alexander Thomson. The problems with ACID, and how to fix them without NoSQL. DBMS Musings, August 31, 2010, http://dbmsmusings.blogspot.com/2010/08/problems-with-acid-and-how-to-fix-them.html.
Перевод на русский язык: Дэниел Абади и Александер Томсон. Проблемы с ACID, и как их устранить, не прибегая к использованию NoSQL, 2010, http://citforum.ru/gazeta/164/.
[9] Tim Kraska, Martin Hentschel, Gustavo Alonso, Donald Kossmann. Consistency Rationing in the Cloud: Pay only when it matters. Proceedings of the 35th VLDB Conference, August 24-28, 2009, Lyon, France, pp. 253-264, http://www.dbis.ethz.ch/research/publications/ConsistencyRationing.pdf.
Перевод на русский язык: Тим Краска, Мартин Хеншель, Густаво Алонсо, Дональд Коссман. Рационализация согласованности в "облаках": не платите за то, что вам не требуется, 2010, http://citforum.ru/database/articles/kossmann_vldb_2009/.
[10] Домашняя страница проекта H-store, 2010, http://hstore.cs.brown.edu/.
[11] Официальный сайт компании VoltDB, 2010, http://voltdb.com/.
[12] Michael Stonebraker. Errors in Database Systems, Eventual Consistency, and the CAP Theorem. BLOG@CACM, April 5, 2010, http://cacm.acm.org/blogs/blog-cacm/83396-errors-in-database-systems-eventual-consistency-and-the-cap-theorem/fulltext.
Перевод на русский язык: Майкл Стоунбрейкер. Ошибки в системах баз данных, согласованность "в конечном счете" и теорема CAP, 2010, http://citforum.ru/gazeta/154/.
[13] Michael Stonebraker. Clarifications on the CAP Theorem and Data-Related Errors. VoltDB.com, October 21, 2010, http://voltdb.com/blog/clarifications-cap-theorem-and-data-related-errors.

Перевод на русский язык: Майкл Стоунбрейкер. Уточнения по поводу теоремы CAP и ошибок, связанных с данными, 2010, http://citforum.ru/gazeta/169/.
[14] Carlo Curino, Evan Jones, Yang Zhang, Sam Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. 36th International Conference on Very Large Data Bases, September 13-17, 2010, Singapore. Proceedings of the VLDB Endowment, Vol. 3, No. 1, 2010, pp. 48-57, http://db.csail.mit.edu/pubs/schism-vldb2010.pdf.
Перевод на русский язык: Карло Курино, Эван Джонс, Янг Жанг и Сэм Мэдден. Schism: управляемый рабочей нагрузкой подход к репликации и разделению баз данных, 2010, http://citforum.ru/database/articles/madden_vldb2010/.
[15] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, Anastasia Ailamaki. Data-Oriented Transaction Execution. 36th International Conference on Very Large Data Bases, September 13-17, 2010, Singapore. Proceedings of the VLDB Endowment, Vol. 3, No. 1, 2010, pp. 928-939, http://infoscience.epfl.ch/record/141326/files/pvldb10pandis.pdf.
Перевод на русский язык: Иппократис Пандис, Райан Джонсон, Никос Харадавеллас и Анастасия Айламаки. Выполнение транзакций, ориентированное на данные, 2010, http://citforum.ru/database/articles/ailamaki_vldb2010/.
[16] Theo Haerder, Andreas Reuter. Principles of transaction-oriented database recovery. ACM Computing Surveys, Volume 15, Issue 4, December 1983, pp. 287 – 317, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.88.5548&rep=rep1&type=pdf
[17] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, Irving Traiger. The recovery manager of the System R database manager. ACM Computing Surveys, Volume 13, Issue 2, June 1981, 223-242, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.94.9008&rep=rep1&type=pdf
[18] Eric Brewer. A certain freedom: thoughts on the CAP theorem. Proceeding of the 29th ACM SIGACT-SIGOPS Symposium on Principles of distributed Computing, 2010, p. 335, http://portal.acm.org/citation.cfm?id=1835701.
[19] Seth Gilbert, Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News, Volume 33 Issue 2, June 2002, pp. 51-59, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.67.6951&rep=rep1&type=pdf
[20] Julian Browne. Brewer's CAP Theorem, January 11, 2009, http://www.julianbrowne.com/article/viewer/brewers-cap-theorem.
[21] Daniel Abadi. Problems with CAP, and Yahoo’s little known NoSQL system. DBMS Musings, April 23, 2010, http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html.
[22] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, Pat Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). Proceedings of the 33rd International Conference on Very Large Data Bases, 2007, pp. 1150-1160, http://cs-www.cs.yale.edu/homes/dna/papers/vldb07hstore.pdf.
Перевод на русский язык: Майкл Стоунбрейкер, Сэмюэль Мэдден, Дэниэль Абади, Ставрос Харизопулос, Набил Хачем, Пат Хеллэнд. Конец архитектурной эпохи, или Наступило время полностью переписывать системы управления данными, 2007, http://citforum.ru/database/articles/end_of_arch_era/.
[23] M. Stonebraker, C. Bear, U. Cetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. Zdonik. One Size Fits All?— Part 2: Benchmarking Results. Proceedings of the Third Biennial Conference on Innovative Data Systems Research (CIDR 2007), January 7-10, 2007, Asilomar, Pacific Grove, CA USA, http://nms.csail.mit.edu/~stavros/pubs/osfa.pdf.

Перевод на русский язык: Майкл Стоунбрейкер, Чак Беэ, Угур Кетинтемел, Мич Черняк, Тиньян Ге, Набил Хачем, Ставрос Харизопулос, Джон Лифтер, Дженни Роджерс, Стэн Здоник. Пригоден ли один размер для всех? Часть 2: результаты тестовых испытаний, 2007, http://citforum.ru/database/articles/one_size_fits_all_2/.
[24] Pat Helland. Life beyond Distributed Transactions: an Apostate's Opinion. Proceedings of the Third Biennial Conference on Innovative Data Systems Research (CIDR 2007), January 7-10, 2007, Asilomar, Pacific Grove, CA USA, http://web.mit.edu/tibbetts/Public/CIDR_2007_Proceedings/papers/cidr07p15.pdf.
[25] Using VoltDB, V1.2, VoltDB, Inc., June 13, 2010, http://community.voltdb.com/docs/UsingVoltDB/index.
[26] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, Daniel J. Abad. HStore: A HighPerformance, Distributed Main Memory Transaction Processing System. Proceedings of the VLDB Endowment, Volume 1 Issue 2, August 2008, pp. 1496-1499, http://cs-www.cs.yale.edu/homes/dna/papers/hstore-demo.pdf.
[27] Evan P.C. Jones, Daniel J. Abadi, Samuel Madden. Low Overhead Concurrency Control for Partitioned Main Memory Databases. SIGMOD'10, Indianapolis, Indiana, USA, June 6–11, 2010, http://cs-www.cs.yale.edu/homes/dna/papers/hstore-cc.pdf.
Перевод на русский язык: Эван Джонс, Дэниэль Абади и Сэмуэль Мэдден. Управление параллелизмом с низкими накладными расходами для разделенных баз данных в основной памяти, 2010, http://citforum.ru/database/articles/h-store-sigmod2010/.
[28] Daniel Abadi, Alexander Thomson. The Case for Determinism in Database Systems. 36th International Conference on Very Large Data Bases, September 13-17, 2010, Singapore. Proceedings of the VLDB Endowment, Vol. 3, No. 1, 2010, pp. 70-80, http://db.cs.yale.edu/determinism-vldb10.pdf.
Перевод на русский язык: Дэниел Абади и Александер Томсон. Доводы в пользу детерминизма в системах баз данных, 2010, http://citforum.ru/database/articles/abadi_vldb2010/.
[29] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, Michael Stonebraker. OLTP Through the Looking Glass, and What We Found There, Proceedings of the ACM SIGMOD International Conference on Management of Data, Vancouver, BC, Canada, June 2008, pp. 981-992, http://db.cs.yale.edu/hstore/oltpperf-sigmod08.pdf.
Перевод на русский язык: Ставрос Харизопулос, Дэниэль Абади, Сэмюэль Мэдден, Майкл Стоунбрейкер. OLTP в Зазеркалье, 2010, http://citforum.ru/database/articles/oltp_lg/.
[30] Домашняя страница проекта Shore, 2010, http://www.cs.wisc.edu/shore/.
[31] G. Karypis. METIS— Family of Multilevel Partitioning Algorithms, 2010, http://glaros.dtc.umn.edu/gkhome/views/metis.
[32] Домашняя страница проекта WEKA (Waikato Environment for Knowledge Analysis), 2010, http://www.cs.waikato.ac.nz/ml/index.html.
[33] Домашняя страница проекта Shore-NT, 2010, http://www.cs.wisc.edu/shore-mt/.
[34] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: A Scalable Storage Manager for the Multicore Era. Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology (EDBT 2009), 2009, pp. 24-35, http://diaswww.epfl.ch/shore-mt/papers/edbt09johnson.pdf.

[35] Spinlock. Материал из Википедии – свободной энциклопедии, 2010, http://ru.wikipedia.org/wiki/Spinlock.
[36] Сергей Кузнецов. Базы данных. Вводный курс. 13.3.1. Синхронизационные блокировки, 2008, http://citforum.ru/database/advanced_intro/41.shtml#13.3.1.
[37] TPC BENCHMARK C. Standard Specification. Revision 5.11. Transaction Processing Performance Council, 2010, http://www.tpc.org/tpcc/spec/tpcc_current.pdf.
[38] Daniela Florescu, Donald Kossmann. Rethinking Cost and Performance of Database Systems. SIGMOD Record, Vol. 38, No. 1, March 2009, pp. 43-48, http://www.dbis.ethz.ch/research/publications/sigrec08.pdf.
Перевод на русский язык: Даниела Флореску, Дональд Коссман. Переосмысление стоимости и производительности систем баз данных, 2009, http://citforum.ru/database/articles/rethinking/.
[39] С.Д. Кузнецов. Год эпохи перемен в технологии баз данных. Труды Института системного программирования, т. 19, М., ИСП РАН, 2010, стр. 9-34, http://citforum.ru/database/articles/epoch/.
[40] Werner Vogels. Data Access Patterns in the Amazon.com Technology Platform. Proceedings of the 33rd International Conference on Very Large Data Bases, Sep 2007, p. 1, http://www.vldb.org/archives/website/2007/program/videos/p1-vogels.wmv.
[41] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a Database on S3. Proceedings of the ACM SIGMOD International Conference on Management of Data, Vancouver, BC, Canada, June 2008, pp. 251–264, http://www.dbis.ethz.ch/research/publications/sigmod08-s3.pdf.
[42] A. Tanenbaum and M. van Steen. Distributed Systems: Principles and Paradigms. Prentice Hall, Upper Saddle River, NJ, 2002.
Перевод на русский язык: Э. Таненбаум, М. ван Стеен. Распределенные системы. Принципы и парадигмы. СПб.: Питер, 2003, 889 стр.
[43] Н.А. Олифер, В.Г. Олифер, П.Б. Храмцов, В.И. Артемьев, С.Д. Кузнецов. Стратегическое планирование сетей масштаба предприятия. Центр Информационных Технологий, 1997, http://citforum.ru/nets/spsmp/.
[44] Домашняя страница компании 28msec/проекта Sausalito, 2010, http://www.28msec.com/.
[45] Домашняя страница Amazon Simple Storage Service (Amazon S3), 2010, http://aws.amazon.com/s3/.
[46] Домашняя страница Amazon Elastic Compute Cloud (Amazon EC2), 2010, http://aws.amazon.com/ec2/.
[47] XQuery 1.0: An XML Query Language (Second Edition), W3C Recommendation, 14 December 2010, http://www.w3.org/TR/xquery/.
[48] Werner Vogels. Eventually Consistent. ACM Queue, Vol. 6 No. 6, October 2008, pp. 15-19, http://queue.acm.org/detail.cfm?id=1466448.
[49] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, Tim Kraska. Building a Database in the Cloud. Technical Report, ETH Zurich, 2009, http://www.dbis.ethz.ch/research/publications/dbs3.pdf.

# Transactional Massive-Parallel DBMSs: A New Wave

*Sergey D. Kuznetsov*
*kuzloc@ispras.ru*

**Abstract.** A possibility to build unlimitedly scalable cluster-based systems has lead to strong activation of research and development of "shared-nothing" architectures of data management systems. Two camps has been established: "NoSQL" that refutes the main principles related with DBMSs and "one size doesn't fit all" that emphasizes needs of systems' specialization saving the most important features of DBMS. The most interesting seems to be a confrontation of these camps in the area of "transactional" data management systems. Based on the CAP-"theorem" of Eric Bruwer, representatives of the camp of NoSQL declines to support traditional features of ACID of database transactions. This paper discusses the essence of the Bruwer's "theorem" and proves that this theorem does not any relation with the ACID features. The paper also overviews the most interesting modern research projects provided classic ACID-transactions in parallel shared-nothing environments and the soundest approaches that partially relaxes requirements of ACID by purely pragmatic reasons (but not at all in relation with the CAP "theorem").

**Keywords:** transactional massive-parallel DBMS, "one size doesn't fit all", NoSQL, ACID-transaction**.**

# Нацеленная генерация данных для тестирования приложений над базами данных

*Е. А. Костычев, В. А. Омельченко, С.В. Зеленов*
*{kostychev, vitaly, zelenov}@ispras.ru*

**Аннотация.** Приложения, обрабатывающие большие массивы данных, являются широко распространенным типом программного обеспечения. В частности, такие приложения решают задачи интеграции данных в области интеграции корпоративных приложений. При решении таких задач используются специальные инструментальные среды, поддерживающие разработку, выполнение и мониторинг приложений, реализующих шаблон извлечения, трансформации и загрузки данных (Extract, Transform and Load - ETL). Специфика функционального тестирования таких приложений заключается в большом количестве возможных комбинаций входных данных. Подходы, реализованные в инструментах генерации данных для функциональной проверки приложений над базами данных, в лучших случаях основываются на схемах баз данных и запросах на SQL, используемых в тестируемых приложениях. Гарантированное покрытие функциональности тестируемого приложения при таких подходах может быть достигнуто только полным перебором возможных комбинаций исходных данных. При помощи предложенного в статье подхода к генерации данных возможно достижение покрытия функциональности приложения с близким к оптимальному объему данных (один набор данных для одной функциональной ветви приложения).

**Ключевые слова:** генерация тестовых данных; приложения над базами данных; функциональное тестирование; интеграция данных.

## *1. Введение*

Постоянно растущая доступность все больших объемов для хранения информации в электронном виде и все больших вычислительных ресурсов для их обработки приводит к все большей распространенности приложений над все большими объемами данных в различных областях. Так в начале 90х годов прошлого века появился британский национальный корпус – специальным образом размеченный и обработанный большой массив, содержащий 100 миллионов слов в виде коллекции письменных и устных текстов за более чем двадцатилетний промежуток [1]. С тех пор такие национальные корпуса появились и продолжают появляться и пополняться для многих языков. Такие

корпуса языков не только делают более тривиальными многие традиционные задачи лингвистики, но и позволяют ставить и решать ранее невыполнимые задачи, прежде всего связанные с обработкой больших объемов данных [2]. С обработкой больших объемов данных так же связаны задачи в таких областях как статистика, социология, георазведка, некоторые разделы физики, молекулярная генетика, климатология, метеорология.

Хранение и обработка больших объемов данных является критической задачей и для современного корпоративного бизнеса. Корпоративные системы содержат огромные объемы взаимосвязанных данных о потребителях и заказчиках, поставщиках, партнерах, различных бизнес-транзакциях, об оборудовании и персонале, финансовых потоках. Большие объемы корпоративных данных, как правило, хранятся в структурированном виде в одной или нескольких базах данных под управлением одной или нескольких СУБД. Для обеспечения поддержки решений различных задач бизнеса зачастую требуется функциональность, которая может быть реализована только путем интеграции нескольких подсистем корпоративной системы, а иногда и интеграции с подсистемами внешних корпоративных систем. Часто в таких случаях возникает проблема доступа к большим массивам данных из разных хранилищ и имеющих разные форматы. Поэтому приложения репликации, обновления и синхронизации больших объемов данных являются широко распространенным частным случаем интеграционных корпоративных приложений.

Задачи таких приложений могут быть элементарными, но не всегда простыми в реализации, как, например, выборка полных или частичных данных, касающихся физического или юридического лица. Они могут быть и достаточно сложными, как по условиям выборки и агрегации входных данных, так и по формированию конечного результата, как, например, выставление счетов пользователям в соответствии с объемом реально потребленных услуг, тарифными планами и с учетом предоставляемых скидок. Высокая частота возникновения таких задач при решении проблемы интеграции корпоративных приложений привела к появлению специализированных платформ, предоставляющих универсальную, относительно платформ хранения данных, среду выполнения и инструменты для разработки приложений, эффективно решающих задачи извлечения, трансформации и загрузки больших массивов данных [3]. Далее такие приложения будем называть ETL-приложениями (от англ. Extract, Transform and Loading).

## *2. Постановка задачи*

После разработки любого нового приложения или доработки уже существующего требуется проверка корректности его поведения на соответствие функциональным требованиям, реализующим необходимую

бизнес-логику. Бизнес-логику приложений над большими объемами данных, в том числе и ETL-приложений, можно детализировать следующим образом:

- извлечение из источника тех и только тех данных, которые удовлетворяют соответствующим условиям, определенным в требованиях;
- изменение в источнике значений тех и только тех данных, которые удовлетворяют соответствующим условиям, определенным в требованиях;
- трансформация (изменение значений, формата представления) входных данных в соответствии с требованиями;
- загрузка в приемник тех и только тех данных, которые должны быть загружены в соответствии с требованиями;
- изменение в приемнике значений тех и только тех данных, которые удовлетворяют соответствующим условиям, определенным в требованиях.

Проверка корректности поведения реализации осуществляется, как правило, при помощи функционального тестирования, которое состоит из выполнения приложения на специально подготовленных входных данных и сравнения полученных выходных данных с ожидаемыми результатами. В данной статье рассматривается задача генерации входных данных для функционального тестирования приложений над большими объемами данных (далее тестовые данные). В качестве примеров таких приложений рассматриваются ETL-приложения, содержащие все этапы, которые могут встречаться в приложениях над большими объемами данных: извлечение данных, их преобразование и загрузка.

Критерием качества функционального тестирования является полнота покрытия функциональных требований. Чем больше функциональных ветвей алгоритма, реализующего функциональные требования к тестируемому приложению, покрыто при выполнении тестов, тем качественнее тестирование. Из сформулированных выше подзадач функциональности ETL-приложений следует, что для покрытия функциональности таких приложений в массиве тестовых данных должны присутствовать такие данные, что при выполнении приложения:

- выборка данных из источника и их фильтрация происходит по всем возможным комбинациям условий, определенных в требованиях;
- при трансформации данных покрываются все возможные функциональные ветви алгоритма трансформации, соответствующего требованиям.
- обеспечивается проверка отсутствия лишних изменений данных в приемнике и источнике;

- обеспечивается проверка обработки специальных данных (пустые поля, строки ограниченной длины и т.д.).

Множество всех возможных комбинаций значений входных данных заведомо содержит комбинации необходимые для покрытия функциональности тестируемого приложения. Однако такой подход в реальных ситуациях очень быстро вызывает комбинаторный взрыв и приводит к неприемлемым затратам по времени генерации и по суммарному объему полученных тестов. Кроме того, в подавляющем большинстве случаев при проведении функционального тестирования полученное таким способом множество тестов заведомо является чрезмерным: очень многие тесты отличаются лишь такими значениями полей, которые не различимы с точки зрения функциональности. Это создает проблемы при анализе результатов тестирования: одна и та же ошибка будет проявляться на тысячах подобных «неразличимых» тестов, и все эти ситуации потребуют трудоемкого анализа.

Оптимальными с точки зрения покрытия функциональных требований являются такие данные, которые обеспечивают на каждое ветвление функциональности по одной возможной комбинации булевских выражений. В статье рассматривается метод получения тестовых данных, по объемам близких к оптимальному набору.

## *3. Обзор существующих инструментов*

Большинство существующих инструментов генерации тестовых данных (DTM Data Generator [4], Turbo Data [5], DBMonster [6]) обеспечивают поддержку заполнения таблиц БД большим количеством синтаксически корректных данных и предоставляют следующие возможности:

- генерация случайных данных с возможностью задать интервал для числовых типов, длину для строк и формат генерируемых данных;
- генерация данных из списка, с возможностью указать процентное соотношение каждой строки из списка ко всем сгенерированным строкам;
- генерация данных путем их выбора из другой таблицы;
- генерация данных путем их выбора из файла;
- генерация автоувеличивающихся данных, с указанием начального значения и шага;
- генерация данных из существующих библиотек;
- генерация случайных данных по маске;
- генерация данных на основе возвращаемого результата SQL – выражений;
- генерация данных для зависимых таблиц;

- генерация данных на основе внешних процедур генерации, пользователь имеет возможность создавать свои процедуры.

Некоторые инструменты (AGENDA [7], HTDGen [8]) поддерживают генерацию данных не только на основе ограничений, заданных пользователем или схемой, но и на основе SQL-запросов тестируемого приложения. В этом случае гарантируется, как минимум, что SQL-запросы, используемые в реализации приложения, будут возвращать не пустой результат. Однако при генерации данных на основе реализации нет гарантии покрытия всех требуемых функциональных ветвей, так как реализация может содержать ошибочные ветви или не содержать требуемые. Так же такой подход не обеспечивает покрытия всех требуемых функциональных ветвей трансформации и фильтрации данных.

В общем случае, используя доступные инструменты, гарантировать полное покрытие всех функциональных ветвей тестируемого приложения возможно только полным перебором комбинаций значений полей, в зависимости от которых происходят ветвления при выборке, трансформации и фильтрации данных в приложении. Как было сказано выше, полный перебор вызывает комбинаторный взрыв и приводит к неприемлемым затратам по времени генерации и по суммарному объему полученных тестов.

Часто, с точки зрения покрытия функциональности, значения нескольких полей бывают связаны некоторой семантикой, которая выражается в том, что значение одного поля зависит от значений каких-то других полей. При этом, как правило, перебор комбинаций значений, удовлетворяющих этой семантике, сильно снижает общее количество вариантов. Например, пусть имеются два поля A и B, принимающие целые значения в диапазоне [1 .. 30], причем значение поля B зависит от значения поля A следующим образом (см. Рис. 1):

- если $1 \le A \le 10$, то $B = 1$;
- если $11 \le A \le 20$, то $11 \le B \le 20$;
- если $21 \le A \le 30$, то $B = A$.

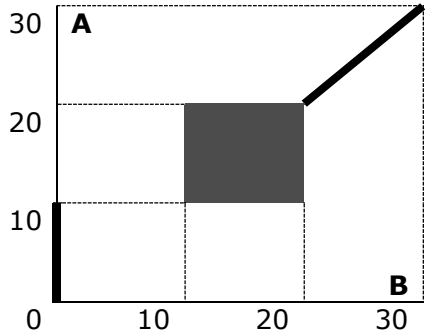



*Рис. 1. Пример зависимости значений полей.*

Если перебирать не все возможные комбинации полей, а только удовлетворяющие перечисленным условиям, то количество получаемых в результате комбинаций снизится практически на порядок (с 900 до 120).

Кроме того, функциональность тестируемого приложения может разбиваться на несколько независимых друг от друга аспектов. В этом случае существенно снизить количество перебираемых комбинаций значений полей позволяет так называемый диагональный комбинатор, перебирающий такое множество $S$ кортежей, что для любого $i$ от 1 до $n$ и для любого $s$ из $S_i$ существует такой кортеж $(s_1, \ldots, s_n)$ в $S$, что $s = s_i$, где $S_i$ – это множество значений, итерируемое $i$-м подчиненным итератором. Такой метод гарантирует, что в полученном множестве тестов будет содержаться каждое значение каждого поля. При этом общее количество тестов будет существенно меньше, чем в случае использования прямого произведения: вместо произведения мощностей множеств значений всех полей получится лишь максимальное из значений этих мощностей.

Инструмент Pinery [9], [10], [11], разработанный в ИСП РАН, предназначен для генерации тестов, обладающих сложной структурой. Генерация тестов с данной синтаксической структурой настраивается путем задания в терминах этой структуры ограничений на фрагменты тестов. При этом ограничения могут быть условными, т.е. зависеть от значений каких-то полей.

При задании ограничения на значения поля в Pinery можно пользоваться, в частности, следующими средствами:

- указать непосредственные значения;
- задать функцию вычисления в зависимости от значений других полей;
- указать принадлежность некоторому множеству, например:
    - отрезку целых чисел;
    - множеству значений другого поля в данном тесте (например, в случае задания значений для внешнего ключа).

Важным ограничением другого рода в Pinery является указание способа комбинирования полей. В частности, наряду с использованием прямого произведения в Pinery есть возможность использовать диагональный комбинатор, а также указать наличие зависимости между значениями полей, причем сами эти комбинаторы могут образовывать иерархию.

Эти возможности позволяют в Pinery более тонко, по сравнению с другими инструментами, настраивать процесс генерации и получать тестовые данные по объемам близкие к оптимальным.

## *4. Описание метода*

Метод нацеленной генерации тестовых данных реализует предложенную в ИСП РАН технологию разработки тестов UniTESK [12], [13], [14], основанную на использовании формальной модели тестируемой системы и

общей схемы, представленной на Рис. 2. Краеугольным камнем этой схемы является формализация требований [15] к тестируемой системе.

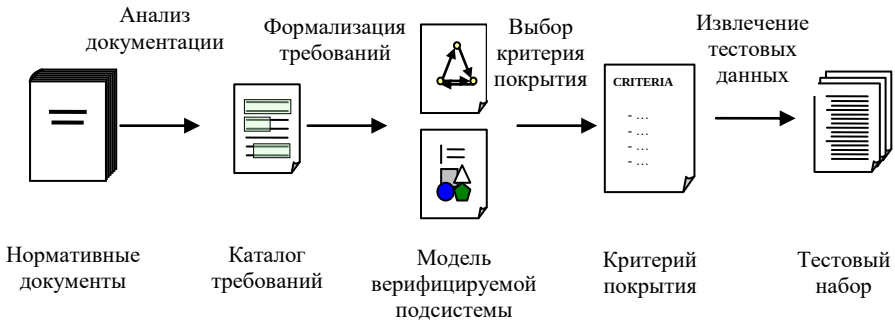



*Рис. 2. Общая схема нацеленной генерации тестов.*

Эта схема состоит из нескольких этапов. На первом этапе требования к тестируемой системе извлекаются из нормативных документов и систематизируются. В результате получается каталог требований, в котором требования сформулированы максимально однозначно, они классифицированы, и, возможно, установлены связи между отдельными требованиями. Каталог требований используется на последующих этапах.

Второй этап нацелен на представление требований в формальном виде. Требования из каталога записываются с использованием того или иного математического формализма. Такая запись требований называется формальной моделью.

На третьем этапе в терминах построенной модели формулируется критерий покрытия. Для этого в множестве входных данных выделяется конечное число подмножеств (возможно, пересекающихся между собой), объединение которых совпадает со всем множеством, так чтобы было единообразно функциональное поведение тестируемой системы на входных данных из одного подмножества. Отправной точкой для выделения подмножеств входных данных с единообразным поведением приложения являются условия, содержащиеся в требованиях каталога. Однако, кроме того, конкретная семантика этих условий может явиться почвой для формулирования дополнительных гипотез о работе тестируемого приложения, которые индуцируют некоторое подразбиение входных данных на более мелкие подмножества.

На четвертом этапе на основе построенной модели и в соответствии со сформулированным критерием покрытия автоматически генерируются тестовые данные. В нашем подходе автоматизированная генерация

проводится с использованием инструмента Pinery. Для запуска генератора ему требуется предоставить формальное описание модели тестовых данных, а также настройки генерации, нацеленные на достижение критерия покрытия.

Вопросы, связанные с прогоном тестов, анализом результатов, обнаружением дефектов и их исправлением на тестируемой системе, выходят за рамки данной статьи.

## *5. Пример*

Проиллюстрируем, как применяется предложенный метод, на примере тестирования приложения, работающего в рамках упрощенной системы банковского кредитования.

**Нормативными документами** здесь являются неформальное описание системы кредитования и работы приложения:

- Клиент с определенным типом регистрируется в базе, получая идентификатор.
- Клиент получает в кредит определенную сумму денег, которую необходимо выплатить в течение несколько месяцев.
- Клиент ежемесячно должен вносить платеж, сумма которого вычисляется исходя из оставшегося долга, деленного на количество месяцев, оставшихся до окончания срока действия кредита.
- На клиента может быть наложен штраф, если сумма очередного внесенного взноса меньше суммы необходимого ежемесячного платежа, а также клиенту может быть начислена премия, если сумма внесенного взноса больше суммы ежемесячного платежа. Размер штрафов и премий зависит от типа клиента и от типа кредита.
- Если клиент желает ежемесячно получать уведомления о текущем состоянии кредита, он может оставить свой адрес электронной почты.
- Приложение запускается в конце месяца и осуществляет следующие действия:
    A. рассчитывает штраф/премию клиента и списывает с долга внесенную сумму и рассчитанный штраф/премию по следующим правилам:
        I. Если взнос, внесенный клиентом, меньше суммы ежемесячного платежа, то на абонента налагается штраф:
            1. Если абонент имеет статус «VIP», то штраф за просроченный платеж равен 0.
            2. Если абонент имеет статус «USUAL», то он облагается штрафом за просроченный платеж с коэффициентом 0.03.

II. Если взнос, внесенный клиентом, не меньше, чем сумма ежемесячного платежа, и не больше общей суммы долга, то абонент получает премию:
    1. Если абонент имеет статус «VIP» и тип кредита «B» или имеет статус «USUAL» и тип кредита «A», то коэффициент при расчете премии равен 0.02.
    2. Если абонент имеет статус «VIP» и тип кредита “A”, то коэффициент при расчете премии равен 0.05.
    3. Если абонент имеет статус «USUAL» и тип кредита “B”, то коэффициент при расчете премии равен 0.01.
III. Если взнос, внесенный клиентом, больше общей суммы долга, то приложение должно сигнализировать о необходимости возврата лишних денег клиенту.

B. Если для клиента определен адрес электронной почты, то после вычисления оставшегося долга приложение отправляет клиенту уведомление о текущем состоянии кредита.

| Короткое имя | Полное имя | Тип | Описание |
|---|---|---|---|
| MP | MONTH_PAYMENT | NUMBER | Ежемесячный взнос |
| D | DEBT | NUMBER | Общая сумма оплаты |
| P | PAYMENT | NUMBER | Внесенная сумма |
| CL | CLIENT_TYPE | CHAR | Тип клиента:<br>“V” – VIP клиент<br>“U” – обычный клиент (USUAL) |
| CR | CREDIT_TYPE | CHAR | Тип кредита:<br>“A” – тип “A”<br>“B” – тип “B” |
| E | EMAIL | STRING | Адрес электронной почты |

*Табл. 1. Параметры работы системы кредитования.*

На **первом этапе** из нормативной документации извлекается каталог требований к тестируемой системе. В нашем случае каталогом требований являются пункты правил поведения тестируемого приложения (A с подпунктами и B). В целях более удобного восприятия, требования к поведению приложения могут быть полуформально представлены в виде следующей схемы, приведенной на Рис. 3.

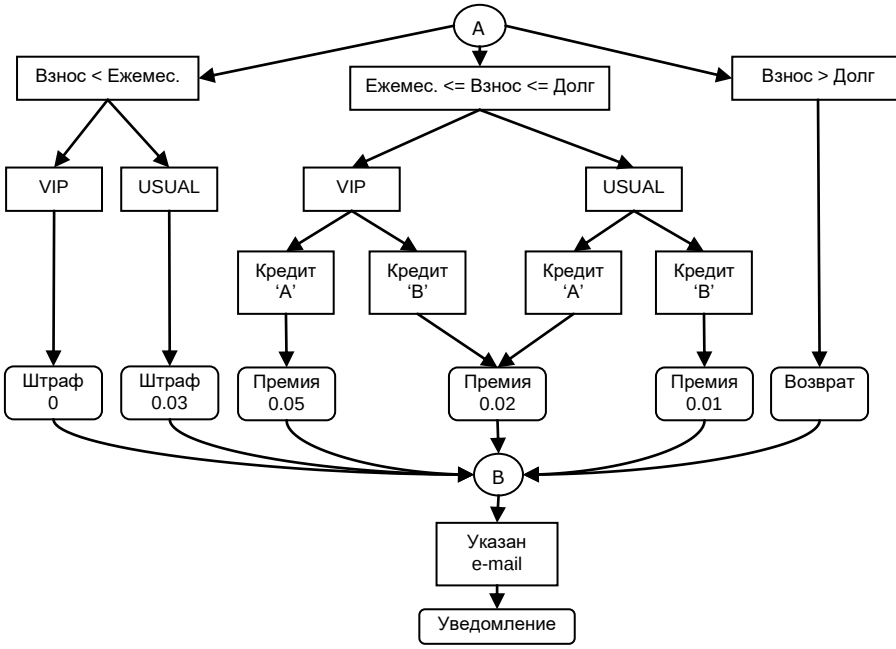



*Рис. 3. Схематичное изображение требований к системе кредитования.*

Приступим ко **второму этапу** – построению формальной модели. Параметры работы тестируемого приложения представлены в Табл. 1.

Условие консистентности данных в терминах введенных параметров таково: MP <= D. Требования к поведению приложения формально описываются так:

A.I.1. P < MP **&&** CL = "V" => Штраф 0;

A.I.2. P < MP **&&** CL = "U" => Штраф -0.03;

A.II.1. MP <= P <= D **&&** ( CL = "V" **&&** CR = "B" || CL = "U" **&&** CR = "A" )

=> Премия 0.02;

A.II.2. MP <= P <= D **&&** CL = "V" **&&** CR = "A" => Премия 0.05;

A.II.3. MP <= P <= D **&&** CL = "U" **&&** CR = "B" => Премия 0.01;

A.III. P > D => Возврат.

B. E ≠ "" => Уведомление.

Перейдем к **третьему этапу** – формулировке критерия покрытия. Для этого, помимо условий, содержащихся в требованиях, мы введем дополнительно следующие гипотезы относительно поведения тестируемого приложения:

1. При выполнении условия MP <= P <= D приложение может вести себя по-разному в следующих случаях:
    - MP = P = D;
    - MP = P < D;
    - MP < P < D;
    - MP < P = D.
2. При выполнении условий требования A.II.1 приложение может вести себя по-разному в следующих случаях:
    - CL = “V” **&&** CR = “B”;
    - CL = “U” **&&** CR = “A”.

Формулировать правила разбиения области входных данных на подмножества с единообразным поведением тестируемого приложения мы будем последовательно для каждого из параметров. Начнем с параметров MP, D, P, CL и CR, относящихся к требованиям из группы A.

Исходя из гипотезы 1, область входных данных в части параметров MP и D разбивается на два подмножества:

- MP = D;
- MP < D.

Исходя из условий требований, в случае MP = D область входных данных в части параметра P разбивается на три подмножества:

- P < MP;
- MP = P = D;
- P > D.

Исходя из условий требований и из гипотезы 1, в случае MP < D область входных данных в части параметра P разбивается на пять подмножеств:

- P < MP;
- MP = P < D;
- MP < P < D;
- MP < P = D;
- P > D.

Исходя из условий требований группы A.I, в случае P < MP область входных данных в части параметров CL и CR разбивается на два подмножества, соответствующих всем возможным значениям параметра CL.

Исходя из условий требований группы A.II и из гипотезы 2, в случае MP <= P <= D область входных данных в части параметров CL и CR разбивается на

четыре подмножества, соответствующих всем возможным сочетаниям значений этих параметров.

Исходя из условия требования A.III, в случае P > D область входных данных в части параметров CL и CR составляет единственное подмножество.

Перейдем к параметру E, относящемуся к требованию B. Исходя из условия требования B, область входных данных в части параметра E разбивается на два подмножества: с пустым и непустым значением этого параметра.

Разбиение области входных данных в части параметров, относящихся к требованиям из группы A, индуцируется прямым произведением (с учетом зависимостей) разбиений областей данных каждого из этих параметров.

Поскольку функциональность приложения разбивается на два независимых аспекта A и B, то разбиение всей области входных данных индуцируется диагональным комбинированием разбиений областей данных, соответствующих каждой из этих групп.

Перейдем к **четвертому этапу** – автоматизированной генерации тестовых данных с использованием инструмента Pinery.

Для генерации тестовых данных на Pinery прежде всего необходимо задать формальное описание схемы базы данных (например, с использованием DDL-подмножества языка SQL). После этого в терминах заданной схемы БД описываются ограничения на генерируемые данные. В рассматриваемом примере база данных содержит одну таблицу CREDITS, состоящую из пяти полей: MONTH_PAYMENT, DEBT, PAYMENT, CLIENT_TYPE, CREDIT_TYPE. Далее мы будем ссылаться на эти поля по их коротким именам.

Ограничения в этом примере бывают двух видов:

- ограничение на значения конкретного поля;
- ограничение на способ комбинирования нескольких полей.

Начнем с описаний ограничений первого вида.

Для получения ситуаций MP = D и MP < D достаточно положить, например, MP = 6, а D = 6 и D = 30. Для получения ситуаций, когда значение поля E пусто и непусто, достаточно положить, например, E = “” и E = “…@…” (некоторый адрес). Для задания этих значений в Pinery требуется описать следующие ограничения, задающие список конкретных значений полей[1]:

```
MP = { 6 };
D = { 6, 30 };
E = { “”, “…@…” };
```

[1] В Pinery ограничения записываются в XML-форме. Здесь, в целях экономии места, а также для повышения понятности записи, мы приводим ограничения, записанные на полу-формальном псевдо-коде.

Это примеры так называемых *безусловных* ограничений – значения этих полей *во всех случаях* являются такими, как это перечислено в соответствующем ограничении. Однако, не всегда можно обойтись лишь безусловными ограничениями. Например, множество значений поля P зависит от значений полей MP и D. Таким образом, нам потребуются два условных ограничения: для случаев MP = D и MP < D.

Для получения ситуации P < MP достаточно положить P = MP – 1; для получения ситуации P > D достаточно положить P = D + 1; для получения ситуации MP < P < D при MP < D достаточно положить P = (MP + D)/2. В результате, ограничения для поля P запишутся так:

```
P[ MP<D ] = { MP - 1, MP, (MP + D)/2, D, D + 1 };
P[ MP=D ] = { MP - 1, D, D + 1 };
```

Множество значений полей CL и CR зависит от значений полей P, MP и D:

```
CL[ P<MP ] = { "V", "U" };
CR[ P<MP ] = { "A" };
CL[ MP<=P && P<=D ] = { "V", "U" };
CR[ MP<=P && P<=D ] = { "A", "B" };
CL[ P>D ] = { "V" };
CR[ P>D ] = { "A" };
```

Перейдем к рассмотрению того, как задается комбинатор значений полей для генерации кортежей таблицы CREDITS. Начнем с того, что множества значений полей CL и CR во всех случаях можно комбинировать методом прямого произведения:

```
Product( CL, CR )
```

Аналогично, множества значений полей MP и D можно комбинировать методом прямого произведения:

```
Product( MP, D )
```

Поскольку значения полей CL и CR зависит от значений полей P, MP и D, а значение поля P в свою очередь также зависит от значений полей MP и D, то наличие таких зависимостей выражается использованием специального «зависимого» комбинатора, который задает комбинации полей, относящихся к требованиям из группы A:

```
Depend(  Product( MP, D )
      => P
      => Product( CL, CR )
      );
```

Как было указано выше, комбинации полей, относящихся к требованиям из групп A и B, производятся методом диагонали, который задается комбинатором значений полей для генерации кортежей таблицы CREDITS:

```
combinator( CREDITS ) =
    Diagonal( Depend(
Product( MP, D )
                      => P
                      =>
Product( CL, CR )
                      )
            , E
            );
```

Итоговые тестовые данные представлены в следующей Табл. 2.

| **D** | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **MP** | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
| **P** | 5 | 5 | 6 | 6 | 6 | 6 | 7 | 5 | 5 | 6 | 6 | 6 | 6 | 18 | 18 | 18 | 18 | 30 | 30 | 30 | 30 | 31 |
| **CL** | V | U | V | V | U | U | V | V | U | V | V | U | U | V | V | U | U | V | V | U | U | V |
| **CR** | A | A | A | B | A | B | A | A | A | A | B | A | B | A | B | A | B | A | B | A | B | A |
| **E** |  | @ |  | @ |  | @ |  | @ |  | @ |  | @ |  | @ |  | @ |  | @ |  | @ |  | @ |

*Табл. 2. Тестовые данные для системы кредитования.*

Заметим, что применение зависимого комбинирования позволило задать единую конфигурацию генератора вместо двух (для случаев MP = D и MP < D). А применение зависимого и диагонального комбинирования сократило количество тестовых данных более, чем на 65%, по сравнению с общим прямым произведением: в нашем случае получилось 22 кортежа, в то время как прямое произведение дает 64 (3*2*2*2 = 24 для случая MP = D, плюс 5*2*2*2 = 40 для случая MP < D).

Экономия увеличивается при увеличении количества возможных значений полей. Так, например, в случае добавления еще одного типа клиента и одного типа кредита описанный подход дает уже экономию в 70% (3*3*3*2+

5*3*3*2 = 144 для прямого произведения против (3 + 1*3*3 + 1) + (3 + 3*3*3 + 1) = 44 в нашем подходе).

Еще больше заметно увеличение экономии при добавлении полей, относящихся к другим независимым аспектам работы приложения. Пусть, например, для взноса указывается тип оплаты с двумя возможными значениями: 'O' – оплата по безналичному расчету и 'E' – оплата наличными. Тогда количество записей в нашем подходе не увеличивается (поскольку применяется диагональный комбинатор), а при использовании прямого произведения количество вариантов удваивается.

## *6. Заключение*

В статье предложен способ генерации данных для функционального тестирования приложений над большими объемами данных, нацеленных на покрытие функциональных ветвей приложения. Отличие предложенного подхода от реализованных в существующих инструментах состоит в том, что полученные тестовые данные, обеспечивая покрытие функциональности тестируемого приложения, являются более оптимальными относительно временных затрат на тестирование, а также временных и трудозатрат на анализ результатов тестирования.

Генерация данных при помощи декартова комбинирования всех полей обеспечивает полное покрытие, но практически всегда является избыточной, причем избыточность стремительно нарастает при увеличении количества комбинируемых полей и подмножеств их значений, от которых зависит поведение приложения.

Предлагаемый подход основан на формализации и анализе требований к тестируемой системе. Использование наряду с прямым произведением комбинаторов зависимых множеств и диагональных комбинаторов позволяет очень существенно сократить тестовый набор без потери качества тестирования и в результате получить тестовые данные, по объемам близкие к оптимальным.

Подход инструментально поддерживается генератором данных сложной структуры Pinery.

**Литература**


[1] British National Corpus (BNC), 2009, http://www.natcorp.ox.ac.uk/corpus/index.xml
[2] . N. Oostdijk and P. Haan, Corpus-Based Research into Language. In honour of Jan Aarts, Amsterdam/Atlanta, GA, 1994, VII.
[3] M. L. Songini, QuickStudy: Extract, Transform and Load (ETL), 2004, Computerworld, http://www.computerworld.com/s/article/89534/QuickStudy_ETL
[4] DTM Data Generator, test data generator for database testing purposes, 2004, http://www.sqledit.com/dg/
[5] Test Data Generator - TurboData - "Out of the Box", 2009, http://www.turbodata.ca/
[6] DBMonster - The dbMonster home page – About, 2003, http://dbmonster.kernelpanic.pl/

[7] D. Chays, Y. Deng, P.G. Frankl, E.J. Weyuker, An AGENDA for testing relational database applications, Software testing, verification and reliability, 2004, VOL 14; PART 1, pages 17-44
[8] C. Binnig, D. Kossmann, E. Lo, Testing database applications, 2006, Proceedings of the 25th ACM SIGMOD international conference on management of data / Principles of database systems, Chicago
[9] А.В. Демаков, С.В. Зеленов, С.А. Зеленова. Генерация тестовых данных сложной структуры с учетом контекстных ограничений. // Труды ИСП РАН, 2006, том 9, 83–96.
[10] А.В. Демаков, С.В. Зеленов, С.А. Зеленова. Генератор сложных данных Pinery: реализация новых возможностей UniTESK // Труды ИСП РАН, Москва, 2008, т. 14, часть 1, 119–136.
[11] А.В. Демаков, С.В. Зеленов, С.А. Зеленова. Использование абстрактных моделей для генерации тестовых данных сложной структуры // Программирование, Москва, 2008, том. 34, № 6, 341–350.
[12] В. В. Кулямин, А. К. Петренко, А. С. Косачев, И. Б. Бурдонов. Подход UniTesK к разработке тестов. Программирование, 29(6):25-43, 2003.
[13] 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, 2004, том. 5, 121–156.
[14] UniTESK, индустриальная технология надежного тестирования, 2006, http://unitesk.ru/
[15] В. В. Кулямин, Н. В. Пакулин, О. Л. Петренко, А. А. Сортов, А. В. Хорошилов. Формализация требований на практике. // Препринт ИСП РАН. М.: ИСП РАН, 2006.

## Covering functionality of database applications by generating special data

*Evgeny Kostychev, Vitaly Omelchenko, Sergey Zelenov*
*{kostychev, vitaly, zelenov}@ispras.com*

**Abstractю.** Software processing huge volumes of data is the one of the most used kind of software. In particular it is an important part of enterprise integration called data integration. There are tools supporting development and execution of applications implementing extract, transformation and load pattern. In regard of functional testing of such applications there is a specific is a number of combinations of input data. There are tools supporting test data generation for database application on the basis on schemes of databases or SQL queries of application under tests. Using their one can ensure the functionality covering only by means of a brute force of all available combinations. We propose a method reducing excessiveness of data generation. By means of the method the functionality coverage is achieved with fewer amounts of test data that is close to optimal (one set of test data for one functionality branch).

**Keywords:** test data generation; database applications; functional testing; data integration.

# Извлечение ключевых терминов из сообщений микроблогов с помощью Википедии

*А.В. Коршунов*
*korshunov@ispras.ru*

**Аннотация.** В статье описывается способ извлечения ключевых терминов из сообщений микроблогов с использованием информации, полученной путём анализа структуры и содержимого интернет-энциклопедии Википедия. Работа алгоритма основана на расчёте для каждого термина его "информативности", т.е. оценки вероятности того, что он может быть выбран ключевым в тексте. В ходе тестирования разработанный алгоритм показал удовлетворительные результаты в условиях поставленной задачи, существенно опережая аналоги. В качестве демонстрации возможного применения разработанного алгоритма был реализован прототип системы контекстной рекламы. Сформулированы также варианты использования информации, полученной путём анализа сообщений Twitter, для реализации различных вспомогательных сервисов.

**Ключевые слова:** Информационный поиск; извлечение ключевых терминов; обработка естественного языка; анализ текста; семантический анализ; микроблоггинг; Твиттер; Википедия; контекстно-зависимая реклама.

## *1. Введение*

На сегодняшний день одной из самых важных и заметных областей Web 2.0, ключевым принципом которой является участие пользователей в работе сайтов, являются сетевые дневники, или веб-логи, сокращённо называемые *блогами*. Концептуальным развитием блогов, обусловленным их широкой социализацией, являются микроблоги, которые имеют ряд характерных особенностей: ограниченная длина сообщений, большая частота публикаций, разнообразная тематика, различные пути доставки сообщений и т.д.

Первый и наиболее известный сервис микроблогов *Twitter* был запущен в октябре 2006 г. компанией *Obvious* из Сан-Франциско. К настоящему времени постоянно растущая аудитория сервиса составляет десятки миллионов человек. Очевидно, что автоматизированное выделение наиболее значимых терминов из потока сообщений, генерируемого сообществом Twitter, имеет практическое значение как для определения интересов различных групп

пользователей, так и для построения индивидуального профиля каждого из них.

Однако нужно отметить, что классические статистические методы экстракции ключевых терминов, основанные на анализе коллекций документов, малоэффективны в данном случае. Это обусловлено чрезвычайно малой длиной сообщений (до 140 символов), их разнообразной тематикой и отсутствием логической связи между собой, а также обилием редко используемых аббревиатур, сокращений и элементов специфического микросинтаксиса.

Для решения этой проблемы в представленной работе относительная значимость терминов в анализируемом контексте определяется с помощью данных о частоте их использования в качестве ключевых в интернет-энциклопедии Википедия. Работа алгоритма основана на расчёте для "информативности" каждого термина, т.е. оценки вероятности того, что он может быть выбран ключевым в тексте. В дальнейшем к анализируемому набору терминов применяется ряд эвристик, результатом которых является список терминов, сочтённых ключевыми.

## *2. Обзор Twitter*

Возникновение и последующее развитие идеи создания сервиса *микроблогов* современные исследователи Интернета вполне обоснованно считают результатом процесса интеграции концепции *социальных сетей* с сетевыми дневниками – *блогами* [1].

По определению, данному Walker в [2], *блогами* называются часто обновляемые веб-сайты, состоящие из содержащих различную информацию записей, размещённых в обратном хронологическом порядке.

Характерные черты *блоггинга* могут быть описаны с использованием 3 ключевых принципов [3]:

- содержимое блогов представляет собой короткие сообщения;
- сообщения имеют общее авторство и находятся под контролем автора;
- возможна агрегация множества потоков сообщений от разных авторов для удобного чтения.

Эти принципы применимы также и для *микроблоггинга* [4]. Однако в то время как для блоггинга публикация и агрегация сообщений являются задачами для разных программных продуктов, сервисы микроблоггинга предоставляют все перечисленные возможности сразу.

Кроме того, микроблоггинг учитывает потребность пользователей в более быстром режиме коммуникации, чем при обычным блоггинге. Поощряя более короткие сообщения, он уменьшает время и мысленную работу, необходимые для создания контента. Это также является одним из главных отличительных

от блоггинга признаков. Другое отличие заключается в частоте обновлений. Автор обычного блога обновляет его, в среднем, раз в несколько дней, в то время как автор микроблога может обновлять его несколько раз в день.

Основные функции наиболее популярного на сегодняшний день сервиса микроблогов Twitter представляют собой очень простую модель. Пользователи могут отправлять короткие сообщения, или *твиты*, длиной не больше 140 символов. Сообщения отображаются как *поток* на странице пользователя. В терминах социальных связей, Twitter даёт возможность пользователям *следовать* (*follow*) за любым числом других пользователей, называемых *друзьями*. Сеть контактов Twitter асимметрична, т.е. если один пользователь следует за другим, то второй не обязан следовать за ним. Пользователи, следующие за другим пользователем, называются его *последователями* (*followers*).

Пользователи имеют возможность указать, желают ли они, чтобы их твиты были доступны *публично* (появляются в обратном хронологическом порядке на главной странице сервиса и на странице самого пользователя, называемой *микроблогом*) или *приватно* (только последователи пользователя могут видеть его сообщения). По умолчанию, все сообщения доступны любому пользователю.

Для облегчения понимания внутреннего устройства Twitter целесообразно ввести 2 концепции: *элемент* как отдельное сообщение и *канал* как поток элементов, чаще всего принадлежащих одному пользователю [3]. Способы взаимодействия концепций между собой представлены в табл. 1.

| | **Канал** | **Элемент** |
|---|---|---|
| **Канал** | следование | ретвит |
| **Элемент** | @-ссылка | ответ |

*Табл. 1. Способы взаимодействия концепций Twitter между собой.*

Ниже приведено краткое описание каждого из способов взаимодействия:

- *следование* — один канал имеет другой в своей сети и читает его обновления;
- *@-ссылка* — текст элемента может ссылаться на другой канал с помощью конструкции @<имя_канала>;
- *ретвит* — пользователи берут элементы из чужих каналов и помещают в свои с добавлением @-ссылки на источник и, в некоторых случаях, своего комментария;
- *ответ* — один элемент является прямой реакцией на предыдущий.

В заключение общего описания особенностей Twitter можно отметить, что необходимым условием для успешного принятия пользователями новой технологии (или нового способа применения существующего инструмента) является позитивное отношение к её потенциалу. Компания Gartner добавила микроблоггинг в свой *"цикл ажиотажа"* (*hype cycle*) в 2008 году [5], предсказывая резкий рост популярности этого явления. Согласно Gartner, передовые компании исследуют потенциал микроблоггинга, чтобы усовершенствовать другие социальные информационные средства и каналы. Всё это говорит о том, что микроблоги являются одним из самых перспективных и динамично развивающихся сегментов Интернета.

Вместе с тем, микроблоги являются всё ещё относительно новым явлением среди онлайновых социальных сетей и на данном этапе недостаточно исследованы.

## 2.1. Особенности сообщений

Помимо наличия «ретвитов» и @-ссылок, в «твитах» могут также присутствовать и другие элементы специфического *микросинтаксиса*, задачей которого является представить часто используемые понятия в сокращённой общеупотребительной форме, а также расширить набор инструментов для повышения информативности сообщений в условиях ограниченного их размера.

Основной частью микросинтаксиса являются *слэштеги*, каждый из которых состоит из символа «/» и *указателя*, который и определяет смысл слэштега. Эти элементы используют для различных целей. Например, «*/via*» - для ссылки на автора сообщения при «ретвите»; «*/by*» - для ссылки на автора исходного сообщения, если оно является результатом цепочки «ретвитов»; «*/cc*» – для указания тех подписчиков микроблога, которым в первую очередь адресовано сообщение и т.д.

Использование слэштегов полностью является инициативой самих пользователей, поэтому не существует единых правил по их использованию. Приведённое описание отражает наиболее популярные способы применения слэштегов к настоящему моменту. Существуют, однако и другие рекомендации, которые тоже заслуживают внимания, т.к. каждый пользователь применяет элементы микросинтаксиса по своему усмотрению. Например, возможно объединение всех использованных в сообщении слэштегов без символа «/» в одну группу, ограниченную скобками. Некоторые пользователи размещают слэштеги строго в конце сообщения, предваряя лишь первый из них символом «/» с целью экономии символов.

Поскольку в Twitter не существует простого и удобного способа для группирования «твитов» разных пользователей по тематике, сообщество пользователей пришло к собственному решению: использование *хэштегов*. Они похожи на другие примеры использования тегов (например, для

аннотирования записей в обычных блогах) и позволяют добавить «твиты» в какую-либо категорию.

Хэштеги начинаются с символа «#», за которым следует любое сочетание разрешённых в Twitter непробельных символов; чаще всего это слова или фразы, в которых первая буква каждого слова приведена к верхнему регистру. Они могут встречаться в любой части «твита», зачастую пользователи просто добавляют символ «#» перед каким-либо словом. Другим вариантом использования является добавление популярных хэштегов, таких, как «*#haiku*». При добавлении в сообщение хэштега оно будет отображаться при поиске в потоке сообщений Twitter по этому хэштегу.

К неофициальным, но общепринятым правилам использования хэштегов относится выбор в качестве них терминов, релевантных теме сообщения, а также добавление лишь небольшого количества их в одно сообщение. Это позволяет рассматривать их в качестве терминов, которые с достаточной степенью вероятности отражают общую тематику сообщения.

## 3. Существующие подходы к извлечению ключевых терминов

Одной из задач извлечения информации из текста является выделение ключевых терминов, с определённой степенью достоверности отражающих тематическую направленность документа. Автоматическое *извлечение ключевых терминов* можно определить как автоматическое выделение важных тематических терминов в документе. Оно является одной из подзадач более общей задачи – автоматической *генерации ключевых терминов*, для которой выделенные ключевые термины не обязательно должны присутствовать в данном документе [6]. В последние годы было создано множество подходов, позволяющих проводить анализ наборов документов различного размера и извлекать ключевые термины, состоящие из одного, двух и более слов.

Важнейшим этапом извлечения ключевых терминов является расчёт их *весов* в анализируемом документе, что позволяет оценить их значимость относительно друг друга в данном контексте. Для решения этой задачи существует множество подходов, которые условно делятся на 2 группы: требующие обучения и не требующие обучения. Под *обучением* подразумевается необходимость предварительной обработки исходного корпуса текстов с целью извлечения информации о частоте встречаемости терминов во всём корпусе. Другими словами, для определения значимости термина в данном документе необходимо сначала проанализировать всю коллекцию документов, к которой он принадлежит. Альтернативным подходом является использование лингвистических онтологий, которые являются более или менее приближёнными моделями существующего набора слов заданного языка. На базе обоих подходов были созданы системы для автоматической экстракции ключевых терминов, однако в этом направлении

постоянно ведутся работы с целью повышения точности и полноты результатов, а также с целью использования методов извлечения информации из текста для решения новых задач [7-13].

Самыми распространёнными схемами для расчёта весов терминов являются *TF-IDF* и различные его варианты, а также некоторые другие (*ATC, Okapi, LTU*). Однако общей особенностью этих схем является то, что они требуют наличия информации, полученной из всей коллекции документов. Другими словами, если метод, основанный на TF-IDF, используется для создания представления о документе, то поступление нового документа в коллекцию потребует пересчёта весов терминов во всех документах. Следовательно, любые приложения, основанные на значениях весов терминов в документе, также будут затронуты. Это в значительной мере препятствует использованию методов извлечения ключевых терминов, требующих обучения, в системах, где динамические потоки данных должны обрабатываться в реальном времени, например, для обработки сообщений микроблогов [14].

Для решения этой проблемы было предложено несколько подходов, таких как алгоритм *TF-ICF* [15]. В качестве развития этой идеи Mihalcea и Csomai в 2007 году предложили использовать в качестве обучающего тезауруса Википедию [16]. Они применили для расчётов информацию, содержащуюся в аннотированных статьях энциклопедии с вручную выделенными ключевыми терминами. Для оценки вероятности того, что термин будет выбран ключевым в новом документе, используется формула:

$$P\ \text{ключевой термин}|W\ » \frac{\text{число}\ D_{\text{ключевой}}}{\text{число}\ D_W}, \qquad (1)$$

где $W$ - термин;

$D_{ключевой}$ - документ, в котором термин был выбран ключевым;

$D_W$ - документ, в котором термин появился хотя бы один раз.

Эта оценка была названа авторами *keyphraseness* (в данной работе определена как *информативность*). Она может быть интерпретирована следующим образом: «чем чаще термин был выбран ключевым из числа его общего количества появлений, тем с большей вероятностью он будет выбран таковым снова».

Информативность может принимать значения от 0 до 1. Чем она выше, тем выше значимость термина в анализируемом контексте. Например, для термина «*Of course*» в Википедии существует только одна статья, посвящённая песне американского исполнителя, поэтому он редко выбирается ключевым, хотя встречается в тексте очень часто. Значение его информативности, таким образом, будет близко к 0. Напротив, термин «*Microsoft*» в тексте любой

статьи почти всегда будет выделен ключевым, что приближает его информативность к 1.

Данный подход является довольно точным, т.к. все статьи в Википедии вручную аннотируются ключевыми терминами, поэтому предложенная оценка их реальной информативности является лишь результатом обработки мнений людей.

Вместе с тем, эта оценка может быть ненадёжной в тех случаях, когда используемые для расчётов значения слишком малы. Для решения этой проблемы авторы рекомендуют рассматривать только те термины, которые появляются в Википедии не менее 5 раз.

В заключение обзора методов извлечения ключевых терминов нужно сказать,

$$w_i = TF_i * K_i , \qquad (2)$$

что для расчёта веса термина $w_i$ в данной работе использовалась формула:

где $i$ - порядковый номер термина;

$TF_i$ - частота термина в анализируемом сообщении;

$K_i$ - информативность термина по данным Википедии.

TF означает *частоту термина* (*Term Frequency*). Значение этого компонента формулы равно отношению числа вхождения некоторого термина к общему количеству терминов сообщения. Таким образом, оценивается важность термина $t_i$ в пределах отдельного сообщения.

# *4. Описание алгоритма*

## 4.1. Извлечение информации из Википедии

Одним из важнейших этапов разработки системы явилась обработка XML-дампа всех статей английской Википедии по состоянию на июль 2009 г. Целью анализа был расчёт *информативности* для всех терминов Википедии по формуле (1).

Нужно отметить, что для одной *концепции* в словаре Википедии может быть несколько *синонимов*. Например, термин «*IBM*» имеет несколько синонимов: «*International Business Machines*», «*Big Blue*» и т.д. Так как в разработанной системе отсутствует этап *разрешения лексической многозначности* терминов, то было недопустимо, чтобы синонимы имели различные значения информативности. Поэтому было принято считать, что информативность всех синонимов одной концепции становится одинаковой, исходя из общей статистики для всех них.

Кроме того, согласно рекомендациям авторов методики расчёта информативности [16], были исключены термины, которые были найдены *менее чем в 5 статьях*. Если пропустить этот шаг, то результирующее значение зачастую становится недостоверным и не позволяет корректно оценить относительную значимость термина в контексте. В результате данного этапа БД содержит 5 445 377 терминов с рассчитанной для них информативностью.

## 4.2. Извлечение ключевых терминов

Общая архитектура разработанной системы представлена на рис. 1.


*Рис. 1. Общая архитектура системы.*

Для получения информации о сообщениях пользователя с сервера Twitter использовался Perl-модуль *Net::Twitter*. Результатом взаимодействия с Twitter API является получение некоторого числа последних сообщений его аккаунта, иначе называемых *timeline*.

Для решения поставленной задачи был выбран метод *statuses/friends_timeline*, возвращающий сообщения друзей пользователя. Для тестирования были созданы аккаунты, подписанные на обновления статуса единственного друга. При этом в самом аккаунте не публиковалось никаких сообщений. Таким образом, результат вызова данного метода содержит лишь необходимое количество сообщений одного пользователя Twitter, что и требуется в качестве исходных данных.

В процессе *предварительной обработки* текста, или *препроцессинга*, содержимое полученных с сервера Twitter сообщений преобразуется к формату входных данных для алгоритма извлечения ключевых терминов.

Помимо стандартных для этого этапа операций, производится также удаление *слэштегов* и *@-ссылок*, а также служебных слов «*RT*» и «*RETWEET*», то есть тех элементов специфического синтаксиса Twitter, которые не несут

смысловой нагрузки и являются *стоп-словами* в терминах обработки естественного языка. На этом этапе также производится извлечение *хэштегов*, которые в последующем обрабатываются наравне с остальными словами.

Очевидно, однако, что для составления списка кандидатов в ключевые термины недостаточно лишь исходной последовательности слов. Исходя из небольшой длины сообщений, представляется возможным сделать алгоритм максимально избыточным с целью повышения *полноты* результатов, то есть не пропустить ни один возможный ключевой термин, из скольких бы слов он ни состоял. С этой целью производится поиск *всех* возможных *N-грамм*, то есть последовательностей идущих друг за другом слов. Количество N-грамм $Q_k$, которые можно получить из *k* слов (включая N-граммы из одного слова), таким образом, равно:

$$Q_k = \sum_{i=1}^{k} i \qquad (3)$$

Все полученные на этом этапе термины добавляются в массив возможных ключевых терминов.

Завершающим этапом препроцессинга является *стоплистинг*, т.е. удаление из полученного массива тех слов, которые не несут существенной смысловой нагрузки. Важно отметить, что стоплистинг выполняется лишь после извлечения N-грамм. Таким образом, стоп-слова могут входить в составные термины, но удаляются из списка кандидатов, если встречаются в нём сами по себе. На этом этапе используется стоп-лист системы *SMART* [17].

На этапе расчёта весов терминов-кандидатов для каждого из них запрашивается значение *информативности* из базы данных. Вторым необходимым для расчётов показателем является частота встречаемости термина *TF*. Для каждого найденного в БД термина его вес рассчитывается по формуле (2).

Общий принцип извлечения ключевых терминов заключается в анализе заданного числа сообщений и определении *порогового значения веса* для каждого из них. Те термины, веса которых *больше или равны* пороговому значению, считаются ключевыми.

Изначально пороговым считается *среднее арифметическое* значение веса для всех терминов-кандидатов. Все последующие операции призваны уточнить его и улучшить результаты работы алгоритма в целом.

Следующим этапом является обработка массива *хэштегов*, полученного на этапе препроцессинга. Предполагается, что с их помощью пользователь явно указывает термины, определяющие тематику сообщения. Поэтому логично предположить, что пороговое значение для всего сообщения не должно быть выше минимального веса среди его хэштегов. Основываясь на этом предположении, вес каждого из хэштегов (если он был найден в БД)

сравнивается с текущим пороговым значением и понижает его в случае, если оно больше.

Если после обработки хэштегов пороговое значение осталось равным 0 (в случае, когда они не были указаны или когда ни один из них не был найден в БД) либо превышает найденное среднее значение, оно принимается равным среднему. Такая ситуация на практике встречается чаще всего, так как пользователи редко явно указывают тематические термины. Однако в противном случае такой подход существенно улучшает результаты работы.

Нужно отметить, что сообщения обрабатываются в порядке, обратном поступлению с сервера, то есть в прямом хронологическом. Такой подход представляется логичным и учитывает специфику сервиса блогов в целом: пользователь может написать сообщение на какую-то тему, а затем вернуться к ней снова. Однако во втором сообщении, помимо тех терминов, которые были выбраны ключевыми в первом, могут быть другие, более информативные термины, за счёт которых пороговое значение для второго сообщения будет завышено, и ключевые термины из первого сообщения не будут выделены. Чтобы избежать этого, в системе имеется отдельный массив, содержащий все ранее извлечённые ключевые термины. Тогда при обработке очередного сообщения термины из этого массива будут безусловно извлечены и снизят пороговое значение веса для данного сообщения.

В этом контексте важно, что при непосредственном выборе ключевых терминов кандидаты обрабатываются в порядке *возрастания* их весов. Таким образом, если вес какого-либо из кандидатов ниже порогового, но он выбирается ключевым из-за того, что присутствует в массиве ранее извлечённых терминов, то пороговое значение становится равным его весу, и все следующие термины автоматически попадают в список ключевых.

Результатом работы алгоритма является список отсортированных в порядке *убывания* весов ключевых терминов.

## *5. Взаимодействие с Amazon API*

В качестве демонстрации возможного применения разработанного алгоритма было реализовано получение с сервера интернет-магазина *Amazon* описаний товаров, *релевантных* найденным ключевым терминам.

Для взаимодействия с Amazon REST API используется Perl-модуль *Net::Amazon*, который предоставляет удобный доступ к большинству функций этого программного интерфейса. При этом осуществляется поиск по всем товарам интернет-магазина, в названии или описании которых встречается искомый термин. Для необходимого количества наиболее подходящих товаров выводятся название, цена, год издания и изображение. Название товара представляет собой ссылку на его страницу на сайте Amazon.

При подключении к серверу используется *секретный ключ разработчика*, который предоставляется при регистрации в сервисе Amazon и позволяет

вести статистику переходов на страницы товаров с сайтов сторонних разработчиков. Если после перехода по ссылке пользователь приобретёт товар, то, согласно *партнёрской программе* Amazon, владелец сайта может получить некоторое вознаграждение, равное части стоимости товара. Подобная схема может быть реализована не только для Amazon, но также и для любого другого интернет-магазина, имеющего партнёрскую программу.

## 6. Результаты экспериментов

Результатом работы системы является HTML-страница, разбитая на блоки, каждый из которых соответствует одному сообщению. В блоке выводится текст оригинального сообщения с указанием его автора, затем – текст после препроцессинга и, наконец, тот же текст после обработки. В тексте сообщения на выходе работы системы найденные ключевые термины являются ссылками на соответствующие статьи Википедии.

Для всех найденных ключевых терминов строится таблица, каждая строка которой содержит термин, его вес и найденные релевантные товары из интернет-магазина. Ниже выводятся среднее и пороговое значения веса. Последней частью выходных данных является список терминов, которые были найдены в базе, но не были отнесены к ключевым.

Эффективность алгоритмов извлечения ключевых терминов обычно оценивается путём сравнения результатов их работы с ключевыми терминами, извлечёнными *вручную*. Критерии качества работы основаны на числе соответствий между фразами, выбранными алгоритмом и человеком [6].

Для тестирования работы системы было создано несколько тестовых аккаунтов, каждый из которых был «подписан» на обновления статусов различных известных в IT-сообществе пользователей Twitter. В качестве основного аккаунта для тестирования был выбран *semtweettest2*, который был «подписан» на обновления блога Tim O'Reilly (*timoreilly*), книгоиздателя и общественного деятеля, который имеет свыше 1 400 000 подписчиков. Сообщения в этом блоге отличаются чрезвычайно разнообразной тематикой, в них часто используются различные именованные сущности (имена людей, названия компаний и мероприятий, географические названия), которые представляют реальный интерес в настоящий момент. Кроме того, автор блога полностью использует возможности микросинтаксиса Twitter. Всё это в совокупности даёт основания полагать, что результаты работы разработанной системы на сообщениях блога *timoreilly* позволяют достоверно оценить эффективность алгоритма.

Для сравнения результатов работы алгоритма с существующими аналогами была выбрана система *Alchemy API* [18], которая предоставляет демонстрационный доступ к своим функциям в онлайн-режиме. В качестве исходных данных эта система использует текстовый документ, а возвращает этот же документ с выделенными ключевыми терминами.

Каждое из выбранных для тестирования сообщений было проанализировано с помощью реализованной системы и демонстрационной версии Alchemy API. Так как нет необходимости рассчитывать точность, полноту и F-меру для каждого из них, то весь массив сообщений был принят за один документ, из которого извлекались ключевые термины.

Всего из выбранных 50 сообщений вручную было выделено 180 ключевых терминов, 28 из которых являются частями других, более длинных составных терминов. Максимальная длина выделенного вручную термина равна 3 словам. Максимальная длина термина, выделенного системой, равна 6 словам.

Результаты тестирования приведены в табл. 2.

| **Метод** | **Точность, %** | **Полнота, %** | **F-мера, %** |
|---|---|---|---|
| Alchemy API | 18,9 | 43,6 | 26,4 |
| Разработанная система | 40,0 | 68,6 | 50,5 |

*Табл. 2. Результаты тестирования работы системы*

По результатам тестирования можно сделать вывод, что разработанная система достаточно эффективно функционирует в условиях поставленной задачи. Кроме того, по качеству результатов она превосходит выбранную для сравнения систему Alchemy API.

Одной из возможных причин снижения качества результатов является большое количество *именованных сущностей* в текстах обработанных сообщений, причём большинство из них указывают на людей, события или компании, ставшие популярными лишь недавно. Очевидно, что, так как БД терминов из Википедии соответствует её состоянию на июль 2009 года, то многие из актуальных в настоящий момент именных сущностей отсутствуют в ней (например: «*CityCamp*», «*CrisisCamps*»). Другой причиной служит частое использование *сокращений*, многие из которых не являются общепринятыми, например: «*webops*» для «*web operations*», «*gov't*» для «*government*», «*gov20*» и «*Gov 2.0*» для «*Government 2.0*».

## *7. Заключение*

В ходе выполнения работы был разработан алгоритм извлечения ключевых терминов из минимально структурированных текстов сообщений микроблогов. Проведённое экспериментальное исследование показало, что алгоритм работает корректно и эффективно. В качестве примера возможного практического использования результатов в рамках разработанной системы реализован прототип системы контекстной рекламы с отображением описаний релевантных товаров из интернет-магазина.

В продолжение начатой работы планируется создание сервиса, основанного на механизме аутентификации *OAuth*, который используется в большинстве современных приложений к Twitter. Пользователь при этом предоставляет доступ к данным своего аккаунта, в том числе к текстам всех сообщений. Одной из возможных перспектив использования разработанного алгоритма является возможность указания пользователем нескольких ключевых терминов с целью просмотра только тех сообщений, которые их содержат. Возможна фильтрация не только по извлечённым с помощью Википедии терминам, но и путём простого поиска по текстам сообщений после препроцессинга. Такой сервис был бы востребованным [19] и позволял бы привлечь целевую аудиторию для показа контекстной рекламы.

**Литература**


[1] Martin Ebner. *Microblogging - more than fun?* - Proceedings of IADIS Mobile Learning Conference 2008, Inmaculada Arnedillo Sánchez and Pedro Isaías ed., Portugal, 2008, pp. 155-159.

[2] Herman David, Janh Manfred, Ryan Marie-Laure. (éd.), *The Routledge Encyclopedia of Narrative Theory*. London, Routledge, 2005.

[3] Böhringer, M. *Really Social Syndication: A Conceptual View on Microblogging*. - Sprouts: Working Papers on Information Systems, 9(31), 2009.

[4] D.R. Karger, D. Quan (2005). *What would it mean to blog on the semantic web?* - Web Semantics: Science, Services and Agents on the World Wide Web, Selected Papers from the International Semantic Web Conference, Hiroshima, Japan, 07-11 November 2004, 3 (2-3), 2005, 147-157.

[5] *Gartner Highlights 27 Technologies in the 2008 Hype Cycle for Emerging Technologies*. - http://www.gartner.com/it/page.jsp?id=739613, 2008.

[6] P. Turney. *Learning to extract keyphrases from text*. Technical report, National Research Council, Institute for Informational Technology, 1999.

[7] D. Turdakov. *Word sense disambiguation methods*. Programming and Computer Software, 2010, Vol. 36, No. 6, pp. 309-326.

[8] D. Turdakov, S. Kuznetsov. *Automatic word sense disambiguation based on document networks*. Programming and Computer Software, 2010, Vol. 36, No. 1, pp. 11–18.

[9] Dmitry Lizorkin, Pavel Velikhov, Maxim Grinev, Denis Turdakov. *Accuracy estimate and optimization techniques for SimRank computation*. - The International Journal on Very Large Data Bases archive. Volume 19 Issue 1, February 2010.

[10] Dmitry Lizorkin, Pavel Velikhov, Maxim Grinev, Denis Turdakov. *Accuracy Estimate and Optimization Techniques for SimRank Computation*. - Proceedings of the VLDB Endowment. Volume 1 Issue 1, August 2008.

[11] Maria Grineva, Maxim Grinev, Dmitry Lizorkin. *Effective Extraction of Thematically Grouped Key Terms From Text*. - Proc. of the AAAI 2009 Spring Symposium on Social Semantic Web. - pp. 39-44.

[12] D. Turdakov, D. Lizorkin. *HMM Expanded to Multiple Interleaved Chains as a Model for Word Sense Disambiguation*. - PACLIC 2009: The 23rd Pacific Asia Conference on Language, Information and Computations. - pp. 549-559.

[13] M. Grineva, M. Grinev, D. Lizorkin. *Extracting Key Terms From Noisy and Multitheme Documents*. - WWW2009: 18th International World Wide Web Conference.

[14] M. Grineva, M. Grinev, Alexander Boldakov, Leonid Novak, Andrey Syssoev, D. Lizorkin. *Sifting Micro-blogging Stream for Events of User Interest*. - Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval, 2009.
[15] Joel W. Reed, Yu Jiao, Thomas E. Potok, Brian A. Klump, Mark T. Elmore, Ali R. Hurson. *TF-ICF: A New Term Weighting Scheme for Clustering Dynamic Data Streams*. - Proc. Machine Learning and Applications, 2006, ICMLA '06, pp. 258-263.
[16] Mihalcea, R., and Csomai, A. 2007. *Wikify!: linking documents to encyclopedic knowledge*. - Proceedings of the sixteenth ACM conference on Conference on information and knowledge management, pp. 233-242. New York, NY, USA: ACM.
[17] Salton, G. (1971). *The SMART Retrieval System - experiments in automatic document processing*. Prentice-Hall, Inc., Englewood Cliffs, NJ.
[18] *Alchemy API – Demo.* - http://www.alchemyapi.com/api/demo.html.
[19] Zhao, Dejin and Mary Rosson. *How and why people Twitter: the role that micro-blogging plays in informal communication at work*. - Proceedings of the ACM 2009 international conference on Supporting group work, 2009.
[20] McFedries, P. *All A-Twitter*. IEEE Spectrum, October 2007, 84.
[21] Java, A., Song, X., Finin, T., Tseng, B. *Why we twitter: understanding microblogging usage and communities*. - Proc. WebKDD/SNA-KDD '07, ACM Press (2007).
[22] Krishnamurthy, B., Gill, P., and Arlitt, M. *A few chirps about twitter*. - Proc. WOSP '08. ACM Press (2008).
[23] Honeycutt, C., Herring, S. *Beyond microblogging: Conversation and collaboration via Twitter*. - Proc. HICSS '09. IEEE Press (2009).
[24] Naaman, M., Boase, J., Lai, C.-H. *Is it really about me? Message content in social awareness streams*. - Proc. CSCW 2010, February 6-10, 2010, Savannah, Georgia, USA.
[25] Huberman, B., Romero, D., Wu, F. *Social networks that matter: Twitter under the microscope*. First Monday [Online] 14, 1 (2008).

## Keyterm extraction from microblogs' messages using Wikipedia

*Anton V. Korshunov*
*korshunov@ispras.ru*

**Abstract.** The paper describes a method for keyterm extraction from messages of microblogs. The described approach utilizes the information obtained by analysis of Wikipedia structure and content. The algorithm is based on computation of “keyphraseness” for each term, i.e. an estimation of probability that it can be selected as a key in text. The experimental study demonstrated that the proposed technique performs significantly better comparing to analogues. As a demonstration of possible application, the prototype of context-sensitive advertising system has been implemented. This system is able to obtain the descriptions of goods relevant to found keyterms from Amazon online store. Several ways have been proposed also on how the information derived from Twitter messages may be utilized in different auxiliary services.

**Keywords:** Information retrieval; keyterm extraction; natural language processing; text mining; semantic analysis; microblogging; Twitter; Wikipedia; context-sensitive advertising.

# Использование префиксного дерева для хранения и поиска строк во внешней памяти

*Таранов И. С.*
*epsilon@ispras.ru*

**Аннотация.** Поиск среди больших объёмов текстовых данных, хотя и изучается в computer science давно, не теряет своей актуальности. В работе представлена структура данных для поиска и эффективного хранения во внешней памяти массивов текстовых строк, реализованная для поддержки индексов в XML СУБД Sedna. Описываются алгоритмы для вставки, удаления и поиска строк переменной длинны в префиксных деревьях, хранимых на дисках. Мы также сравниваем нашу реализацию с существующей реализацией B-дерева. В работе показано, что в некоторых случаях предложенная структура данных занимает в несколько раз меньше места во внешней памяти при той же скорости поиска.

**Ключевые слова:** словарные структуры данных; префиксные деревья; B-деревья; индексы в СУБД

## *1. Введение*

Проблемы реализации структур данных, реализующих основные словарные операции (добавление, удаление и поиск элемента), хотя и изучаются в computer science очень давно, тем не менее, не теряют своей актуальности. В данной работе описана структура данных, предназначенная для хранения множества строковых ключей во внешней памяти, приведено её сравнение с B-деревьями, и описаны основные алгоритмы работы с ней. Предложенная структура данных в некоторых случаях позволяет значительно сократить объём индекса во внешней памяти.

Задача разработки специальной структуры данных встала в ходе работы над XML-СУБД Sedna [1, 2]. Данная XML-СУБД поддерживает индексы по значению узлов над XML-документом. В результате к структуре данных, используемой для поддержки индексов, предъявляются следующие требования:

1. Возможность хранить ключи произвольно большой длины. Необходимость этого связана с подходом к хранению XML, где все

значения при отсутствии предписывающей схемы являются строками, при таком хранении нельзя заранее ограничить длину строки. Одним из примеров длинных строк являются URI, поиск по которым зачастую необходим, но которые при этом могут быть достаточно длинными.

2. Возможность хранить мультимножество пар "ключ/значение". В произвольном XML-документе при произвольном индексе могут существовать одинаковые ключи с разным значением и одинаковые значения с одинаковыми ключами. Причём при обновлении документа (например, удалении узлов) может быть удалена только одна из одинаковых пар "ключ/значение".

3. Эффективный поиск пары "ключ/значение" необходим для быстрого удаления пары из индекса при обновлении документа.

## 2. Обзор существующих работ

Стандартной структурой данных для создания такого индекса является *B-дерево* [3] и его варианты [4, 5, 6] [1]. B-деревья очень широко применяются в базах данных для задач индексации [8]. Хранение ключей произвольной длины в B-деревьях хотя и возможно, однако представляет ряд проблем. Во-первых, достаточно сложно реализовать эффективное хранение ключей переменной длины. На практике обычно в узле дерева хранят только ограниченную часть ключа, а остаток хранят в отдельных *страницах переполнения (overflow pages)* [9, 10]. Такой подход достаточно эффективен в тех случаях, когда ключи короткие или различаются в первых символах. Во-вторых, в случаях, когда одному ключу соответствует множество различных значений, сложно реализовать достаточно эффективный поиск по паре "ключ/значение". Часто B-деревья не предусматривают хранения мультимножества ключей. Для поставленной же задачи характерно наличие дубликатов пар "ключ/значение". Для возможности хранения одинаковых логических ключей в качестве физического ключа используется конкатенация пары "ключ/значение" [11][2].

В статье предложен подход к организации индексов над строками во внешней памяти, который удовлетворяет всем вышеописанным требованиям. Его

[1] Во всей статье под B-деревьями на самом деле подразумеваются $B^+$ деревья [7].

[2] Здесь мы будем понимаем под физическим ключом тот ключ, который действительно используется при сравнении и поиске в описываемой структуре данных. Под логическим ключом понимается ключ, который передаётся интерфейсу работы с данной структурой данных.

можно рассматривать как расширение структуры данных, известной как trie [12], которую в дальнейшем будем называть *префиксное дерево*[3].

Идея использования данной структуры для реализации индексов не нова. В работе [14] предложена структура данных S(b)-tree, которая представляет собой разновидность B-дерева, в узлах которого находится бинарная *"Патриция" (patricia tree)* [15]. Особенностью S(b)-tree является то, что в узлах хранятся не ключи как таковые, а количество пропускаемых при сравнении бит. В процессе поиска искомую строку, возможно, придётся сравнивать со строкой, хранящейся во внешней памяти. Однако само по себе такое сравнение не представляет больших проблем. Для поставленной задачи проблемой было бы хранить все строки-ключи в отдельном месте. S(b)-tree позиционируется как структура данных для поддержки полнотекстового индекса и достаточно хорошо справляется с этой задачей [16].

Наиболее похожей на описанную в работе структуру данных является предложенная в работе B-trie [17]. За основу в этой работе была взята реализация префиксного дерева от тех же разработчиков, которая предусматривает эффективное использование процессорного кеша [18]. В обеих структурах предложено эффективное для этого разбиение префиксного дерева на блоки (buckets).

Кроме того, существуют совсем простые реализации подобного подхода, такие как Index Fabric [19], который является тем же B-деревом, в узлах которого хранение и поиск ключей осуществляется с помощью префиксного дерева.

## 3. Описание предложенной структуры

В статье предложена структура данных для хранения множества строк *K*, которую в дальнейшем мы будем называть **BST** (**B**lock **S**tring **T**rie), и над которой определены следующие (словарные) операции:

- *insert*(*s*) - добавить строку *s* в множество *K*;
- *delete*(*s*) - удалить строку *s* из множества *K*;
- *find*(*s*) - найти все строки в *K* с префиксом *s*, включая саму строку *s*.

Данная структура данных, как видно, сама по себе не предусматривает хранения пар "ключ/значение". Учитывая наши требования, мы будем хранить значения, как части физически сохраняемого ключа. Рассмотрим пример. Пусть надо сохранить пару $(k, v)$. Для этого мы строим строковый ключ $k' = k + c + string(v)$, где под $c$ понимается символ, которого нет в алфавите символов логических ключей (так, для текстовых строк можно

[3] В русском переводе [13] для перевода термина trie используется слово "луч". Также в литературе встречается слово "бор".

использовать нулевой символ), а $string(v)$ — любое строковое представление значения. Если надо найти все значения, соответствующие некоторому ключу $k$, необходимо вызвать $find(k+c)$.

В нашей структуре данных мы разделяем несколько уровней объединения вершин дерева. Внутренние вершины дерева объединяются в *ветки*, а ветки целиком хранятся в страницах внешней памяти. Ниже мы опишем последовательно эти уровни.

## 3.1. Префиксное дерево

Описываемая структура данных представляет собой *корневое дерево* $T$, хранящее множество ключей $K$ и устроенное следующим образом:

1. Каждая вершина $x$ содержит следующие поля: префикс $prefix(x)$ (возможно, пустой), массив $E(x)$ из $n(x)$ исходящих из неё рёбер, помеченных *различными* символами (упорядоченный в лексикографическом порядке), а также некоторые флаги (булевские переменные), например $final(x)$, который определяет, соответствует ли этой вершине какой-либо ключ. Флаги вершины будут описываться по мере их введения.

2. Ребра $e=(x,L_i(x))$ (будем обозначать $L_i(x)$ вершину, в которую ведёт $i$-е ребро из массива $E(x)$) помечены символами $c(e)$. В данном случае мы не отличаем отдельные символы от строк и будем считать их строками единичной длины.

3. Любой путь $S(x_1,x_n)=x_1e_1x_2e_2\ldots e_{n-1}x_n$ в дереве задаёт строку $s$, которая получается из этого пути $S$ конкатенацией строк $s=prefix(x_1)+c(e_1)+prefix(x_2)+c(e_2)+\ldots+c(e_{n-1})+prefix(x_n)$ Строка $s$, заданная путём $S$, принадлежит хранимому в дереве множеству ключей $K$ в том и только в том случае, если конечная вершина $x_n$ пути $S$ помечена флагом $final(x_n)$.

Отметим, что в общем случае некоторому множеству ключей $K$ может соответствовать более одного дерева $T$. Это объясняется тем, что в дереве, в котором есть хотя бы одна вершина, из которой исходят не все возможные рёбра, можно добавить вершину $x'$ с произвольным префиксом, не помеченную $final(x')$. Полученное дерево будет задавать то же самое множество ключей $K$. Поэтому введём дополнительное свойство,

выполнения которого требовать мы не будем, позволяя реализовать тем самым *отложенное удаление* (которое будет обсуждаться ниже):

1. Любая вершина $x \in T$, для которой $n(x) \le 1$, помечена как $final(x)$. Т.е. (единственному) пути, ведущему от корня к такой вершине, соответствует ключ множества $K$.
2. Заметим, что при выполнении последнего свойства множество $K$ однозначно задаёт дерево $T$. Доказательство этого факта не представляет сложности, однако выходит за рамки данной статьи.

Дерево $T$, для которого выполняется последнее свойство, будем называть *минимальным*. Кроме того, вершину $x$, для которой выполняется (не выполняется) условие $n(x) \le 1 \Rightarrow final(x)$, будем называть *неизбыточной* (*избыточной*). В дальнейшем (если не оговорено обратного) будем рассматривать только минимальные деревья.

Заметим также, что для хранения мультимножества достаточно ввести дополнительный параметр вершины $count(x) \ge 1$, определённый только для вершин, для которых установлен флаг $final(x)$.

## 3.2. Разделение на страницы

Описанная в предыдущем разделе структура удобна для хранения и поиска строковых ключей в оперативной памяти. Но для использования её с большими объёмами данных во внешней памяти необходимо эффективно распределять её на страницы фиксированной длины. Заметим, что это важно не только для структур данных, предназначенных непосредственно для хранения во внешней памяти (например для поддержания индексов баз данных), но также для структур данных в оперативной памяти [13].

Для такого разделения выделим особый тип *ссылочных вершин*, которые не хранят префиксов и ссылок на другие вершины, а хранят лишь ссылку на другой узел, находящийся в другой странице. Такие вершины пометим флагом $external(x)$, каждая из которых ссылается на некоторую вершину $y = J(x)$ такую, что $external(x) \Rightarrow not\ external(J(x))$. Введём также понятие ветки. Назовём *веткой корневое дерево* $B$ такое, что *конечные вершины всех путей от корня к листовым вершинам* в нём помечены либо $final(x)$, либо $external(x)$. Если в дереве $T$ нет ссылочных вершин, то оно состоит из одной ветки.

Заметим, что любое дерево $T$ может произвольным образом быть разбито на ветки путём вставки перед любой вершиной новой ссылочной вершины. Ссылочная вершина разбивает ветку на две. Если ветки считать узлами, то они

также образуют дерево $T_b$. Отметим также, что ссылочные вершины никак не влияют на минимальность дерева *по определению* и не являются избыточными.

*Страницы* могут хранить одинаковый объём данных $W$ байт[4]. В странице могут храниться одна или несколько веток исходного дерева, причём в одной странице могут храниться только ветки с общим прямым предком $P(B)$. Последнее условие необходимо, во-первых, для обеспечения локальности изменений в дереве [20], а, во-вторых, для обеспечения эффективных блокировок на уровне страниц. Из этого также следует, что на странице, в которой хранится корневая ветка, других веток быть не может.

## 4. Алгоритмы

Алгоритмы изменения дерева должны обеспечивать, во-первых, локальность изменений (ограниченное число изменяемых страниц), во-вторых, компромисс между оптимальным заполнением страниц и высотой дерева.

### 4.1. Поиск пути

Алгоритм BST-Search(r, k) возвращает вершину $x_{n+1}$ такую, что:

- строка $s(x_n)$, задаваемая путём $S(r, x_n)$, если такая существует, является префиксом искомой либо совпадает с ней,
- искомая строка $k$ является префиксом строки $s(x_{n+1})$ либо совпадает с ней.

Т.е. выполняется неравенство: $s(x_n) \le k \le s(x_{n+1})$, где под операцией $A \le B$ понимается то, что $A$ является префиксом $B$ или совпадает с ней. Процедура получает на вход указатель $x$ на корневой узел поддерева, а также строку $k$. Промежуточные результаты поиска сохраняются в стек $S$. В процедуре также используется функция $y = L(x, c)$, которая находит среди исходящих рёбер вершины $x$ ребро, помеченное символом $c$, и возвращает узел $y$, в который оно ведёт, либо $NIL$, если такого ребра нет. Такую функцию можно реализовать, например, бинарным поиском, т.к. массив рёбер упорядочен. Также используется функция $Cut(p, s)$, которая возвращает строку, получающуюся из $s$ удалением префикса $p$.

Алгоритм достаточно компактен, поэтому приведём его здесь:

[4] В СУБД Sedna, для которой реализована описываемая структура, размер страницы по умолчанию равен 64K.

```
1.  BST-Search( x , k , S )
2.  Push( S , x )
3.  if external(x) then
4.      Disk-Read( J(x) )
5.      return BST-Search( J(x) , k , S )
6.  endif
7.  if not Is-Prefix( prefix(x) , k )) then
8.      if Is-Prefix( k , prefix(x) ) then
9.          return x
10.     else
11.         return NIL
12.     endif
13. else
14.     s ← Cut( prefix(x) , k )
15.     if Empty( s ) then
16.         return x
17.     elseif L(x, s[1]) = NIL
18.         return NIL
19.     else
20.         return BST-Search( L(x, s[1]) , Cut( s[1] , s ), S )
21.     endif
22. endif
```

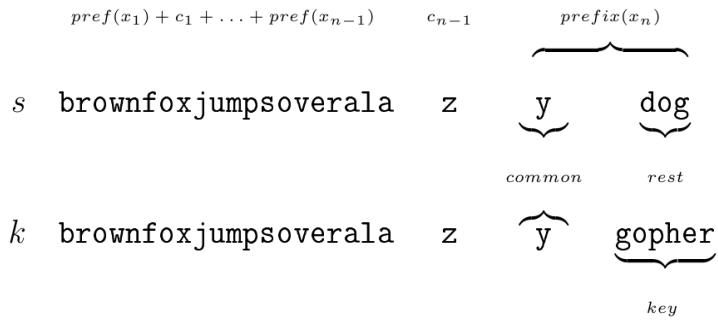
Процедура возвращает такой узел $x$, что все пути, ведущие из корня в $final$-вершины и содержащие $x$, задают строки, для которых искомая строка является префиксом, либо $NIL$, если такого узла нет. Таким образом, нам остаётся найти все эти пути (обходом поддерева от возвращённой вершины).

## 4.2. Вставка

Процедура добавления строки устроена таким образом, что она сначала строит структуру, описывающую изменение страницы, в которую надо вставить новые вершины. Может случиться, что для вставки нет места в странице. В этом случае процедура вызывает расширение дерева на необходимую величину, и вставка не происходит. Она должна быть вызвана ещё раз. Заметим также, что корень дерева может измениться в процессе работы процедуры.

Сама процедура добавления достаточно проста, однако она слишком велика, чтобы приводить её здесь. Опишем только основной подход.

Начинается вставка с поиска пути по дереву к ключу $k$, который надо вставить. Алгоритм практически идентичен алгоритму поиска пути, только нас интересует лишь путь $S$, получающийся в результате. Кроме того, нам понадобятся три дополнительные строки, получаемые из найденного пути и строки $k$: *common*, *rest* и *key*. Строятся они следующим образом. Возьмём строку $s$, которая получилась из строки, соответствующей найденному пути $S$ удалением последнего префикса. Она, очевидно, является префиксом добавляемой строки $k$ (либо совпадает с ней). Будем рассматривать строку $k$, которая получилась из $k$ удалением этого префикса $s$. Тогда *common* — это наибольший общий префикс строк $k$ и $p = prefix(S[n])$ (где $S[n]$ — последняя вершина пути $S$). *rest* — это остаток от строки $p$, полученный удалением префикса *common*, и *key* — остаток от $k$, полученный удалением префикса *common*. Все три строки удобно считать одновременно, и их значение иллюстрирует следующая схема:




Алгоритм вставки рассматривает пять случаев:

1. В дереве нет узлов. В этом случае нам надо выделить страницу, на которой расположить единственный новый узел дерева.
2. *rest* и *key* — пустые строки. В этом случае нам достаточно пометить последнюю вершину пути как *final* (в случае минимального дерева она уже будет помечена как *final*).
3. *key* — пустая строка, *rest* — непустая. В этом случае нам достаточно разбить последнюю вершину пути на две, одна их которых будет содержать префикс *common* и будет помечена как *final*.
4. *key* — непустая строка, *rest* — пустая. В этом случае нам надо добавить дополнительную вершину с префиксом *key*, дочернюю по отношению к последней вершине пути.

5. *key* и *rest* — суть непустые строки. В этом случае последняя вершина $x_n$ разбивается на три: одну с префиксом *common*, с двумя исходящими из неё — $x_n$, префиксом которой становится *rest*, и final-вершину с префиксом *key*.

Основной момент, который связан с разделением на блоки, заключается в том, что в случае 4 и 5, если ветка, в которую мы производим добавление, имеет дочерние ветки, мы создаём новую вершину с *key* в новой ветке. Для этого мы находим среди дочерних веток ту, на странице которой осталось больше всего места. Это самая дорогая из всех приведённых операций, т.к. требует в худшем случае чтение такого количества страниц, которое равно количеству различных возможных дочерних веток. На практике это неприемлемо. Поэтому в нашей реализации мы ограничиваем количество просматриваемых страниц некоторой константой *D* (в реализации равна 2). В случае, если среди первых просмотренных *D* страниц не нашлось той, в которую помещается добавляемая вершина, пытаемся расширить самую занятую из просмотренных. При таком подходе затрагивается не более *D*+2 страниц.

## 4.3. Разделение страниц

В отличие от B-деревьев, предложенные нами BST-деревья не сбалансированы. Заметим, что так как минимальное дерево $T$ полностью определяется множеством ключей $K$, которые оно хранит, мы не можем никак повлиять на его высоту. Под высотой BST-дерева подразумевается высота дерева его веток, т.к. именно этим определяется количество блоков, которые необходимо прочитать, чтобы найти вершину в худшем случае. Кроме того, нашей задачей является избежать большого числа “полупустых” блоков.

Процедура выделения места в странице подразумевает разделение страницы и вызывается только в том случае, если на странице не хватает места для вставки нового узла. Мы используем два очень простых алгоритма разделения страниц.

Первый из них вызывается в случае, если на странице расположено несколько веток. В этом случае мы разделяем ветки страницы на два непересекающихся набора $P_1$ и $P_2$, такие, что $|\sum_{w\in P_1} w(B) - \sum_{w\in P_2} w(B)|$ минимальна по всем возможным разделениям $P_1$ и $P_2$. Т.е. эти наборы должны разделять ветки примерно пополам по их общему размеру. Это можно сделать, например, с помощью жадного алгоритма. Каждый из этих наборов записывается в отдельную страницу, а ссылки обновляются. Данная операция затрагивает ровно три блока.

Второй алгоритм вызывается в случае, если на странице, в которую происходит добавление, осталась только одна ветка. В этом случае мы

выделяем корень $P_p$ этой ветки следующим образом. Находим первую вершину от корня, которая содержит $N > 1$ ссылок. У этой вершины есть $N$ дочерних вершин, которые будут корнями $N$ новых веток, оставшихся на данной странице. Корневое множество вершин $P_p$ мы удаляем из данной страницы и целиком переносим в ветку-предок. Чтобы гарантировать возможную вставку в исходную страницу, применяем к ней первый алгоритм разделения. Если исходная ветка являлась корневой, то множество вершин $P_p$ помещается в новую страницу. Может оказаться, что в странице, хранящей ветку-предок, не хватит места для вставки $P_p$. В этом случае вызывается процедура выделения свободного места для страницы предка. Данная процедура затрагивает ровно три страницы без учёта возможного рекурсивного вызова.

Проблемой могут оказаться строки, по размеру превышающие $h$ страниц, где $h$ — высота дерева в страницах. В этом случае мы можем не найти в ветке вершину, которую можно “перенести” в верхнюю ветку. Тогда, в нашей реализации, строка разбивается на две (возможно нарушая минимальность дерева), и листовую часть строки представляется правильным вынести в отдельную страницу. Это практически является аналогом страниц переполнения для B-деревьев, но не будет приводить к постоянному чтению этой страницы, т.к. любая исходная строка сравнивается в нашем дереве не более одного раза.

## 4.4. Удаление

Следуя примеру большинства работ по B-деревьям, мы вводим процедуру отложенного удаления [20, 21, 22]. Данный подход широко используется в базах данных, т.к. процедура удаления для B-деревьев может быть сложнее процедуры вставки. Отложенное (lazy) удаление позволяет, во-первых, значительно упростить само удаление; во-вторых, ускорить обновления базы данных. Удаление заключается в простом снятии флага *final* с найденного узла. После удаления узла таким образом мы получаем *избыточный* узел, и, следовательно, неминимальное дерево.

При таком подходе необходима процедура минимизации дерева (или, возможно, отдельной ветки). Она заключается в удалении всех избыточных вершин. Избыточные вершины бывают двух типов:

1. Вершины, которые не помечены как *final*, и у которых нет исходящих рёбер. Такие вершины можно просто удалить.

2. Вершины, которые не помечены как *final* из которых есть ровно одно исходящее ребро. В этом случае вершины достаточно объединить конкатенацией префиксов через символ ребра.

В результате применения этой операции не остаётся избыточных вершин. В результате такой операции может получится так, что в некоторой ветке не останется вершин, либо останется единственная *external*-вершина. Такую ветку необходимо удалить, а (единственную) дочернюю ветку перенести вверх на данную страницу. Может оказаться так, что страницу необходимо будет для этого разделить. Таким образом минимизация одной ветки затрагивает не более четырёх страниц (три страницы может затронуть разделение, и с одной страницы мы переносим ветку)

## 5. Эксперименты

В качестве набора данных для тестирования мы использовали базу данных DBLP [23]. Тестирование производилось с использованием СУБД Sedna. При тестировании индексировались публикации по идентификатору (ID) и по двум типам URI-ссылок (EE и URL), которые есть в базе DBLP. Всего индексировалось 861473 публикации. Результаты тестирования показаны в следующей таблице:

| Набор данных | Объем данных | Время поиска (сек.) | |
|---|---|---|---|
| | | Большой буфер | Маленький буфер |
| BST | | | |
| DBLP ID | 27Mb | 6.0 | 6.2 |
| DBLP EE | 17Mb | 15.9 | 16.0 |
| DBLP URL | 31Mb | 15.7 | 17.0 |
| B-дерево | | | |
| DBLP ID | 31Mb | 6.0 | 6.0 |
| DBLP EE | 44Mb | 16.0 | 16.0 |
| DBLP URL | 46Mb | 16.0 | 17.0 |

Каждая серия поисковых запросов выполнялась в двух условиях: при большом буфере оперативной памяти (большая часть дерева помещалась в памяти) и при маленьком буфере (в памяти помещались всего 32 страницы). В тестах производился поиск 10% всех возможных значений каждого набора в случайном порядке. Как видно, предложенные деревья BST показывают практически одинаковую производительность по сравнению с B-деревьями, однако занимают в некоторых случаях гораздо меньше места за счёт сжатия ссылок. Кроме того, из результатов можно сделать вывод о том, что количество сравнений строк не оказывает существенного влияния на производительность. За счёт того, что они занимают меньше места, BST-деревья медленнее растут в глубину, что может в некоторых случаях серъёзно повлиять на производительность. Однако для того, чтобы это продемонстрировать, нужно генерировать большие искусственные наборы данных.

Кроме того, было произведено сравнение скорости вставки в B-деревья и BST-деревья. Различие скорости вставки было также практически неразличимо для двух типов деревьев, поэтому мы не приводим здесь его результатов.

В работе [17] очень похожая структура данных сравнивается с различными реализациями B-деревьев (их префиксным вариантом и Berkeley B-Tree). Описанное в работе B-дерево принципиально не отличается от B-деревьев, используемых в СУБД Sedna (со всеми его особенностями). Результаты этой работы также показывают, что префиксные деревья во внешней памяти и B-деревьях отличаются по скорости поиска незначительно. Кроме того, как и отмечалось в данной работе, видно, что скорость поиска в хранимых вариантах префиксных деревьев в большей степени зависит от размера буфера оперативной памяти. Основным преимуществом нашей структуры данных по сравнению с предложенной в работе [17] является то, что нам удалось достичь гораздо более значительного сжатия похожих ключей за счёт хранения общих префиксов. В итоге это может означать более медленный рост дерева в высоту.

## 6. Заключение

Структура данных BST была реализована в качестве возможной структуры данных для индексов в СУБД Sedna. Существует ещё много возможностей её оптимизации (в основном, алгоритмов разделения), которые необходимо дополнительно рассмотреть в будущем. В настоящее время проводится достаточно много работ, связанных с использованием префиксных деревьев.

Различные эксперименты с разными наборами данных подтверждают, что структура данных BST показывает производительность не хуже, чем у B-деревьев. Поэтому она может использоваться как полная их замена для строковых ключей. При этом, в случае, если строки достаточно длинные и по своей природе могут иметь длинные общие префиксы, можно достичь значительного сжатия индекса. Например, значительного выигрыша можно достичь при хранении индекса по нумерующим числам в XML-документах [24] и быстрого поиска по ним, например для поддержки блокировок [25].

Однако стоит отметить, что реализация BST достаточно сложна, и при выборе структуры данных для каких-то задач в общем случае есть смысл использовать хорошо проверенные и более надёжные реализации B-деревьев.

**Литература**


[1] Andrey Fomichev and Maxim Grinev and Sergei D. Kuznetsov. Sedna: A Native XML DBMS. SOFSEM, pages 272-281, 2006.

[2] Ilya Taranov et al. Sedna: native XML database management system (internals overview). *SIGMOD Conference*, pages 1037-1046, 2010.

[3] Rudolf Bayer and Edward M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Inf.*, 1:173-189, 1972.

[4] Rudolf Bayer and Karl Unterauer. Prefix B-Trees. *ACM Trans. Database Syst.*, 2(1):11-26, 1977.

[5] Nikolaus Walczuch and Herbert Hoeger. Using individual prefixes in $B^+$-trees. *Journal of Systems and Software*, 47(1):45-51, 1999.

[6] Ratko Orlandic and Hosam M. Mahmoud. Storage Overhead of O-Trees, B-Trees and Prefix B-Trees: A Comparative Analysis. *Int. J. Found. Comput. Sci.*, 7(3):209-226, 1996.

[7] Douglas Comer. The Ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121-137, 1979.

[8] Eugene Inseok Chong et al. A mapping mechanism to support bitmap index and other auxiliary structures on tables stored as primary $B^+$-trees. *SIGMOD Record*, 32(2):78-88, 2003.

[9] Ricardo A. Baeza-Yates. An Adaptive Overflow Technique for B-trees. *EDBT*, pages 16-28, 1990.

[10] B. Srinivasan. An Adaptive Overflow Technique to Defer Splitting in B-Trees. *Comput. J.*, 34(5):397-405, 1991.

[11] SQLLite File Format: B-Tree Structures. http://www.sqlite.org/fileformat.html#btree_structures

[12] Walter A. Burkhard. Hashing and Trie Algorithms for Partial Match Retrieval. *ACM Trans. Database Syst.*, 1(2):175-187, 1976.

[13] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[14] Paolo Ferragina and Roberto Grossi. The String B-tree: A New Data Structure for String Search in External Memory and Its Applications. *J. ACM*, 46(2):236-280, 1999.

[15] Wojciech Szpankowski. Patricia Tries Again Revisited. *J. ACM*, 37(4):691-711, 1990.

[16] Joong Chae Na and Kunsoo Park. Simple Implementation of String B-Trees. *SPIRE*, pages 214-215, 2004.

[17] Nikolas Askitis and Justin Zobel. B-tries for disk-based string management. *VLDB J.*, 18(1):157-179, 2009.

[18] Steffen Heinz and Justin Zobel and Hugh E. Williams. Burst tries: a fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, 20(2):192-223, 2002.

[19] Neal Sample and Brian F. Cooper and Michael J. Franklin and Gsli R. Hjaltason and Moshe Shadmon and Levy Cohe. Managing Complex and Varied Data with the IndexFabric(tm). *ICDE*, pages 492-493, 2002.

[20] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[21] Theodore Johnson and Dennis Shasha. B-Trees with Inserts and Deletes: Why Free-at-Empty Is Better Than Merge-at-Half. *J. Comput. Syst. Sci.*, 47(1):45-76, 1993.

[22] Garcia-Molina, Hector and Ullman, Jeffrey D. and Widom, Jennifer. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[23] Michael Ley. Die Trierer Informatik-Bibliographie DBLP. *GI Jahrestagung*, pages 257-266, 1997.

[24] N.A. Aznauryan, S.D. Kuznetsov, L.G. Novak, and M.N. Grinev. SLS: A numbering scheme for large XML documents, *Programming and Computer Software*, vol. 32, Jan. 2006, pp. 8-18.

[25] Peter Pleshachkov and Petr Chardin and Sergei D. Kuznetsov. A DataGuide-Based Concurrency Control Protocol for Cooperation on XML Data. *ADBIS 2005*, Tallinn, Estonia, September 12-15, 2005, pp. 268-282

# Using prefix trees for searching text strings with disk-based storage

*Taranov I.*
*epsilon@ispras.ru*

**Abstract.** Searching and storing large amounts of text data is a common problem in computer science, though it have been discussed for a long time. We introduce data structure for searching a set of string values and storing it on disk efficiently which is used for indexing XML data in Sedna XML DBMS. This paper describes algorithms for insertion, deletion and searching of variable-length strings in disk-resident trie structures. We also compare our implementation with existent B-tree implementation and show that proposed data structure in some cases occupies several times less disk space than B-tree does with the same search efficiency.

**Keywords:** dictionary data structures; tries; B-trees; DBMS indicies