ИСП Институт Системного Программирования Российской Академии наук

ISSN 2079-8156 (Print)
ISSN 2220-6426 (Online)

Труды Института Системного Программирования РАН

Proceedings of the Institute for System Programming of the RAS

Том 27, выпуск 1

Volume 27, issue 1

Москва 2015

ISSN 2079-8156 (Print), ISSN 2220-6426 (Online)

Труды Института системного программирования РАН

Proceedings of the Institute for System Programming of the RAS

Труды ИСП РАН – это издание с двойной анонимной системой рецензирования, публикующее научные статьи, относящиеся ко всем областям системного программирования, технологий программирования и вычислительной техники. Целью издания является формирование научно-информационной среды в этих областях путем публикации высококачественных статей в открытом доступе.

Издание предназначено для исследователей, студентов и аспирантов, а также практиков. Оно охватывает широкий спектр тем, включая, в частности, следующие:

- операционные системы;
- компиляторные технологии;
- базы данных и информационные системы;
- параллельные и распределенные системы;
- автоматизированная разработка программ;
- верификация, валидация и тестирование;
- статический и динамический анализ;
- защита и обеспечение безопасности ПО;
- компьютерные алгоритмы;
- искусственный интеллект.

Журнал издается по одному тому в год, шесть выпусков в каждом томе.

Поддерживается открытый доступ к содержанию издания, обеспечивая доступность результатов исследований для общественности и поддерживая глобальный обмен знаниями.

Труды ИСП РАН реферируются и/или индексируются в:

Proceedings of ISP RAS are a double-blind peer-reviewed journal publishing scientific articles in the areas of system programming, software engineering, and computer science. The journal's goal is to develop a respected network of knowledge in the mentioned above areas by publishing high quality articles on open access.

The journal is intended for researchers, students, and practitioners. It covers a wide variety of topics including (but not limited to):

- Operating Systems.
- Compiler Technology.
- Databases and Information Systems.
- Parallel and Distributed Systems.
- Software Engineering.
- Software Modeling and Design Tools.
- Verification, Validation, and Testing.
- Static and Dynamic Analysis.
- Software Safety and Security.
- Computer Algorithms.
- Artificial Intelligence.

The journal is published one volume per year, six issues in each volume.

Open access to the journal content allows to provide public access to the research results and to support global exchange of knowledge.

Proceedings of ISP RAS is abstracted and/or indexed in:

УДК004.45

Адрес: 109004, г. Москва, ул. А. Солженицына, дом 25.
Телефон: +7(495) 912-44-25
E-mail: info-isp@ispras.ru
Сайт: http://www.ispras.ru/proceedings/

Address: 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
Tel: +7(495) 912-44-25
E-mail: info-isp@ispras.ru
Web: http://www.ispras.ru/en/proceedings/

Содержание

Table of Contents

Применение статической бинарной инструментации с целью проведения динамического анализа программ для платформы ARM

М.К.Ермаков <mermakov@ispras.ru>
С.П.Вартанов <svartanov@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. В настоящее время динамический анализ программ активно используется для контроля качества программных продуктов, задач профилирования и поиска уязвимостей. В данной работе рассматриваются особенности одного из методов обработки кода программы в рамках динамического анализа — статической инструментации исполняемого кода, подразумевающей предварительное изменение исполняемых файлов или файлов динамических библиотек. Предлагается метод статической инструментации, позволяющий обрабатывать файлы исполняемого кода в формате ELF для архитектуры ARM.

Предлагаемый метод включает возможности настройки инструментации с помощью пользовательских спецификаций, задающих точки внедрения кода и необходимый внедряемый код. В статье описываются основные шаги метода: обработка пользовательских спецификаций, разбор кода программы и создание блоков инструментационного кода по числу точек инструментации, внедрение блоков инструментационного кода в программу, модификация кода программы с целью осуществления передачи управления на инструментационный код во время выполнения программы.

Модификация файлов исполняемого кода программы проводится в рамках ограничений, накладываемых спецификациями формата ARM ELF — распределение кода, данных и управляющей информации по секциям и сегментам, обрабатываемым динамическим загрузчиком операционной системы, и дополнительными внешними и внутренними зависимостями между секциями, порождаемыми во время генерации исполняемого кода. Непосредственный метод инструментации включает замену блоков инструкций в точках инструментации на инструкции безусловного перехода на соответствующий блок инструментационного кода и перенос заменённых инструкций в блок инструментационного кода для сохранения исходной функциональности программы. Для поддержания корректности работы в блок инструментационного кода также добавляются инструкции сохранения и восстановления состояния и инструкция безусловного перехода для возврата управления после выполнения инструментационного кода. Дополнительно в статье описываются модификации, направленные на добавление информации о внешних символах, используемых в инструментационном коде.

Данные модификации необходимы для поддержания корректности работы динамического загрузчика на этапе разрешения внешних зависимостей.
В статье приведены результаты практических экспериментов по применению разработанной программной системы, реализующей предлагаемый метод. На примере задачи подсчёта базовых блоков разработанная система показала более высокую производительность по сравнению с распространённым средством инструментации Valgrind.

Ключевые слова: статическая бинарная инструментация; динамический анализ; архитектура ARM; формат ELF; Android.

DOI: 10.15514/ISPRAS-2015-27(1)-1

Для цитирования: Ермаков М.К., Вартанов С.П. Применение статической бинарной инструментации с целью проведения динамического анализа программ для платформы ARM. Труды ИСП РАН, том 27, вып. 1, 2015 г., стр. 5-24. DOI: 10.15514/ISPRAS-2015-27(1)-1.

1. Введение

1.1 Автоматический анализ программного обеспечения

В настоящее время возрастание сложности программного обеспечения и повышение степени строгости требований к качеству программного обеспечения приводят ко всё более активному использованию средств автоматического анализа в рамках цикла разработки. Средства статического анализа, обычно интегрированные в среды разработки, позволяют обнаруживать программные дефекты, а также фрагменты, не соответствующие заданным спецификациям и стандартам, непосредственно во время создания программного кода. Средства динамического анализа позволяют проводить оценку различных аспектов программы, таких как производительность, эффективность использования ресурсов и др.; подобные особенности программ являются критичными для конечного пользователя и обычно могут быть рассмотрены только во время реального выполнения программы.

Средства статического анализа обычно предполагают некоторые стандартные подходы к рассмотрению исходного кода (построение таких информационных структур, как синтаксические деревья и базы знаний по модулям и функциям кода) и ориентированы на общие задачи поиска дефектов, обнаружения шаблонов кода, проведение автоматического рефакторинга и т. д. Средства динамического анализа предлагают решения для более широкого спектра задач по исследованию программного обеспечения и отличаются разнообразием применяемых подходов и методов. Данная особенность привела к тому, что было разработано и разрабатывается значительное количество комплексных систем, предоставляющих базовую инфраструктуру для реализации пользовательских инструментов анализа, производящих извлечение релевантной информации во время выполнения программы.

1.2 Инструментация кода

В основе большой группы подобных систем лежит технология инструментации кода — изменение исходного или исполняемого кода с целью добавления функциональности или изменения имеющейся функциональности. Внедрённый таким образом код осуществляет сбор информации и/или непосредственно проводить более сложную обработку для получения результатов анализа. В число распространённых задач, эффективно решаемых данным образом, входит поиск дефектов различного рода, сбор статистики по использованию ресурсов, взаимодействие с внутренним состоянием и данными программы для извлечения другой специфичной информации.

Инструментация исполняемого кода представлена в современных средствах в значительно большей степени, чем инструментация исходного кода; это связано, в первую очередь, с более высоким уровнем организации исходного кода и, соответственно, большей сложности его обработки и модификации.

Методы инструментации классифицируются также по стадии проведения инструментации на статическую и динамическую. Статическая инструментация предполагает предварительную модификацию целевых файлов с целью получения итоговых изменённых вариантов данных файлов. Использование модифицированных файлов вместо исходных приведёт к реальному выполнению внедрённого кода. Динамическая инструментация предполагает перехват управления при выполнении программы и замену блоков кода, подаваемых на процессор, на модифицированные блоки кода с помощью некоторого программного модуля, загружаемого в виртуальную память вместе с целевой программой. Оба подхода предоставляют схожие возможности, однако имеют несколько значительных различий, обуславливающих превосходство одного или другого подхода в условиях конкретной задачи:

- Статическая инструментация производится как некоторый предварительный этап и позволяет использовать модифицированную версию целевой программы, которая может быть использована неограниченное количество раз для получения необходимых результатов на различных наборах входных данных. Накладные расходы при выполнении модифицированной версии целевой программы обычно включают в себя только затраты на непосредственное выполнение внедрённого кода.
- Динамическая инструментация производится непосредственно во время выполнения и включает в себя этап декодирования исходного исполняемого кода и кодирования модифицированного исполняемого кода из некоторого промежуточного представления; эта особенность значительно увеличивает накладные расходы на выполнение целевой программы в режиме инструментации. Несмотря на увеличенные накладные расходы, динамическая инструментация предоставляет более широкие возможности по модификации кода и доступ к текущему состоянию программы уже на этапе инструментации, что значительно

уменьшает сложность структуры дополнительного исполняемого кода для проведения необходимого анализа.

1.3 Статическая инструментация исполняемого кода

Проведение статической инструментации исполняемого кода предусматривает непосредственное изменение одного или нескольких файлов, которые будут использованы вместо исходных. Как правило, форматы, описывающие структуру исполняемых файлов, обладают довольно высокой степенью связности их содержимого (например, смещения относительных переходов между фрагментами кода, смещения констант и специфических конструкций и др.). Поэтому, в отличие от динамической инструментации, при которой инструментационный код не обязан формировать часть целостного образа выполняемой программы в памяти, обеспечение корректности итогового результата является одной из важнейших задач при реализации системы статической инструментации.

Для статической инструментации также более остро стоит проблема разбора исходного исполняемого файла. В то время как при проведении динамической инструментации присутствует значительное количество косвенных признаков, которые можно использовать для идентификации необходимых областей программы, эффективность статической инструментации напрямую зависит от количества информации, которую можно извлечь из целевых файлов.

Несмотря на указанные выше возможные факторы снижения точности статической инструментации, а также более ограниченные возможности инструментационного кода по сравнению с динамической инструментацией, статический подход значительно лучше подходит к решению задач, для которых минимизация накладных расходов является критическим требованием. Методы статической инструментации также являются более подходящим решением при наличии ограничений среды выполнения (если данные ограничения усложняют применение механизмов по перехвату и генерации кода, используемых при динамической инструментации).

1.4 Цели работы

Данная работа представляет описание системы инструментов, позволяющих проводить инструментацию исполняемых файлов и динамических библиотек в формате ELF (Executable and Linkable Format – формат объектных и исполняемых файлов, используемый семейством Unix-подобных операционных систем) для архитектуры ARM. В качестве одной из основных платформ для применения данного инструмента рассматривается система Android. Именно ряд особенностей системы Android, а также относительная ограниченность ресурсов устройств на базе архитектуры ARM являются факторами, определяющими преимущество статической инструментации.

Данная статья имеет следующую структуру:

- Секция 2 описывает общие положения задачи статической инструментации исполняемого кода в формате ARM ELF и особенности целевых файлов, требующие применения специфических механизмов обработки; данная секция также предлагает подробный обзор характеристик системы, разработанной для решения задачи статической инструментации ARM ELF,
- Секция 3 содержит оценку результатов применения полученной реализации для задачи трассировки функций и подсчёта базовых блоков; в качестве целевых объектов выступают динамические библиотеки ARM ELF.
- Секция 4 приводит краткий обзор работ в области инструментации исполняемого кода.
- Секция 5 содержит общую оценку проведённых работ и рассматривает приоритетные направления дальнейшего развития.

2. Статическая инструментация ARM ELF

2.1 Общая структура формата ARM ELF

Исполняемые файлы и динамические библиотеки в формате ARM ELF имеют модульную структуру и разбиваются на заголовок и произвольное количество секций. Секции представляют из себя именованные неразрывные блоки данных, внутренняя организация которых определяется типом секции. Информация о количестве секций, их положении в файле, размерах и управляющих флагах содержится в заголовке файла.

Количество секций и положение секций друг относительно друга не являются фиксированными параметрами, что позволяет свободно добавлять, удалять и перемещать секции внутри одного ARM ELF файла.

Секции, присутствующие в ARM ELF файле делятся на загружаемые и незагружаемые:

- Загружаемые секции объединяются в сегменты и имеют фиксированное смещение в виртуальной памяти от начала образа. Данные смещения назначаются компоновщиком на этапе сборки итогового файла и напрямую связаны с внутренними смещениями, используемыми элементами секции.
- Незагружаемые секции содержат вспомогательную информацию, используемую компоновщиками (только для объектных файлов ELF) и другими инструментами (отладчиками, инструментами анализа и т. д.)

В отличие от положения секции в файле, положение секции в сегменте и её виртуальный адрес не всегда могут быть изменены без внесения дефектов в

итоговой образ. При проведении статической инструментации целевой ARM ELF файл либо полностью преобразовывается с перерасчётом всех смещений, зависимых от положений секций в сегментах, либо производится ограниченная работа с виртуальными адресами, требующая минимальных коррекций для поддержания правильной работы итогового файла.

В рамках представляемой работы применяется второй подход, предполагающий минимизацию изменений, вносимых в имеющиеся секции.

2.2 Общая схема проведения инструментации

Отсутствие ограничений на количество, размер и содержимое секций в ARM ELF файле предоставляет возможность организации инструментации следующим образом:

1. Обработка спецификаций типа инструментационных точек и кода обработчиков инструментационных точек, задаваемых пользователем, для разметки целевого файла и генерации непосредственного инструментационного кода.
2. Компиляция инструментационного кода в объектный файл ARM ELF.
3. Добавление секций исполняемого кода и данных из полученного объектного файла в целевой файл ARM ELF в качестве дополнительных секций.
4. Изменение секции исполняемого кода целевого файла для осуществления перехвата управления и передачи его в инструментационный код в точках инструментации.
5. Расширение и модификация специализированных секций целевого файла ARM ELF, используемых динамическим загрузчиком для идентификации внешних зависимостей (только если в инструментационном коде используются внешние зависимости, которые не присутствовали в коде целевого файла).
6. Корректировка смещений и внутренних зависимостей в итоговом файле для поддержания корректности выполнения.

2.3 Подключение инструментационного кода

Первым шагом проведения статической инструментации файла исполняемого кода является добавление непосредственно инструментационного кода; передача управления на инструментационный код и его выполнение будет позволять получать необходимую информацию при работе целевой программы.

Добавление инструментационного кода в целевой ARM ELF файл осуществляется путём добавления двух новый секций в конец файла и размещения непосредственно кода и данных в эти секции. Так как инструментационный код необходимо также использовать во время выполнения, требуется изменить структуру сегментов, описывающих образ, загружаемый в виртуальную память, добавив две секции в конец последнего сегмента.

Передача контроля управления в точках инструментации осуществляется путём замены одной или нескольких инструкций, находящихся в данной точке, на инструкцию безусловного перехода. Обратная передача осуществляется аналогичным образом путём вставки перехода в конец блока инструментационного кода. Схема, представленная на рисунке 1, иллюстрирует структуру исполняемого кода при применении описанного механизма.

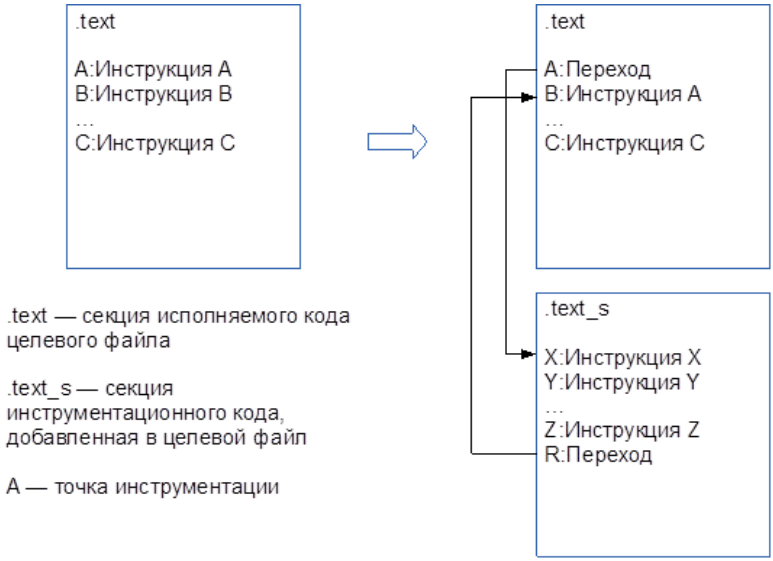


Рисунок 1: Передача управления на код инструментации

Для поддержания корректности выполнения целевой программы и обеспечения полноты инструментации инструментационный код должен удовлетворять следующим требованиям:

1. Инструментационный код должен быть функционально совместим с целевым файлом;
2. Инструментационный код должен сохранять состояние программы во избежание нарушения исходной функциональности;

2.3.1 Функциональная совместимость

В контексте данной задачи под функциональной совместимостью понимается соответствие инструментационного кода по набору инструкций для обеспечения корректного декодирования, осуществляемого процессором. Инструментационный код задаётся пользователем в виде исходного кода и должен быть скомпилирован в объектный файл. На платформе ARM существует несколько наборов инструкций, которые могут быть использованы параллельно; базовыми наборами инструкций являются ARM32 и Thumb-2. Для обеспечения функциональной зависимости производится полный разбор исполняемого кода целевого файла для выявления фрагментов, использующих конкретный набор инструкций. Генерация инструментационного кода производится таким образом, чтобы точке инструментации, находящейся в блоке инструкций ARM32, был сопоставлен блок инструкций, реализующих функциональность, необходимую пользователю, также в формате ARM32.

2.3.2 Сохранение состояния

Модифицированный целевой файл должен полностью сохранять исходную функциональность и выполнять инструментационный код исключительно как некоторый побочный эффект. В контексте схемы инструментации, описанной выше, для обеспечения корректности исходного исполняемого кода необходимо выполнение следующих условий:

1. Сохранение состояния программы после осуществления безусловного перехода на инструментационный код и до его непосредственного выполнения.
2. Восстановление состояния программы после выполнения инструментационного кода и до осуществления обратного безусловного перехода в основной исполняемый код.
3. Выполнение инструкций исполняемого кода целевого файла, заменённых на инструкцию безусловного перехода в инструментационный код.

Механизмы 1 и 2 реализуются с помощью добавления в инструментационный код инструкций сохранения регистров общего пользования и регистра флагов в стек программы и восстановление их из стека. Корректная работа со стеком внутри инструментационного кода гарантируется компилятором (блоки инструментационного кода оформляются в виде отдельных функций, что позволяет пользоваться положениями, зафиксированными в стандартном протоколе вызовов).

Механизм 3 реализуется путём добавления заменённых инструкций непосредственно в блок инструментационного кода после инструкций восстановления состояния, что гарантирует корректность выполнения данных инструкций и полное соответствие исходной функциональности целевого файла. Исключение составляют некоторые классы инструкций, зависимые от конкрет-

ного положения в виртуальной памяти (такие как, например, инструкции перехода, реализующие сдвиги относительно текущего значения счётчика инструкций). Для поддержания корректности в случае, если были заменены именно такие инструкций, необходимо провести их модификацию или заменить функционально аналогичным блоком инструкций.

Инструментационный код конструируется таким образом, чтобы иметь набор зарезервированных фрагментов, заполненных инструкциями, не имеющими эффекта. Именно на данные зарезервированные места осуществляется добавление дополнительных инструкций (сохранение состояния, заменённые инструкции целевого кода). Подробную схема трансформации инструментационного кода представлена на рисунке 2.

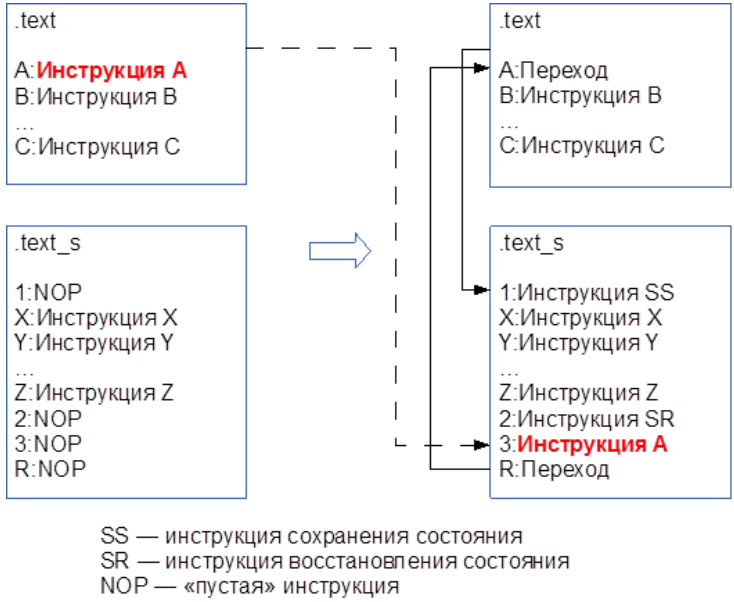


Рисунок 2: Схема структуры инструментационного кода

2.4. Подключение внешних зависимостей инструментационного кода

Одним из этапов работы динамического загрузчика операционной системы Linux (и систем, базирующихся на ней) при работе с файлами в формате ELF является разбор внешних зависимостей, заявленных загружаемым файлом и поиск необходимых ресурсов для удовлетворения данных зависимостей. Для задания внешних зависимостей и последующей работы с ними в файле формата ELF используется целый набор специфических секций, включающих в себя следующие:

1. .dynsym и .dynstr — секции, содержащие статическую информацию о внешних зависимостях: имена символов динамических библиотек и функций, контрольные флаги символов функций. Имена символов непосредственно используются динамическим загрузчиком для осуществления поиска по реестру библиотек, доступных в системе.
2. .got и .plt — секции, загружаемые в образ исполняемого файла в виртуальной памяти и используемые исполняемым кодом напрямую для осуществления перехода на инструкции в динамических библиотеках, реализующих внешние зависимости.
3. .rel.plt – секция, содержащая указания для динамического загрузчика о соответствии статической информации внешних зависимостей явным относительным смещениям в секции .plt.
4. .dynamic — секция, содержащая сокращённую информацию о наиболее важных параметрах образа целевого файла, загружаемого в виртуальную память. В данную информацию входит список внешних зависимостей по динамическим библиотекам.

Информация, заключённая в данных секциях, а также данные и код, находящиеся в секциях .got и .plt обеспечивают корректную загрузку и использование внешних библиотек во время выполнения целевого файла. Только зависимости, отражённые в данных секциях, будут обрабатываться корректно — использование в коде зависимостей, не известных динамическому загрузчику по указанным секциям, приводит к невозможности выполнения файла.

В случае, если инструментационный код имеет внешние зависимости, не отражённые в целевом файле, необходимо осуществить добавление информации о данных зависимостях в рассмотренные выше секции целевого файла. Данная задача обычно выполняется компоновщиком при создании исполняемых файлов и динамических библиотек из объектных файлов. Так как компоновщик не поддерживает работу уже с исполняемым ELF файлом, его применение невозможно.

Непосредственные модификации включают в себя следующие этапы:

1. Расширение секций .dynstr, .dynsym, .got, .plt, rel.plt и .dynamic на размер соответствующих блоков по числу внешних зависимостей инструментационного кода, отсутствующих в исходном целевом файле.
2. Конструирование блоков данных секций для работы с внешними зависимостями инструментационного кода и вставка в «пустые» фрагменты секций, появившиеся после их расширения.
3. Реорганизация таблицы сегментов и обновление данных в секции .dynamic.

В то время как непосредственное расширение и модификация секций являются исключительно техническими задачами, проведение реорганизации секций требует учёта многих особенностей отдельных секций. Как было отмечено ранее, загружаемые секции объединяются в сегменты и обычно располагаются

в данных сегментах как общий единый блок (т. е. в виртуальной памяти соседние секции не имеют «пустого» промежутка между собой). Расширение загружаемых секций (все указанные выше секции являются загружаемыми) приводит к образованию наложений в таблице сегментов. Данные наложения приводят к порче данных при загрузке образа в виртуальную память и должны быть разрешены путём сдвига секций и изменения порядка следования в образе.

Некоторые загружаемые секции (такие как секции исполняемого кода и динамических релокаций) содержат данные и управляющие конструкции, зависимые от относительного положения в образе, загружаемом в виртуальную память. Поэтому при проведении реорганизации структуры сегментов данные особенности необходимо учитывать. Для минимизации необходимых последующих корректировок больший приоритет имеют операции перемещения независимых секций. В случае, если это невозможно, производятся разбор затронутых секций и автоматическая модификация необходимых элементов.

Стандартные спецификации таблиц сегментов и секций, используемые в базовых компиляторных системах, организованы таким образом, что при практическом проведении инструментации, реорганизация секций приводит к минимальным изменениям.

2.5. Поддержание корректности ARM ELF

Этап проведения постобработки файла ARM ELF после проведения инструментации включает в себя задачи, относящиеся к корректировке смещений, используемых некоторыми инструкциями исполняемого кода. Следующие три группы инструкций рассматриваются в качестве целевых:

- Инструкции, осуществляющие обработку данных инструментационного кода;
- Инструкции перехода из секции кода A в секцию кода B, если одна из этих секций была модифицирована во время инструментации;
- Инструкции, которые были заменены на переходы в инструментационный ход и перенесены в блоки инструментационного кода.

Так как заданный пользователем исходный инструментационный код непосредственно компилируется в объектный файл, но добавляется в целевой ELF файл напрямую в виде двух секций (секции кода и секции данных), то стандартные модификации, проводимые компоновщиком для корректировки смещений из секции кода в секцию данных не производятся. Информация о позициях, в которых необходимо проставить смещение между виртуальными адресами данных секций, находится в объектном файле в секции .rel.text. В рамках проведения инструментации производится автоматический перенос информации из секции .rel.text (с учётом конкретных значений виртуальных адресов уже в целевом ELF файле) для модификации смещений в секции исполняемого кода инструментации.

Инструкции условных и безусловных переходов из секции A в секцию B используют относительные смещения, учитывающие разницу виртуальных адресов данных секций. Перенос одной из секций приводит к нарушению корректности инструкций перехода и требует автоматической коррекции.

При проведении инструментации производится перенос инструкций в точках инструментации в блоки инструментационного кода. В наборе инструкций ARM32 и Thumb-2 присутствуют типы инструкций, напрямую использующие значение регистра PC — счётчика инструкций. За счёт данного свойства результат выполнения данных инструкций зависит от их положения в образе целевого файла, загружаемого в виртуальную память; перенос инструкций из одной секции в другую может нарушить корректность их выполнения. Для устранения подобного эффекта необходимо проведение дополнительных корректировок данных инструкций для учёта разности виртуальных адресов положения в исходной секции исполняемого кода и положения в секции инструментационного кода.

Сложность модификаций, необходимых для инструкций, зависит от конкретного типа инструкций. Инструкции безусловного перехода требует простого изменения относительного смещения, закодированного в инструкцию (в некоторых случаях требуется использование альтернативных кодировок, поддерживающих более широкие диапазоны значений). Некоторые типы инструкций, такие как, например, инструкций загрузки из памяти (сохранения в память) по смещению от значения регистра PC, требует замены на блок арифметических инструкций, проводящий конструирование итогового эффективного адреса загрузки (сохранения).

3. Практические результаты

Для проведения эффективности реализации средства инструментации был проведён ряд тестов с использованием набора базовых библиотек платформы Android и набора библиотек для обработки файлов мультимедиа форматов. Тестовые запуски проводились на устройстве Pandaboard (Dual-core ARM Cortex A-9 MPCore, 1 GHz) с установленной версией системы Android 4.0.4.

При проведении первого этапа тестирования был рассмотрен ряд библиотек для работы с файлами специализированных форматов с целью извлечения необходимой пользователю информации:

- libjpeg — библиотека для работы с изображениями формата JPEG;
- libmpeg2 — библиотека для работы с видео в формате MPEG-2;
- libxml2 — библиотека для работы с файлами в формате XML;
- swftools — набор средств для работы с файлами Flash.

В рамках задачи оценки эффективности средства инструментации была проведена обработка библиотек с целью добавления кода, осуществляющего подсчёт количества базовых блоков инструкций, выполненных во время работы приложения. Практическая реализация необходимой функциональности зани-

мает 14 инструкций из набора ARM32 (учитывая инструкции, используемые базовыми механизмами инструментации) при замене одной инструкции из набора ARM32 в каждой из точек инструментации.

Для сравнения с имеющимися аналогами, предоставляющими поддержку архитектуры ARM, аналогичная по функциональности инструментация была проведена для рассматриваемых библиотек с помощью средства Valgrind.

Средние результаты запусков и информация о количестве точек инструментации по отношению к общему числу инструкций исполняемого кода в целевом файле и числу отдельных функций в составе исполняемого кода представлена в таблице 1.

Таблица 1: Инструментация с целью проведения подсчёта базовых блоков

Целевая программа	Базовое время работы, с	Время работы, с (статическая инструментация)	Точки инструментации	Время работы, с (Valgrind)
cjpeg (libjpeg)	3.965	9.944 (+150.8 %)	5649/78080/483	14.33 (+261 %)
djpeg (libjpeg)	9.135	22.168 (+142.7 %)	5649/78080/483	33.60 (+268 %)
mpeg2dec (libmpeg2)	10.469	15.493 (+48 %)	3849/44729/133	37.21 (+255 %)
xmllint (libxml2)	5.022	14.563 (+190 %)	61558/380491/3065	24.64 (+391 %)
png2swf (swftools)	39.382	47.32 (+19 %)	3157/23252/432	132.2 (+231 %)

Для оценки эффективности применения статической бинарной инструментации для проведения анализа библиотек платформы Android была проведена модификация набора базовых библиотек Android, использующихся для отображения графического интерфейса системы. В качестве целевой инструментации рассматривалось добавление функциональности по трассировке точек входа и выхода из функций, описанных в библиотеках. Замеры временных задержек проводились на одном из возможных сценариев работы с системой Android — запуске и инициализации графической оболочки.

В то время как непосредственно затраты на вставку инструментационного кода и выполнение инструкций базовых механизмов инструментации вносят минимальные задержки, отдельные аспекты организации инструментационного кода приводят к появлению более значительных накладных расходов. К основным аспектам, приводящим к замедлению работы, относятся следующие:

- запись данных в файл на устройстве;
- вычисление текущего абсолютного времени события входа или выхода из функции;
- использование средств синхронизации из стандартной библиотеки pthread для защиты доступа к внутренним буферам и ресурсам.

Таблица 2 приводит результаты измерений времени загрузки среды Android (от момента запуска устройства до внутреннего индикатора о завершении необходимой инициализации, после которой пользователь может выполнять действия с устройством). В таблице представлены следующие значения — время загрузки системы без инструментации каких-либо библиотек, время загрузки системы с использованием библиотеки, инструментированной «пустым» кодом, время загрузки системы с использованием библиотеки, инструментированной трассировочным кодом, и количество отслеживаемых событий входа/выхода из функций, зафиксированных за время выполнения.

Таблица 2: Инструментация библиотек Android

Библиотека	Загрузка, с	Загрузка, с (базовая инструментация)	Загрузка, с (целевая инструментация)	Вызовы инстр. кода
libsurfaceflinger.so	42.431	43.204 (+1.8 %)	51.706 (+21.8 %)	452880
libui.so	42.431	43.667 (+2.9 %)	58.964 (+38.9 %)	928812
libgui.so	42.431	43.364 (+2.1 %)	50.043 (+17.9 %)	406780
libEGL.so	42.431	43.105 (+1.5 %)	45.373 (+6.9 %)	129132
libGLES_android.so	42.431	43.226 (+1.8 %)	50.381 (+18.7 %)	500677

Полученные результаты укладываются в рамки ожидаемых результатов замедления при использовании инструментированных библиотек, активно применяемых несколькими ключевыми процессами системы Android. Замедление, вносимое инструментационным кодом, равномерно распределено по коду целевых библиотек и не влияет на корректность результатов профилирования библиотек по производительности отдельных функций.

4. Обзор существующих решений

Проекты систем инструментации, позволяющие пользователю создавать средства для проведения специализированного анализа, начали появляться уже в конце XX века. Некоторые из них были разработаны для определённых платформ, что позволяло использовать специфические особенности данных платформ для повышения гибкости и эффективности инструментов; современные системы, наиболее широко применяемые при разработке программ, покрывают целый ряд архитектур и основаны на относительно общих принципах работы с целевым кодом. Среди инструментов статической инструментации

следует отметить такие проекты, как ATOM[1], EEL[2], Etch[3], BIRD[4], PEBIL[5] и Dyninst[6]. Для проведения анализа на основе динамической инструментации широко применяются инструменты Pin [7], Valgrind [8] , DynamoRIO [9] и Dyninst.

Система ATOM (Analysis Tools with OM) была разработана для процессорной архитектуры DEC Alpha и предоставляла возможности для анализа целевых программ, доступных в виде объектных модулей. Система включала в себя интерфейс генерации кода, обращения к данным и работы с потоком управления целевой программы. Данный интерфейс позволял создать непосредственно код инструмента анализа, который впоследствии будет взаимодействовать с целевой программой, а также разметить объектные модули целевой программы для обозначения точек инструментации, т.е. точек в которых происходит выполнение инструментационного кода. ATOM осуществляет вставку инструкций инструментационного кода в указанные точки и, используя систему работы с объектными модулями OM, создаёт итоговый исполняемый файл. При запуске полученного файла исходная целевая программа работает с инструментом, заданным пользователем, параллельно в рамках одного процесса на устройстве. На основе системы ATOM были разработаны инструменты профилирования кэша и динамической памяти, счётчики инструкций и вызовов функций, а также система повышения эффективности предсказания переходов.

В рамках проекта EEL была разработана система, предоставляющая возможность внесения новой функциональности и изменения имеющейся функциональности программного кода, на основе работы со структурными компонентами — функциями, базовыми блоками и отдельными инструкциями, и комплексными производными сущностями, такими как граф потока управления. Данные компоненты в рамках инструмента обобщались под общим термином «абстракция». Абстракции позволяли производить архитектурно-независимую инструментацию — инструмент анализа, разработанный пользователем, также был выражен на языке абстракций и операций с ними; кодирование и декодирование в исполняемый код конкретной архитектуры выполнялось автоматически на основе описания языка соответствия. На основе системы EEL был реализован ряд инструментов профилирования и трассировки в рамках процессорной архитектуры SPARC.

Инструменты Etch и BIRD были разработаны для проведения инструментации программ на платформе Windows/x86.

Система Etch предоставляет модуль обхода программного кода с целью выявления иерархической структуры (модуль, функция, блок инструкций, инструкция) и разметки точек инструментации. Код инструмента анализа оформляется в виде обработчиков, применяющихся к отдельным узлам выявленной структуры. В результате выполнения инструментации целевой исполняемый файл полностью трансформируется для включения дополнительной функциональности. Помимо стандартных применений методов инструмента-

ции (профилирование и трассировка), Etch предлагает возможности модификации исходного исполняемого кода с целью оптимизаций (например, путём перестановки инструкций для повышения эффективности конвейера).

Система BIRD осуществляет инструментацию, минимизируя количество изменений, которые необходимо внести в целевой исполняемый код. Инструментационный код оформляется в виде динамической библиотеки, вызовы к которой вставляются в обрабатываемый файл путём замены инструкций (в тех местах, в которых замены возможны). Для разбора исполняемого кода используется статический декодер, производящий разметку точек инструментации. BIRD также предоставляет модуль динамического анализа для покрытия тех фрагментов кода, которые не удалось разметить статически. На основе системы BIRD был разработан инструмент, осуществляющий внедрение модуля защиты от несанкционированного изменения кода во время выполнения.

Инструмент PEBIL является одним из наиболее близких аналогов системы, разрабатываемой в рамках данной работы. PEBIL нацелен на работу с исполняемыми файлами и динамическими библиотеками в формате ELF для архитектур x86 и x86_64. Инструмент PEBIL производит расширение секций ELF в целевом файле, тем самым создавая пространство для внесения дополнительного кода. В коде целевой программы инструкции в точках инструментации заменяются на инструкции перехода на дополнительный код, откуда в свою очередь производится вызов методов динамической библиотеки, включающей непосредственную реализацию кода инструментации (однако присутствуют и ограниченные возможности по использованию прямых ассемблерных вставок). Инструмент PEBIL применяется в рамках нескольких систем оценки производительности работы программы (подсчёт статистики вызовов функций и блоков инструкций определённого типа, предсказание переходов).

Система Dyninst предоставляет возможности как статической, так и динамической инструментации исполняемого кода. Реализация подхода статической инструментации аналогична инструменту PEBIL — внедрение дополнительных фрагментов кода, замена инструкций на инструкции перехода и перенос заменённых инструкций для выполнения после инструментации. В отличие от PEBIL Dyninst максимизирует размер заменяемого блока инструкции с целью минимизации секций кода, которые необходимо расширить.

Системы Dyninst, Valgrind, Pin и DynamoRIO предоставляют широкие возможности по проведению динамической инструментации бинарного кода. Специализированные инструменты анализа разрабатываются на основе интерфейсов, предоставляемых данными системами для обработки кода в некотором внутреннем представлении. При выполнении программы блоки инструкций переводятся в данной представление ядром системы и передаются инструменту анализа, который осуществляет трансформацию кода для получения нужной функциональности. Изменённый блок инструкций декодируется обратно в машинное представление и передаётся на выполнение процессору.

5. Заключение

Реализованный в рамках проекта комплекс инструментов предоставляет пользователю возможность провести статическую бинарную инструментацию с указанием типа точек перехвата управления; дополнительная функциональность, которая будет выполняться во время выполнения также задаётся пользователем в форме исходного кода на языке C. Полнота реализации инструмента сравнима с рассмотренными аналогами, проводящими статическую инструментацию исполняемого кода. Инструмент предоставляет поддержку файлов исполняемого кода в формате ELF для платформы ARM; поддержка данной архитектуры отсутствует у прямых рассмотренных аналогов и представлена только среди инструментов динамической инструментации.

Разработанная система была применена в рамках задачи анализа производительности набора нативных библиотек платформы Android путём добавления трассирующего инструментационного кода в точки входа и выхода из внутренних функций библиотек. Дополнительно система инструментации была использована при анализе реализации подсистемы Android Binder, осуществляющей межпроцессное взаимодействие, путём трассировки внутренних функций библиотеки реализации Binder и извлечения параметров вызовов функций во время выполнения.

5.1 Направления дальнейших исследований

В настоящее время проводятся дополнительные исследования по расширению гибкости системы инструментации и оценка эффективности применения системы для реализации произвольных инструментов динамического анализа. В качестве прямых задач по развитию функциональности системы рассматриваются следующие:

- Расширение набора поддерживаемых типов точек инструментации (дополнительные типы инструкций, поддержка замены блоков).
- Расширение встроенных механизмов работы с внутренним состоянием программы и обработки инструкций (поддержка пользовательского интерфейса, доступного при разработке кода инструментации для таких целей как доступ к параметрам функций, извлечение явных значений операндов инструкций и т. д.).
- Оптимизация базовых механизмов обработки конкретных инструкций и блоков исполняемого кода, используемых в специализированных секциях (например, секции .plt) для упрощения добавления поддержки наборов инструкций отличных от ARM архитектур.

Практические результаты применения методов статической инструментации по сравнению с методами динамической инструментации, представленные в работах [5], [10] и в данной статье, показывает превосходство методов первой группы по производительности и величине накладных расходов. При этом современные средства динамической инструментации предоставляют более

широкие возможности по разработке инструментационного кода за счёт простоты встраивания дополнительной функциональности уже во время исполнения и отсутствия необходимости проводить комплексную корректировку целевого файла. На основе данных утверждений можно предположить, что поддержка дополнительных механизмов, которые позволят достигнуть уровня гибкости динамической инструментации при проведении статической инструментации, является в значительной степени важной задачей. Указанные выше направления развития инструмента посвящены решению данной задачи.

Список литературы

[1]. Srivastava Amitabh, Eustace Alan. ATOM: A System for Building Customized Program Analysis Tools, WRL Research Report 94/2, Western Research Laboratory, Palo Alto, CA, USA (http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-94-2.pdf)

[2]. Larus James R., Scnharr Eric. EEL: Machine-Independent Executable Editing. PLDI '95 Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation, 1995, pp. 291-300.

[3]. Romer Ted, Voelker Geoff, Lee Dennis, Wolman Alec, Wong Wayne, Levy Hank, Bershad Brian, Chen Brad. Instrumentation and Optimization of Win32/Intel Executables Using Etch. Proceedings of the USENIX Windows NT Workshop, 1997.

[4]. Susanta Nanda, Wei Li, Lap-Chung Lam, Tzi-cker Chiueh. BIRD: Binary Interpretation using Runtime Disassembly. International Symposium on Code Generation and Optimization, 2006. doi:10.1109/CGO.2006.6

[5]. Laurenzano Michael A., Tikir Mustafa M., Carrington Laura, Snavely Allan. PEBIL: Efficient static binary instrumentation for Linux. 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS), 2010, pp. 175-183. doi:10.1109/ISPASS.2010.5452024

[6]. Miller Barton P. and Bernat Andrew R., Anywhere, Any Time Binary Instrumentation, ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE), Szeged, Hungary, 2011, pp. 9-16. doi: 10.1145/2024569.2024572

[7]. Luk Chi-Keung, Cohn Robert, Muth Robert, Patil Harish, Klauser Artur, Lowney Geoff, Wallace Steven, Reddi Vijay Janapa, Hazelwood Kim. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, 2005, pp.190-200. doi:10.1145/1065010.1065034

[8]. Nethercote Nicholas and Seward Julian. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), San Diego, California, USA, 2007, pp. 89-100. doi:10.1145/1250734.1250746

[9]. Bruening Derek L. Efficient, Transparent, and Comprehensive Runtime Code Manipulation. Doctor of Philosophy Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

[10]. Hazelwood Kim, Klauser Artur. A Dynamic Binary Instrumentation Engine for the ARM Architecture. Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems (CASES'06). New York, NY, USA, 2006, pp. 261-270

Dynamic Analysis of ARM ELF Shared Libraries Using Static Binary Instrumentation

M.K. Ermakov <mermakov@ispras.ru>
S.P. Vartanov <svartanov@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, Russia, 109004.

Abstract. Dynamic program analysis is a prominent approach towards software quality control allowing to perform automatic profiling, defect detection and other activities during software development. In this paper we focus on static binary code instrumentation – a technique to automatically modify program executable code in order to extract data necessary for dynamic analysis. We discuss the key features of this technique within context of dynamic analysis and propose a method to perform static binary code instrumentation for ELF executable and shared library files specifically targeting the ARM architecture.

We describe the main steps of the proposed method including the following: instrumentation specification and target code parsing, executable instrumentation code generation and finally target executable code file modification in order to insert instrumentation code and ensure that control transfer from original code to instrumentation code and vice versa will happen at runtime.

Executable code file modification is performed within bounds of ARM ELF specifications and is designed to minimize the changes introduced in actual executable code blocks. Instrumentation code is appended to target files as a set of separate sections; we implement control transfer to instrumentation code through unconditional jump instructions which replace small blocks of original instructions at instrumentation points. In order to preserve the original functionality we wrap instrumentation code blocks with instructions that save and restore program state; additionally, instructions replaced at instrumentation points are transferred to the instrumentation code blocks. We also describe a set of modifications performed in order to introduce instrumentation code external dependencies to the target executable files.

The proposed method was implemented in an instrumentation framework. We provide a brief overview of practical experiments using basic block counting and function entry/exit tracing as base instrumentation applications. The results show better performance in comparison to popular dynamic instrumentation framework Valgrind and low overhead for system-wide tracking of native Android libraries.

Key words: static binary instrumentation; dynamic analysis; ARM architecture; ELF format; Android.

DOI: 10.15514/ISPRAS-2015-27(1)-1

For citation: Ermakov M.K., Vartanov S.P. Dynamic Analysis of ARM ELF Shared Libraries Using Static Binary Instrumentation. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 1, 2015, pp. 5-24 (in Russian). DOI: 10.15514/ISPRAS-2015-27(1)-1

References

[11]. Srivastava Amitabh, Eustace Alan. ATOM: A System for Building Customized Program Analysis Tools, WRL Research Report 94/2, Western Research Laboratory, Palo Alto, CA, USA (http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-94-2.pdf)

[12]. Larus James R., Scnharr Eric. EEL: Machine-Independent Executable Editing. PLDI '95 Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation, 1995, pp. 291-300.

[13]. Romer Ted, Voelker Geoff, Lee Dennis, Wolman Alec, Wong Wayne, Levy Hank, Bershad Brian, Chen Brad. Instrumentation and Optimization of Win32/Intel Executables Using Etch. Proceedings of the USENIX Windows NT Workshop, 1997.

[14]. Susanta Nanda, Wei Li, Lap-Chung Lam, Tzi-cker Chiueh. BIRD: Binary Interpretation using Runtime Disassembly. International Symposium on Code Generation and Optimization, 2006. doi:10.1109/CGO.2006.6

[15]. Laurenzano Michael A., Tikir Mustafa M., Carrington Laura, Snavely Allan. PEBIL: Efficient static binary instrumentation for Linux. 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS), 2010, pp. 175-183. doi:10.1109/ISPASS.2010.5452024

[16]. Miller Barton P. and Bernat Andrew R., Anywhere, Any Time Binary Instrumentation, ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE), Szeged, Hungary, 2011, pp. 9-16. doi: 10.1145/2024569.2024572

[17]. Luk Chi-Keung, Cohn Robert, Muth Robert, Patil Harish, Klauser Artur, Lowney Geoff, Wallace Steven, Reddi Vijay Janapa, Hazelwood Kim. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, 2005, pp.190-200. doi:10.1145/1065010.1065034

[18]. Nethercote Nicholas and Seward Julian. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), San Diego, California, USA, 2007, pp. 89-100. doi:10.1145/1250734.1250746

[19]. Bruening Derek L. Efficient, Transparent, and Comprehensive Runtime Code Manipulation. Doctor of Philosophy Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

[20]. Hazelwood Kim, Klauser Artur. A Dynamic Binary Instrumentation Engine for the ARM Architecture. Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems (CASES'06). New York, NY, USA, 2006, pp. 261-270

Применение статической инструментации байт-кода языка Java для динамического анализа программ

С. П. Вартанов <svartanov@ispras.ru>,
М. К. Ермаков <mermakov@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25

Аннотация. В статье рассматривается задача проведения динамического анализа программ на языке Java при условии, что исходный код программы отсутствует, а запуск программы может происходить на виртуальных машинах, которые интерпретируют байт-код формата, отличного от формата Java Virtual Machine. Приводится обзор методов инструментации и особенностей инструментации байт-кода языка Java для проведения итеративного динамического анализа с целью покрытия наибольшего числа путей выполнения программы. Для такого рода анализа используется построение входных данных для покрытия ранее не пройденных базовых блоков при помощи отслеживания потока помеченных данных, построения ограничений пути и проверки их выполнимости. В качестве метода решения поставленной задачи рассматривается применение статической инструментации байт-кода. Основными достоинствами подобного подхода являются увеличение скорости анализа (за счёт того, что инструментация проводится один раз, до начала работы итеративного механизма) и возможность конвертировать инструментированный байт-код в другие форматы для запуска на нестандартных виртуальных машинах (например, DEX для виртуальной машины Dalvik). В статье также рассматривается реализация предложенных методов в инструменте Coffee Machine. Инструментация осуществляется с помощью BCEL (библиотеки для манипулирования байт-кодом) и разделяется на три этапа: определение классов и методов для инструментации, инструментация на уровне классов и методов, инструментация на уровне отдельных инструкций. На основе Coffee Machine показано, как статическая инструментация может быть применена для печати информации о выполнившихся инструкциях, отслеживания помеченных данных, построения ограничений пути выполнения, а также для построения трассы синхронизационных событий. В качестве одного из ограничений предложенного подхода рассматривается невозможность доступа к динамическим данным в ходе выполнения программы и некоторым методам системных классов. Эти ограничения могут быть сняты за счёт увеличения накладных расходов на повторную инструментацию анализируемой программы и написание методов, симулирующих работу требуемых системных методов. Для использования методов-симуляторов используется специальный механизм, который сопоставляет их имена и имена реальных методов в процессе работы программы и производит дублирование вершины стека для передачи фактических параметров методу-симулятору.

Ключевые слова: статическая инструментация, динамический анализ, анализ программ.
DOI: 10.15514/ISPRAS-2015-27(1)-2
Для цитирования: Вартанов С.П., Ермаков М.К.. Применение статической инструментации байт-кода языка Java для динамического анализа программ. Труды ИСП РАН, том 27, вып. 1, 2015 г., стр. 25-38. DOI: 10.15514/ISPRAS-2015-27(1)-2.

1. Введение

Поиск ошибок в программном обеспечении — неотъемлемая часть процесса разработки программ, ввода их в эксплуатацию и дальнейшей поддержки. Причём стоимость обнаружения и исправления ошибок постоянно увеличивается с течением времени в жизненном цикле программного обеспечения.

Методы ручного или полуавтоматического тестирования программ требуют от программиста, во-первых, больших временных затрат, во-вторых, достаточно высокого уровня понимания принципов и деталей работы конкретной программы.

Для полностью автоматического анализа программ с целью поиска ошибок используются методы статического и динамического анализа. Динамический анализ известен как более тяжеловесный подход и требует значительных временных и вычислительных ресурсов. Основное достоинство динамического анализа по сравнению со статическим — отсутствие ложных срабатываний при определённых условиях, накладываемых на исходную программу (таких как детерминированность). Проведение динамического анализа требует внесения определённых изменений в ход выполнения программы. Это может быть осуществлено за счёт использования инструментации кода программы или преобразования среды выполнения программы в зависимости от требований, предъявляемых к анализу.

В статье рассматривается применение статической инструментации программ для проведения динамического анализа с целью поиска ошибок и уязвимостей в программах на языке Java, либо в программах, транслируемых в Java байт-код. Реализация описанного метода будет рассмотрена на примере инструмента Coffee Machine, разрабатываемого в ИСП РАН.[1]

2. Динамический анализ программ на языке Java с целью поиска ошибок и уязвимостей

Под динамическим анализом программ понимается процесс, требующий запуска анализируемой программы на исполнение. В противоположность ему, статические методы анализа основываются на различных моделях,

[1] Работа проводится в рамках научно исследовательских работ Института системного программирования РАН в 2014—2017 годах.

построенных на базе кода анализируемой программы. Динамический анализ программ имеет ряд достоинств и недостатков по сравнению со статическими методами.

К достоинствам динамического анализа можно отнести отсутствие ложных срабатываний при определённых ограничениях на анализируемую часть программы: отсутствие элементов, вносящих недетерминированность в ход выполнения, отсутствие элементов взаимодействия с пользователем или детерминированное описание этого взаимодействия в виде последовательности событий. Также достоинством динамического анализа можно назвать возможность доступа к фактическим данным, значения которых известны лишь на этапе выполнения программы.

К недостаткам динамического анализа относятся значительные временные затраты на его проведение по сравнению со статическим анализом. В случае, если целью анализа является обход всех возможных путей выполнения программы или достижение максимального покрытия кода программы по одному из критериев, возникает проблема экспоненциального роста числа путей анализируемой программы.

В рамках задачи поиска ошибок в приложениях на языке Java было принято решение применить методы полностью автоматического динамического анализ, поскольку эти методы наиболее эффективны для поиска редко проявляющихся критических ошибок, которые приводят к аварийному завершению программы.

2.1 Требования к решению

В связи с особенностями анализируемых программ к решению предъявлялся ряд требований:

- возможность проведения анализа на виртуальных машинах, интерпретирующих байт-код формата, отличного от де-факто принятого за стандарт формата Java Virtual Machine;
- также для анализа может быть недоступен исходный код программы — анализируемая программа может представлять собой набор класс-файлов или файлов типа JAR.

Решение основано на принципах, заложенных в инструменте динамического анализа Avalanche [1]. В цели инструмента входит обеспечение как можно более полного покрытия базовых блоков исходной программы и поиск ошибок на обнаруженных путях выполнения. Для этих целей используется инвертирование условных переходов. После каждого инвертирования условия в условном переходе при помощи решателя булевых формул STP осуществляется проверка выполнимости ограничений пути. В случае выполнимости строится новый набор входных данных, который в силу детерминированности анализируемой программы определяет ранее не пройденный путь выполнения.

Сбор информации о ходе выполнения программы, который необходим для построения ограничений пути, осуществляется при помощи механизма динамической инструментации, предоставляемого фреймворком Valgrind.

Применение инструмента Avalanche нецелесообразно при анализе Java программ в связи с тем, что анализ проводится на уровне машинного кода процессора компьютера, на котором выполняется виртуальная машина Java, и основные вычислительные мощности в основном будут затрачены на анализ работы самой виртуальной машины. К тому же на уровне инструментации, применяемом в инструменте Valgrind, нет непосредственного доступа к высокоуровневым структурам языка Java, что также неоправданно усложняет анализ Java программ, например обнаружение необработанных исключительных ситуаций в Java программе.

В связи с этим было принято решение о создании нового инструмента предназначенного для непосредственного анализа байт-кода Java. Принципы, на которых построен инструмент, подробно описаны в статьях [2, 3]. Данная статья посвящена описанию механизма статической инструментации байт-кода Java программы и описанию ограничений данного подхода, обнаруженных в процессе разработки инструмента.

3. Инструментация программ

Под инструментацией понимается процесс внедрения дополнительных действий в ход выполнения программы таким образом, чтобы исходная функциональность не была затронута или влияние на неё было бы минимальным.

Инструментация может применяться для извлечения различной информации о ходе выполнения программы в рамках задачи профилирования — определение времени событий (начала и завершения работы подпрограмм), длительности работы отдельных инструкций или фрагментов, время, затраченное на обмен информацией между различными компонентами, извлечение информации о состоянии памяти и т. д. Также среди извлекаемой информации могут быть данные, необходимые для оптимизации работы систем, управляющих выполнением программ, — точность предсказаний условных переходов, частота возникновения ситуаций, когда запрашиваемые данные отсутствуют в кэше, эффективность систем автоматической сборки мусора, а также для обнаружения дефектов.

Анализ извлекаемой информации может происходить как непосредственно в ходе выполнения (on-the-fly analysis), так и после завершения работы (postmortem analysis). Во втором случае извлекаемая информация записывается в так называемую трассу событий. Построенная трасса в дальнейшем может использоваться без изменений или преобразовываться для построения более абстрактных моделей работы программы и их

последующего анализа (например, поиск ошибочных ситуаций, допускаемых моделью, либо проверка модели на соответствие спецификации).

В качестве способа построения новых путей выполнения программы выбран метод инвертирования условных переходов, который включает в себя отслеживание потока помеченных данных для определения возможности инвертирования отдельного перехода. Помеченными называются входные данные, а также данные, которые от них зависят. Для инвертирования условного перехода необходимо, чтобы в его условии присутствовали помеченные данные. В ином случае не существует возможности в рамках принятых ограничений повлиять на выполнение программы таким образом, чтобы оно пошло по альтернативному пути. Отслеживание помеченных данных требует хранения информации о том, является ли отдельный блок памяти, используемый в программе, помеченным, а также сбора информации о том, как помеченные блоки связаны между собой.

Как и методы анализа программ, методы инструментации подразделяют на статические и динамические в зависимости от того, на каком этапе относительно непосредственного процесса выполнения программы происходит процесс инструментации.

3.1 Динамическая инструментация

Динамическая инструментация происходит в процессе выполнения программы.

В качестве примера среды динамической инструментации бинарного кода можно привести фреймворк Valgrind [4]. Принцип работы инструментации Valgrind заключается в разбиении исходного бинарного кода на блоки нефиксированной длины по заранее определённым правилам (не содержащие условных переходов, содержащие ограниченное число безусловных переходов и т. д.). Полученные блоки переводятся во внутреннее представление, инструментируются, конвертируются обратно в машинный код и запускаются на исполнение.

Для увеличения эффективности динамической инструментации часто используется кэширование инструментированных блоков таким образом, что при запросе на выполнение блока кода, вместо его инструментации будет использован ранее сохранённый блок, если он был проинструментирован ранее.

3.2 Статическая инструментация

В отличие от динамической, статическая инструментация полностью завершается до момента начала выполнения программы (однократного или многократного).

Основное достоинство статической инструментации состоит в том, что повторно исполняемые блоки программы не требуют кэширования или

переинструментации. Также в случае статической инструментации, дополнительные действия, помимо самого инструментационного кода, минимальны или отсутствуют вовсе.

Выбор в пользу одного из представленных способов инструментации осуществляется в зависимости от того, с какой целью производится преобразование программы. Статическая инструментация эффективней, если основное требование к инструментации — минимальное снижение производительности инструментируемой программы. Если производится многократное выполнение программы на путях, имеющих общие фрагменты кода, статическая инструментация также может быть эффективней за счёт отсутствия затрат на кэширование и возможных промахов кэша. Динамическая инструментация эффективней, если запуск программы производится один раз и заранее неизвестно, какая часть программы будет выполнена.

3.3 Инструментация байт-кода языка Java

Программы, написанные на Java, обычно транслируются в специальный байт-код, исполняемый виртуальной машиной. Для инструментации программ в языке Java, начиная с версии 1.5.0, существует специальный пакет `java.lang.instrument`.

Для инструментации байт-кода существует множество различных инструментов и библиотек. ASM [5] — одна из популярных библиотек для динамической инструментации байт-кода, использующих интерфейс `java.lang.instrument`. К сожалению, `java.lang.instrument` поддерживается не всеми виртуальными машинами. В частности, опция `-javaagent` отсутствует у виртуальной машины Dalvik платформы Android, что делает невозможным использование средств инструментации, подобных ASM.

Для статической инструментации байт-кода существует другая популярная библиотека — BCEL (bytecode engineering library) [6]. Она позволяет осуществлять генерацию и модификацию байт-кода на уровне отдельных инструкций и отдельных записей таблиц class-файлов.

4. Реализация инструментации в инструменте Coffee Machine

Инструментация байт-кода в инструменте Coffee Machine осуществляется на трёх основных уровнях:

- Предварительная подготовка классов и методов.
- Инструментация классов и методов.
- Инструментация инструкций.

Предварительная подготовка заключается в создании списка классов и методов, которые необходимо проинструментировать. Наиболее простое

решение этой проблемы заключается в инструментации всех методов всех классов, которые доступны в ходе выполнения программы. Для того, чтобы класс мог быть использован Java-машиной, он должен находится в файле с расширением `.jar` или `.class`, который располагается в одном из перечисленных мест:

- в системной директории Java-машины (где располагаются такие системные библиотеки как `rt.jar`);
- в специальной папке, предназначенной для расширений;
- в директории из списка `classpath`, который может быть передан Java-машине при помощи опции или может быть прочитан из переменной окружения.

В общем случае инструментация всех методов во всех возможных классах избыточна. Статически построенный список методов может, с одной стороны, содержать лишние методы (которые не будут вызваны), с другой стороны, может быть неполным. Первое происходит из-за того, что в большинстве случаев путь, по которому будет проходить выполнение, заранее не известен, а задача определения недостижимости кода является алгоритмически неразрешимой в общем случае. Второе происходит по причине того, что метод класса может быть вызван динамически.

Если число всех доступных методов сильно превышает число методов, действительно используемых в ходе выполнения программы, инструментация всех доступных методов приводит к чрезмерным накладным расходам. Данная проблема может быть решена использованием статического определения списка методов с рядом дополнений, реализованных в инструменте.

Если пользователю анализатора заранее известен список методов, которые с большой долей вероятности будут вызваны динамически, инструмент предоставляет возможность указать их для инструментации. На следующем шаге в этот список добавляются методы, вызовы которых явно присутствуют в байт-коде программы. Далее этот список может быть расширен целиком классами, в которых присутствуют эти методы, исходя из предположения, что если из класса динамически вызывается метод, с большой долей вероятности может быть вызван и другой метод того же класса.

Если в ходе выполнения программы обнаруживается метод, который не был проинструментирован ранее, выполнение останавливается и происходит инструментация пропущенного метода, а также других на основе информации о классе, которому он принадлежит. После этого программа перезапускается на тех же входных данных для продолжения анализа.

4.1 Инструментация классов и методов

После определения списка необходимых методов, управление передаётся компоненту, который разделяет методы на отдельные инструкции. Для этих целей используется BCEL.

4.2 Инструментация инструкций

Компонент, производящий инструментацию отдельных инструкций, имеет модульную структуру. Основной модуль представляет собой класс с набором абстрактных методов, соответствующих каждому типу инструкции байт-кода. Каждый конкретный модуль инструментации является наследником основного с необходимым образом реализованными методами инструментации. Для каждого типа инструкции переопределяются два метода: первый создаёт список инструкций, которые необходимо вставить до инструментируемой инструкции, второй — список инструкций, которые нужно вставить после.

В инструменте реализованы три модуля, производящих инструментацию отдельной инструкции:

- печать трассы инструкций,
- печать трассы условий на помеченные данные,
- печать трассы синхронизационных событий.

4.2.1 Печать трассы инструкций

Первый из модулей инструментации тривиален и демонстрирует возможность использования инструментации для снятия трасс работы анализируемой программы. Его работа заключается в печати информации о выполнившейся инструкции.

Для большинства инструкций это достигается путём вставки следующего набора инструкций после её выполнения:

- `getstatic` – получение потока вывода,
- `push` – запись выводимой строки в стек,
- `invokevirtual` – вызов функции `java.lang.String.println`.

Таким образом, вместо инструкции `new` для класса `java.io.FileReader` в байт-коде появятся четыре следующие команды (список команд приведён в формате библиотеки BCEL, в скобках указаны номера записей в константном пуле):

```
0: getstatic      java.lang.System.out Ljava/io/PrintStream; (51)
3: ldc            "Main.main 0 187 java/io/FileReader" (67)
5: invokevirtual  java.io.PrintStream.println (Ljava/lang/String;)V
(56)
8: new            java.io.FileReader (2)
```

Листинг 1. Инструкции байт-кода, в котрые будет преобразована инструкция `new` для класса `java.io.FileReader`

Исключением являются инструкции перехода, такие как `ifeq` (условный переход в случае равенства значений аргументов) или `return` (возврат из

метода), поскольку инструкции, помещаемые после них, не выполняются или могут не выполняться в случае условных переходов.

Решение данной задачи заключается в том, чтобы помещать печать дополнительных инструкций до инструкций перехода и изменять адреса переходов на эти инструкции на адрес первой дополнительной инструкции.

4.2.2 Печать трассы условий на помеченные данные

Одна из функций инструмента Coffee Machine — поиск ошибок в программах на языке Java с помощью итеративной генерации наборов входных данных, каждый из которых увеличивает покрытие кода программы, и запуска программы на этих входных данных.

Первый запуск анализируемой программы производится на введённых пользователем или случайным образом сгенерированных входных данных. Построение новых путей выполнения программы происходит при помощи инвертирования условий в условных переходах в уже известных путях выполнения. Для того, чтобы проверить в каждом конкретном случае, является ли инвертирование корректным, т. е. приводит ли оно к действительно существующему новому пути выполнения, в ходе выполнения программы производится отслеживание помеченных данных.

Для этих целей служит специальный инструментационный класс, методы которого вызываются в ходе выполнения программы. Этот класс также содержит структуры данных, соответствующие стеку, регистрам и памяти Java-машины, на которой происходит выполнение программы. Ячейки каждой структуры данных соответствуют реальным переменным Java-машины и хранят в себе ссылки на метапеременные. В терминах метапеременных описываются связи между реальными переменными в ходе выполнения программы. Эти связи записываются в виде булевых ограничений на языке запросов инструмента STP [7].

Для печати ограничений переопределены методы модуля инструментации для большинства типов инструкций байт-кода. В большинстве случаев метод содержит в себе вставку в байт-код инструкций, дублирующих необходимые параметры стека, помещающие в стек определённые параметры, и вставку вызова соответствующего метода инструментационного класса. Параметры инструментационных методов должны в точности соответствовать дублированным и добавленным значениям, чтобы не повлиять на результат работы инструкций, которые будут выполнены после.

4.2.3 Печать трассы синхронизационных событий

Другая функция инструмента Coffee Machine – поиск состояний гонки при помощи инструмента ThreadSanitizer Offline [8].

Инструментация в этом случае производится для ограниченного набора инструкций — работы с памятью, полями, исключениями, вызовов методов и специальных инструкций `monitorenter` и `monitorexit`. Каждая инструкция указанного типа инструментируется с целью печати информации в трассу синхронизационных событий в формате инструмента ThreadSanitizer Offline.

Пример формата трассы приведён ниже. В него входят информация о начале и завершении потоков и методов, установка и снятие блокировок, чтение и запись объектов.

```
...
RTN_EXIT 2 22c 0 0
SBLOCK_ENTER 2 937 0 0
WRITER_LOCK 2 938 2a1fc2b466 0
READ 2 939 1abbb49a1967433 1
WRITE 2 939 1abbb49a1967433 1
UNLOCK 2 93a 2a1fc2b466 0
RTN_EXIT 2 93b 0 0
SBLOCK_ENTER 2 475 0 0
RTN_EXIT 2 476 0 0
THR_END 2 383 0 0
THR_JOIN_AFTER 0 383 2 0
...
```

Листинг 2. Пример трассы синхронизационных событий

Рассмотрим теперь подробнее инструментацию вызовов методов.

4.2.4 Инструментация методов

Особые действия выполняются при инструментации инструкции вызова метода. Всего в байт-коде языка Java для вызова метода используются пять инструкций:

- `invokevirtual` (код $B6_{16}$),
- `invokespecial` (код $B7_{16}$),
- `invokestatic` (код $B8_{16}$),
- `invokeinterface` (код $B9_{16}$),
- `invokedynamic` (код BA_{16}).

В параметры перечисленных инструкций входят ссылка на вызываемый метод, а также (за исключением инструкций `invokestatic` и `invokedynamic`) ссылка на объект, метод которого вызывается. При возможности, на этапе инструментации извлекается имя и сигнатура вызываемого метода. Если реализация метода отсутствует в теле класса, соответствующий класс ищется среди классов-предков.

При использовании статической инструментации нет возможности проинструментировать ряд не нативных системных методов. Это происходит

в связи с возникающими зависимостями, которые не удаётся разрешить на этапе инициализации виртуальной машины. Такие методы, как и нативные, требуют симуляции, если принимают на вход помеченные данные.

Всего различаются три типа методов.

- Не нативные методы, к байт-коду которых у инструмента имеется доступ. Эти методы добавляются в список инструментации, а значит либо были проинструментированы ранее, либо будут проинструментированы. Инструментационный код до или после инструкции вызова в этом случае не добавляется.

Методы, которые либо не имеют байт-кода (нативные методы), либо это системные методы, которые невозможно проинструментировать статически. Среди них можно выделить два типа методов:

- Методы, которые принимают в качестве параметров или генерируют помеченные данные. В этом случае метод либо будет проигнорирован и помеченные данные будут потеряны, либо в качестве инструментационного кода будет использован вызов метода, симулирующего его функциональность.
- Методы, среди параметров которых нет помеченных данных, и о которых известно, что они не генерируют помеченные данные. В этом случае вызов метода игнорируется.

Вызовы методов первого и третьего типа игнорируются. Для инструкции вызова метода второго типа используется следующий способ инструментации, аналогичный методу использования методов-симуляторов инструмента Java ThreadSanitizer [8].

Каждый метод, реализующий симуляцию функциональности исходного метода имеет имя вида

```
[a|b]_<имя класса>_<имя исходного метода>
```

и располагается в классе с именем, соответствующим имени пакета, в котором располагается исходный класс. Первая буква в имени метода-симулятора означает позицию, в которую будет добавлен инструментационный код относительно инструкции вызова метода: `a` (after) — после инструкции вызова, `b` (before) – до инструкции вызова. Сигнатура метода-симулятора должна повторять сигнатуру исходного метода с дополнительным первым параметром, в котором содержится ссылка на объект, метод которого вызывается (если метод не статический), а также с дополнительным последним параметром, в котором методу передаётся его возвращаемое значение (если происходит вставка вызова после инструкции и если возвращаемое значение имеет тип, отличный от `void`).

Более детально описание процесса инструментации для случаев добавления вызова метода-симулятора до, после и одновременно до и после инструкции вызова исходного метода, приведено на рис. 1.

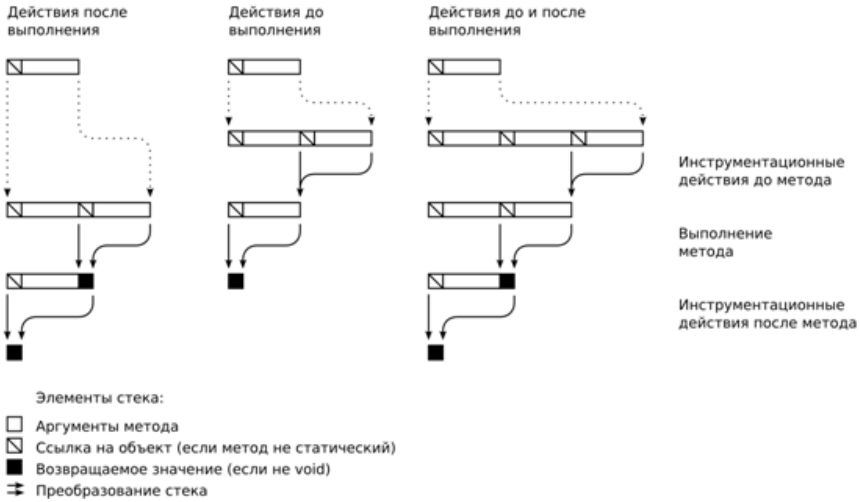


Рис. 1. Инструментация инструкции вызова метода

Например, метод-симулятор для метода `char java.lang.String.charAt(int)`, вызов которого должен быть вставлен после вызова исходного метода, будет иметь следующие имя и сигнатуру: `char a_String_charAt(String, int, char)`. Последней инструкцией этого метода должен быть возврат последнего параметра.

5. Заключение

Различные методы инструментации применяются в зависимости от особенностей задач, которые решаются при помощи модификации исходной программы. Статическая инструментация оправдывает затраты на повторную инструментацию кода, который не будет вызван в ходе выполнения программы в том случае, если проинструментированная один раз программа используется для запуска достаточное число раз. В случае же, когда целью анализа программы является проход по всевозможным путям выполнения с целью увеличения покрытия кода, понятие кода, который не вызывается в ходе всех произведённых вызовов стремится к понятию недостижимого кода.

Статическая инструментация имеет ряд ограничений, связанных с невозможностью доступа к динамическим данным в ходе выполнения программы. Эти ограничения влияют на возможности определения списка методов, которые будут вызваны. Проблема решается при помощи

итеративной повторной инструментации исходной программы. К тому же статически невозможно проинструментировать некоторые системные классы, которые используются в ходе инициализации Java-машины. Эта проблема, как и проблема инструментации нативных методов, решается при помощи написания методов-симуляторов. Также эту проблему можно решить при помощи методов изменения среды выполнения, чему будет посвящено дальнейшая работа в рамках исследования.

Список литературы

[1]. И. К. Исаев, Д. В. Сидоров. Применение динамического анализа для генерации входных данных, демонстрирующих критические ошибки и уязвимости в программах. Программирование, 2010, № 4, с. 1–16

[2]. С. П. Вартанов, А. Ю. Герасимов. Применение динамического анализа для поиска дефектов в программах на языке Java. Стр. 9–28. Труды Института системного программирования РАН, том 25, 2013 г. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print)

[3]. С. П. Вартанов, А. Ю. Герасимов. Динамический анализ программ с целью поиска ошибок и уязвимостей при помощи целенаправленной генерации входных данных. Стр. 375–394. Труды Института системного программирования РАН, том 26, 2014 г. Выпуск 1. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print)

[4]. N. Nethercote, J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation

[5]. E. Bruneton ASM 4.0. A Java bytecode engineering library, 2011 [PDF] (http://download.forge.objectweb.org/asm/asm4-guide.pdf)

[6]. Apache Commons Byte Code Engineering Library [HTML] (http://commons.apache.org/bcel)

[7]. V. Ganesh, D. L. Dill, A Decision Procedure for Bit-Vectors and Arrays // In Proceeding of Computer Aided Verification. 2007. P. 524–536.

[8]. K. Serebryany, T. Iskhodzhanov. ThreadSanitizer-data race detection in practice

Applying Java bytecode static instrumentation for software dynamic analysis

S. P. Vartanov <svartanov@ispras.ru>,
M. K. Ermakov <mermakov@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, Russia, 109004.

Abstract. This paper focuses on dynamic analysis of Java programs. We consider the following limitations: analysis tool may not have access to target program source code, and the program may be interpreted by a non-standard virtual machine with bytecode format different from Java Virtual Machine specifications. The paper describes an approach to bytecode instrumentation which is used to perform iterative dynamic analysis for new

execution path discovery. Path discovery is performed through automatic input data generation by tracing tainted data, collecting path conditions, and satisfiability checking. The proposed approach is based on static bytecode instrumentation. The main advantages of this approach are analysis speedup (because of one-time instrumentation) and explicit access to statically generated instrumented bytecode which makes it possible to run instrumented program on different virtual machines with different bytecode formats. Proposed approaches were implemented in the Coffee Machine tool. Paper sections dedicated to this tool provide a detailed description of taint data tracing and automatic branch traversing techniques as well as a set of instrumentation utilities based on Coffee Machine allowing executed instructions printing, taint trace dumping, and synchronization events trace generation. Coffee Machine uses BCEL (bytecode instrumentation library) for instrumentation. The paper concludes with an overview of practical restrictions existing for discussed methods and possible future work directions. Main disadvantage of proposed approach is the inability to access dynamic data at run-time and instrument a set of system class methods. It may be resolved by method simulation and execution environment modifications.
Keywords: static instrumentation, dynamic analysis, program analysis.

DOI: 10.15514/ISPRAS-2015-27(1)-2

For citation: Vartanov S.P., Ermakov M.K. Applying Java bytecode static instrumentation for software dynamic analysis. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 1, 2015, pp. 25-38 (in Russian). DOI: 10.15514/ISPRAS-2015-27(1)-2.

References

[1]. Isaev I. K., Sidorov D. V. Primenenie dinamicheskogo analiza dlya generatsii vkhodnykh dannykh, demonstriruyushhikh kriticheskie oshibki i uyazvimosti v programmakh [The Use of Dynamic Analysis for Generation of Input Data that Demonstrates Critical Bugs and Vulnerabilities in Programs]. Programmirovanie [Programming and Computer Software]. 2010. # 4. P. 1–16. (in Russian)

[2]. S. Vartanov, A. Gerasimov. Primenenie dinamicheskogo analiza dlyz poiska defektov v programmakh na yazyke Java. [Applying dynamic analysis for defect detection in Java-applications]. Trudy ISP RAN [The Proceedings of ISP RAS], vol 25, 2013. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), P. 9–28. (in Russian)

[3]. S. Vartanov, A. Gerasimov. Dinamicheskiy analiz programm c tselyu poiska oshibok i uyazvimostey pri pomoshchi tselenapravlennoy generatsii vkhodnykh dannykh [Dynamic program analysis for error detection using goal-seeking input data generation]. Trudy ISP RAN [The Proceedings of ISP RAS], vol 26, issue 1. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), P. 375–394. (in Russian)

[4]. N. Nethercote, J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation

[5]. Bruneton E. ASM 4.0. A Java bytecode engineering library, 2011 [PDF] (http://download.forge.objectweb.org/asm/asm4-guide.pdf)

[6]. Apache Commons Byte Code Engineering Library [HTML] (http://commons.apache.org/bcel)

[7]. V. Ganesh, D. L. Dill, A Decision Procedure for Bit-Vectors and Arrays // In Proceeding of Computer Aided Verification. 2007. P. 524–536.

[8]. K. Serebryany, T. Iskhodzhanov. ThreadSanitizer-data race detection in practice

Масштабируемый инструмент поиска клонов кода на основе семантического анализа программ*

Севак Саргсян, <sevaksargsyan@ispras.ru>
Шамиль Курмангалеев, <kursh@ispras.ru>
Андрей Белеванцев, <abel@ispras.ru>
Айк Асланян, <hayk@ispras.ru>
Артем Балоян, <artyom@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25

Аннотация: В статье обсуждаются существующие методы поиска семантически сходных участков кода (клонов). Анализируются недостатки каждого метода, на основе чего предлагается новый метод поиска клонов кода и описывается архитектура инструмента для языков C/C++ на основе компиляторной инфраструктуры LLVM, в которой реализован предложенный метод. Работу инструмента можно разделить на два основных этапа. На первом этапе программа компилируется в промежуточное представление LLVM компилятором Clang. По этому представлению строится граф зависимостей программы (Program Dependence Graph – PDG) для каждой единицы компиляции. На втором этапе производится анализ поиска клонов кода в построенных графах. В инструменте существует отдельный этап тестирования алгоритмов, который будет подключен при запуске инструмента в режиме тестирования. Это дает возможность автоматической генерации тестов и проверки точности реализованных алгоритмов.

Ключевые слова: семантический анализ, поиск клонов, PDG, LLVM.

DOI: 10.15514/ISPRAS-2015-27(1)-3

Для цитирования: Саргсян Севак, Курмангалеев Шамиль, Белеванцев Андрей, Асланян Айк, Балоян Артем. Масштабируемый инструмент поиска клонов кода на основе семантического анализа программ. Труды ИСП РАН, том 27, вып. 1, 2015 г., стр. 39-50. DOI: 10.15514/ISPRAS-2015-27(1)-3.

1. Введение

Повторное использование фрагментов исходного кода часто встречается при разработке программного обеспечения (ПО). Разработчик путем копирования

* Работа поддержана грантом РФФИ 15-07-07541 А

и дальнейшего изменения некоторого участка кода может получить желаемый результат. Фрагменты кода будем называть *клонами*, если они схожи друг с другом по заданной функции схожести. Клоны могут возникнуть не только в результате копирования неких участков кода. Примером может быть реализация схожей функциональности (многие драйверы в операционной системе делают аналогичную работу). Или ограничения конкретного языка программирования, что не позволяет создать одну общую версию нескольких функций, которые имеют маленькие отличия. Исследования показали, что до 20 процентов исходного кода могут являться клонами [1, 2]. Потребность в поиске клонов кода возникает при поиске функционально похожих частей программы в бинарном или исходном коде программ, при решении задач автоматического рефакторинга, поиска семантических ошибок, возникающих при некорректном копировании участков кода. В данной работе будет описана архитектура инструмента для поиска клонов кода, который обладает высокой точностью и масштабируема. Благодаря ряду новых техник и алгоритмов стало возможно анализировать миллионы строк исходного кода. Описание конкретных алгоритмов будут приведены позже.

2. Типы клонов

Есть три основных типа клонов (классификация приведена из [3]). Первый тип (**T1**) – это те фрагменты кода, которые отличаются только пробелами и комментариями. Второй тип (**T2**) – это те фрагменты кода, которые отличаются пробелами, комментариями, именами переменных, типами переменных и значениями переменных. Третий тип (**T3**) – это те фрагменты кода, которые отличаются пробелами, комментариями, именами переменных, типами переменных, значениями переменных. Где в конкретном фрагменте могут быть также добавлены или удалены некоторые строки.

3. Подходы поиска клонов кода

3.1 Текстовой подход

Алгоритмы поиска клонов кода считают хеш коды одной или нескольких строк исходного кода и сравнивают их. Если хеш коды совпадают, считается, что соответствующие строки исходного кода являются клонами [4]. Алгоритм может объединить последовательные клоны после того, как все клоны найдены. Некоторые алгоритмы могут найти схожие файлы. Для этого рассматривается некое подмножество строк для каждого файла и считают это подмножество его отпечатком. Дальше, если два отпечатка совпадают, считается, что соответствующие файлы похожи [5]. Алгоритмы, работающие на основе этого подхода, находят в основном клоны типа **T1**.

3.2 Лексический подход

Алгоритмы, основанные на этом подходе, в первую очередь получают последовательность лексических единиц (token) путем разбора исходного кода. После чего производится поиск совпадающую подпоследовательность таких единиц. Для этого существуют несколько эффективных алгоритмов, основанных на параметризованном суффиксом дереве [6]. Алгоритмы, работающие на основе этого подхода, в основном находят клоны типа **T1** и **T2**.

3.3 Синтаксический подход

Алгоритмы этого типа работают на абстрактном синтаксическом дереве (Abstract Syntax Tree – AST). Клонами считаются совпадающие AST-поддеревья. Некоторые алгоритмы сразу сравнивают пару деревьев для нахождения совпадающих поддеревьев [7]. Другой алгоритм строит суффиксное дерево (из вершин AST получается последовательность элементов (символы), на основе чего и строится суффиксное дерево) для каждого поддерева AST и сравнивает их [8]. Существует еще один эффективный алгоритм, который строит векторы для каждого AST-поддерева и сравнивает эти векторы [9]. Длина вектора зафиксирована и равна количеству всех возможных типов инструкций в AST. Для конкретного AST поддерева каждый элемент вектора – это количество соответствующих инструкций в этом поддереве. Алгоритмы, работающие на основе AST, находят все три типа клонов.

3.4 Подходы, основанные на метриках

Эти алгоритмы считают ряд метрик для фрагментов кода и сравнивают векторы полученных метрик вместо фрагментов. Обычно метрики считаются на AST или графе зависимостей (PDG) данного фрагмента [10, 11]. Другой метод кластеризует вычисленные метрики с помощью нейронных сетей [12]. Эти методы находят три основные типы клонов. Обычно у этих алгоритмов производительность лучше, чем у алгоритмов, основанных на AST или PDG. Но точность, как правило, гораздо хуже [13].

3.5 Семантический подход

Граф зависимостей программы (PDG) – это объединенный граф потока данных и графа потока управления. Вершины PDG – это инструкции программы, а ребра – зависимости между ними. Есть две основных типов ребер: ребра выражающие зависимости по данным и ребра выражающие зависимости по управлению. PDG наиболее общее представления программы, в нем хранится вся информацию о его семантике и структуре, что позволяет более точно анализировать программу. Оно используется в задачах оптимизации и поиска семантический схожих участков кода (пример PDG-графа приведен на рис.2, пунктирные стрелки это зависимости по

управлению). Алгоритмы, работающие на PDG, пытаются найти изоморфные подграфы в паре PDG-графов [14, 15, 16]. Эти алгоритмы неточные, так как задача изоморфизма графов NP-сложная. Как правило, эти методы обладают большой точностью, но довольно медленны.

4. Сравнения подходов

Описанные выше три типа клонов не могут быть найдены лексическим и текстовым подходом.. Следовательно, эти подходы не могут быть применены к ряду задач таких как, автоматический рефакторинг. Остальные подходы находят все три основные типы клонов. Недостаток подхода основанного на метриках, низкая точность, в результате чего у инструментов использующих этот подход большие погрешности. В общем случае алгоритмы, основанные на PDG, обладают большей точностью, чем методы AST, потому что в AST хранится информация только о структуре программы. PDG содержит информацию о семантике (через потоки данных) и о структуре (через потоки управления) программы, что дает преимущество по сравнению с AST. Следует, что высокая точность инструмента может быт обеспечена только при использования семантического подхода.

5. Модель инструмента поиска клонов кода

Наша модель инструмента поиска клонов кода основана на семантическом подходе, так как цель инструмента – найти все клоны с наибольшей точностью. Инструмент также должен быть масштабируемым для анализа проектов размером в несколько миллионов строк исходного кода. Инструмент состоит из двух основных частей. Первая часть создает PDG-граф из внутреннего представления LLVM (биткода) в ходе компиляции проекта (рис.1.) и реализован как компиляторный проход. Вторая часть инструмента отвечает за анализ PDG-графов в целях нахождения клонов. Она реализована как инструмент пакета LLVM [17].

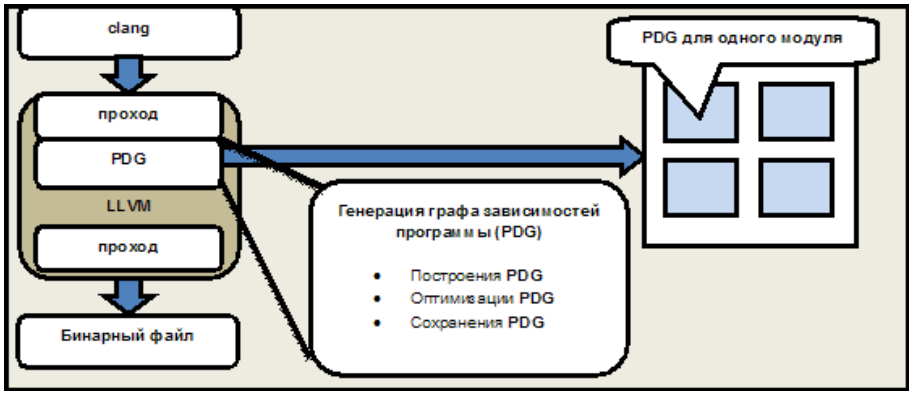


Рис. 1. Архитектура инструмента: генерация PDG.

5.1 Генерация PDG

Во время компиляции генерируется PDG-граф для каждой LLVM-функции [17]. Вершинами PDG являются инструкции LLVM (рис. 2).

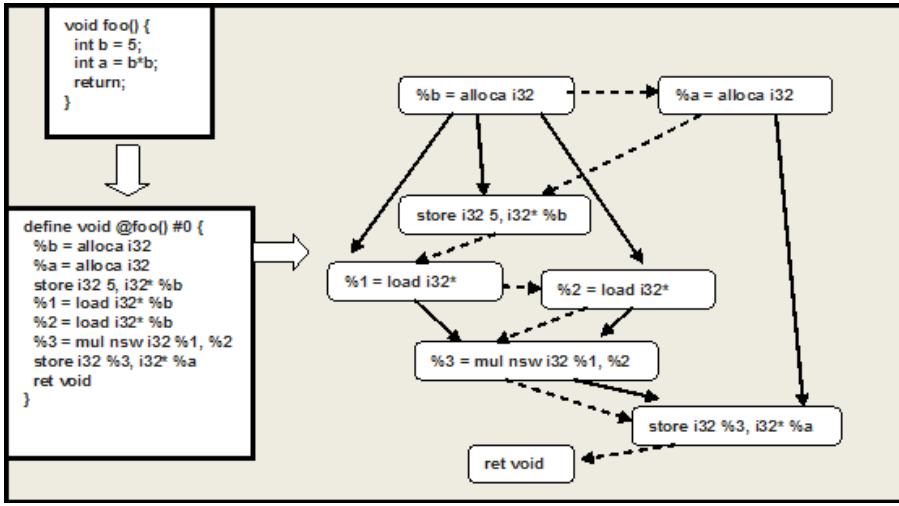


Рис. 2. Пример PDG.

Ребрам соответствуют зависимости по управлению между инструкциями, зависимости, получаемые анализом алиасов и use-def анализом LLVM [17]. Инструмент дает возможность генерировать PDG с тремя разными уровнями детализации. Что дает возможность эффективно искать клоны конкретного типа. В задачах где требуется быстро найти только клоны **T1** и **T2**, используется граф первого уровня. Если требуется дополнительный анализ, например, анализ алиасов, тогда используется граф второго уровня. Для эффективного поиска клонов **T3**, надо использовать полный граф (третьего уровня), что по сравнению с остальными будет работать медленнее. В графе первого уровня включаются только ребра, полученные LLVM use-def анализом. Граф второго уровня также содержит ребра, полученные с использованием анализа алиасов. Максимальную информацию будет содержать граф третьего уровня детализации – в нем есть все ребра первого и второго уровня, а также ребра, отображающие зависимости по управлению. По умолчанию генерируется PDG первого уровня (только на основе LLVM use-def анализа). Для генерации графа второго и третьего уровня детализации предусмотрены отдельные опции. После того, как PDG будет построен, оно оптимизируется и сохраняется в файле. Оптимизации PDG включают в себя:

1. Удаление тех вершин, которые не имеют никаких ребер.

2. Удаление тех вершин, которым не соответствует исходный код. Такие вершины могут возникнуть из-за того, что LLVM bitcode [17] представляется в виде SSA (Single Static Assignment) формы (рис.2 “alloca” инструкция).

5.2 Анализ PDG в целях нахождения клонов кода

Для того, чтобы инструмент был масштабируемым, сохраненные PDG должны быть разделены на части для параллельной обработки. Размер каждой части и количество параллельных процессов зависят от архитектуры машины, на которой будет работать инструмент.

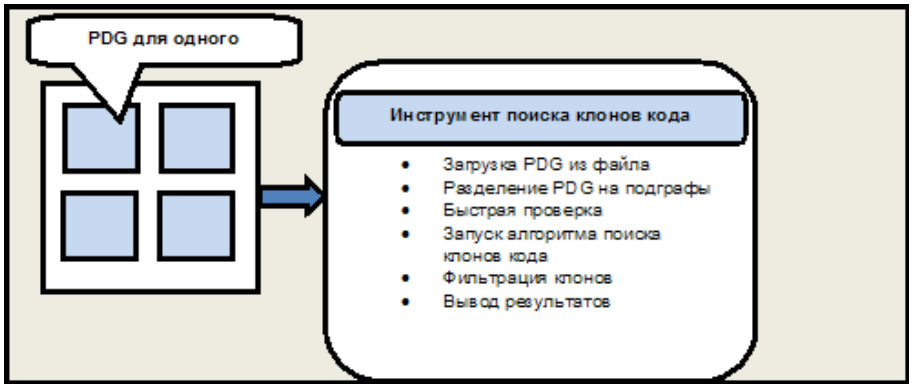


Рис. 3. Архитектура инструмента: анализ PDG.

5.3 Разделения PDG на подграфы

После загрузки PDG в память, они разделяются на единицы сравнения (**EC**) (р.3.). EC-ы представляют собой подграфы PDG и рассматриваются как потенциальные клоны друг друга. Разделение графа на EC-ы должно производится так, чтобы клоны оказались в различных EC-ах. В противном случае в EC попадет только часть клона, в результате чего он найдется частично или вообще не будет найден. Задача состоит в том, что каждый EC полностью содержал потенциальный клон.

5.4 Поиск клонов

После разделения PDG на EC, начинаются их сравнения в целях нахождения клонов. Инструмент содержит два типа алгоритмов сравнения. Первый тип алгоритмов проверяет пару EC на то, что они не клоны (быстрая проверка). Сложность таких алгоритмов O(n)) или O(n*log(n)), где n – количество вершин в обоих EC. Второй тип – это приближенные алгоритмы поиска изоморфных

подграфов. Вычислительная сложность этих алгоритмов довольно велика, она может достигать кубической степени от количества вершин графа. Так как большинство пар ЕС не являются клонами, для них нецелесообразно будет применить алгоритмы второго типа. Таким образом, сначала будут работать алгоритмы первого типа, которые за линейное время, докажут что большинство пар ЕС-ев не клоны. Алгоритмы второго типа будут запущены для тех пар ЕС-ев, которые не были обработаны алгоритмами первого типа. Приблизительно 80 процентов (согласно [1, 2] в среднем клонами может быть до 20 процентов кода) кода будет проанализировано алгоритмами первого типа. Только для малой части будут запущены тяжеловесные алгоритмы второго типа.

5.5 Фильтрация

Последний шаг в процессе поиска клонов кода – фильтрация результатов. Найденные пары изоморфных подграфов дополнительно проверяются алгоритмами фильтрации. Необходимость применения фильтра возникает из-за того, что мы определяем понятие клона для исходного кода программы, а ищем клоны как изоморфные подграфы. Получается, что клон должен быть некой последовательностью строк в файле (не обязательно друг за другом, но обязательно не сильно разбросанным). Цель фильтрации проверить, что исходный код соответствующий изоморфным подграфам не сильно разбросан. Эту проверку необходимо проводить после того как изоморфные подграфы найдены, в противном случае алгоритм поиска клонов может пропустить некоторые клоны кода.

6.Автоматическая генерация тестов

Для проверки точности реализованных алгоритмов разработана система автоматической генерации клонов кода. Для PDG проекта создается новый список графов, в котором попарно объединены графы оригинального проекта (рис.4). После чего алгоритм поиска клонов запускается на паре графов, где первый взят из оригинального списка, а второй из объединенного. Таки образом, «идеальный» алгоритм для всех графов из оригинального списка должен найти клон из объединенного списка. Количество найденных клонов будет характеризовать алгоритм.

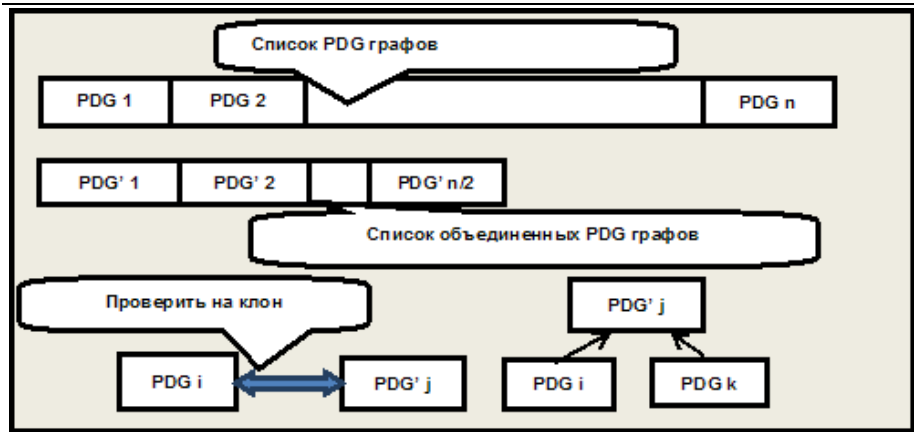


Рис. 4. Схема автоматического тестирования.

7. Заключение

В данной работе рассмотрены существующие подходы к поиску клонов коде. Для достижения высокой точности надо применить семантический подход. Масштабируемость обеспечивается применением двух типа алгоритмов. Первый тип пытается за линейное время определить, что пара PDG не может быть клонами. Второй тип алгоритмов это приближенные алгоритмы поиска изоморфных подграфов. Они запускаются в случае отрицательного результата первого типа алгоритмов. В целях эффективного построения PDG предложен модель инструмента основанной на компиляторной инфраструктуре LLVM, что позволяет получать эти графы в течении компиляции проекта. После чего происходит анализ полученных графов. Так же был предложен метод для автоматического тестирования точности алгоритмов поиска клонов.

Благодаря предложенному подходу, стало возможно сокращение времени генерации PDG. Алгоритмы поиска изоморфных подграфов применяются только для некоторых пар EC, что дает возможность создать масштабируемый инструмент поиска клонов кода.

Список литературы

[1]. Baker B., On finding duplication and near-duplication in large software systems, in: Proceedings of the 2nd Working Conference on Reverse Engineering, 1995, pp. 86-95. DOI: 10.1109/WCRE.1995.514697.

[2]. Roy C.K., Cordy J.R., An empirical study of function clones in open source software systems, in: Proceedings of the 15th Working Conference on Reverse Engineering, 2008, pp. 81-90, DOI: 10.1109/WCRE.2008.54.

[3]. Bellon S., Koschke R., Antoniol G., Krinke J., Merlo E., Comparison and evaluation of clone detection tools, Transactions on Software Engineering 33, 2007, pp. 577–591. DOI: 10.1109/TSE.2007.70725.

[4]. Ducasse S., Rieger M., Demeyer S., A language independent approach for detecting duplicated code, in: Proceedings of the 15th International Conference on Software Maintenance, 1999, pp. 109-119, DOI: 10.1109/ICSM.1999.792593.

[5]. Manber U., Finding similar files in a large file system, in: Proceedings of the Winter 1994 Usenix Technical Conference, 1994, pp. 2-2.

[6]. Kamiya T., Kusumoto S., Inoue K., CCFinder: A multilinguistic tokenbased code clone detection system for large scale source code, IEEE Transactions on Software Engineering, 2002, vol. 28, no. 7, pp. 654-670, DOI: 10.1109/TSE.2002.1019480.

[7]. Baxter I., Yahin A., Moura L., Anna M., Clone detection using abstract syntax trees, in: Proceedings of the 14th IEEE International Conference on Software Maintenance, IEEE Computer Society, 1998, pp. 368-377, DOI: 10.1109/ICSM.1998.738528.

[8]. Tairas R., Gray J., Phoenix-based clone detection using suffix trees, in: Proceedings of the 44th Annual Southeast Regional Conference, 2006, pp. 679-684, DOI: 10.1145/1185448.1185597.

[9]. Jiang L., Misherghi G., Su Z., Glondu S., DECKARD : Scalable and accurate tree-based detection of code clones, in: Proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society, 2007, pp. 96-105, DOI: 10.1109/ICSE.2007.30.

[10]. Mayrand J., Leblanc C., Merlo E., Experiment on the automatic detection of function clones in a software system using metrics, in: Proceedings of the 12th International Conference on Software Maintenance, 1996, pp. 244-253, DOI: 10.1109/ICSM.1996.565012.

[11]. Sargsyan S., Kurmangaleev S., Baloian A.., Aslanyan H., Scalable and Accurate Clones Detection Based on Metrics for Dependence Graph, Mathematical Problems of Computer Science, 2014, Volume 42, pp. 54-62.

[12]. Davey N., Barson P., Field S., Frank R., The development of a software clone detector, International Journal of Applied Software Technology, 1995, Volume 1, no. 3/4, pp. 219-236.

[13]. Gupta S., Gupta P. C., Literature Survey of Clone Detection Techniques, International Journal of Computer Applications, 2014, Volume 99, no. 3, pp. 41-44, DOI: 10.5120/17355-7858.

[14]. Komondoor R., Horwitz S., Using slicing to identify duplication in source code, in: Proceedings of the 8th International Symposium on Static Analysis, 2001, pp. 40-56, DOI: 10.1007/3-540-47764-0_3.

[15]. Krinke J., Identifying similar code with program dependence graphs, in: Proceedings of the 8th Working Conference on Reverse Engineering, 2001, pp.301-309, DOI: 10.1109/WCRE.2001.957835.

[16]. Gabel M., Jiang L., Su Z., Scalable detection of semantic clones, in: Proceedings of 30th International Conference on Software Engineering, 2008, pp. 321-330, DOI: 10.1145/1368088.1368132.

[17]. www.llvm.org.

Scalable code clone detection tool based on semantic analysis*

Sevak Sargsyan, <sevaksargsyan@ispras.ru>
Shamil Kurmnagaleev, <kursh@ispras.ru>
Andrey Belevantsev, <abel@ispras.ru>
Hayk Aslanyan, <hayk@ispras.ru>
Artiom Baloian, <artyom@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, Russia, 109004.

Annotation. This article describes the methods of code clones detection. New approach of code clones detection is proposed for C/C++ languages based on analysis of existed methods. The method based on semantic analysis of the project, which allows detecting code clones with high accuracy. It is realized as part of LLVM compiler, which allows exceeding existed methods. The tool is consisted of three basic parts. The first part is Program Dependence Graph (PDG) generation and serialization. PDG is constructed during compilation time of the project based on LLVM's intermediate representation. Several simple optimizations are applied on these graphs, then they are serialized to file. The second stage is analyzing of stored PDGs. PDGs are loaded from files and split to subgraphs. Every subgraph is considered as clone candidate. New method is purposed for the splitting, which increases number of detected clones. There are two types of algorithms for clone detection. The first types of algorithms try to prove that the pair of PDGs cannot be clones. These algorithms have linear complexity, which allows processing huge amount of PDGs pairs. In case of failure graph isomorphism algorithms are applied for similar subgraphs detection. The last part is integrated system for automatic testing of algorithm's accuracy. For the project, set of clones are automatically generated, then clone detection algorithms are applied for original source and generated one.

Keywords: semantic analysis; code clones; PDG; LLVM.

DOI: 10.15514/ISPRAS-2015-27(1)-3

For citation: Sargsyan Sevak, Kurmnagaleev Shamil, Belevantsev Andrey, Aslanyan Hayk, Baloian Artiom. Scalable code clone detection tool based on semantic analysis. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 1, 2015, pp. 39-50 (in Russian). DOI: 10.15514/ISPRAS-2015-27(1)-3

References.

[1]. Baker B., On finding duplication and near-duplication in large software systems, in: Proceedings of the 2nd Working Conference on Reverse Engineering, 1995, pp. 86-95. DOI: 10.1109/WCRE.1995.514697.

* The paper is supported by RFBR grant 15-07-07541 A

[2]. Roy C.K., Cordy J.R., An empirical study of function clones in open source software systems, in: Proceedings of the 15th Working Conference on Reverse Engineering, 2008, pp. 81-90, DOI: 10.1109/WCRE.2008.54.

[3]. Bellon S., Koschke R., Antoniol G., Krinke J., Merlo E., Comparison and evaluation of clone detection tools, Transactions on Software Engineering 33, 2007, pp. 577–591. DOI: 10.1109/TSE.2007.70725.

[4]. Ducasse S., Rieger M., Demeyer S., A language independent approach for detecting duplicated code, in: Proceedings of the 15th International Conference on Software Maintenance, 1999, pp. 109-119, DOI: 10.1109/ICSM.1999.792593.

[5]. Manber U., Finding similar files in a large file system, in: Proceedings of the Winter 1994 Usenix Technical Conference, 1994, pp. 2-2.

[6]. Kamiya T., Kusumoto S., Inoue K., CCFinder: A multilinguistic tokenbased code clone detection system for large scale source code, IEEE Transactions on Software Engineering, 2002, vol. 28, no. 7, pp. 654-670, DOI: 10.1109/TSE.2002.1019480.

[7]. Baxter I., Yahin A., Moura L., Anna M., Clone detection using abstract syntax trees, in: Proceedings of the 14th IEEE International Conference on Software Maintenance, IEEE Computer Society, 1998, pp. 368-377, DOI: 10.1109/ICSM.1998.738528.

[8]. Tairas R., Gray J., Phoenix-based clone detection using suffix trees, in: Proceedings of the 44th Annual Southeast Regional Conference, 2006, pp. 679-684, DOI: 10.1145/1185448.1185597.

[9]. Jiang L., Misherghi G., Su Z., Glondu S., DECKARD : Scalable and accurate tree-based detection of code clones, in: Proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society, 2007, pp. 96-105, DOI: 10.1109/ICSE.2007.30.

[10]. Mayrand J., Leblanc C., Merlo E., Experiment on the automatic detection of function clones in a software system using metrics, in: Proceedings of the 12th International Conference on Software Maintenance, 1996, pp. 244-253, DOI: 10.1109/ICSM.1996.565012.

[11]. Sargsyan S., Kurmangaleev S., Baloian A.., Aslanyan H., Scalable and Accurate Clones Detection Based on Metrics for Dependence Graph, Mathematical Problems of Computer Science, 2014, Volume 42, pp. 54-62.

[12]. Davey N., Barson P., Field S., Frank R., The development of a software clone detector, International Journal of Applied Software Technology, 1995, Volume 1, no. 3/4, pp. 219-236.

[13]. Gupta S., Gupta P. C., Literature Survey of Clone Detection Techniques, International Journal of Computer Applications, 2014, Volume 99, no. 3, pp. 41-44, DOI: 10.5120/17355-7858.

[14]. Komondoor R., Horwitz S., Using slicing to identify duplication in source code, in: Proceedings of the 8th International Symposium on Static Analysis, 2001, pp. 40-56, DOI: 10.1007/3-540-47764-0_3.

[15]. Krinke J., Identifying similar code with program dependence graphs, in: Proceedings of the 8th Working Conference on Reverse Engineering, 2001, pp.301-309, DOI: 10.1109/WCRE.2001.957835.

[16]. Gabel M., Jiang L., Su Z., Scalable detection of semantic clones, in: Proceedings of 30th International Conference on Software Engineering, 2008, pp. 321-330, DOI: 10.1145/1368088.1368132.

[17]. LLVM www.llvm.org.

Обход неизвестного графа коллективом автоматов. Недетерминированный случай

Игорь Бурдонов <igor@ispras.ru>,
Александр Косачев <kos@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. Исследование графов автоматами является корневой задачей во многих приложениях. К таким приложениям относятся верификация и тестирование программных и аппаратных систем, а также исследование сетей, в том числе сети интернета и GRID на основе формальных моделей. Модель системы или сети, в конечном счёте, сводится к ориентированному графу переходов, свойства которого нужно исследовать. За последние годы размер реально используемых систем и сетей и, следовательно, размер их графовых моделей непрерывно растёт. Проблемы возникают тогда, когда исследование графа одним автоматом (компьютером) либо требует недопустимо большого времени, либо граф не помещается в памяти одного компьютера, либо и то и другое. Поэтому возникает задача параллельного и распределённого исследования графов. Эта задача формализуется как задача исследования графа коллективом автоматов. В основе такого исследования лежит обход графа (проход по всем его дугам, достижимым из начальной вершины). Автоматы могут генерироваться в начальной вершине графа, перемещаться по дугам графа в направлении их ориентации и обмениваться между собой сообщениями через независимую (от графа) сеть связи. Суммарная память автоматов используется для хранения описания пройденной части графа. Для перемещения из вершины по выходящей из неё дуге графа автомат каким-то образом должен идентифицировать эту дугу: указать её номер. В нашей статье «Обход неизвестного графа коллективом автоматов» предложен алгоритм такого обхода для случая детерминированных графов. Задача усложняется, если граф недетерминирован. В таком графе одному номеру дуги соответствует, вообще говоря, несколько дуг, из которых для перехода выбирается одна дуга недетерминированным образом. Для того, чтобы обход графа был возможен, должна быть гарантия, что при неограниченном числе экспериментов каждая выходящая из вершины дуга с данным номером может быть пройдена. Такой недетерминизм мы называем справедливым. Решению задачи обхода справедливо недетерминированных графов посвящена данная работа.

Ключевые слова: недетерминированные графы, исследование графа, обход графа, взаимодействующие автоматы, параллельная обработка, распределенные системы, тестирование.

DOI: 10.15514/ISPRAS-2015-27(1)-4

Для цитирования: Бурдонов Игорь, Косачев Александр. Обход неизвестного графа коллективом автоматов. Недетерминированный случай. Труды ИСП РАН, том 27, вып. 1, 2015 г., стр. 51-68. DOI: 10.15514/ISPRAS-2015-27(1)-4.

1. Введение

Задача обхода неизвестного ориентированного графа автоматом (прохода автоматом по всем дугам графа, достижимым из начальной вершины) используется во многих приложениях. В данной статье подразумевается тестирование, когда граф — это граф автомата тестируемой системы, автомат на графе — тестирующая система, а проход по дуге — это тестовое воздействие и наблюдение результата [[4]]. В качестве практического примера можно привести работу [[11]], где выполнялось функциональное тестирование различных подсистем модели процессора: кэш третьего уровня, управление прерываниями и пр. Модельные графы содержали от нескольких тысяч до нескольких миллионов узлов и несколько миллионов дуг. Тест выполнялся максимально на 150 компьютерах.

Автомат-обходчик выполняется на одной машине (процессор с памятью), а наличие нескольких автоматов на разных машинах позволяет существенно распараллелить работу. При тестировании клонирование тестируемой системы обычно возможно только в начальном состоянии. Поэтому автомат начинает работу с начальной вершины графа. Будем считать, что за один такт создаётся не более одного клона тестируемой системы, то есть не более одного автомата-обходчика.

Для того, чтобы автомат мог обходить любой конечный граф, требуется доступ по чтению/записи к неограниченной рабочей памяти, в которой накапливается информация о пройденной части графа. Если эта память — часть памяти автомата (машины), автомат не конечен на классе всех графов. Алгоритм обхода графа одним автоматом предложен в [[5]]. Если число автоматов больше одного, но ограничено, то распараллеливание ускоряет обход, но не меняет порядок времени обхода в наихудшем случае [[10]].

Проблема возникает, когда граф не помещается в память машины, что эквивалентно конечности автомата. Есть два подхода.

Первый подход применим, когда рабочая память существует отдельно от памяти автоматов и реализуется на вершинах графа: автоматы не могут обмениваться между собой сообщениями, но могут писать/читать из текущей вершины символы конечного алфавита и перемещаться по дугам графа. Это эквивалентно «обратной» модели, когда автоматы неподвижно «сидят» в вершинах графа, а по дугам передаются сообщения, которые автоматы посылают друг другу. Такой подход может применяться, например, для сети интернета, когда вершина — это узел сети, а проход по дуге — передача сообщения между узлами. Алгоритм работы одного конечного автомата

предлагается в [[3],[7],[8]]. Если конечных автоматов несколько, это уменьшает время обхода [[15]].

Второй подход применяется при тестировании, когда вершина графа — это состояние тестируемой системы, и автомат ничего не может в неё писать. Тогда рабочая память — это суммарная память коллектива автоматов, обменивающихся сообщениями. Для k машин можно обходить графы в k раз большие, чем для одной машины. Если размер графа не ограничен, число автоматов в коллективе также должно быть не ограничено. Именно этот подход для конечных графов рассматривается в настоящей статье.

Когда автомат создаётся, ему выделяется отдельный клон (экземпляр) исследуемого графа. Считается, что автомат находится в выделенной начальной вершине графа, которую мы будем называть *корнем*. Этому соответствует рестарт тестируемой системы, т.е. её перевод в начальное состояние. Суммарная память автоматов используется для хранения описания пройденной части графа. После создания автоматы могут перемещаться по дугам графа в направлении их ориентации (каждый автомат по своему клону графа) и обмениваться между собой сообщениями через независимую (от графа) сеть связи.

Для перемещения из вершины по выходящей из неё дуге графа автомат каким-то образом должен идентифицировать эту дугу: указать её номер. Этому соответствует то или иное тестовое воздействие на тестируемую систему. Предполагается, что все дуги, выходящие из вершины перенумерованы, начиная с 1, и известно число выходящих дуг (число допустимых тестовых воздействий). После прохода по дуге автомат должен каким-то образом узнать, в какой вершине он оказался, и сколько у неё выходящих дуг. Для этого используется уникальный идентификатор вершины. Этому соответствует тестирование с открытым состоянием [[9]], когда тест после тестового воздействия может узнать текущее состояние реализации с помощью примитива *status message*. Взаимодействие автомата с графом сводится к примитиву ***проход по дуге*** с параметром «номер дуги» и ***ответа на проход***, в котором сообщается идентификатор конца дуги. Кроме того, автомат должен знать число дуг, выходящих из вершины. При тестировании это число допустимых тестовых воздействий, которое может зависеть от предыистории взимодействия. В графовой модели для простоты мы будем считать, что это число сообщается автомату после прохода по дуге вместе с идентификатором конца дуги. Мы будем считать, что имеется некий автомат графа, а указанный примитив и ответ на него реализуются как сообщения, посылаемые из автомата-обходчика автомату графа и обратно.

При обмене сообщениями автомат-получатель (включая автомат графа) идентифицируется адресом автомата. Автомат получает адрес при его создании, и этот адрес является параметром ответа на сообщение ***создай автомат***. Сами автоматы и клоны графа создаются и уничтожаются неким неописываемым внешним автоматом. Он же инициирует обход,

создавая первый автомат, и ему же посылается сообщение о завершении обхода.

Этот подход впервые был применён в нашей работе [[12]], а в [[13],[14]] описана улучшенная модификация алгоритма обхода. В этих работах граф предполагается детерминированным. Это означает, что в каждой вершине номер дуги однозначно определяет выходящую из этой вершины дугу. Когда автомат в текущей вершине указывает номер дуги как параметр примитива ***проход по дуге***, он может перейти по дуге только в одну вершину – конец дуги с указанным номером. Если это не так, граф считается недетерминированным. В данной работе мы предлагаем алгоритм обхода такого графа при некоторых предположениях о характере его недетерминизма.

2. Недетерминизм

Граф с нумерацией дуг, выходящих из каждой вершины, недетерминирован, если одному номеру дуги может соответствовать несколько дуг. Множество дуг с общим началом и одним номером дуги будем называть *дельта-дугой*. Будем считать, что недетерминированный выбор дуги по дельта-дуге осуществляется с помощью недетерминированной функции выбора s : *«множество дельта-дуг»* $\rightarrow$ *«множество дуг»*, которая для каждой дельта-дуги X недетерминированным образом возвращает дугу $x \in X$. Автомат указывает в текущей вершине номер дуги, который однозначно определяет дельту-дугу, и проходит некоторую дугу из дельта-дуги, определяемую функцией выбора. Мы будем говорить также, что автомат проходит дельта-дугу.

Поскольку после прохода по дуге автомат узнаёт только идентификатор конца дуги, кратные дуги с одним номером неразличимы между собой. Тем самым, без дополнительных предположений невозможно гарантировать проход по всем дугам графа. Мы будем считать, что в графе нет кратных дуг с одним номером. Тогда любая дуга уникально идентифицируется идентификатором её начала, номером дуги и идентификатором конца дуги.

Для каждой дельта-дуги X будем рассматривать различные последовательности значений функции выбора $s(X), s(X), \ldots$. Интуитивно ясно, что если может быть получена некоторая такая последовательность, то может быть получен и любой её префикс.

Будем говорить, что граф *абсолютно недетерминирован*, если для любой дельта-дуги X любая бесконечная последовательность дуг из этой дельта-дуги $S \in X^{\infty}$ может быть получена как последовательность значений функции выбора $S = s(X), s(X), \ldots$. Это означает, что если на недетерминизм функции выбора не налагается никаких ограничений, обход недетерминированного графа невозможно выполнить за конечное время, поскольку нет никаких гарантий, что за конечное число проходов по дельта-дуге удастся пройти по всем её дугам, если их больше одной. Например, всё время может выбираться одна и та же дуга.

Будем говорить, что граф *справедливо недетерминирован*, если для любой дельта-дуги X в любой бесконечной последовательности значений функции выбора $S = s(X), s(X), ...$ каждая дуга $x \in X$ встречается бесконечное число раз. Это означает, что для прохода по всем дугам дельта-дуги достаточно конечного числа проходов по дельта-дуге. Поэтому, если известно число дуг в дельта-дуге, автомат может за конечное время не только пройти по всем дугам дельта-дуги, но и узнать об этом. А, кроме того, по одной и той же дуге можно проходить любое неограниченное число раз, просто повторяя достаточно большое число раз проход по дельта-дуге.

Можно отметить, что гипотеза о справедливом недетерминизме графа, фактически, эквивалентна гипотезе о глобальном тестировании [[1],[2]], которая предполагает, что при бесконечном числе прогонов теста будут получены все возможные варианты поведения тестируемой системы. Для графа такому варианту поведения системы соответствует маршрут в графе, а для того, чтобы можно было пройти любой маршрут, функция выбора должна быть справедливой.

Частным случаем справедливо недетерминированного графа является *ограниченно недетерминированный граф*. Это такой граф, для которого существует такое число t, что для любой дельта-дуги X и любой бесконечной последовательности значений функции выбора $S = s(X), s(X), ...$ каждая дуга $x \in X$ встречается хотя бы один раз на каждом отрезке последовательности длиной t, т.е. на отрезке $S[i..i+t-1]$ для каждого натурального числа i. Для данного числа t такой граф будем называть также *t-недетерминированным*.

Очевидно, что при $t=1$ мы имеем детерминированный граф.

Имеет место следующая последовательность строгих вложений классов графов:

класс детерминированных графов

$\subset$ класс t-недетерминированных графов

$\subset$ класс ограниченно недетерминированных графов

$\subset$ класс справедливо недетерминированных графов

$\subset$ класс абсолютно недетерминированных графов.

В следующем разделе будет предложен алгоритм обхода коллективом автоматов любого справедливо недетерминированного конечного графа, для которого 1) в дельта-дуге нет кратных дуг, 2) известно число дельта-дуг, выходящих из каждой вершины, 3) известно число дуг в каждой дельта-дуге. Кроме того, мы будем предполагать, что при передаче сообщения «от точки к точке» не происходит потери, искажения, генерации лишних или обгона сообщений.

Замечание: Для t-недетерминированного графа можно ослабить требования, оставив только знание числа дельта-дуг в каждой вершине. Действительно, если мы t раз выполним проход по дельта-дуге X, каждый раз запоминая дугу, по которой мы проходим, то, во-первых, мы гарантированно пройдём по всем

дугам дельта-дуги, включая кратные дуги, а, во-вторых, узнаем число дуг в дельта-дуге с точностью до кратности, т.е. узнаем число различных концов дуг этой дельта-дуги.

3. Алгоритм обхода

Алгоритм основан на трёх режимах работы автоматов: *генератор*, *регулятор* и *движок*. Обход графа, то есть движение по его дугам, выполняют движки, регуляторы предназначены для управления перемещением движков по дугам, а генератор создаёт и уничтожает (с помощью внешнего автомата) движки и связанные с ними клоны графов. С каждой нетерминальной (имеющей выходящие дуги) вершиной связан один регулятор, в котором хранится описание дельта-дуг и дуг, выходящих из вершины. Регулятор корня выполняет также функции генератора. Для управления перемещением движков регулятор сообщает движку, находящемуся в вершине регулятора, номер дуги, однозначно определяющий дельту-дугу. Движок выполняет проход по этой дельта-дуге, что означает проход по одной из дуг этой дельта-дуги, выбираемой функцией выбора.

3.1 Описание дуги

В процессе работы алгоритма в суммарной памяти регуляторов строится описание пройденной части графа. Для этого в регуляторе вершины создаётся описание каждой выходящей из вершины дуги. *Пройденные* дуги делятся на *прямые* и *хорды*. Прямые дуги образуют остов пройденной части графа. *Законченной* дугой будем называть дугу, по которой больше не нужно посылать движки. *Непройденная* дуга всегда незаконченная, хорда всегда законченная, а прямая дуга может быть сначала незаконченной, а потом стать законченной. Прямая дуга становится законченной, когда выше неё по остову нет вершин, из которых выходят непройденные дуги. Заметим, что, поскольку регулятор посылает движки по дельта-дуге, а не отдельной дуге, движок всё равно может пройти по законченной дуге, но только в том случае, если в дельта-дуге есть незаконченная дуга. В целом имеются следующие состояния дуг: непройденная, хорда, незаконченная прямая и законченная прямая. В конце обхода все дуги законченные, т.е. хорды и законченные прямые.

Вершину будем называть *законченной*, если все выходящие из неё дуги законченные.

Описание дуги состоит из следующих полей:

- состояние дуги;
- идентификатор конца дуги [несущественен для непройденной дуги];
- адрес регулятора конца дуги [несущественен для непройденной дуги].

3.2 Сообщения

Сообщение состоит из названия (тега) и параметров сообщения. Ниже перечислены все используемые сообщения и их параметры, кроме адреса получателя, который является обязательным параметром каждого сообщения. В скобках указаны отправитель и получатель сообщения.

1. ***ты генератор*** (внешний автомат → первый созданный автомат):
   - адрес внешнего автомата,
   - адрес генератора, то есть адрес получателя сообщения;
2. ***создай граф*** (генератор → внешний автомат):
   - адрес генератора, то есть адрес отправителя сообщения;
3. ***граф создан*** (ответ на ***создай граф***):
   - адрес клона графа,
   - идентификатор корня,
   - число выходящих из корня дельта-дуг;
   - список чисел дуг для каждой дельта-дуги в корне;
4. ***создай автомат*** (генератор или движок, ставший регулятором, → внешний автомат):
   - адрес отправителя сообщения;
5. ***автомат создан*** (ответ на ***создай автомат***):
   - адрес созданного автомата;
6. ***ты движок*** (создатель движка → созданный движок):
   - адрес движка, т.е. адрес получателя сообщения,
   - адрес регулятора корня (он же генератор),
   - адрес клона графа,
   - идентификатор текущей вершины,
   - адрес регулятора текущей вершины;
7. ***куда идти*** (движок → регулятор текущей вершины):
   - адрес движка, т.е. адрес отправителя сообщения;
8. ***иди по дуге*** (ответ на ***куда идти***):
   - номер выходящей дуги (может быть равным нулю);
9. ***проход по дуге*** (движок → клон ассоциированного с движком графа):
   - адрес движка, т.е. адрес отправителя сообщения,
   - номер выходящей дуги;
10. ***ответ на проход*** (ответ на ***проход по дуге***):
    - идентификатор вершины конца пройденной дуги,
    - число выходящих из этой вершины дельта-дуг,
    - список чисел дуг для каждой дельта-дуги;
11. ***извещение*** (движок, прошедший по дуге, → регулятор начала дуги):
    - номер пройденной дуги,
    - описание пройденной дуги;

12. ***опрос*** (движок → регулятор корня как первый в списке регуляторов, или регулятор → следующий регулятор в списке регуляторов):
    - адрес движка, инициировавшего опрос,
    - идентификатор вершины;
13. ***ответ на опрос*** (найденный регулятор → движок, инициировавший опрос):
    - адрес найденного регулятора,
    - номер прямой дуги, входящей в вершину найденного регулятора;
    - адрес регулятора начала прямой дуги, входящей в вершину найденного регулятора;
14. ***уничтожь граф*** (движок перед самоуничтожением → внешний автомат):
    - адрес клона графа;
15. ***уничтожь автомат*** (самоуничтожающийся движок → внешний автомат):
    - адрес самоуничтожающегося движка, т.е. адрес отправителя.

3.3 Состояние автомата

Состояние автомата – это состояние памяти, в которой имеются следующие поля:

- режим работы: генератор (он же регулятор корня), регулятор, движок;
- собственный адрес автомата;
- адрес внешнего автомата;
- адрес клона графа;
- адрес генератора;
- адрес следующего в списке регуляторов (у последнего в списке регулятора адрес пустой);
- идентификатор текущей вершины;
- адрес регулятора текущей вершины;
- номер входящей дуги;
- адрес регулятора начала входящей дуги;
- число выходящих дельта-дуг;
- список по выходящим дельта-дугам (от 1 до числа выходящих дельта-дуг):
    - число дуг в дельта-дуге;
    - список описаний дуг дельта-дуги (от 1 до числа дуг в дельта-дуге).

3.4 Описание алгоритма

3.4.1 Генератор

Работа алгоритма начинается, когда извне создаётся первый автомат и ему посылается сообщение ***ты генератор***. Генератор выполняет функции регулятора корня, описанные ниже, и, кроме того, непрерывно генерирует

движки. Сначала создаётся клон графа с помощью сообщений ***создай граф*** и ответного сообщения ***граф создан***. Затем создаётся автомат с помощью сообщений ***создай автомат*** и ответного сообщения ***автомат создан***. Созданному автомату посылается сообщение ***ты движок*** с передачей ему адреса клона графа. Генерация движков происходит до тех пор, пока корень не станет законченным. Когда это произойдёт, генератор посылает вовне ***извещение***, означающее конец обхода.

3.4.2 Регулятор

Регулятор связан с одной вершиной графа и регулирует движение движков, попадающих в эту вершину, по выходящим дугам. Для этого регулятор ожидает от приходящих движков сообщений ***куда идти***. Получив такое сообщение, регулятор проверяет, закончена ли вершина. Если есть незаконченная дуга, регулятор посылает движку её номер в ответном сообщении ***иди по дуге***. Если вершина закончена, регулятор посылает движку нулевой номер дуги в ответном сообщении ***иди по дуге***.

Также регулятор может получить сообщение ***извещение***, содержащее описание дуги, выходящей из вершины регулятора. Регулятор проверяет по номеру дуги и идентификатору конца дуги есть ли уже описание этой дуги или ещё нет. Если такой дуги ещё нет, регулятор создаёт её описание на месте описания одной из непройденных дуг. Если такая дуга уже есть, регулятор меняет её состояние на то, которое указано в ***извещении***, за исключением случая, когда в ***извещении*** указана незаконченная прямая дуга, а в регуляторе дуга уже законченная.

Это исключение может возникнуть из-за асинхронности поведения автоматов. Если по непройденной дуге, ведущей в непройденную вершину, проходит первый движок, он посылает ***извещение*** №1 о незаконченной прямой дуге. Однако это ***извещение*** №1 может идти достаточно долго, и за это время по дуге может пройти много движков. В конце конйов один из этих движков пошлёт ***извещение*** №2 о том, что дуга законченная, и это ***извещение*** №2 обгонит ***извещение*** №1.

3.4.3 Движок

Движок начинает свою работу с получения сообщения ***ты движок***. Движок спрашивает у регулятора текущей вершины ***куда идти*** и ожидает ответа ***иди по дуге***.

Если в ответном сообщении ***иди по дуге*** указан нулевой номер дуги, движок должен уничтожить граф и самого себя с помощью сообщений ***уничтожь граф*** и ***уничтожь автомат***.

Если в ответном сообщении ***иди по дуге*** указан ненулевой номер дуги, движок выполняет проход по дуге.

Мы опишем цикл работы движка, начиная с прохода по дуге.

Для прохода по дуге движок посылает клону графу сообщение ***проход по дуге*** и ожидает ответа. В сообщении ***ответ на проход***, движок получает идентификатор текущей вершины (конца пройденной дуги). По этому идентификатору движок должен найти регулятор текущей вершины или убедиться, что такого регулятора нет. Для этого движок выполняет опрос регуляторов.

Опрос начинается с того, что движок посылает сообщение ***опрос*** регулятору корня, который всегда находится в начале списка регуляторов. Каждый регулятор, получив такое сообщение, сравнивает идентификатор вершины из сообщения с идентификатором вершины, регулятором которой он является. Если они совпадают, регулятор отправляет движку, инициировавшему опрос, сообщение ***ответ на опрос***. В противном случае регулятор пересылает сообщение ***опрос*** дальше по списку регуляторов. При этом последний в списке регулятор (у него пустой адрес следующего в списке регуляторов) становится предпоследним, запоминая адрес движка, инициировавшего опрос, как адрес следующего в списке регуляторов и пересылая ***опрос*** этому движку.

1. Если регулятор текущей вершины не найден (движок получил сообщение ***опрос***), движок сам становится регулятором текущей вершины и посылает регулятору начала пройденной дуги ***извещение*** о незаконченной прямой дуге. Затем движок, ставший регулятором, создаёт новый движок с помощью сообщения ***создай автомат*** и ответного сообщения ***автомат создан*** и посылает созданному автомату соообщение ***ты движок***, в котором передаёт ему адрес клона графа.
2. Если регулятор текущей вершины найден (движок получил сообщение ***ответ на опрос***), движок узнаёт адрес регулятора текущей вершины, а также номер прямой дуги, входящей в вершину найденного регулятора, и адрес регулятора начала прямой дуги, входящей в вершину найденного регулятора. Если найден регулятор корня, то у него номер входящей прямой дуги равен нулю. Движок определяет, в какое состояние должна перейти пройденная им дуга: хорда или прямая дуга.

   Здесь нужно отметить, что эта дуга была отмечена как непройденная в тот момент, когда движок получил номер этой дуги от регулятора её начала в сообщении ***иди по дуге***. Однако после этого движок выполнял проход по дуге и опрос регуляторов. Из-за асинхронности поведения автоматов могло оказаться, что по этой дуге уже прошёл другой движок, попал в ещё непройденную вершину, стал её регулятором, послал ***извещение*** о прямой дуге, но это ***извещение*** не поступило в регулятор начала дуги к указанному выше моменту. В этом случае наш движок не должен посылать ***извещение*** о хорде, а повторное ***извещение*** о прямой дуге может понадобиться только в том случае, если она должна быть отмечена как законченная. Если же эта дуга хорда, и предыдущий движок, прошедший по той же дуге, уже послал ***извещение*** об этом, то повторное ***извещение*** о хорде ничего не изменит в описании этой дуги, поэтому его

можно послать. Если же наш движок был первым, прошедшим по этой дуге, то такое ***извещение*** о хорде нужно посылать обязательно.

Как движок определяет состояние пройденной дуги? Это прямая дуга, если номер пройденной дуги и адрес регулятора её начала совпадают с номером прямой дуги, входящей в вершину найденного регулятора, и адресом регулятора начала прямой дуги, входящей в вершину найденного регулятора. В противном случае это хорда.

2.1. Если движок прошёл по хорде, он посылает регулятору её начала ***извещение*** о хорде. После этого движок спрашивает у регулятора текущей вершины ***куда идти*** и ожидает ответа ***иди по дуге***. Если в ответном сообщении ***иди по дуге*** указан нулевой номер дуги, движок уничтожает граф и самого себя с помощью сообщений ***уничтожь граф*** и ***уничтожь автомат***. Если в ответном сообщении ***иди по дуге*** указан ненулевой номер дуги, движок выполняет проход по дуге.

2.2. Если движок прошёл прямую дугу, он спрашивает у регулятора текущей вершины ***куда идти*** и ожидает ответа ***иди по дуге***. Если в ответном сообщении ***иди по дуге*** указан нулевой номер дуги, движок сначала посылает регулятору начала пройденной дуги ***извещение*** о законченной прямой дуге, а потом уничтожает граф и самого себя с помощью сообщений ***уничтожь граф*** и ***уничтожь автомат***. Если в ответном сообщении ***иди по дуге*** указан ненулевой номер дуги, движок выполняет проход по дуге, и никакого ***извещения*** не посылает.

3.5 Оптимизация

В этом разделе мы опишем оптимизации, позволяющие ускорить работу алгоритма.

3.5.1 Терминальный корень

Если корень – терминальная вершина, то генератору не нужно создавать движок. После создания первого клона графа генератор проверяет, является ли корень терминальной вершиной. Если да, то генератор уничтожает клон графа с помощью сообщения ***уничтожь граф*** и посылает вовне ***извещение***, означающее конец обхода.

3.5.2 Терминальная некорневая вершина

Аналогично, если движок прошёл по дуге в терминальную вершину, и это первый движок, попавший в эту вершину, то после того, как он становится регулятором, ему не нужно создавать новый движок. В этом случае движок, ставший регулятором, посылает регулятору начала пройденной дуги

извещение о законченной прямой дуге. Затем движок, ставший регулятором, уничтожает граф с помощью сообщения ***уничтожь граф***.

3.5.3 Петля

Пройденная петля по определению является хордой. Движок может распознать этот случай без опроса регуляторов. Для этого после прохода дуги достаточно сравнить идентификатор текущей вершины с идентификатором той вершины, из которой движок вышел. Если они совпадут, то движок прошёл по петле.

3.5.4 Пройденная дуга

Если движок проходит по уже пройденной дуге, он может определить её конец без опроса регуляторов. Для этого достаточно воспользоваться описанием пройденных дуг с тем же номером, которое имеется в регуляторе начала дуги. Это описание регулятор может передать движку как дополнительный параметр сообщения ***иди по дуге***: список описаний пройденных дуг с данным номером дуги.

Сначала по идентификатору текущей вершины движок проверяет, прошёл он уже пройденную дугу или ещё не пройденную. Для этого он сравнивает идентификатор текущей вершины с идентификаторами концов пройденных дуг пройденной дельта-дуги. Если движок прошёл непройденную дугу, его поведение описано в основном алгоритме. Если движок прошёл пройденную дугу, его работа зависит от состояния этой дуги.

1. Если движок прошёл пройденную хорду, движок знает также адрес регулятора текущей вершины, спрашивает у него ***куда идти*** и ожидает ответа ***иди по дуге***. Если в ответном сообщении ***иди по дуге*** указан нулевой номер дуги, движок уничтожает граф и самого себя с помощью сообщений ***уничтожь граф*** и ***уничтожь автомат***. Если в ответном сообщении ***иди по дуге*** указан ненулевой номер дуги, движок выполняет проход по дуге.
2. Если движок прошёл пройденную незаконченную прямую дугу, движок знает также адрес регулятора текущей вершины, спрашивает у него ***куда идти*** и ожидает ответа ***иди по дуге***. Если в ответном сообщении ***иди по дуге*** указан нулевой номер дуги, движок сначала посылает регулятору начала пройденной дуги ***извещение*** о законченной прямой дуге, а потом уничтожает граф и самого себя с помощью сообщений ***уничтожь граф*** и ***уничтожь автомат***. Если в ответном сообщении ***иди по дуге*** указан ненулевой номер дуги, движок выполняет проход по дуге.
3. Если движок прошёл пройденную законченную прямую дугу, движок уничтожает граф и самого себя с помощью сообщений ***уничтожь граф*** и ***уничтожь автомат***.

3.6 Теорема о конце обхода

Будем предполагать, что время передачи одного сообщения, время срабатывания каждого автомата и время перемещения автомата по дуге (т.е. время срабатывания автомата клона графа) ограничено.

Теорема о конце обхода: Через конечное время после начала работы обход графа будет завершён, то есть по каждой дуге графа хотя бы один раз прошёл хотя бы один движок, генератор пошлёт внешнему автомату ***извещение***. Кроме того, все дуги будут законченными, а прямые дуги будут образовывать остов графа, ориентированный от корня.

Доказательство:

Из описания алгоритма следует, что состояние дуги может меняться так, как это изображено на **Ошибка! Источник ссылки не найден.**.

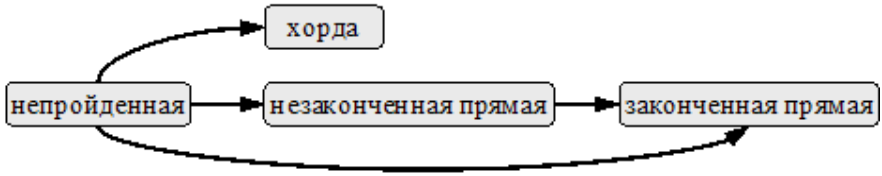


Рис. 1. Схема изменения состояния дуги

Поскольку в графе изменения состояний дуги нет ориентированных циклов, а число дуг конечно, начиная с некоторого времени T_1 все дуги перестают менять своё состояние. Далее будем рассматривать работу алгоритма после времени T_1.

Дуга ab становится прямой, если она была непройденной и вела в непройденную вершину b, после того, как по этой дуге прошёл движок, оказавшийся первым среди всех движков попадающих в вершину b. Поэтому прямые дуги не образуют ориентированных циклов и в каждую вершину входит не более одной прямой дуги. Также, поскольку движки начинают двигаться по графу, начиная с корня, все пройденные вершины достижимы из корня по пройденным дугам, и в каждую пройденную вершину, кроме корня, входит прямая дуга. Отсюда следует, что после времени T_1 прямые дуги образуют остов пройденной части графа, а хорды являются хордами этого остова, т.е. начинаются и заканчиваются в вершинах остова, но сами остову не принадлежат.

Покажем, что через конечное время генератор прекратит генерацию движков.

Допустим, это не так и движки генерируются бесконечно долго. Поскольку число путей в графе конечно, существуют пути, по которому движки двигаются бесконечное число раз. Как их префиксы существуют пути из прямых незаконченных дуг, начинающиеся в корне, по которым движки двигаются бесконечное число раз. Возьмём

максимальный из таких путей путь *P*. Покажем, что каждый движок, проходящей путь *P*, должен двигаться дальше из конца пути.

Допустим, это не так. Движок не двигается дальше, если конец пути законченный. Если путь *P* имеет нулевую длину, то его конец совпадает с его началом – корнем, а тогда корень законченный, и генератор должен был бы прекратить генерацию движков, что противоречит допущению. Если путь *P* имеет ненулевую длину, то через конечное время последняя дуга пути *P* станет законченной, что противоречит тому, что после времени T_1 все дуги перестают менять своё состояние. Мы пришли к противоречию, следовательно, наше допущение не верно, и утверждение доказано.

Поскольку число дульта-дуг с началом в конце пути *P* конечно, по одной из этих дельта-дуг движки двигаются бесконечно долго. Но тогда в этой дельта-дуге есть незаконченная дуга. В силу справедливости недетерминизма и конечности числа дуг в дельта-дуге по этой дуге также бесконечно долго двигаются движки. Эта дуга не может быть прямой, так как это противоречило бы максимальности пути *P*. Также эта дуга не может быть непройденной, так как после первого же прохода по этой дуге через конечное время она изменит своё состояние, что противоречит тому, что после времени T_1 все дуги перестают менять своё состояние. Однако других незаконченных дуг не бывает. Мы пришли к противоречию, следовательно, наше допущение не верно, и утверждение доказано.

Итак, мы доказали, что через конечное время генератор прекратит генерацию движков. Покажем, что в этот момент времени все дуги графа пройдены и закончены, т.е. помечены в регуляторах их начал как хорды или законченные прямые дуги, и, следовательно, прямые дуги образуют остов графа.

Допустим, это не так: существует незаконченная дуга. Поскольку все вершины достижимы из корня, должна существовать незаконченная дуга *ab*, начало которой, вершина *a*, принадлежит остову пройденной части графа. Тогда существует прямой путь *Q* из корня в вершину *a*. Вершина, из которой выходит незаконченная дуга, по определению незаконченная. Если вершина незаконченная, то входящая в неё прямая дуга также незаконченная. Отсюда следует что весь путь *Q* незаконченный: все его дуги и вершины. В частности, не закончено начало пути – корень, но это противоречит тому, что генератор прекратил генерацию движков. Мы пришли к противоречию, следовательно, наше допущение не верно, и утверждение доказано.

Теорема доказана.

4. Заключение

Дальнейшие исследования обхода недетерминированных графов коллективом автоматов возможны по нескольким направлениям.

Одно направление – это то или иное уточнение справедливой функции выбора, которое позволило бы оценить время работы алгоритма. Одним из таких уточнений является t-недетерминизм. В [[14]] для детерминированного случая ($t=1$) доказана оценка $\boldsymbol{O}(m+nD)$, где m – число дуг в исходном графе, n – число вершин, D – диаметр графа (длина максимального пути). В [[9]] для неавтоматного обхода (что эквивалентно обходу одним автоматом, в память которого помещается весь граф) t-недетерминированных графов доказана оценка $\boldsymbol{O}(bt^n)$, где b – ограничение на число дельта-дуг, выходящих из одной вершины. Можно предположить, что для коллектива автоматов оценка останется экспоненциальной. Это определяется тем, что для прохода пути длиной k мы должны k раз применить функцию выбора и каждый раз получить требуемую дугу, следовательно, для гарантированного прохода этого пути нам может понадобиться t^k попыток. Было бы интересно рассмотреть другие функции выбора, для которых оценка меньше экспоненциальной. В качестве простого примера можно рассмотреть граф с числом недетерминированных дуг, ограниченным константой k. В этом случае экспоненциальная составляющая времени обхода не превысит t^k.

Другое направление – снижение требований к обходу. Можно потребовать прохода не по каждой дуге, а по каждой дельта-дуге графа. В терминах тестирования это означает, что нужно попробовать каждое тестовое воздействие в каждом состоянии системы, но необязательно получать все возможные варианты поведения системы на эти тестовые воздействия. Такой обход в [[6]] называется Δ-обходом. Доказаны необходимые и достаточные условия существования Δ-обхода, которые базируются на понятии Δ-достижимости вершин. Говоря неформально, из вершины a Δ-достижима вершины b, если существует алгоритм движения из вершины a в вершину b, который гарантированно обеспечивает достижение вершины b при любой (даже не справедливой) функции недетерминированного выбора. Для того, чтобы Δ-обход можно было было выполнить без рестарта граф должен быть сильно-Δ-связным, т.е. каждая вершина должна быть Δ-достижима из каждой вершины. При наличии рестарта достаточно Δ-достижимости каждой вершины из корня графа, в этом случае Δ-обход назван Δ-покрытием. Неавтоматный Δ-обход можно выполнить за время $\boldsymbol{O}(nm)$. Было бы интересно попытаться модифицировать этот алгоритм для коллектива взаимодействующих автоматов.

Литература:

[1]. R. Milner. "A modal characterisation of observable machine-behaviour". In G. Astesiano & C. Bohm, editors: Proceedings CAAP 81, LNCS 112, Springer-Verlag, 1981, pp. 25-34.

[2]. van Glabbeek R.J. The linear time - branching time spectrum II; the semantics of sequential processes with silent moves. Proceedings CONCUR '93, Hildesheim, Germany, August 1993 (E. Best, ed.), LNCS 715, Springer-Verlag, 1993, pp. 66-81.

[3]. Y. Afek and E. Gafni, Distributed Algorithms for Unidirectional Networks, SIAM J. Comput., Vol. 23, No. 6, 1994, pp. 1152-1178.

[4]. И.Б. Бурдонов, А.С. Косачев, В.В. Кулямин, А.К. Петренко. "Подход UniTesK к разработке тестов" // Программирование, 2003 г., №6, с. 25-43.

[5]. И.Б. Бурдонов, А.С. Косачев, В.В. Кулямин. "Неизбыточные алгоритмы обхода ориентированных графов. Детерминированный случай" // Программирование, 2003 г., №5, с. 59-69.

[6]. И.Б. Бурдонов, А.С. Косачев, В.В. Кулямин. "Неизбыточные алгоритмы обхода ориентированных графов. Недетерминированный случай" // Программирование, 2004 г., №1, с. 2-17.

[7]. И.Б. Бурдонов. "Обход неизвестного ориентированного графа конечным роботом" // Программирование, 2004 г., № 4, с. 11-34.

[8]. И.Б. Бурдонов. "Проблема отката по дереву при обходе неизвестного ориентированного графа конечным роботом" // Программирование, 2004 г., № 6, с. 6-29.

[9]. И. Бурдонов, А. Косачев. "Полное тестирование с открытым состоянием ограниченно недетерминированных систем". Программирование, 2009, №6, стр. 3-18.

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

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

[12]. И. Бурдонов, А. Косачев. "Обход неизвестного графа коллективом автоматов". Труды Международной суперкомпьютерной конференции "Научный сервис в сети Интернет: все грани параллелизма". 2013, изд. МГУ, стр. 228-232.

[13]. И. Бурдонов, А. Косачев. "Обход неизвестного графа коллективом автоматов". Труды Института системного программирования РАН, том 26-2, 2014 г., стр. 43-86.

[14]. И. Бурдонов, А. Косачев. "Исследование графа взаимодействующими автоматами". Новые информационные технологии в исследовании сложных структур. Материалы 10-ой российской конференции с международным участием. 2014, изд. Томского госуниверситета, стр.47-48.

[15]. И. Бурдонов, А. Косачев. "Параллельные вычисления на графе". Программирование, 2015, №1 (в печати).

Graph Learning by a Set of Automata. The Nondeterministic Case

Igor Burdonov <igor@ispras.ru>,
Alexander Kosachev <kos@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, Russia, 109004.

Abstract. Graph learning with automata is a basic task in many applications. Among such applications is formal model-based verification and testing of software and hardware systems, as well as network exploration including Internet and GRID. The model of a system or a network, in the final analysis, is reduced to an oriented graph of transitions with properties to be examined. In the recent years, the size of the real-life systems and networks and, consequently, the size of their graph models continuously grows. The problems arise when the graph learning by a single automaton (computer) either requires unacceptable long time, or the graph does not fit in the single computer memory, or both. Therefore, there is a task of parallel and distributed graph learning. This task is formalized as graph learning by a set of automata. The basis for such learning is graph traversal (traversing all its arcs accessible from the initial node). Automata can be generated in the initial node, move along the arcs of the graph according with their orientation and exchange messages through independent (of the graph) communication network. The total memory of the automata is used for storing the descriptions of the already traversed part of the graph. To move from node along an arc outgoing from it, the automaton should somehow identify this arc: indicate its number. In our paper "Graph learning by a set of automata" we proposed an algorithm of such traversal for deterministic graphs. The task is more complicated if the graph is nondeterministic. In such graph, to one arc number correspond, generally speaking, multiple arcs, one of which is chosen nondeterministically for traversal. To make the traversal possible, the following should be guaranteed: in unlimited number of experiments, each outgoing arc with the given number can be traversed. We call such nondeterminism fair. This paper covers the solution of the problem of traversal of fairly nondeterministic graphs.

Keywords: nondeterministic graphs, graph learning, graph traversal, communicating automata, parallel processing, distributed systems, testing.

DOI: 10.15514/ISPRAS-2015-27(1)-4

For citation: Burdonov Igor, Kosachev Alexander. Graph Learning by a Set of Automata. The Nondeterministic Case. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 1, 2015, pp. 51-68 (in Russian). DOI: 10.15514/ISPRAS-2015-27(1)-4

References:

[1]. R. Milner. A modal characterisation of observable machine-behaviour. In G. Astesiano & C. Bohm, editors: Proceedings CAAP 81, LNCS 112, Springer-Verlag, 1981, pp. 25-34.

[2]. van Glabbeek R.J. The linear time - branching time spectrum II; the semantics of sequential processes with silent moves. Proceedings CONCUR '93, Hildesheim, Germany, August 1993 (E. Best, ed.), LNCS 715, Springer-Verlag, 1993, pp. 66-81.
[3]. Y. Afek and E. Gafni, Distributed Algorithms for Unidirectional Networks, SIAM J. Comput., Vol. 23, No. 6, 1994, pp. 1152-1178.
[4]. Bourdonov I.B., Kossatchev A.S., Petrenko A.K., Kuliamin V.V. The UniTesK Approach to Designing Test Suites. Programming and Computer Software, Vol. 29, No. 6, 2003, pp. 310-322.
[5]. Bourdonov I.B., Kossatchev A.S., Kuliamin V.V. Irredundant Algorithms for Traversing Directed Graphs: The Deterministic Case. Programming and Computer Software, Vol. 29, No. 5, 2003, pp. 245-258.
[6]. Bourdonov I.B., Kossatchev A.S., Kuliamin V.V. Irredundant Algorithms for Traversing Directed Graphs: The Nondeterministic Case. Programming and Computer Software, Vol. 30, No. 1, 2004, pp. 2-17.
[7]. Bourdonov I.B. Traversal of an Unknown Directed Graph by a Finite Robot. Programming and Computer Software, Vol. 30, No. 4, 2004, pp. 188-203.
[8]. Bourdonov I.B. Backtracking Problem in the Traversal of an Unknown Directed Graph by a Finite Robot. Programming and Computer Software, Vol. 30, No. 6, 2004, pp. 305-322.
[9]. Bourdonov I.B., Kossatchev A.S. Complete OpenState Testing of Limitedly Nondeterministic Systems. Programming and Computer Software, Vol. 35, No. 6, 2009, pp. 301-313.
[10]. Bourdonov I.B., Groshev S.G., Demakov A.V., Kamkin A.S., Kossatchev A.S, Sortov A.A. Parallel'noe testirovanie bol'shikh avtomatnykh modelej [Parallel testing of large automata models], Vestnik NNGU [Vestnik of UNN], №3920, 2011, pp. 187-193. (in Russian).
[11]. A. Demakov, A. Kamkin, A. Sortov. "High-Performance Testing: Parallelizing Functional Tests for Computer Systems Using Distributed Graph Exploration". Open Cirrus Summit 2011, Moscow.
[12]. Bourdonov I.B., Kossatchev A.S. Obkhod neizvestnogo grafa kollektivom avtomatov [Unknown graph traversing by automata group]. Trudy Mezhdunarodnoj superkomp'yuternoj konferentsii "Nauchnyj servis v seti Internet: vse grani parallelizma "(21-26 sentyabrya 2009 g., g. Novorossijsk) [The proceeding of Russian Supercomputer conference 'Scientific service of Internet' (2013, Novorossijsk)] – Moscow, MSU publ., 2013, pp. 228-232. (in Russian).
[13]. Bourdonov I.B., Kossatchev A.S. Obhod neizvestnogo grafa kollektivom avtomatov [Unknown graph traversing by automata group]. Trudy ISP RAN [The proceeding of ISP RAS], Vol. 26-2, 2014, pp. 43-86 (in Russian).
[14]. Bourdonov I.B., Kossatchev A.S. Issledovanie grafa vzaimodejstvuyushhimi avtomatami. [The graph learning by interacting automata] Novye informatsionnye tekhnologii v issledovanii slozhnykh struktur. Materialy 10-oj rossijskoj konferentsii s mezhdunarodnym uchastiem [New information technicks for complicated structure learning. The proceeding of 10-th Russian conference with international participation], Tomsk state university, 2014, pp. 47-48 (in Russian).
[15]. Bourdonov I.B., Kossatchev A.S. Parallel calculation on the graph. Programming and Computer Software, Vol. 41, No. 1, 2015 (in publication).

Мониторинг динамически меняющегося графа

Игорь Бурдонов <igor@ispras.ru>
Александр Косачев <kos@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. Исследование ориентированных графов является корневой задачей во многих приложениях. Такое исследование имеет особую специфику тогда, когда граф моделирует сеть связи, в том числе сеть интернета и GRID. Узел сети имеет локальную информацию о сети: он «знает» только о дугах, выходящих из этой вершины, но «не знает», куда (в какие вершины) эти дуги ведут. Узлы сети обмениваются сообщениями, передаваемыми по сетевым связям, которые в графе изображаются как дуги и играют роль каналов передачи сообщений. Исследование графа базируется на его обходе, когда сообщение проходит по каждой дуге графа. Пока не пройдена какая-то дуга, нет уверенности, что она не ведёт в ещё не исследованную часть графа. Обычно рассматривается обход графа с помощью одного сообщения, циркулирующего в сети. Обход выполняется быстрее, если выполнять его параллельно: по сети одновременно циркулирует не одно, а множество сообщений. В данной работе рассматривается параллельное исследование сильно связного графа, целью которого является не просто обход графа, а сбор полной информации о графе в каждой его вершине. Вторая особенность работы – исследование динамически меняющегося графа: его дуги могут исчезать, появляться или менять свои конечные вершины. Предлагается алгоритм работы автоматов, который обеспечивает сбор полной информации о графе в каждой вершине графа.

Ключевые слова: ориентированные графы, исследование графа, обход графа, взаимодействующие автоматы, параллельная работа, динамически меняющиеся графы.

DOI: 10.15514/ISPRAS-2015-27(1)-5

Для цитирования: Бурдонов Игорь, Косачев Александр. Мониторинг динамически меняющегося графа. Труды ИСП РАН, том 27, вып. 1, 2015 г., стр. 69-96. DOI: 10.15514/ISPRAS-2015-27(1)-5.

1. Введение

Исследование ориентированных графов является корневой задачей во многих приложениях. Такое исследование имеет особую специфику тогда, когда граф

моделирует сеть связи, в том числе сеть интернета и GRID. В этом случае узлы сети обычно «не знают» всего графа и имеют только локальную информацию о нём. В частности, узел сети, соответствующий вершине графа, может «знать» только о дугах, выходящих из этой вершины, и даже «не знать», куда (в какие вершины) эти дуги ведут. Исследование графа можно свести к задаче сбора полной информации о графе в каких-то или во всех узлах сети. Для решения этой задачи узлы сети могут обмениваться сообщениями, передаваемыми по сетевым связям, которые в графе изображаются как дуги.

Исследование графа базируется на его обходе, когда сообщение проходит по каждой дуге графа. Пока не пройдена какая-то дуга, нет уверенности, что она не ведёт в ещё не исследованную часть графа. Обход графа – это уже классическая задача обхода лабиринта. Эта задача нетривиальна, если граф ориентирован, то есть в лабиринте «улицы с односторонним движением».

Обход ориентированного сильно-связного графа требует времени порядка nm, где n – число вершин графа, а m – число дуг. Такое время обхода достигается многими хорошо известными алгоритмами: обход в глубину, обход в ширину, «жадный» алгоритм и др. [1,2,3].

В 1966 г. М.О. Рабин поставил задачу обхода ориентированного графа конечным автоматом [4]. Автомат на графе аналогичен машине Тьюринга: ячейке ленты соответствует вершина графа, а движение влево или вправо по ленте заменяется переходом по одной из дуг, выходящих из текущей вершины графа. Это эквивалентно движению по графу одного сообщения, которое пересылается по дугам графа автоматами, неподвижно «сидящими» в вершинах графа. Дуги графа играют роль каналов передачи сообщений. Автоматы в вершинах графа идентичны друг другу, но могут находиться в разных состояниях. Автомат, находящийся в вершине, посылает сообщение по одной из дуг, выходящих из этой вершины, и через какое-то время такое сообщение принимается автоматом в конце дуги, который, в свою очередь, может модифицировать это сообщение и послать его дальше.

На сегодняшний день наиболее быстрый алгоритм обхода графа с помощью одного сообщения предложен в [5], он имеет оценку $nm+n2\log\log n$. При повторном обходе, когда автоматы в вершинах находятся не в начальном состоянии, а в конечных состояниях после первого обхода, оценка уменьшается до $nm+n2l(n)$, где $l(n)$ — число логарифмирований, при котором достигается соотношение $1 \le \log(\log...(n)...) < 2$ [6]. Отличие от нижней оценки nm объясняется тем, что сообщению бывает нужно «вернуться» в начало только что пройденной дуги.

Обход графа можно выполнить быстрее, если по дугам графа может параллельно пересылаться не одно, а много сообщений. Оценка времени работы алгоритма зависит от числа сообщений, которые могут одновременно передаваться по дуге. Такое число называется *ёмкостью дуги* и обозначается

k. В [7] предложен алгоритм такого обхода, имеющий оценку порядка *n/k+D*, где *D* – длина максимального пути (маршрута без самопересечений) в графе. Правда, число состояний автоматов в вершинах графа зависит от числа *n* вершин графа и ограничения *s* на число дуг, выходящих из вершины, и имеет порядок *nDlogs*. Этот алгоритм строит на графе с выделенной начальной вершиной (корнем) структуру из прямого и обратного остовов графа: прямой остов ориентирован от корня, а обратный – к корню. В дальнейшем эта структура может использоваться для параллельного вычисления любой требуемой функции от мультимножества значений, записанных в вершинах графа (в памяти автоматов, находящихся в вершинах графа).

В данной работе мы также рассматриваем исследование графа с помощью многих параллельно передаваемых сообщений, но с двумя существенными отличиями от предыдущих работ по исследованию графа. Во-первых, целью такого исследования является не просто обход графа или построение какой-то структуры на графе, а сбор полной информации о графе в каждой его вершине. Во-вторых, мы рассматриваем граф, который может динамически изменяться: какие-то его дуги могут исчезать, появляться или менять свою конечную вершину.

2. Постановка задачи

Рассматривается ориентированный граф с *n* вершинами и *m* дугами, в котором вершины не меняются, а дуги могут меняться со временем. Будем предполагать, что в каждый момент времени граф сильно связный. Кроме того, будем считать, что если изменения в графе прекращаются, то с этого момента времени длина максимального пути в графе не превышает *D*.

В вершинах графа находятся автоматы, которые могут получать специальные сигналы от графа и обмениваться между собой сообщениями, передаваемыми по дугам графа в направлении их ориентации. Каждая вершина имеет уникальный (среди вершин) *идентификатор вершины*. Будем считать, что существует *пустой* идентификатор, отличный от идентификатора любой вершины. Автомат может узнать идентификатор своей вершины с помощью примитива «*Дай идентификатор*». Для краткости везде, где это не приведёт к недоразумениям, мы будем вместо «автомат вершины» писать просто «вершина».

Все дуги, выходящие из вершины, перенумерованы, начиная с номера 1 до наибольшего номера, не превосходящего некоторого числа *s*. Очевидно, $s \le m$. Дуга однозначно идентифицируется парой (идентификатор начала дуги, номер дуги), которую мы будем называть *идентификатор дуги*. Число дуг *m* – это число различных идентификаторов дуг в графе.

Автомат, посылая сообщение по дуге, указывает её номер. Будем говорить, что дуга занята, если по ней передаётся сообщение. Иначе дуга свободна. Посылать сообщение можно только по свободной дуге. В самом начале все

дуги свободны. Когда сообщение передано по дуге, начало дуги извещается об этом сигналом *освобождения* дуги с параметром номер дуги.

Дуги графа могут изменяться следующим образом:

- Дуга может появиться, о чём начало дуги извещается сигналом *появления* дуги с параметром номер дуги.
- Дуга может исчезнуть. Если по дуге передавалось сообщение, то оно пропадает, и в этом случае начало дуги извещается сигналом *исчезновения* дуги с параметром номер дуги. Заметим, что если по дуге никакого сообщения не передавалось, то сигнала исчезновения дуги не будет; однако если вершина попытается послать сообщение по исчезнувшей дуге, сообщение не будет передано, а вершина получит сигнал исчезновения дуги.
- Дуга может поменять свой конец; в этом случае никаких сигналов не предусмотрено.

Из-за того, что дуги могут исчезать и появляться, номера дуг, выходящих из одной вершины и имеющихся в данный момент времени, могут не идти подряд, т.е. не образовывать отрезок натурального ряда, но в любом случае они располагаются на отрезке от 1 до s. Хотя конец дуги может меняться, в данный момент времени у дуги может быть только один конец; если дуга исчезла, считается, что её конец «пустой», то есть имеет пустой идентификатор. В каждый момент времени дуга описывается парой (идентификатор дуги, идентификатор конца дуги) или, что то же самое, тройкой (идентификатор начала дуги, номер дуги, идентификатор конца дуги). *Временем существования* дуги будем называть время между изменениями дуги, то есть время от появления дуги или изменения её конца до исчезновения дуги или изменения её конца.

Итак, если не считать идентификатора вершины, который автомат получает с помощью примитива «Дай идентификатор», входными символами автомата, находящегося в вершине, являются сигналы от дуг, выходящих из вершины, и сообщения, получаемые по дугам, входящим в вершину. Срабатывание автомата – это обработка одного такого входного символа. Мы будем предполагать, что входные символы обрабатываются автоматом в порядке их возникновения. Это означает, что существует очередь на обработку автоматом входных символов. Если во время обработки автоматом очередного сигнала или сообщения поступает новый входной символ, он ставится в конец этой очереди. Одновременно поступающие входные символы ставятся в конец очереди в произвольном порядке. При этом сигнал освобождения дуги вырабатывается не тогда, когда сообщение ставится в очередь входных символов, а тогда, когда оно выбирается из этой очереди автоматом конца дуги. Поэтому в очереди входных символов может быть не более одного сообщения для каждой входящей дуги. Если сигнал от некоторой выходящей

дуги вырабатывается в тот момент времени, когда предыдущий сигнал от этой дуги ещё находится в очереди входных символов, то новый сигнал замещает собой старый сигнал. Можно отметить, что таким новым сигналом может быть только сигнал появления дуги. Поэтому в очереди входных символов может быть не более одного сигнала для каждой выходящей дуги. Тем самым, длина очереди входных символов автомата не превосходит степени вершины, то есть не больше $2m$ ($2m$ достигается для $n=1$, когда все дуги – петли, поскольку петля считается два раза: как входящая – для сообщений и как выходящая – для сигналов).

Мы ставим задачу разработки алгоритма работы автоматов, который обеспечивает сбор полной информации о графе в каждой вершине графа. Такая информация может быть задана как набор описаний всех дуг. Описание дуги содержит идентификатор дуги и идентификатор её конца. Понятно, что если граф постоянно меняется, то мы не можем гарантировать, что описания текущего состояния всех его дуг отражены во всех его вершинах: сообщения о последних изменениях дуг просто «не дошли» до некоторых вершин. Поэтому требуется только, чтобы через время T_0 после изменения дуги все вершины графа «узнали» об этом или более позднем изменении дуги. Если после данного изменения дуга больше не меняется, по крайней мере, в течение времени T_0, то во всех вершинах графа будет одинаковое (и верное) описание этой дуги.

Для того чтобы время T_0 было конечным, нужно, чтобы сообщения могли распространяться по графу, доходя до каждой вершины. Для этого нужно, чтобы время x пересылки сообщения по дуге и время y срабатывания автомата были конечными. Кроме того, время z существования дуги должно быть достаточно велико, чтобы по дуге успевало пройти хотя бы одно сообщение. Такими «долгоживущими» дугами могут быть не все дуги: достаточно, чтобы в каждый момент времени «долгоживущие» дуги порождали сильно связный суграф (подграф, содержащий все вершины графа). Это усиление требования сильно связности графа в каждый момент времени.

Для ограниченности времени T_0 необходимо, чтобы x и y были ограничены сверху: $x \le X$ и $y \le Y$, а z – снизу: $z \ge Z$. Выразим эту нижнюю границу Z через X и Y. Сообщение можно посылать по дуге сразу после того, как она появилась или освободилась. Однако в этот момент времени сигнал появления или освобождения дуги ещё только ставится во входную очередь символов автомата начала дуги, длина которой может достигать $2m$. Поэтому сообщение посылается по дуге не сразу, а через время, которое может достигать $2mY$. Следовательно, сообщение дойдёт до конца дуги и будет поставлено во входную очередь символов автомата конца дуги через время, которое может достигать $2mY+X$, если в течение этого времени дуга не менялась. Поэтому будем предполагать, что нижняя граница времени существования «долгоживущих» дуг $Z \ge 2mY+X$.

Мы предложим алгоритм решения поставленной задачи, а время T_0 будем оценивать для простоты в предположении, что временем срабатывания автомата можно пренебречь, то есть $Y=0$, время пересылки по дуге не превосходит 1 такта, то есть $X=1$, а время существования «долгоживущей» дуги может быть минимально, то есть $Z=2mY+X=1$. Кроме того, оценим время T_1, за которое будет выполнен сбор полной информации о графе в каждой вершине графа после того, как вообще прекратились все изменения в графе. Очевидно, что $T_1 \leq T_0$, однако прекращение изменений в графе позволяет существенно уменьшить T_1 по сравнению с T_0.

3. Идея алгоритма

Идея алгоритма заключается в том, чтобы каждый раз, когда обнаруживается изменение дуги, корректировать описание дуги и рассылать сообщение с этим описанием «веером» по всем дугам графа для того, чтобы каждая вершина его получила. Мы будем говорить, что рассылается описание дуги. Первое описание дуги создаётся в начале дуги по сигналу появления дуги при первом появлении этой дуги. Дуга, которая имеется в графе с самого начала, считается появляющейся в начальный момент времени.

Как выполнить рассылку описания дуги p «веером»? Для этого вершина, принимая по некоторой входящей дуге q описание дуги p первый раз, рассылает его по всем выходящим дугам, а повторные принимаемые описания дуги p игнорирует. Однако дуга может меняться несколько раз, поэтому если вершина принимает описание более позднего состояния дуги, чем то, о котором она уже «знает», она не должна игнорировать принимаемое описание. Для этого состояния дуги, соответствующие её последовательным изменениям, перенумеровываются. Этот номер называется *рангом* дуги и хранится в описании дуги. Вершина игнорирует принимаемое описание дуги только в том случае, если в нём ранг дуги меньше или равен рангу дуги в её описании, хранящемся в вершине. Если же ранг дуги в принимаемом описании больше, то принимаемое описание дуги замещает собой описание дуги в вершине, и далее вершина должна разослать это обновлённое описание дуги по всем выходящим дугам.

Каким образом обнаруживается изменение дуги? Схема работы алгоритма показана на рис.1.

О появлении дуги извещается начало дуги с помощью сигнала появления дуги. В этот момент времени конец дуги ещё не известен, поэтому в описании дуги идентификатор конца дуги делается пустым. По дуге посылается её описание, причём указывается идентификатор дуги, по которой посылается описание. Когда это описание дойдёт до конца дуги, в описание дуги будет помещён идентификатор конца дуги и далее полученное описание дуги рассылается «веером». Когда это описание дойдёт до начала дуги, там тоже появится непустой идентификатор конца дуги.

Для того чтобы отслеживать изменение конца дуги, по дуге периодически посылается её описание. Если конец дуги не меняется, идентификатор конца дуги в посылаемом описании совпадает с идентификатором вершины, получившей описание, то есть с идентификатором конца дуги. В этом случае ничего делать не нужно. Если же конец дуги сменился, то идентификатор конца дуги в посылаемом описании будет отличаться от идентификатора вершины, получившей опрос, т.е. от идентификатора нового конца дуги. В этом случае новый конец дуги рассылает «веером» описание дуги со своим собственным идентификатором как идентификатором конца дуги. Когда такое описание получит начало дуги, оно поменяет идентификатор конца дуги в своём описании дуги.

Об исчезновении дуги извещается начало дуги с помощью сигнала исчезновения дуги. Это происходит тогда, когда по дуге передаётся сообщение. Начало дуги делает пустым идентификатор конца дуги в своём описании дуги и рассылает это описание «веером».

Для того, чтобы описанная схема алгоритма работала, нужна правильная процедура изменения ранга дуги. Смысл этой процедуры в том, чтобы в описании более нового состояния дуги ранг был больше, чем во всех других имеющихся описаниях дуги.

Пусть в начале дуги в описании этой дуги указан максимальный ранг j. Если идентификатор конца дуги в этом описании неправильный (пустой после появления дуги или старый после изменения конца дуги), то конец дуги, получив по дуге такое её описание и обновив его (добавив свой идентификатор как идентификатор конца дуги), должен увеличить на 1 ранг дуги. Далее обновлённое описание дуги с максимальным рангом $j+1$ будет распространяться по графу «веером», в том числе и до начала дуги.

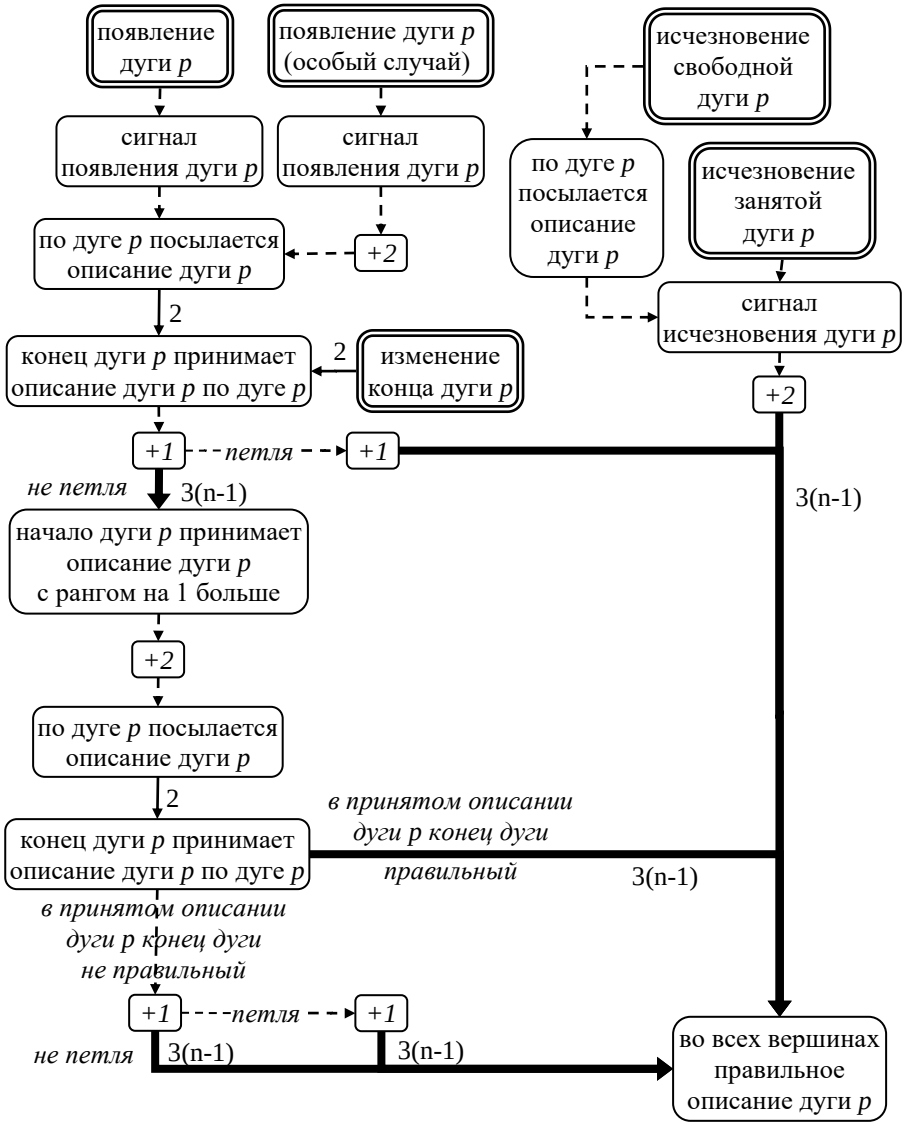
Однако здесь возникает проблема многократного изменения конца дуги. В этом случае уже не в одной, а в нескольких вершинах может оказаться максимальный ранг $j+1$. Поскольку только одна из этих вершин в данный момент времени является текущим концом дуги, нужно, чтобы по графу распространялось описание дуги именно с текущим концом дуги, а не с каким-то другим, более ранним. Для решения этой проблемы начало дуги, получив описание с рангом $j+1$ от какого-нибудь, возможно, более раннего конца дуги, ещё раз увеличивает ранг дуги на 1 так, что он становится равным $j+2$. Но теперь такой ранг не просто максимальный: он имеется только в начале дуги. Если идентификатор конца дуги в этом описании правильный (т.е. начало дуги получило описание с рангом $j+1$ от текущего конца дуги), то это описание с максимальным рангом $j+2$ будет распространяться по всему графу, пока не произойдёт следующего изменения дуги. Если же идентификатор конца дуги в описании не правильный (т.е. начало дуги получило описание с рангом $j+1$ от более раннего конца дуги), то это описание с максимальным рангом $j+2$ снова по дуге попадёт в текущий конец

дуги, где ранг ещё раз будет увеличен на 1. Теперь в описании в конце дуги идентификатор конца правильный, а ранг, равный *j+3*, не просто максимальный: он имеется только в конце дуги. Это описание будет распространяться по всему графу, пока не произойдёт следующего изменения дуги. В частности, такое описание с рангом *j+3* попадёт в начало дуги, где ранг снова будет увеличен до *j+4*, а идентификатор конца дуги останется правильным.

Разумеется, во время выполнения этой процедуры возможно исчезновение дуги. В начале дуги идентификатор конца дуги в её описании станет пустым, что правильно после исчезновения дуги. Но нужно ещё увеличить ранг дуги, чтобы он стал больше, чем в любой другой вершине. Поскольку по приведённому выше описанию процедуры изменения ранга видно, что ранг дуги в начале дуги либо максимальный, либо на *1* меньше максимального, при исчезновении дуги в начале дуги достаточно увеличить ранг дуги на *2*.

Что делать с рангом дуги, если сигнал исчезновения дуги пропущен, поскольку был замещён последующим сигналом появления дуги? Пропущенный сигнал исчезновения дуги означает и пропущенное состояние дуги – «дуга исчезла». Этому состоянию дуги соответствует описание с пустым идентификатором конца дуги. Поэтому, если при появлении дуги идентификатор конца дуги в её описании в начале дуги не пустой, нужно в начале дуги не только сделать идентификатор конца дуги пустым, но и увеличить ранг так, чтобы он стал больше ранга дуги в любой другой вершине, то есть увеличить на 2. Это как раз и будет соответствовать пропущенному состоянию дуги. Если же при появлении дуги идентификатор конца дуги в её описании в начале дуги пустой, в начале дуги достаточно сделать идентификатор конца дуги пустым, а ранг увеличивать не нужно, поскольку описание дуги в её начале уже соответствует пропущенному состоянию дуги.

На рис.1 выделен также случай, когда дуга является петлёй, то есть конец дуги совпадает с её началом. В этом случае не нужно сначала увеличить ранг на *1* в конце дуги, затем распространить это описание по графу до начала дуги, после чего ещё раз увеличить ранг на *1*. Вместо этого нужно сразу увеличить ранг на *2*.



Пунктирная линия – переход за 0 тактов, сплошная – за ≤ 2 такта, жирная – за ≤ 3(n-1) тактов. В двойной рамке – изменения дуги. Особый случай появления дуги: конец дуги известен, а сигнал исчезновения пропущен.

Рис. 1. Схема работы алгоритма

Также на рис.1 показаны переходы, которые происходят либо за нулевое время (при условии, что временем срабатывания автомата мы пренебрегаем), либо за время не более *2* тактов, либо за время не более *3(n-1)* тактов. На рисунке видно, что максимальная длительность цепочки переходов от изменения дуги до распространения описания этой дуги по состоянию после этого или последующих изменений дуги равна *2+3(n-1)+3(n-1)=6n-4*. Если добавить ещё один такт, то информация о дуге окажется гарантированно правильной также и во всех описаниях, передаваемых по дугам. Строгое доказательство этих оценок приведено ниже в разделе 5.1.

До сих пор мы молчаливо предполагали, что распространению по графу описания дуги ничто не мешает, кроме описаний этой же дуги с большими рангами, что соответствует более поздним состояниям дуги. Однако такому распространению могут препятствовать также изменения дуг. По «короткоживущим» дугам сообщения могут просто не успевать проходить, поэтому вся надежда на «долгоживущие» дуги, которые не меняются, по крайней мере, в течение 1 такта. В постановке задачи предполагается, что в каждый момент времени суграф, порождённый «долгоживущими» дугами, сильно связен.

Однако для того, чтобы воспользоваться этими предположениями, нужно успевать передавать по «долгоживущей» дуге все требуемые описания. А это означает, что в одном сообщении, посылаемом по некоторой дуге, нам нужно объединить все описания дуг, которые нужно передать по этой дуге. Что это за описания? Если дуга q появляется или меняет свой конец, то по ней нужно передавать все имеющиеся в начале дуги описания, поскольку возможен случай, когда после такого изменения дуги q в её конец ещё не передавались эти описания. Если дуга q исчезает, то по ней, конечно, ничего передавать нельзя. И только в том случае, когда не происходит изменение дуги q, по ней можно было бы передавать только те описания, которые по ней ещё не передавались.

Однако когда мы посылаем сообщение по дуге, содержащее лишь часть хранящихся в начале дуги описаний, мы не знаем, сменит ли дуга свой конец до завершения передачи сообщения. Если через время $x \le 1$ тактов после посылки по дуге сообщения дуга меняет свой конец, то дуга ещё не освобождается; освобождение может произойти через *1* такт, после чего нам следует послать по дуге второе сообщение со всеми (или со всеми остальными) описаниями. Это второе сообщение может передаваться тоже 1 такт, поэтому в конец дуги все описания могут попасть через 2 такта. Однако время существования «долгоживущей» дуги ограничено снизу одним тактом, а не двумя.

Но даже если бы мы увеличили эту нижнюю границу времени существования «долгоживущей» дуги до 2 тактов, это не помогло бы. Дело в том, что о смене конца дуги узнаёт лишь автомат в конце дуги, а в начало дуги информация об

этом попадает через длительное время в результате распространения описания по графу. В течение этого длительного времени дуга может много раз менять свой конец, и каждый раз по дуге в её очередной конец попадёт только часть описаний.

Эти эвристические соображения приводят к выводу, что по дуге нужно посылать сообщение, содержащее сразу все описания, хранящиеся в начале дуги.

4. Описание алгоритма

4.1 Сообщения

Сообщение содержит:

- $I^{\sim}$ – идентификатор начала дуги, по которой сообщение передаётся,
- $i^{\sim}$ – номер дуги, по которой сообщение передаётся,
- $l^{\sim}$ – длина массива описаний дуг,
- $L^{\sim}[0..l^{\sim}-1]$ – массив описаний дуг.

Описание e дуги содержит: 1) идентификатор начала дуги, обозначаемый $N(e)$, 2) номер дуги, обозначаемый $i(e)$, 3) идентификатор конца дуги (быть может, пустой), обозначаемый $K(e)$ и 4) ранг дуги, обозначаемый $r(e)$. Идентификатор дуги – $(N(e),i(e))$.

4.2 Память автомата

Память автомата содержит:

- управляющее состояние (начальное или рабочее),
- I – идентификатор вершины,
- l – длина массива описаний дуг, хранящихся в вершине,
- $L[0..l-1]$ – массив описаний дуг.

4.3 Работа автомата

4.3.1 Начальное состояние

Автомат опрашивает идентификатор вершины и запоминает его в I. Инициализируется: $l:=0$. Автомат переходит в рабочее состояние.

4.3.2 Рабочее состояние

В этом состоянии автомат ожидает любого сигнала и любого сообщения.

4.3.2.1 Сигнал появления дуги, параметр: номер дуги *i*.

В массиве L ищется описание e с идентификатором дуги (I,i).

Если такого описания нет, то в массив L вставляется описание $e = (I,i,\varepsilon,0)$, $l:=l+1$.

Если $K(e)\neq\varepsilon$, то $r(e):=r(e)+2$ и $K(e):=\varepsilon$.

В любом случае по дуге i посылается сообщение (I,i,l,L).

4.3.2.2 Сигнал исчезновения дуги, параметр: номер дуги *i*.

В массиве L ищется описание дуги e с идентификатором дуги (I,i).

$r(e):=r(e)+2$ и $K(e):=\varepsilon$.

4.3.2.3 Сигнал освобождения дуги, параметр: номер дуги *i*.

По дуге i посылается сообщение (I,i,l,L).

4.3.2.4 Сообщение

Просматривается массив $L^{\sim}$ описаний дуг в принятом сообщении. Для каждого $e^{\sim}=L^{\sim}(j^{\sim})$, где $j^{\sim}=1..l^{\sim}-1$ выполняются следующие действия.

1. В массиве L ищется описание дуги e с тем же идентификатором дуги, что в $e^{\sim}$. Если такого описания нет, то в массив L вставляется описание $e^{\sim}$, $l:=l+1$.

   Далее п.2, п.3, п.4 или п.5 в зависимости от условий этих пунктов.
2. Дуга e – это петля, по которой пришло сообщение, т.е. идентификатор дуги в e равен $(I^{\sim},i^{\sim})$ и $I^{\sim}=I$.

   Если $K(e)\neq I$, то $r(e):=r(e)+2$ и $K(e):=I$.
3. Дуга e – не петля (т.е. $I^{\sim}\neq I$) и это дуга, по которой пришло сообщение, т.е. идентификатор дуги в e равен $(I^{\sim},i^{\sim})$.

   Если $r(e^{\sim})\geq r(e)$ и $K(e^{\sim})=I$, то $r(e):=r(e^{\sim})$ и $K(e):=I$.

   Если $r(e^{\sim})\geq r(e)$ и $K(e^{\sim})\neq I$, то $r(e):=r(e^{\sim})+1$ и $K(e):=I$.
4. Дуга e – не петля (т.е. $I^{\sim}\neq I$), это не дуга, по которой пришло сообщение, т.е. идентификатор дуги в e не равен $(I^{\sim},i^{\sim})$, и это выходящая дуга ($N(e)=I$).

   Если $r(e^{\sim})>r(e)$, то $r(e):=r(e^{\sim})+1$ и $K(e):=K(e^{\sim})$.
5. Дуга e – не петля (т.е. $I^{\sim}\neq I$), это не дуга, по которой пришло сообщение, т.е. идентификатор дуги в e не равен $(I^{\sim},i^{\sim})$, и это не выходящая дуга ($N(e)\neq I$).

   Если $r(e^{\sim}) > r(e)$, то $r(e):=r(e^{\sim})$, $K(e):=K(e^{\sim})$.

5. Оценка времени и памяти

5.1 Время работы алгоритма

Пусть выбрана дуга p и некоторый момент времени t_0, который будем называть *началом отсчёта*. Мы будем рассматривать распространение по графу информации о дуге p после момента времени t_0. До первого появления дуги p во всех вершинах отсутствует описание дуги p. Такое «отсутствие описания» распространять не требуется, поэтому можно считать, что t_0 – момент времени после первого появления дуги p, включая обработку автоматом сигнала появления дуги. Текущий момент времени обозначим через t, $t \geq t_0$.

Начало дуги p обозначим через a, а её текущий конец (конец в момент времени t) – через b. Если дуги нет, будем считать, что $b=\varepsilon$. Через B обозначим множество концов, которые дуга p имела в интервале времени $[t_0,t]$, включая ε, если дуга p исчезала в этом интервале времени. Заметим, что множество B может только расти, за исключением случая выбора другого начала отсчёта.

Обозначим: V – множество идентификаторов вершин, E – множество идентификаторов дуг графа.

Для краткости ранг дуги p в её описании в вершине x будем называть *рангом в вершине* x и обозначать r_x. Если такое описание отсутствует, то будем считать, что $r_x = -1$. *Рангом на дуге* q будем называть ранг дуги p в её описании в сообщении, которое передаётся или (если сейчас по дуге сообщение не передаётся) последний раз передавалось по дуге q, и обозначать r_q. Если по дуге q ещё не передавалось ни одного сообщения или ни в одном сообщении не было описания дуги p, то будем считать, что $r_q = -1$. Просто *рангом* будем называть ранг в вершине или ранг на дуге. *Максимальным рангом* будем называть такой ранг, что во всех вершинах и на всех дугах ранг не больше.

Идентификатор конца дуги p в её описании в вершине x будем называть *концом в вершине* x, и обозначать K_x. Если такое описание отсутствует, то будем считать, что K_x не определено. *Концом на дуге* q будем называть идентификатор конца дуги p в её описании в сообщении, которое передаётся или (если сейчас по дуге сообщение не передаётся) последний раз передавалось по дуге q, и обозначать K_q. Если по дуге q ещё не передавалось ни одного сообщения или ни в одном сообщении не было описания дуги p, то будем считать, что K_q не определено. Просто *концом* будем называть конец в вершине или на дуге. Будем говорить, что конец K_z *правильный*, если $K_z \in B$, иначе конец K_z – *неправильный*.

Состояние графа относительно выбранной дуги p в текущий момент времени t полностью описывается величинами V, E, a, t_0, b, B, K_z, r_z, где $z \in V \cup E$. При этом V, E, a не зависят от t. Для события, которое может произойти в момент

времени t и изменить состояние графа, обозначим значения величин после события через, соответственно, $t`_0$, $b`$, $B`$, $K`_z$, $r`_z$, где $z \in V \cup E$. Также обозначим: $r = r_a$, $r` = r`_a$, $K = K_a$, $K` = K`_a$.

Из правил изменения дуг и описания алгоритма следует следующий перечень событий, меняющих состояние графа. Некоторые из этих событий являются последовательностью «микрособытий», приводящих к обработке автоматом входного символа: сигнала или принимаемого сообщения. Например, повторное появление дуги p – это последовательность следующих «микрособытий»: исчезновение дуги, выработка сигнала исчезновения дуги и постановка его в очередь входных символов автомата, появление дуги, выработка сигнала появления дуги и замещение им в очереди входных символов автомата находящегося там сигнала исчезновения, обработка автоматом сигнала появления. Важно отметить, что в любом случае событие происходит в течение времени, сравнимого с временем срабатывания автомата, которым мы пренебрегаем; иными словами, мы можем считать, что событие происходит «мгновенно». Мы рассматриваем состояние графа между событиями, но не «внутри события», то есть не между «микрособытиями» внутри одного события. В перечне событий мы указываем для каждого события только те значения величин, определяющих состояние графа, которые меняются.

1. *Смена начала отсчёта* – выбор текущего момента времени как начала отсчёта.

   $t`_0 = t$, $B` = \{b\}$.

2. *Появление* дуги p с концом $x \neq \varepsilon$, включая обработку автоматом сигнала появления. Если в начале отсчёта t_0 дуги p не было, то $\varepsilon \in B$, следовательно, $\varepsilon \in B`$. Если в начале отсчёта $t_0$ дуга $p$ была, то это повторное появление дуги, перед которым должно было быть исчезновение дуги после начала отсчёта, даже если сигнал исчезновения замещался в очереди входных символов автомата сигналом появления, и поэтому тоже $\varepsilon \in B`$.

   $b` = x$, $B` = B \cup \{\varepsilon, x\}$. Если $K \neq \varepsilon$, то $r` = r+2$ и $K` = \varepsilon$. В любом случае $K`_p = K`$, $r`_p = r`$.

3. *Исчезновение* дуги p, если сигнал исчезновения не замещается последующим сигналом появления, включая обработку автоматом сигнала исчезновения.

   $b` = \varepsilon$, $B` = B \cup \{\varepsilon\}$, $r` = r+2$, $K` = \varepsilon$.

4. *Смена конца* дуги p на конец $x \neq \varepsilon$.

   $b` = x$, $B` = B \cup \{x\}$.

5. *ab-событие* – поступление сообщения по дуге p в её конец b, когда $b = a$ (дуга p является петлёй) и $K \neq a$, включая срабатывание автомата по приёму этого сообщения. Заметим, что сначала автомат принимает

сообщение, приходящее по дуге-петле, и только после этого вырабатывается сигнал освобождения. Заметим, что если $K = a$, то состояние графа не меняется.

$r`=r+2$, $K`=a$.

6. *b-событие* – поступление сообщения по дуге p в её конец $b \neq \varepsilon$, когда $a \neq b$ (дуга p не является петлёй), включая срабатывание автомата по приёму этого сообщения.

   Если $r_p \geq r_b$ и $K_p=b$, то $r`_b=r_p$ и $K`_b=b$.

   Если $r_p \geq r_b$ и $K_p \neq b$, то $r`_b=r_p+1$ и $K`_b=b$.

7. *a-событие* – поступление сообщения по дуге $q \neq p$ в вершину a, когда $r_q > r$, включая срабатывание автомата по приёму этого сообщения. Заметим, что если $r_q \leq r$, то состояние графа не меняется.

   $r`=r_q+1$ и $K`=K_q$.

8. *Приём* – поступление сообщения по дуге $q \neq p$ в вершину $x \neq a$, когда $r_q > r_x$, включая срабатывание автомата по приёму этого сообщения. Заметим, что если $r_q \leq r_x$, то состояние графа не меняется.

   $r`_x=r_q$, $K`_x=K_q$.

9. *Посылка* – освобождение дуги q или появление дуги $q \neq p$, имеющей начало в x, включая срабатывание автомату по сигналу освобождения или появления дуги, соответственно.

   $r`_q=r_x$, $K`_q=K_x$.

Определим следующие состояния графа:

1. $b \neq \varepsilon \;\&\; \exists z \in V\; r_z > r \;\&\; K_z \notin B$.

2. $b \neq \varepsilon \;\&\; \forall z \in V \cup E\; r_z \leq r \;\&\; K \notin B$.

3(w). $\exists z \in V\; r_z \geq w \;\&\; \forall z \in V \cup E\; (r_z \geq w \Rightarrow K_z \in B)$.

4. $\forall z \in V \cup E\; K_z \in B$.

Состояние 3(w) определяется для некоторого ранга w. Понятно, что если $w`>w$ и $\exists z \in V\; r_z \geq w`$, то состояние 3(w) является также состоянием 3($w`$). Состояние 4 является, на самом деле, подсостоянием состояния 3(w), когда во всех вершинах и на всех дугах ранг не меньше w.

Лемма 1. Ранг дуги в вершине не уменьшается: $\forall x \in V\; r`_x \geq r_x$.

Доказательство. Непосредственно следует из описания событий.

Лемма доказана.

Лемма 2. Ранг на дуге не превосходит ранга в начале дуги.

Доказательство. Как видно из описания событий, ранг на дуге может измениться только при посылке сообщения по этой дуге (события *появление* и *посылка*), а в этот момент времени ранг на дуге равен рангу в начале дуги. Поскольку по лемме 1 ранг в начале дуги не уменьшается, ранг на дуге не превосходит ранга в начале дуги.

Лемма доказана.

Лемма 3.

1) Ранги r и r_p чётны.

2) Любой ранг не превосходит ранга в вершине a плюс 1: $\forall z \in V \cup E\ r_z \le r+1$.

3) Если ранг совпадает с рангом вершины a, то концы также совпадают: $\forall z \in V \cup E\ (r_z = r \Rightarrow K_z = K)$.

4) Если событие увеличивает ранг $r`>r$, то $r`=r+2$ и после события ранг $r`$ максимальный и имеется только в вершине a и, быть может, на дуге p.

Доказательство. Будем вести доказательство индукцией по событиям. Как видно из описания алгоритма, после первого появления дуги p имеем: $r_p=r=0$, $K_p=K=\varepsilon$, а для $z \ne a$ и $z \ne p$: $r_z=-1$, следовательно, утверждения леммы верны.

Пусть до события утверждения леммы верны. Утверждения 1 и 4 могли бы быть нарушены только в результате событий *появление*, *исчезновение*, *ab-события* или *a-события*, которые меняют r или r_p. Утверждения 2 и 3 дополнительно могли бы быть нарушены в результате *b-события*, меняющего r_b или K_b.

Появление. Если $K \ne \varepsilon$, то $r`=r+2$, $K`=\varepsilon$. В любом случае $K`_p=K`$, $r`_p=r`$. По предположению шага индукции для утверждения 1 r чётно, следовательно, $r` = r$ или $r` = r+2$ и $r`_p=r`$ тоже чётны, поэтому утверждение 1 остаётся верным. По предположению шага индукции для утверждения 2 $r_z \le r+1$, следовательно, для $z \ne a$ и $z \ne p$ имеем $r`_z = r_z \le r+1 \le r`+1$, а $r`_p=r`$, поэтому утверждение 2 остаётся верным. Если $K = \varepsilon$, то $r`_p=r`$, $K`_p=K`$, а для $z \ne p$ ранг и конец не меняются, поэтому утверждение 3 остаётся верным и, поскольку $r`=r$, утверждение 4 тоже верно. Если $K \ne \varepsilon$, то $r`_p=r`=r+2$, $K`_p=K`$, а для $z \ne a$ и для $z \ne p$ ранг не меняется. Поскольку по предположению шага индукции для утверждения 2 $r_z \le r+1$, ранг $r`$ после события максимальный и имеется только в вершине a и на дуге p, поэтому утверждение 3 остаётся верным и утверждение 4 тоже верно.

Исчезновение. Имеем $r`=r+2$, $K`=\varepsilon$. По предположению шага индукции для утверждения 1 r чётно, следовательно, $r`=r+2$ тоже чётно, поэтому утверждение 1 остаётся верным. По предположению шага индукции для утверждения 2 $r_z \le r+1$, следовательно, для $z \ne a$ имеем $r`_z = r_z \le r+1 < r+2 = r`$, поэтому утверждение 2 остаётся верным. Ранг $r`$ после события максимальный и имеется только в вершине a, поэтому утверждение 3 остаётся верным и утверждение 4 тоже верно.

ab-событие. Имеем $r`=r+2$, $K`=a$. По предположению шага индукции для утверждения 1 r чётно, следовательно, $r`=r+2$ тоже чётно, поэтому утверждение 1 остаётся верным. По предположению шага индукции для утверждения 2 $r_z \le r+1$, следовательно, для $z \ne a$ имеем $r`_z = r_z \le r+1 < r+2 = r`$, поэтому утверждение 2 остаётся верным. Ранг $r`$ после события максимальный

и имеется только в вершине a, поэтому утверждение 3 остаётся верным и утверждение 4 тоже верно.

a-событие ($r_q > r$). Имеем $r`=r_q+1$ и $K`=K_q$. Поскольку по предположению шага индукции для утверждения 2 $r_q \le r+1$, имеем $r_q = r+1$ и $r`=r+2$. По предположению шага индукции для утверждения 1 $r$ чётно, следовательно, $r`=r+2$ тоже чётно, поэтому утверждение 1 остаётся верным. Также по предположению шага индукции для утверждения 2 $r_z \le r+1$, следовательно, для $z \ne a$ имеем $r`_z = r_z \le r+1 < r+2 = r`$, поэтому утверждение 2 остаётся верным. Ранг $r`$ после события максимальный и имеется только в вершине a, поэтому утверждение 3 остаётся верным и утверждение 4 тоже верно.

b-событие. 1) Если $r_p \ge r_b$ и $K_p=b$, то $r`_b=r_p$ и $K`_b=b$. 2) Если $r_p \ge r_b$ и $K_p \ne b$, то $r`_b=r_p+1$ и $K`_b=b$. Поскольку *b-событие* не меняет r и r_p, утверждение 1 остаётся верным. По лемме 2 $r_p \le r$. Следовательно, поскольку $r`_b \le r_p+1$, имеем $r`_b \le r+1$, и, поскольку $r`=r$, $r`_b \le r`+1$. По предположению шага индукции для утверждения 2 $r_z \le r+1$, и, поскольку для $z \ne b$ $r`_z=r_z$, имеем $r`_z \le r+1=r`+1$. Поэтому утверждение 2 сохраняется. В случае 1 по предположению шага индукции для утверждения 3, если $r_p=r$, то $K_p=K$, а тогда $r`_b=r$ и $K`_b=b=K_p=K$. В случае 2 $r`_b=r_p+1$ и, поскольку по предположению шага индукции для утверждения 1 $r_p$ и $r$ чётны, $r`_b$ нечётно и, следовательно, $r`_b \ne r=r`$. Поскольку остальные ранги и концы не меняются, утверждение 3 верно. Поскольку *b-событие* не меняется ранг в вершине a, то утверждение 4 верно.

Лемма доказана.

Лемма 4. Если дуги нет ($b=\varepsilon$), то ранг r максимальный $\forall z \in V \cup E$ $r_z \le r$ и $K=\varepsilon$.

Доказательство. Поскольку начало отсчёта t_0 – это момент времени после первого появления дуги p, отсутствие дуги p (между событиями, а не «внутри» одного события между «микрособытиями») возможно только в результате события *исчезновение*, при котором $r`=r+2$, $K`=\varepsilon$, а остальные ранги и концы не меняются. Поскольку по лемме 3 (утверждение 2) $r_z \le r+1$, для $z \ne a$ имеем $r`_z = r_z \le r`$. Поэтому, учитывая, что $K`=\varepsilon$, утверждение леммы верно.

Лемма доказана.

Лемма 5. Пусть в некоторый момент времени в некоторой вершине x ранг $r_x = j$. Тогда не более чем через $3(n-1)$ тактов в каждой вершине y ранг $r_y \ge j$.

Доказательство. Пусть в некоторый момент времени $A \ne \varnothing$ – это множество вершин y, в которых дуга p имеет ранг $r_y \ge j$. В каждый момент времени сураф, порождённый «долгоживущими» дугами, сильно связен, время существования «долгоживущей» дуги не меньше 1 такта, время пересылки по дуге не больше 1 такта, а временем срабатывания автомата мы пренебрегаем. Кроме того, в каждой вершине при появлении или освобождении дуги по ней сразу посылается сообщение.

Докажем *утверждение о расширении* множества A: с момента времени t_1, когда образуется множество $A \neq \varnothing$ и $A \neq V$, не более чем через 3 такта множество A будет расширено.

Поскольку по лемме 1 ранг дуги p в вершине не может уменьшаться, множество A может только расширяться. В любой момент времени, в том числе в момент времени t_1+2, должна существовать «долгоживущая» дуга q, ведущая из вершины $x \in A$ «наружу», т.е. в вершину $y \notin A$. В момент времени t_1+2 по дуге q передаётся сообщение S. Это сообщение отправлено из x не ранее момента времени $(t_1+2)-1=t_1+1$, поэтому в S ранг дуги p не меньше j. Возможны два случая.

1) Сообщение S дойдёт до вершины y. Тогда это произойдёт не позднее момента времени $(t_1+2)+1=t_1+3$. Следовательно, в этом случае не более чем через 3 такта после момента времени t_1 множество A будет расширено.
2) Сообщение S не дойдёт до вершины y. Это означает, что дуга q изменится до того, как по ней до вершины y дойдёт сообщение S. А тогда, поскольку q – «долгоживущая» дуга, по ней должно дойти до вершины y предыдущее сообщение $S`$. Поскольку сообщение $S$ отправлено из $x$ не ранее момента времени $t_1+1$, сообщение $S`$ принято в y не ранее этого же момента времени. Следовательно, сообщение $S`$ отправлено из $x$ не ранее момента времени $(t_1+1)-1=t_1$, поэтому в $S`$ ранг дуги p не меньше j. Следовательно, в этом случае не более чем через 1 такт после момента времени t_1 множество A будет расширено.

При каждом расширении множества A в него добавляется хотя бы одна вершина. Поскольку с самого начала множество A содержит хотя бы одну вершину, а общее число вершин равно n, получается, что число расширений множества A не более $n-1$. Таким образом, требуется не более чем $3(n-1)$ тактов, чтобы в каждой вершине y оказался ранг $r_y \geq j$.

Лемма доказана.

Лемма 6 (о состоянии 1). Граф находится в состоянии 1 не более $3(n-1)$ тактов.

Доказательство. Пусть в момент времени t граф находится в состоянии 1, тогда в некоторой вершине $r_x > r$. По лемме 5 в момент времени $t \leq t^{\wedge} \leq t+3(n-1)$ во всех вершинах ранг будет не меньше $r+1$, в том числе и в вершине a. Рассмотрим событие, в результате которого в вершине a ранг первый раз увеличивается, т.е. $r` > r$. По лемме 3 (утверждение 4) $r`=r+2$ и после события ранг $r`$ максимальный и имеется только в вершине $a$ и, быть может, на дуге $p$. Если дуга $p$ не исчезла ($b` \neq \varepsilon$) и $K` \notin B`$, то мы имеем состояние 2. Если дуга p не исчезла ($b` \neq \varepsilon$) и $K` \in B`$, то, поскольку по лемме 3 (утверждение 3) $K`_p=K`$, мы имеем состояние 3($r`$). Если дуга исчезла ($b` = \varepsilon$), то $\varepsilon \in B`$, по лемме 4 ранг $r`$ максимальный и $K` = \varepsilon$, а по лемме 3 (утверждение 3) для любого z, где $r`_z = r`$, также $K`_z = \varepsilon$. Следовательно, для любого $z$, где $r`_z$

максимальный ранг, имеет место $K`_z \in B`$, поэтому мы имеем состояние $3(r`)$. Таким образом, через время не более $3(n-1)$ тактов граф покинет состояние 1. Лемма доказана.

Лемма 7 (о состоянии 2). Граф находится в состоянии 2 не более 2 тактов, после чего переходит в состояние $3(w)$ для некоторого ранга w.

Доказательство. Рассмотрим различные события.

1. *Смена начала отсчёта*: $t`_0=t$, $B`=\{b\}$.

   Поскольку $b \in B$ и в состоянии 2 $K \notin B$, имеем $K`=K \notin B`$. Поскольку $b`=b$, а в состоянии 2 $b \neq \varepsilon$, имеем $b` \neq \varepsilon$. Поскольку ранги не меняются, граф остаётся в состоянии 2.

2. *Появление* дуги p с концом $x \neq \varepsilon$: $b`=x$, $B`=B \cup \{\varepsilon,x\}$. Если $K \neq \varepsilon$, то $r`=r+2$, $K`=\varepsilon$. В любом случае $K`_p=K`$, $r`_p=r`$.

   Покажем, что ранг $r`$ максимальный после события. Действительно, если $K=\varepsilon$, то ранги не меняются, кроме $r`_p=r`$, и поэтому ранг $r`$ максимальный по условию состояния 2. Если $K \neq \varepsilon$, то по лемме 1 ранг $r`>r$ и тогда по лемме 3 (утверждение 4) ранг $r`$ максимальный.

   По лемме 3 (утверждение 3) везде, где ранг равен $r`$, конец равен $K`$. Также в любом случае $K`=\varepsilon$, а, поскольку $\varepsilon \in B`$, имеем $K` \in B`$. Следовательно, граф переходит в состояние $3(r`)$.

3. *Исчезновение* дуги p: $b`=\varepsilon$, $B`=B \cup \{\varepsilon\}$, $r`=r+2$, $K`=\varepsilon$.

   Поскольку $r`>r$, по лемме 3 (утверждение 4) ранг $r`$ максимальный. По лемме 3 (утверждение 3) везде, где ранг равен $r`$, конец равен $K`$. Поскольку $K`=\varepsilon$ и $\varepsilon \in B`$, имеем $K` \in B`$. Следовательно, граф переходит в состояние $3(r`)$.

4. *Смена конца* дуги p на конец $x \neq \varepsilon$: $b`=x$, $B`=B \cup \{x\}$.

   Если $K=x$, то $K`=K \in B`$. Ранг $r`$ максимальный по условию состояния 2. По лемме 3 (утверждение 3) везде, где ранг равен $r`$, конец равен $K`$. Поэтому граф переходит в состояние $3(r`)$.

   Если $K \neq x$, то, поскольку в состоянии 2 $K \notin B$, имеем $K`=K \notin B`$. Ранг $r`$ максимальный по условию состояния 2, $b`=x \neq \varepsilon$. Поэтому граф остаётся в состоянии 2.

5. *ab-событие*: $r`=r+2$, $K`=a$.

   Поскольку, если дуга p петля, то $a \in B=B`$, имеем $K` \in B`$. Поскольку $r`>r$, по лемме 3 (утверждение 4) ранг $r`$ максимальный. По лемме 3 (утверждение 3) везде, где ранг равен $r`$, конец равен $K`$. Следовательно, граф переходит в состояние $3(r`)$.

6. *b-событие*: если $r_p \geq r_b$ и $K_p=b$, то $r`_b=r_p$ и $K`_b=b$, а если $r_p \geq r_b$ и $K_p \neq b$, то $r`_b=r_p+1$ и $K`_b=b$.

6.1. Если $r_p \geq r_b$ и $K_p=b$, то ранг r остаётся максимальным согласно условию состояния 2. Поскольку в состоянии 2 $K \notin B$, и *b-событие* не меняется конец в вершине a, имеем $K`=K \notin B=B`$. Поскольку $b`=b$, а в состоянии 2 $b \neq \varepsilon$, имеем $b` \neq \varepsilon$. Следовательно, граф остаётся в состоянии 2.

6.2. Если $r_p \geq r_b$ и $K_p \neq b$, то, поскольку в состоянии 2 ранг r максимальный, возможны два случая: $r_p<r$ и $r_p=r$.

6.2.1. В первом случае ($r_p<r$) ранг r остаётся максимальным согласно условию состояния 2. Поскольку в состоянии 2 $K \notin B$, и *b-событие* не меняет конец в вершине a, имеем $K`=K \notin B=B`$. Поскольку $b`=b$, а в состоянии 2 $b \neq \varepsilon$, имеем $b` \neq \varepsilon$. Следовательно, граф остаётся в состоянии 2.

6.2.2. Во втором случае ($r_p=r$) имеем $r`_b>r$ и, поскольку в состоянии 2 ранг $r$ до события был максимальным, после события ранг $r`_b$ максимальный и имеется только в вершине b. Поскольку $b`=b \in B=B`$, а $K`_b=b$, имеем $K`_b \in B`$. Следовательно, граф переходит в состояние 3($r`_b$).

7. *a-событие* ($r_q > r$): $r`=r_q+1$ и $K`=K_q$.

   Условие $r_q > r$ противоречит условию максимальности ранга r в состоянии 2, поэтому это событие в состоянии 2 невозможно.

8. *Приём сообщения*: $r`_x=r_q$, $K`_x=K_q$.

   Это событие, очевидно, сохраняет состояние 2.

9. *Посылка сообщения*: $r`_q=r_x$, $K`_q=K_x$.

   Это событие, очевидно, сохраняет состояние 2.

Таким образом, мы показали, что любое событие либо оставляет граф в состоянии 2, либо переводит его в состояние 3(w) для некоторого ранга w, но не в состояние 1. Теперь покажем, что не более чем через 2 такта граф перейдёт в состояние 3(w). Рассмотрим момент времени t_2, когда граф переходит в состояние 2. Начиная с момента времени t_2 ранг r может измениться только после событий *появление*, *исчезновение* или после *ab-события* (*a-событие* невозможно в состоянии 2). Однако каждое из этих событий гарантированно переводит граф в состояние 3(w). Поэтому достаточно рассмотреть случай, когда эти события не происходят. Тогда ранг r не меняется всё время, пока граф остаётся в состоянии 2. Из описания событий следует, что без изменения ранга r не меняется конец K. Также условие $K \notin B$ не меняется, пока граф остаётся в состоянии 2. Поскольку в состоянии 2 дуга p есть ($b \neq \varepsilon$), в момент времени t_2 по дуге p передаётся сообщение, которое, возможно, было послано ещё до момента времени t_2. Поскольку дуга p не появляется и не исчезает, через время не более 1 такта это сообщение гарантированно будет принято в конце дуги p, после чего по этой дуге будет послано второе сообщение. Это второе сообщение будет послано

тогда, когда граф находится в состоянии 2, поэтому в этот момент времени будет $K_p=K \notin B$ и $r_p=r$. Поскольку дуга p не появляется и не исчезает, это второе сообщение также гарантированно будет принято в конце дуги p через время не более 1 такта, причём, поскольку r и K не меняются и не меняется условие $K \notin B$, будет $K_p \notin B$ и $r_p=r$. Поскольку в момент приёма сообщения в вершине b имеет место $b \in B$, имеем $K_p \neq b$. Поскольку в состоянии 2 ранг r максимальный и поскольку $r_p=r$, имеем $r_p \geq r_b$. А тогда, как показано выше для *b-события* (6.2.2), граф переходит в состояние $3(r`_b)$. Итак, мы показали, что через время не более 2 тактов граф переходит из состояния 2 в состояние $3(w)$. Лемма доказана.

Лемма 8 (о состоянии 3). Если не меняется начало отсчёта, то граф находится в состоянии $3(w)$ для любого заданного w не более $3n-2$ тактов, после чего переходит в подсостояние 4, из которого уже не выходит. Смена начала отсчёта переводит состояние $3(w)$ в состояние 1, 2 или $3(w`)$.

Доказательство. Рассмотрим различные события.

1. *Смена начала отсчёта*: $t`_0=t$, $B`=\{b\}$.

   Если $b=\varepsilon$, то по лемме 4 ранг r максимальный и $K=\varepsilon=b \in B`$. По лемме 3 (утверждение 3) везде, где ранг равен $r$, конец равен $K$. Тогда, поскольку $b`=b$, $r`=r$, $K`=K$ и для любого z $K`_z=K_z$, граф переходит в состоянии $3(r)$.

   Если $b \neq \varepsilon$, то возможны четыре случая:

   1) Ранг r максимальный и $K \neq b$.

      Тогда $K \notin B`$. Граф переходит в состояние 2.

   2) Ранг r максимальный и $K=b$.

      Тогда $K \in B`$. По лемме 3 (утверждение 3) для всех z таких, что $r_z=r$, имеет место $K_z=K$. Граф переходит в состоянии $3(r)$.

   3) Максимальный ранг $r_{max} > r$, для некоторого z, где $r_z = r_{max}$, $K_z \neq b$.

      Тогда $K_z \notin B`$. Граф переходит в состояние 1.

   4) Максимальный ранг $r_{max} > r$, для каждого z, где $r_z = r_{max}$, $K_z = b$.

      Тогда $K_z \notin B`$. Граф переходит в состояние $3(r_{max})$.

Поскольку для любого другого события $B \subseteq B`$, нам достаточно показать, что если меняются ранг $r`_z \neq r_z$ или конец $K`_z \neq K_z$, то конец становится правильным $K`_z \notin B`$.

2. *Появление* дуги p с концом $x \neq \varepsilon$: $b`=x$, $B`=B \cup \{\varepsilon,x\}$. Если $K \neq \varepsilon$, то $r`=r+2$, $K`=\varepsilon$. В любом случае $K`_p=K`$, $r`_p=r`$.

   Поскольку $K`_p=K`=\varepsilon$ и $\varepsilon \in B`$, граф остаётся в состоянии $3(w)$.

3. *Исчезновение* дуги p: $b`=\varepsilon$, $B`=B \cup \{\varepsilon\}$, $r`=r+2$, $K`=\varepsilon$.

   Поскольку $K`=\varepsilon$ и $\varepsilon \in B`$, граф остаётся в состоянии $3(w)$.

4. *Смена конца* дуги p на конец x: $b`=x$, $B`=B \cup \{x\}$.

   Граф остаётся в состоянии $3(w)$.

5. *ab-событие*: $r`=r+2$, $K`=a$.

   Поскольку $K`=a$ и для петли $a \in B=B`$, граф остаётся в состоянии 3(w).

6. *b-событие*: если $r_p \geq r_b$ и $K_p=b$, то $r`_b=r_p$ и $K`_b=b$, а если $r_p \geq r_b$ и $K_p \neq b$, то $r`_b=r_p+1$ и $K`_b=b$.

   Поскольку $K`_b=b$ и $b \in B=B`$, граф остаётся в состоянии 3(w).

7. *a-событие* ($r_q > r$): $r`=r_q+1$ и $K`=K_q$.

   Если $r_q > r$, то, поскольку по лемме 3 (утверждение 2) $r_q \leq r+1$, имеем $r_q = r+1$, следовательно, по лемме 3 (утверждение 2) r_q максимальный ранг. А тогда в состоянии 3(w) $K_q \in B$, что влечёт $K`=K_q \in B=B`$. Поэтому граф остаётся в состоянии 3(w).

8. *Приём сообщения*: $r`_x=r_q$, $K`_x=K_q$.

   Это событие, очевидно, сохраняет состояние 3(w).

9. *Посылка сообщения*: $r`_q=r_x$, $K`_q=K_x$.

   Это событие, очевидно, сохраняет состояние 3(w).

Таким образом, мы показали, что любое событие, кроме смены начала отсчёта, оставляет граф в состоянии 3(w) без изменения ранга w. Теперь покажем, что не более чем через $3n-2$ тактов граф перейдёт в подсостояние 4. По лемме 2 не более чем через $3(n-1)$ тактов во всех вершинах будет ранг не менее w. А тогда ещё не более чем через 1 такт на каждой дуге тоже будет ранг не менее w. Для состояния 3(w) это как раз и означает переход в подсостояние 4 через $3(n-1)+1=3n-2$ тактов.

Лемма доказана.

Теорема 1. Через время не более $6n-3$ тактов в каждой вершине и на каждой дуге z будет $K_z \in B$, т.е. $T_0 = \boldsymbol{O(n)}$.

Доказательство. Состояние 4 – это подсостояние 3(w), когда в каждой вершине и на каждой дуге ранг не меньше w. В этом случае как раз будет выполнено условие $K_z \in B$. Поэтому нам надо показать, во-первых, что при любом выборе начала отсчёта граф окажется в одном из состояний 1, 2 или 3(w), и, во-вторых, что после этого через время не более $4n-1$ тактов он окажется в состоянии 4.

Если t_0 – момент времени непосредственно после обработки автоматом сигнала первого появления дуги p, то имеем: $r_p=r=0$, $K_p=K=\varepsilon$, а для $z \neq a$ и $z \neq p$: $r_z=-1$, что соответствует состоянию 2. По леммам 6, 7, 8 (о состояниях 1, 2, 3) выбор другого начало отсчёта переводит в одно из состояний 1, 2 или 3(w) для некоторого ранга w.

По лемме 6 (о состоянии 1) в состоянии 1 граф находится не более $3(n-1)$ тактов. По лемме 7 (о состоянии 2) в состоянии 2 граф находит не более 2 тактов, после чего переходит в состояние 3(w) для некоторого ранга w. По лемме 8 (о состоянии 3) в состоянии 3(w) граф находит не более $3n-2$ тактов, после чего переходит в подсостояние 4, из которого уже не выходит, если не

меняется начало отсчёта. Следовательно, через время не более $3(n-1)+2+3n-2=6n-3$ из любого состояния граф переходит в состояние 4. Теорема доказана.

Теорема 2. После прекращения изменений в графе полная информация о графе будет собрана в каждой его вершине через время не более $4D+3$ тактов, т.е. $T_1=\boldsymbol{O}(D)$.

Доказательство. После прекращения изменений графа порядок верхней границы времени распространения информации о дуге снижается с n до D. Действительно, рассмотрим множество вершин A из доказательства леммы 5, содержащее вершины, в которых ранг не меньше j.

Поскольку дуги больше не меняются, модифицируется *утверждение о расширении* в доказательстве леммы 5: с момента времени t_1, когда образуется множество $A\neq\varnothing$ и $A\neq V$, множество A будет расширено не более чем через 2 такта (а не 3 такта).

Действительно, в любой момент времени, в том числе в момент времени t_1+1, должна существовать «долгоживущая» дуга q, ведущая из вершины $x\in A$ «наружу», т.е. в вершину $y\notin A$. В момент времени t_1+1 по дуге q передаётся сообщение S. Это сообщение отправлено из x не ранее момента времени $(t_1+1)-1=t_1$, поэтому в S ранг дуги p не меньше j. Поскольку после момента времени t_1 дуги не меняются, сообщение S дойдёт до вершины y. Тогда это произойдёт не позднее момента времени $(t_1+1)+1=t_1+2$. Следовательно, не более чем через 2 такта после момента времени t_1 множество A будет расширено.

Пусть $A_0\neq\varnothing$ – это множество A с самого начала. Для каждой вершины $x\notin A_0$ выберем входящую дугу, по которой в эту вершину первый раз пришло сообщение с рангом не меньше j. Выбранные дуги, очевидно, образуют лес деревьев, ориентированных от своих корней, которыми являются вершины из множества A_0. Этот лес деревьев содержит все вершины. Поскольку дуги не меняются, с момента времени, когда в некоторой вершине x ранг становится не меньше j, по всем выбранным дугам, выходящим из данной вершины x, пройдут сообщения с рангом не меньше j за время не более 2 тактов, согласно модифицированному утверждению о расширении. А это означает, что сообщения с рангом не меньше j пройдут по всем выбранным дугам за время не более чем $2L$ тактов, где L – максимальная длина маршрута из выбранных дуг. Поскольку выбранные дуги образуют лес деревьев, $L\leq D$.

Следовательно, изменяется оценка времени в лемме 5: вместо $3(n-1)$ будет $2D$ тактов. Соответственно, изменяются оценки в леммах 6 (о состоянии 1) и в лемме 8 (о состоянии 3): вместо $3(n-1)$ и $3n-2$ будет $2D$ и $2D+1$. Поэтому максимальное время работы алгоритма уменьшается с $3(n-1)+2+3n-2=6n-3$ тактов до $2D+2+2D+1=4D+3$ тактов.

Теорема доказана.

Заметим, что в доказательстве теоремы 2 не использовано ограничение на время существования дуги: до того, как прекращаются изменения в графе, время существования дуг может быть произвольным. Также до прекращения изменений граф может быть не сильно связным, и сильная связность графа требуется только после прекращения его изменений.

5.2 Размер сообщения

Сообщение содержит идентификатор вершины $\Gamma\tilde{}$, номер дуги $i\tilde{}$ от 1 до s, длину $l\tilde{}$ массива описаний дуг и массив $L\tilde{}[0..l\tilde{}-1]$ описаний дуг.

Обозначим через *sizeofI* размер (в битах) памяти, требуемой для хранения идентификатора вершины. Поскольку разные вершины имеют разные идентификаторы, а число вершин равно n, имеем $sizeofI \geq log_2 n$. Нижняя оценка достигается, если вершины просто нумеруются от 0 до $n-1$.

Номер дуги имеет размер $log_2(s+1)$, поскольку дуги нумеруются от 1 до s.

Как следует из описания алгоритма, каждое изменение дуги приводит к увеличению ранга этой дуги не более чем на 2 (события *появление*, *исчезновение*, *ab-событие* или *b-событие* плюс *a-событие*). Обозначим через v (от *variation*) максимальное число изменений одной дуги. Тогда для данной дуги максимальный ранг не более $2v$, поэтому для ранга достаточно $log_2 2v$ бит памяти. Длина $l\tilde{}$ массива не превосходит числа дуг m.

Итого: размер сообщения $sizeofI + log_2(s+1) + log_2 m + m(2sizeofI + log_2(s+1) + log_2 2v) = \boldsymbol{O}(m(sizeofI+lgs)) + \boldsymbol{O}(mlgv)$ бит. Поскольку $s \leq m$, это равно $\boldsymbol{O}(m(sizeofI+lgm)) + \boldsymbol{O}(mlgv)$. Если идентификатор вершины – это просто её номер от 0 до n, то, учитывая, что $n \leq m$, имеем $\boldsymbol{O}(m(lgn+lgm)) + \boldsymbol{O}(mlgv) = \boldsymbol{O}(mlgm) + \boldsymbol{O}(mlgv)$. Второе слагаемое – это память для хранения рангов, размер которой зависит от числа изменений дуги.

5.3 Память вершины (автомата в вершине)

Для хранения управляющего состояния (начальное или рабочее) достаточно 1 бита. Идентификатор вершины занимает *sizeofI* бит.

Суммарно имеем размер памяти автомата в вершине:

$1 + sizeofI + log_2 m + m(2sizeofI + log_2(s+1) + log_2 2v) =$

$= \boldsymbol{O}(m(sizeofI+lgs)) + \boldsymbol{O}(mlgv)$ бит.

Поскольку $s \leq m$, это равно $\boldsymbol{O}(m(sizeofI+lgm)) + \boldsymbol{O}(mlgv)$. Если идентификатор вершины – это просто её номер от 0 до n, то, учитывая, что $n \leq m$, имеем $\boldsymbol{O}(m(lgn+lgm)) + \boldsymbol{O}(mlgv) = \boldsymbol{O}(mlgm) + \boldsymbol{O}(mlgv)$. Заметим, что память, необходимая для хранения полного описания графа, – это первое из двух слагаемых суммы. Второе слагаемое – это память для хранения рангов, размер которой зависит от числа изменений дуги.

5.4. Модификация алгоритма для уменьшения размера памяти

Как показано выше, размер сообщения и памяти вершины зависит от числа изменений дуги. От этой зависимости можно избавиться, если использовать либо дополнительное предположение об изменениях дуг, либо дополнительную возможность для автомата.

Дополнительное предположение: все дуги долгоживущие. В этом случае каждая дуга меняется не чаще 1 раза в такт. Следовательно, за время работы алгоритма максимум $6n-3$ тактов дуга меняется не более $6n-3$ раз. Поскольку каждое изменение дуги увеличивает ранг не более чем на 2, можно сделать ранг циклическим с максимальным значением $2(6n-3)$. Тогда размер сообщения и размер памяти вершины равен $\boldsymbol{O}(m(sizeofI+lgm)) + \boldsymbol{O}(mlgv) = \boldsymbol{O}(m(sizeofI+lgm)) + \boldsymbol{O}(mlgn)$ или, поскольку $n \le m$, $\boldsymbol{O}(m(sizeofI+lgm))$.

Дополнительная возможность: временные сигналы для автомата. Пусть имеется таймер, посылающий автомату каждый такт специальный временной сигнал. Все те действия, которые автомат выполнял при обработке сигнала освобождения или появления дуги, теперь он будет выполнять при обработке временного сигнала. Это значит, что сообщения будут посылаться по дугам не при их освобождении или появлении, а, вообще говоря, позже – при обработке следующего за освобождением или появлением дуги временного сигнала. Тем самым, сообщения по дугам будут посылаться не чаще, чем раз в такт.

Правда, в этом случае придётся предположить, что время существования «долгоживущих» дуг ограничено снизу не 1, а 2 тактами. Действительно, в противном случае может оказаться, что каждый раз при появлении дуги сообщение посылается по ней не сразу, а спустя какое-то время (до получения временного сигнала), из-за чего сообщение не успевает передаться по дуге до её изменения.

В общем случае рассмотрим τ_1 – интервал между сигналами времени, τ_2 – максимальное время передачи сообщения по дуге, τ_3 – минимальное время жизни долгоживущей дуги. Тогда нужно, чтобы выполнялось следующее соотношение $\tau_1 + \tau_2 \le \tau_3$.

В этом случае максимальное время пребывания графа в состоянии 2 будет не 2 такта, а $2\tau_2 + \tau_1$. В доказательстве леммы 5 время расширения множества A меняется с 3 на $3\tau_2 + 2\tau_1$. Соответственно, оценка времени в лемме 5 меняется с $3(n-1)$ на $(3\tau_2 + 2\tau_1)(n-1)$, время пребывания графа в состоянии 1 меняется с $3(n-1)$ на $(3\tau_2 + 2\tau_1)(n-1)$, время пребывания графа в состоянии 3 меняется с $3n-2$ на $(3\tau_2 + 2\tau_1)(n-1) + (\tau_2 + \tau_1)$. Эти утверждения легко доказываются, но эти доказательства мы опускаем.

Итак, максимальное время работы алгоритма, включая время «притормаживания» в ожидании временного сигнала, останется тем же по порядку, что и в теореме 1, $T_0 = (3\tau_2 + 2\tau_1)(n-1) + 2\tau_2 + \tau_1 + (3\tau_2 + 2\tau_1)(n-1) +$

$(\tau_2+\tau_1) = (6\tau_2+4\tau_1)(n-1) + 3\tau_2+2\tau_1 = (3\tau_2+2\tau_1)(2n-1) = \boldsymbol{O}(n)$. Ранг станет циклическим и будет занимать память $\boldsymbol{O}(lgn)$ бит. Соответственно, размер сообщения и памяти вершины равен $\boldsymbol{O}(m(sizeofI+lgm)) + \boldsymbol{O}(mlgn)$ или, поскольку $n \le m$, $\boldsymbol{O}(m(sizeofI+lgm))$.

Соответственно, модифицируется утверждение теоремы 2: $T_1 = (2\tau_2+\tau_1)D + 2\tau_2+\tau_1 + (2\tau_2+\tau_1)D + (\tau_2+\tau_1) = (4\tau_2+2\tau_1)D + 3\tau_2+2\tau_1 = \boldsymbol{O}(D)$.

Список литературы

[1]. Steven S. Skiena. The Algorithm Design Manual. Springer-Verlag, New York, 1997.

[2]. И.Б. Бурдонов, А.С. Косачев, В.В. Кулямин. Неизбыточные алгоритмы обхода ориентированных графов. Детерминированный случай. Программирование, 2003 г., №5, с. 59-69.

[3]. И.Б. Бурдонов, А.С. Косачев, В.В. Кулямин. Неизбыточные алгоритмы обхода ориентированных графов. Недетерминированный случай. Программирование, 2004 г., №1, с. 2-17.

[4]. M.O. Rabin. Maze Threading Automata. An unpublished lecture presented at MIT and UC. Berkeley, 1967.

[5]. И.Б. Бурдонов. Обход неизвестного ориентированного графа конечным роботом. Программирование, 2004 г., № 4, с. 11-34.

[6]. И.Б. Бурдонов. Проблема отката по дереву при обходе неизвестного ориентированного графа конечным роботом. Программирование, 2004 г., № 6, с. 6-29.

[7]. И. Бурдонов, А. Косачев, В. Кулямин. Параллельные вычисления на графе. Программирование, 2015, №1, с. 3-20.

Monitoring of dynamically changed graph

Igor Burdonov <igor@ispras.ru>
Alexander Kossatchev kos@ispras.ru
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. Monitoring of oriented graphs is a key task in many applications. Such monitoring is very specific when the graph models a communication network including Internet and GRID. A node of the network has local information about the network: if "knows" only about the arcs outgoing from this node, but does not "know" where (to which nodes) these arcs go. The nodes of the network exchange messages through the network links represented as arcs of the graph and act as message transfer channels. The graph monitoring is based on its traversal when message passes each arc in the graph. While there is an untraversed arc, we cannot be certain that it goes to still unmonitored part of the graph. Usually, the graph traversal is performed with a single message circulating in the network. Traversal can be done

faster if performed in parallel: multiple messages simultaneously circulate in the network. In this paper we consider the parallel monitoring of a graph aimed at not just the graph traversal, but also collection of complete information about the graph in each its node. Another feature of this work is monitoring of a dynamically changing graph: its arcs can disappear, appear or change their destination nodes. An algorithm is proposed, which provides the collection of full information on the graph in each its node.

Keywords: directed graphs; graph exploration, graph traversal, communicating automata, parallel processing, dynamically changed graphs.

DOI: 10.15514/ISPRAS-2015-27(1)-5

For citation: Burdonov Igor, Kosachev Alexander. Monitoring of dynamically changed graph. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 1, 2015, pp. 69-96 (in Russian). DOI: 10.15514/ISPRAS-2015-27(1)-5

References

[1]. Steven S. Skiena. The Algorithm Design Manual. Springer-Verlag, New York, 1997.
[2]. I. B. Burdonov, A. S. Kossatchev, V. V. Kuliamin. Irredundant Algorithms for Traversing Directed Graphs: The Deterministic Case. Programming and Computer Software, 29(5):245-258, 2003.
[3]. I. B. Burdonov, A. S. Kossatchev, V. V. Kuliamin. Irredundant Algorithms for Traversing Directed Graphs: The Nondeterministic Case. Programming and Computer Software, 30(1):2-17, 2004.
[4]. M.O. Rabin. Maze Threading Automata. An unpublished lecture presented at MIT and UC. Berkeley, 1967.
[5]. I. B. Burdonov. Traversal of an unknown directed graph by a finite automaton. Programming and Computer Software, 30(4): 11-34, 2004.
[6]. I. B. Burdonov. Backtracking on a tree in traversal of an unknown directed graph by a finite automaton. Programming and Computer Software, 30(6): 6-29, 2004.
[7]. I. B. Burdonov, A. S. Kossatchev, V. V. Kuliamin. Parallel computations on graphs. Programming and computer Software, 41(1): 1-13, 2015.

Обзор методов извлечения моделей из HDL-описаний

С.А. Смолов <smolov@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. В статье дается обзор существующих методов извлечения моделей из описаний цифровой аппаратуры, разработанных на языках семейства HDL (Hardware Description Language). Методы извлечения моделей используются для решения многих задач, связанных с процессом проектирования и обеспечения качества программных и аппаратных систем. В данной работе затрагиваются методы решения следующих актуальных задач – оптимизация кода, оптимизация логического синтеза, абстракция, функциональная верификация. В статье рассматриваются методы извлечения таких семейств моделей, как графы потока и зависимостей, а также автоматные модели. Подробно рассматриваются методы построения программных срезов, конечных автоматов и расширенных конечных автоматов.

Ключевые слова: языки описания аппаратуры; извлечение моделей; статический анализ; оптимизация кода; абстракция; логический синтез; функциональная верификация; программные срезы; графы потока; графы зависимостей; конечные автоматы; расширенные конечные автоматы.

DOI: 10.15514/ISPRAS-2015-27(1)-6

Для цитирования: Смолов С.А. Обзор методов извлечения моделей из HDL-описаний. Труды ИСП РАН, том 27, вып. 1, 2015 г., стр. 97-124. DOI: 10.15514/ISPRAS-2015-27(1)-6.

1. Введение

В настоящее время цифровая аппаратура повсеместно используется во многих сферах человеческой деятельности – производственной, научной, бытовой и так далее. Появление все новых научно-технических задач и необходимость автоматизации производства приводят к постоянному росту сложности отдельных аппаратных компонентов и увеличению их количества в рамках аппаратных комплексах. Временны́е и финансовые затраты на разработку многокомпонентных аппаратных систем также возрастают, что на текущем этапе развития технологий привело к унификации процесса проектирования цифровой аппаратуры. Процесс проектирования включает следующие основные этапы: (1) составление требований, (2) архитектурное проектирование, (3) детальное проектирование, (4) логическое

проектирование (логический синтез), (5) физическое проектирование (физический синтез) [1]. Результатами каждого из этапов соответственно являются: (1) спецификация, (2) программный симулятор, (3) модель уровня регистровых передач (RTL, Register Transfer Level), (4) фотошаблон, (5) готовая микросхема.

В данной статье рассматриваются методы и инструментальные средства анализа моделей уровня регистровых передач (далее – RTL-моделей), полученных на этапе детального проектирования. Причиной служит тот факт, что последующие этапы (логический и физический синтез) автоматизированы средствами современных САПР и, следовательно, детальное проектирование является ключевым и критически важным этапом в контексте обеспечения корректности разрабатываемого аппаратного комплекса. На этапе разработки RTL-модели замеченные ошибки могут быть легко исправлены, чего нельзя сказать о результатах применения процедур синтеза. Пропущенные ошибки могут привести к летальным последствиям и серьезным материальным и финансовым потерям [2].

Модели уровня регистровых передач разрабатываются на специализированных языках описания аппаратуры (HDL, Hardware Description Language). Средствами данных языков можно описывать структуру и поведение аппаратных компонентов [3] (последнее – с потактовой точностью). Примерами языков описания аппаратуры являются Verilog [4] и VHDL [5].

В рамках процесса детального проектирования актуальными являются следующие задачи: (1) ко-симуляция (совместное исполнение моделей, разработанных на разных языках описания, а также совместное исполнение программных и аппаратных моделей), (2) оптимизация кода (например, для упрощения последующего этапа логического синтеза), (3) верификация (проверка соответствия модели её спецификации). В связи с ростом сложности аппаратных систем данные задачи не могут эффективно решаться вручную (методом экспертизы кода), что делает актуальной задачу разработки соответствующих автоматизированных средств. Распространенный подход при разработке методов и инструментальных средств анализа моделей, реализованных на языках описания аппаратуры (в дальнейшем – HDL-описаний), состоит в использовании моделей – абстрактных представлений аппаратных систем. В зависимости от поставленной задачи, уровень абстракции модели может варьироваться от воспроизведения поведения HDL-описания с потактовой точностью, до, например, предоставления информации об интерфейсе или отдельном режиме работы устройства.

В работе представлен обзор существующих методов извлечения моделей из HDL-описаний цифровой аппаратуры. Статья организована следующим образом. В Разделе 2 представлены необходимые базовые сведения о структуре HDL-описаний. В Разделе 3 делается обзор методов извлечения графов потоков и зависимостей. Раздел 4 описывает методы извлечения автоматных моделей. Раздел 5 завершает статью.

2. Структура HDL-описаний

Языки семейства HDL во многом схожи с языками императивного программирования, такими как C и Ada. В частности, используется аналогичный аппарат декларирования и использования процедур и функций, типов данных (представлены булевский и целочисленный типы, битовые векторы и массивы фиксированной длины), арифметических и логических операций (в том числе побитовых). Остановимся на основных отличиях HDL-описаний в контексте структуры и исполнения.

Основным компонентом HDL-описания является модуль (module). Модули в HDL-описаниях соответствуют элементам аппаратных систем. Каждый модуль имеет интерфейс, содержащий входные (input) и выходные (output) сигналы модуля. Как правило, интерфейс модуля содержит специальные входные сигналы, называемые сигналами синхроимпульса (clk, он же сигнал тактовой частоты) и сброса (reset). Функциональная логика модуля описывается в его теле (architecture). Тело модуля может содержать фиксированное количество экземпляров других модулей (тогда HDL-описание называется иерархическим) а также фиксированное количество процессов (process). Процессы – это сущности, исполняемые в рамках одного модуля в параллельном режиме. Каждый процесс может иметь так называемый список чувствительности (sensitivity list), являющийся подмножеством входного интерфейса модуля, а также включать исполняемый код. Ряд типов инструкций исполняемого кода имеет традиционную для императивных языков программирования семантику; это справедливо для последовательных (блокирующих) присваиваний, условных операторов и операторов цикла. Специфические для HDL-описаний инструкции, как правило, представляют сведения о временны́х ограничениях на выполнение процесса. Часто используемыми специфическими инструкциями являются инструкции ожидания (wait), приостанавливающие выполнение соответствующего процесса до выполнения определенного условия или возникновения определенного события, а также параллельные (не блокирующие) присваивания. Под событиями в HDL-описаниях понимают изменения уровня входного сигнала; различают события по приходу переднего фронта сигнала (rising edge), заднего фронта сигнала (falling edge) и произвольного фронта сигнала (event).

Важной особенностью HDL-описаний является специфический режим выполнения процессов. Во время исполнения (симуляции) каждый процесс постоянно находится в ожидании прихода события. Если возникшее в системе событие принадлежит списку чувствительности некоторого процесса, то выполняется содержащийся в нем код и процесс вновь переходит в состояние ожидания. Отличия от описанного режима исполнения возникают при наличии в коде специфических инструкций. В частности, если нить исполнения процесса достигает инструкции ожидания, то исполнение приостанавливается до тех пор, пока не произойдет событие для данной

инструкции ожидания. Что касается блоков параллельных присваиваний, то означивание определяемых в них переменных происходит параллельно, а их обновленные значения оказываются доступны только на следующей итерации исполнения кода процесса. Наличие специфических инструкций, а также парадигма исполнения HDL-описаний обуславливают необходимость разработки собственных методов извлечения моделей и затрудняют использование методов анализа программ.

Пример. 1 HDL-описание на языке VHDL

01:	**library** IEEE;	используемая библиотека
02:	**use** IEEE.std_logic_1164.all;	используемый пакет библиотеки
03:	**entity** SAMPLE **is**	модуль
04:	**port** (	интерфейс модуля
05:	PRESET : **in integer**;	входной целочисленный сигнал
06:	RST, CLK, D : **in std_logic**;	входные однобитные сигналы
07:	Q : **out std_logic**);	выходной однобитный сигнал
08:	**end** SAMPLE;	
09:	**architecture** BODY **of** SAMPLE **is**	тело модуля
10:	**begin**	
11:	**process** (RST, CLK)	объявление процесса и списка
12:	**begin**	чувствительности
13:	**if** (RST = '0') **then**	условный оператор
14:	**if** (PRESET = 0) **then**	условный оператор
15:	Q <= '0';	параллельное присваивание
16:	**else**	
17:	Q <= '1';	параллельное присваивание
18:	**end if**;	
19:	**wait on** D;	оператор ожидания
20:	Q <= D;	параллельное присваивание
21:	**end if**;	
22:	**end process**;	
23:	**end** BODY;	

Отметим, что не весь код HDL-описаний может быть обработан с помощью средств логического синтеза. Так, например, не синтезируемыми (хотя и исполнимыми с помощью специальных компонентов, называемых HDL-симуляторами) являются инструкции, выполняющие вычисления над рациональными числами и функции отладочной печати (для них не могут быть сгенерированы аппаратные компоненты, реализующие их логику). В дальнейшем будет предполагаться, что методы анализа HDL-описаний применяются на синтезируемом подмножестве инструкций данных языков. Не

синтезируемые конструкции, как правило, используются в целях отладки, в тестовых системах (testbench) или модельных примерах для оценки эффективности работы инструментов анализа и синтеза (benchmark).

Синтаксис HDL-описания на языке VDHL приведен в прим. 1.

В современных аппаратных системах размер кода HDL-описаний может составлять сотни тысяч строк. В дальнейшем такие описания, имеющие, как правило, иерархическую структуру, мы будем называть *описаниями повышенной сложности*; именно их анализ, оптимизация и верификация представляют наибольший интерес. В Разделах 3-4 делается обзор методов извлечения моделей с указанием того, имеется ли у методов инструментальная поддержка и были ли эти методы апробированы на HDL-описаниях повышенной сложности.

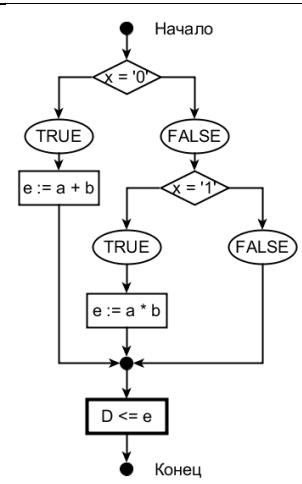
3. Графы потоков и зависимостей

Графы потоков и графы зависимостей – традиционные представления, используемые в области анализа программ [6]. Распространенными их разновидностями являются *графы потока управления* (множества всех возможных путей исполнения) и *графы зависимостей по данным* (множества переменных с наложенными на них отношениями зависимости по данным). В силу широкого распространения в программной инженерии эти модели одними из первых были адаптированы для анализа HDL-описаний. В прим. 2 приведен процесс на языке VHDL и соответствующий ему граф потока управления (CFG, Control Flow Graph). Прямоугольные вершины графа соответствуют базовым блокам, базовый блок с параллельным (не блокирующим) присваиванием выделен утолщенной рамкой. Ромбические вершины соответствуют операторам ветвления, овальные – значениям условного оператора. Круглые черные вершины соответствуют операторам слияния, начальной и конечной вершинам.

Одной из ранних работ в области статического анализа HDL-описаний, в которой фигурируют графы потока управления, является [7]. В ней описывается система SAVE (Static Analysis for VHDL Evaluation) оценивания описаний аппаратуры на языке VHDL. Система позволяет анализировать код с точки зрения эффективности его симуляции, а также оценивать сравнительную сложность различных модулей. Сравнительная эффективность симуляции различных инструкций определяется на основе жестко заданных в системе правил, а те, в свою очередь, строятся на основе обобщения экспериментальных данных. Хотя в работе указывается, что метрики определения сложности HDL-описаний должны отличаться от аналогичных метрик, используемых для программных модулей, сами отличия в статье не описаны. Графы потока управления используются в системе SAVE для визуализации HDL-описаний.

Пример 2. HDL-описание и его граф потока управления

```
01:  process(a, b, x)
02:    variable e : integer range 0 to 9;
03:  begin
04:    if (x = '0') then
05:      e := a + b;
06:    elsif (x = '1') then
07:      e := a * b;
08:    end if;
09:    D <= e;
10:  end process;
```



В работе [8] рассматриваются примеры использования методологий статического анализа, используемых в области программного обеспечения, для анализа HDL-описаний. В частности, для выявления *зависаний* (deadlocks) в наборах однотипных процессов предлагается использовать *информационные графы* (information graph) и графы зависимостей. Вершины информационного графа соответствуют процессам, а ребра – зависимостям по внутренним переменным (если переменная в одном процессе читается, а в другом - определяется). Если информационный граф не содержит циклов, то считается, что целевое HDL-описание может содержать зависания, так как процессы не могут вычислять новые значения переменных состояния на основе обновленных значений входных сигналов. Вершинами графа зависимостей являются сигналы, которыми обмениваются процессы, а ребрами – зависимости по событиям между сигналами. Наличие циклов в таком графе также сигнализирует о возможном наличии зависаний.

В статье также представлены некоторые техники анализа графа потока HDL-описаний. *Временной анализ* (timing analysis) направлен на вычисление для каждой переменной максимальной и минимальной задержек по присваиванию им значений. Для определения этих величин анализируются блоки вида *after x*, где x – величина задержки (например, указанная в наносекундах). Величина задержки присваивания значений переменной определяется как сумма величин задержек на данном пути в графе потока управления. Полученные значения задержек используются для формирования временных ограничений для генерации случайных тестовых последовательностей.

Второй техникой анализа является поиск *неявных элементов памяти* (implicit memory elements) и *прозрачных переменных* (transparent). Для этого для

каждой переменной каждого процесса HDL-описания на основе анализа всех путей в графе потока строится регулярное выражение, каждый элемент которого сигнализирует об операции над переменной. Всего существует три типа операций – присваивание (assignment), чтение (reading), ожидание (waiting). Неявные элементы памяти и прозрачные переменных определяются посредством установления соответствия между извлеченными регулярными выражениями и заданными шаблонами (например, в случае прозрачных переменных необходимо, чтобы регулярные выражения содержали только использования их значений). По словам авторов статьи, определение неявных элементов памяти и прозрачных переменных способно упростить процесс верификации HDL-описаний.

Предложенные в статье методы поясняются на модельных примерах небольшого размера и не имеют инструментальной поддержки.

В статье [9] описывается метод предварительной оптимизации HDL-описаний на языке VHDL, основанный на построении *структуры данных системы* (SDS, System Data Structure). Структура данных системы состоит из *графа потока данных* (Data Flow Graph), *графа управления временными свойствами* (Control Timing Graph) и *связей* (bindings) между ними. Процедура построения представления, предварительно оптимизированного для последующего логического синтеза, содержит следующие шаги:

- построение графа потока управления (используется инструмент VAUL [10]) и выделение базовых блоков;
- анализ потоков данных (на первом этапе – выявление зависимостей по чтению и записи переменных на уровне инструкций для каждого базового блока, на втором этапе – на уровне графа);
- построение графов потоков данных для каждой переменной и графа управления временными свойствами для всего HDL-описания;
- составление очередности выполнения инструкций (scheduling).

Метод реализован в прототипе инструмента VHDL2SDS. Инструмент поддерживает следующие алгоритмы очередности выполнения инструкций: «как можно раньше» (As Soon As Possible, ASAP), «как можно позже» (As Late As Possible, ALAP). Прототип инструмента был апробирован на простых описаниях на языке VHDL.

В работе [11] предложен метод проверки выполнения свойств (в частности, определения недостижимых путей в графе потока управления) для HDL-описаний на языке Verilog. Для этого строится вспомогательная структура данных, называемая графом зависимостей путей (Path Dependency Graph). В граф зависимостей путей попадают только те пути, в которых читаются или

определяются переменные, присутствующие в проверяемом свойстве. Затем методом обратных подстановок [12] (подход предлагается для выделенного класса HDL-описаний, содержащих ровно один процесс), определяется условие попадания в данный путь выполнения.

В работе [13] описывается метод поиска состояний гонки (races) в описаниях цифровой аппаратуры на языке SystemC [14]. Метод основан на построении графов потока управления и анализе процессов на коммутативность с помощью техники предикатной абстракции CEGAR [15] и инструмента проверки моделей Cadence SMV [16]. Инструментальная реализация SCOOT [17] предложенного метода основана на особенностях языка SystemC и его компонента выполнения SystemC Sheduler, поэтому адаптация данного метода для анализа описаний на других языках представляется трудоемкой. Инструмент SCOOT не апробировался на описаниях повышенной сложности.

В работе [18] описывается метод масштабируемой генерации тестовых стимулов на основе гибридной верификации HDL-описаний. Под гибридной верификацией в данном случае понимается комбинация статического анализа кода и имитационного верификации. В качестве входных данных методу подается целевой Verilog-модуль и длина тестовой последовательности. Предполагается, что модуль не содержит гонок и имеет только один входной сигнал синхроимпульса. Под отсутствием гонок следует понимать условие на определения переменных: каждая переменная должна определяться не более одного раза за один такт работы модуля. Длина тестовой последовательности задается вручную; её конкретное значение может быть задано на основе анализа покрытия кода или требований имеющимися тестовыми последовательностями, реализованными, например, методом случайной генерации.

Генерация тестовой последовательности осуществляется на модели вида графа потока управления. В предположении, что все пути выполнения в графе потока однотактные (каждый путь может быть пройден за один такт синхроимпульса), метод строит упорядоченную последовательность графов потока. Количество копий графов потока в последовательности задается равным длине тестовой последовательности; таким образом, для каждой тестовой последовательности, выполняемой за n тактов, существует путь в упорядоченной связанной последовательности графов потока.

На начальном этапе выполняется случайная генерация значений входных сигналов целевого модуля. Для сгенерированных значений выполняется симуляция модуля с параллельным сохранением информации о том, какой путь выполнения был покрыт. Для покрытого пути составляется ограничение в терминах текущих значений переменных (для блокирующих присваиваний) и значений переменных, которые будут им присвоены на следующем такте (для не блокирующих присваиваний). Информация об ограничениях и номере текущего такта сохраняется посредством инструментирования HDL-описания. Ограничения хранятся в стеке (вершиной стека является ограничение,

построенное в результате анализа последней инструкции HDL-описания, а, соответственно, дном стека является ограничение на первую инструкцию). Далее методом поиска в глубину выполняется инвертирование ограничений, начиная с вершины стека. Инвертируются только ограничения, извлеченные из условных блоков, ограничения на переменные, определяемые в блоках присваиваний, не изменяются, а служат только для генерации корректных тестовых воздействий. Если модифицированное таким образом ограничение оказывается неразрешимым, то оставляется исходное ограничение и инвертируется следующий элемент стека ограничений.

Узким местом данного подхода является необходимость частых обращений к решателю ограничений для проверки выполнимости каждого пути. Кроме того, в HDL-описаниях повышенной сложности количество путей выполнения может быть весьма велико и приводить к «комбинаторному взрыву» [19], т.е. делать их перебор чрезмерно длительной процедурой.

Для уменьшения числа путей выполнения, необходимых для генерации покрывающих тестовых последовательностей, предлагается использовать символические состояния. Символические состояния – это наборы ограничений на все внутренние переменные целевого модуля. Если на некотором такте две сгенерированные тестовые последовательности приводят к одному и тому же символическому состоянию, то избыточно генерировать для каждого из них все возможные пути дальнейшего выполнения. В методе предлагается для каждого построенного пути выполнения на каждом такте хранить достигнутое символическое состояние. Если для последующего пути на некотором такте будет достигнуто состояние, уже достигнутое ранее, то дальнейшее построение пути прекращается.

Большое внимание в данной работе уделяется вопросам эффективного хранения символических состояний и их сравнения без использования решателей. Предлагается хранить состояния в виде множеств инструкций ветвления в графе потока, по которым прошел путь, достигающий данного состояния. Тогда сравнение символических состояний сводится к сравнению множеств и может быть выполнено без использования решателя. В работе отмечается, что такой подход не позволяет выявлять эквивалентные ограничения, построенные путем обхода разных путей выполнения. Дополнительные оптимизации, описанные в методе (анализ зависимостей по данным для построения упрощенных ограничений, выявление заведомо противоречивых ограничений при инвертировании), направлены на уменьшение числа обращений к решателю. Метод реализован в проекте STAR [20] (STatic Analysis of RTL) и апробирован на HDL-описаниях небольшой сложности (не более 1500 строк кода в каждом). Эксперименты показали, что тесты, сгенерированные инструментом STAR, обеспечивают более высокое покрытие ветвей и путей, чем случайная генерация при меньшей длине тестов (в ряде случаев отмечено уменьшение длины тестовых последовательностей на три порядка).

В работе [21] предложен гибридный подход к генерации утверждений (assertion) с помощью техник статического анализа кода и анализа данных (data mining). В данном случае под анализом данных следует понимать выявление статистических закономерностей в изменениях значений выходных сигналов. На первом этапе метода выполняются прогон набора случайных тестов на целевом HDL-описании и сбор сведений о значениях, принимаемых выходными сигналами. На втором этапе выполняется анализ зависимостей по данным между переменными. В результате для каждой переменной, для которой предполагается сгенерировать утверждение, строится так называемый конус влияния [22] (cone-of-influence) – множество переменных и входных сигналов, значения которых определяют значение переменной. На третьем этапе выполняется статистический анализ данных, полученных в результате выполнения случайных трасс, с целью выявления статистически достоверных взаимосвязей между элементами конуса влияния и заданными переменными. Выявленные взаимосвязи формулируются в виде утверждений в терминах линейной темпоральной логики. Метод был реализован в инструменте GoldMine [23], построенные утверждения были проверены с помощью коммерческого инструмента формальной верификации Cadence Incisive [24], а также использованы для создания регрессионных тестов. Эксперименты демонстрируют применимость предложенного подхода и его программной реализации для верификации HDL-описаний повышенной сложности. Однако построенные инструментом утверждения могут содержать ошибки, т.к. они извлекаются из исходного кода, а не из спецификаций.

3.1 Программные срезы

Важным направлением в области статического анализа программного кода вообще и HDL-описаний в частности является извлечение и анализ программных срезов (program slicing). Срезом программы является подмножество её инструкций, которые могут привести программу к набору состояний (точек), называемому критерием среза (slicing criterion). Программные срезы можно рассматривать как подграфы графа потока управления исходного HDL-описания.

Исторически, первыми были разработаны техники извлечения срезов для программ на процедурных языках [25]. Одной из первых работ по извлечению программных срезов HDL-описаний является [26]. В ней предложен способ адаптации техник построения срезов процедурных программ для описаний на языке VHDL. Авторами статьи была разработана программная реализация подхода (с использованием коммерческого инструмента Codesurfer [27]), а также предложены варианты его дальнейшего использования для проектирования, тестирования (имитационной верификации, основанной на анализе результатов исполнения HDL-описания в симулируемом окружении) и формальной верификации (математически строгого доказательства выполнения свойств).

В работе [28] предложен подход к извлечению срезов для описаний на языке Verilog. Программная реализация подхода представляет собой надстройку над коммерческим Verilog-симулятором, получающую на вход HDL-описание и критерий среза, задаваемый пользователем. Компонент анализирует Verilog-код с помощью интерфейса VPI [29], строит исполнимый срез, генерирует упрощенное Verilog-описание и передает его симулятору для исполнения. Инструмент был апробирован на описаниях, часть кода которых содержали управляющие конечные автоматы (FSM, Finite State Machine). В качестве критериев среза задавались внутренние переменные, определяющие состояние. Полученные на выходе автоматы состояний вручную сравнивались со спецификациями соответствующих аппаратных компонентов.

В работе [30] предложено развитие подходов к извлечению срезов в HDL-описаниях посредством наложения дополнительных ограничений (в виде предикатов логики первого порядка) на начальные значения выбранных переменных или входных сигналов целевой системы. В таком случае критерий среза называется условным критерием среза (conditioned slicing criterion). С точки зрения верификации, интересным классом условий являются требования на входные стимулы и возвращаемые реакции, выраженные в терминах линейной темпоральной логики [31] (LTL, Linear Temporal Logic). Для такого класса условий вводится понятие предшествующего условного среза (antecedent conditioned slice) как последовательности точек программы, в каждой из которых выполнен условный критерий среза. Предложенный метод был апробирован на Verilog-реализации протокола USB 2.0. По спецификации протокола был составлен набор свойств, заданных в терминах логики LTL. Для каждого свойства извлекался предшествующий условный срез, который проверялся на выполнимость с помощью инструмента проверки моделей Cadence SMV [16]. Использование техники извлечения условных срезов существенно сократило время верификации и не исказило конечный результат.

Статья [32] посвящена использованию срезов для локализации ошибок. Предполагается, что для некоторого иерархического HDL-описания уже имеется набор функциональных тестов, причем каждый тест был выполнен на искомом HDL-описании хотя бы один раз и завершился успешно или с ошибкой. Предложенный метод локализации ошибок основан на извлечении срезов из графа экземпляров (IG, Instantiation Graph) HDL-описания и анализе покрытия инструкций среза имеющимися тестами при динамической верификации. Метод реализован в открытой расширяемой среде проектирования и верификации ZamiaCAD [33] и был применен для выявления ошибочных инструкций в коде процессора ROBSY.

4. Автоматные модели

В теории алгоритмов под абстрактным автоматом понимается пятерка $A = (S,I,O,\delta,\lambda)$, где S – конечное множество состояний, I и O – соответственно

входной и выходной алфавиты, из символов которых формируются строки, считываемые и выдаваемые автоматом, $\delta : S \times X \rightarrow S$ – функция переходов, $\lambda : S \times X \rightarrow Y$ – функция выходов [34]. Автоматные модели широко применяются в верификации программных систем, в том числе таких критически важных, как реализации телекоммуникационных протоколов [35] и операционные системы реального времени [36]. В области аппаратных систем автоматные модели (в частности, конечные автоматы, речь о которых пойдет далее) активно используются в качестве эталонных моделей при имитационной верификации [37]. В данном разделе речь пойдет о методах извлечения автоматных моделей из HDL-описаний.

Здесь уместно отметить, что существующие коммерческие инструменты поддержки проектирования цифровой аппаратуры зачастую тоже используют техники построения моделей. В частности, особое внимание уделяется извлечению моделей вида конечных автоматов (FSM, Finite State Machine). Как правило, данные техники основаны на выявлении в коде специальных шаблонов и конструкций, заранее определенных в документации для описания конечно-автоматных моделей. Например, в работе [38] представлено описание стиля программирования конечных автоматов, принятого в компании Cisco. Использование аналогичных подходов с одной стороны, позволяет эффективным образом строить и анализировать модели даже для описаний повышенной сложности. С другой стороны, использование коммерческих инструментов приводит к необходимости следовать жестко заданным правилам проектирования во всех реализуемых проектах и порождает зависимость процесса проектирования от выбранной технологии.

4.1 Конечные автоматы

В данном разделе представлены существующие методы извлечения моделей типа конечных автоматов. Абстрактный автомат называется конечным, если конечно его множество состояний [39].

В работе [40] предлагается новый метод логического синтеза HDL-описаний в более низкоуровневое представление в виде списка соединений (netlist). В качестве промежуточных стадий метод содержит построение графа потока управления для каждого процесса, а также извлечение конечного автомата (далее – FSM-модели) для каждого графа потока управления. Извлечение FSM-моделей возможно только в том случае, если граф потока управления процесса содержит инструкции wait, зависящие от одного и того же события, связанного с одним и тем же сигналом синхроимпульса. Каждой инструкции wait, обладающей указанным свойством, в FSM-модели сопоставляется состояние. Переходы конечного автомата строятся на основе наличия путей между wait-инструкциями в графе потока управления.

Работа [41] также посвящена решению задачи логического синтеза HDL-описаний в список соединений. В качестве промежуточного этапа используется выявление блоков типа FSM-моделей в исходном коде. Авторы

подхода экспериментально доказывают, что список соединений, сгенерированный на основе комбинации блоков типа конечных автоматов, и блоков, содержащих комбинационную логику, обладает меньшим временем выполнения. Блоки типа конечных автоматов определяются путем сопоставления с шаблонами, заданными авторами подхода. Всего представлено три шаблона конечно-автоматных блоков, каждых из которых содержит циклические зависимости по управлению между блоками, определяющими переменную состояния, и блоками, выполняющими промежуточные вычисления (комбинационная логика). Метод дает нетривиальные результаты, если в исходном HDL-описании есть процессы, имеющие непустые списки чувствительности, а также содержащие ровно одну переменную состояния. Поддерживается иерархическая структура HDL-описаний, никаких других ограничений на исходный код не накладывается.

В работе [42] предложен метод оптимизации логического синтеза за счет извлечения конечных автоматов из описаний на языках VHDL и Verilog. В основе метода лежит классификация процессов целевого HDL-описания по следующим множествам: процессы, выполняемые при возникновении событий над синхросигналами (clock); процессы, содержащие переходы между состояниями конечного автомата (transition); процессы, генерирующие выходные сигналы (output). Предполагается, что конечный автомат может быть «размыт» по процессам различных классов. На код HDL-описания накладывается существенное ограничение: описание должно содержать только FSM-модель. Также должны быть заранее известны переменные состояния, синхросигналы и сигналы сброса.

В работе [43] предложен поход к выявлению в коде HDL-описаний внутренних переменных, задающих состояния конечных автоматов. Авторы утверждают, что протоколы взаимодействия между компонентами индустриальных HDL-описаний зачастую удобно выражать именно в виде FSM-моделей. Предполагается использовать найденные в исходном коде конечные автоматы для декомпозиции HDL-описаний на более простые для верификации подсистемы, а также для оптимизации логического синтеза и, возможно, снижения энергопотребления синтезированных аппаратных систем. Предложенный метод определения переменных состояния состоит из следующих этапов:

- построение абстрактного дерева синтаксиса [14];
- построение графа потока данных на уровне сигналов;
- построение графа зависимостей между внутренними переменными;
- определение переменных состояния.

На первом этапе используется классический подход [44]. На втором этапе строится граф потоков данных между параллельно выполняемыми

компонентами HDL-описания. Вершинами графа являются компоненты, ребрами – зависимости. Существует четыре типа ребер, соответствующих зависимостям по *управлению* (control), по *данным* (data), по *синхросигналу* (clock) и по *сбросу* (reset). В работе не описано, как идентифицируются зависимости по синхросигналу и по сбросу. Зависимости по управлению определяются как использования переменных, присутствующих в условных операторах. В остальных случаях зависимости трактуются как зависимости по данным. На третьем этапе строится граф зависимостей между внутренними переменными (это уточнение графа потока данных до уровня отдельных сигналов). Четвертый этап заключается в выявлении в графе переменных, удовлетворяющих следующим свойствам:

- хотя бы одна исходящая дуга для данной переменной является циклической;
- хотя бы одна исходящая дуга для данной переменной задает зависимость по управлению;
- все входные зависимости по данным являются зацикленными на себя (self-loop), либо являются зависимостями от констант.

Если для некоторой внутренней переменной все условия выполнены, то переменная является переменной состояния FSM.

Предложенный метод был реализован на языке C++ в инструменте Asynchronous Verilog Synthesizer [45] и апробирован на нескольких промышленных Verilog-описаниях. Утверждается, что метод дает ложные срабатывания (в качестве состояний иногда определяются переменные, таковыми не являющиеся). Однако, процент таких срабатываний невысок (на трех проанализированных системах он не превышал 10%). Процедура проверки корректности извлеченных переменных состояния в работе детально не описана.

В работе [46] описан метод извлечения конечных автоматов из описаний на языке Verilog. Предлагается анализировать построенные конечные автоматы автоматически с помощью инструмента проверки моделей NuSMV [47]. Метод позволяет преобразовывать только HDL-описания, реализующие логику FSM. Концептуальной особенностью метода является автоматическое определение внутренних переменных, задающих состояние конечного автомата. В качестве таковых определяются переменные, чтение которых выполняется синхронно с возникновением события над синхросигналом (например, приход переднего фронта), а значения циклическим образом зависят от них самих в смысле зависимостей по данным.

4.2 Расширенные конечные автоматы

Значимым в контексте применимости к моделированию аппаратных комплексов частным случаем автоматных моделей являются *расширенные конечные автоматы*. В дополнение к свойствам конечного автомата, расширенный конечный автомат (EFSM, Extended Finite State Machine) имеет собственные переменные (входные, выходные и внутренние), а его переходы имеют особую структуру. Каждый переход содержит *охранное условие* (булевский предикат) и *действие* (последовательность инструкций). Переход срабатывает только в том случае, если выполнено охранное условие, тогда выполняется действие.

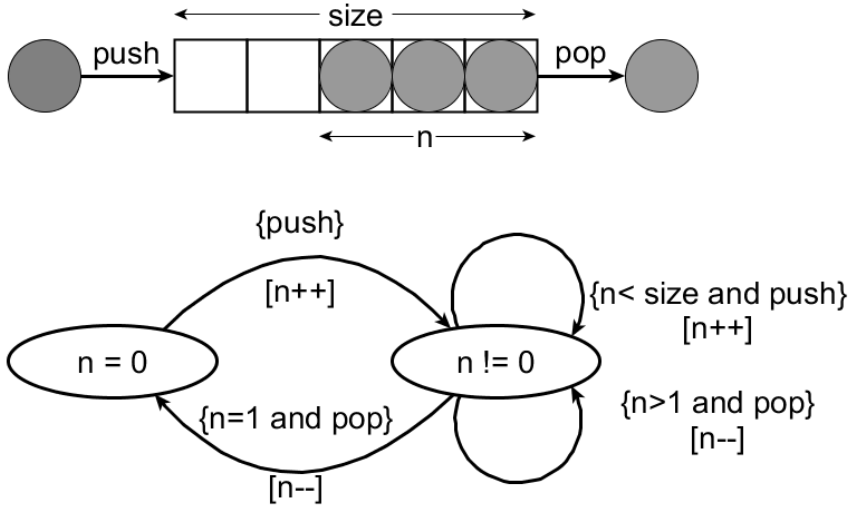


Рис. 1. Пример расширенного конечного автомата

На рис. 1 представлен пример простой очереди (FIFO, First In First Out) и расширенного конечного автомата, моделирующего её поведение. Очередь может принимать на вход воздействия типа push (добавление элемента) и pop (удаление элемента). Размер очереди равен size, а текущее значение степени заполнения – n. Расширенный конечный автомат имеет два состояния, соответствующих пустой и непустой очереди. Для каждого перехода указаны охранные условия (в фигурных скобках) и действия (в квадратных скобках).

В работе [48] описан метод автоматической генерации тестовых последовательностей для HDL-описаний на языках VHDL и BESTMAP-C [49]. Тестовые последовательности нацелены на выявление *постоянных ошибок* (stuck-at faults) типа разрыва цепи, заключающихся в постоянной генерации неверных значений сигналов HDL-описания. Процедура генерации состоит из двух фаз. На первой фазе выполняется автоматизированное извлечение

расширенных конечных автоматов из исходного кода; на второй фазе производится генерация тестовых последовательностей, покрывающих состояния и ветви полученного автомата. Процедура извлечения EFSM-моделей работает при следующих ограничениях:

- заранее определены переменные, задающие состояние HDL-описания;
- тела процессов анализируются по отдельности, возможная иерархическая структура HDL-описаний никак не учитывается, равно как и список чувствительности процесса;
- каждый путь исполнения в HDL-описании содержит две непрерывных (возможно, пустых) и непересекающихся последовательности блоков – условных операторов и операторов присваивания соответственно.

В статье детально обсуждаются процедуры *ортогонализации* охранных условий (enabling function) на пути HDL-описания и стабилизации (т.е. сокращения до одного элемента) множества выходных состояний для каждого перехода EFSM-модели. Однако способы сравнения состояний (для определения эквивалентных) и выявления недостижимых переходов авторами опускаются. Метод генерации функциональных векторов описан неформально. Недостатком метода является экспоненциальный рост числа состояний при стабилизации.

В работе [50] предложен метод автоматизированной генерации модульных функциональных тестов для HDL-описаний на языке VHDL. В основе метода лежит генерация по исходному коду моделей в виде EFSM и генерации по ним тестов методом обхода с помощью техники переходов с возвратами (backjumping). Предполагается наличие дополнительной информации о том, какими переменные задают состояние искомого модуля, а какие – являются синхросигналами.

Первый шаг метода состоит в извлечении исходной модели (REFSM, Reference EFSM), функционально эквивалентной HDL-описанию. Исходная модель всегда содержит ровно одно символическое состояние, а все переходы модели (соответствующие путям выполнения HDL-описания с точностью до выбора ветви в операторах switch/case) являются замкнутыми на него. Следующий шаг метода направлен на устранение условных операторов типа if-then-else из действий переходов. Это свойство достигается путем расщепления действий переходов на более простые, не содержащие условные операторы. Расщепление состоит во введении дополнительных символических состояний, поэтому построенная на данном этапе модель носит название наибольшей модели (LEFSM, Largest EFSM). Помимо увеличения числа состояний, наибольшая модель также обладает следующим недостатком – она не является функционально эквивалентной HDL-описанию. Устранение

указанных недостатков происходит на следующем шаге метода, где выполняется объединение совместных переходов. Под совместными переходами понимаются такие, что их охранные условия могут принимать истинное значение независимо друг от друга, а также не имеющие общих переменных. В данном случае под общими переменными для пары переходов подразумеваются такие, которые определяются в действии первого перехода и используются в охранном условии второго перехода. Если данные условия не выполнены, и переходы объединять нельзя, однако для достижения функциональной эквивалентности это необходимо, то допускается исправление кода HDL-описания вручную с тем, чтобы комбинируемые переходы не содержали общих переменных. Последний шаг метода состоит в построении полу-стабилизированной EFSM-модели (S2EFSM, semi-stabilized EFSM). На данном шаге из охранных условий переходов выделяются ограничения на переменные состояния; эти ограничения сопоставляются построенным символическим состояниям. Предложенный метод был использован для построения на его основе генератора модульных функциональных тестов и был апробирован на HDL-описаниях небольшой сложности.

В работе [51] предлагается использовать EFSM-модели для выполнения полуформальной (semi-formal) верификации HDL-описаний. Подход использует техники имитационной верификации с использованием случайной генерации стимулов на основе ограничений (constrained random simulation), обхода состояний EFSM-модели и анализа достижимости граничных случаев с помощью инструмента проверки свойств NuSMV.

Целью извлечения моделей из HDL-описаний может являться абстрагирование – представление аппаратуры на более высоком уровне, чем потактовое описание поведения. Техника абстрагирования применяется для ускорения симуляции [52], а также для ко-симуляции представлений аппаратуры, описанных на разных уровнях абстракции. В работе [53] представлен метод трансляции HDL-описаний в представление уровня транзакций (TLM, Transaction-level Modeling) [54] с целью их совместной симуляции с модулями, уже реализованными в терминах TLM. Основными примитивами в TLM-представлении являются сообщения (наборы сигналов), которыми модули обмениваются по установленным между ними каналам. Последовательности обменов сообщениями носят название транзакций. Предложенный метод трансляции HDL-описаний основан на построении EFSM-моделей в качестве промежуточного представления (для этого используется метод [49]). Полученные расширенные конечные автоматы разделяются на под-фазы, соответствующие чтению входных сигналов (input subphase), выполнению промежуточных вычислений (elaboration subphase) и генерации выходных сигналов (output subphase). Разделение выполняется по результатам анализа потоков данных. Полученные под-фазы преобразуются в макро-состояния EFSM-модели (логика охранных условий и действий в

переходах под-фаз сохраняется в действиях переходов EFSM-модели), которые затем автоматически транслируется в TLM-представление. На завершающем этапе генерации TLM-представления во входное макро-состояние добавляется примитив TLM-транзакции чтения, а в выходное макро-состояние – примитив TLM-транзакции записи. Метод реализован в виде компонента среды HIFSuite [55] и апробирован на описаниях небольшой сложности. Результаты демонстрируют уменьшение времени симуляции при значительном сокращении времени разработки TLM-модели по сравнению с разработкой вручную.

Работа [56] содержит описание метода извлечения EFSM-моделей из описаний на языках Verilog и VHDL. Метод принимает на вход только HDL-описание, никакой дополнительной информации для анализа не требуется. На первом этапе строится граф потока управления. На втором этапе граф преобразуется в систему охраняемых действий посредством «подъема» внутренних условных операторов на верхний уровень. Под охраняемым действием в данном случае понимается пара «охранное условие - действие»; данное представление активно используется для описания асинхронных систем [57]. Затем с помощью эвристик определяются переменные синхроимпульса и переменные состояния. Состояния EFSM-модели строятся в виде ограничений на переменные состояния, построенные путем ортогонализации охранных условий. Переходы EFSM-модели строятся путем проверки совместности условий охраняемых действий (для начальных вершин переходов) и слабейших предусловий (для конечных вершин) с условиями, задающими состояния. Предложенный метод был реализован в инструменте Retrascope [58] и апробирован на HDL-описаниях небольшой сложности.

4.3 Прочие автоматные модели

Статья [59] посвящена техникам абстракции представлений цифровой аппаратуры. Предлагаемый подход строит на основе конечного автомата и сведений о переменных автомата более компактную структуру данных, называемую автоматом потока управления (ECFM, Extracted Control Flow Machine). Под сведениями о переменных следует понимать классификацию на переменные управления (задающие состояние автомата) и регистры данных. Предполагается, что такие сведения предоставляются разработчиком HDL-описания. Регистры данных, как правило, имеют большую размерность, и именно они зачастую являются причиной экспоненциального роста числа состояний в конечных автоматах. Построенный автомат потока управления предлагается использовать для оценки покрытия имеющихся тестовых наборов (в данной работе речь идет не о функциональных тестах, а о тестах готовой продукции). Используются две основные метрики покрытия – покрытие состояний ECFM-модели и переходов ECFM-модели. Если обнаруживается переход или состояние ECFM-модели, не покрываемые существующим тестовым набором, то генерируется тест, нацеленный на

покрытие указанного компонента. Поскольку тесты, построенные для более абстрактной, нежели исходное HDL-описание, ECFM-модели, в общем случае не могут быть корректно применены к исходной модели, задача генерации нацеленных тестов возлагается на сторонние инструменты (в статье предлагается использовать HITEC [60]).

Работа [61] тоже посвящена проблеме абстракции HDL-описаний. В частности, в ней описан метод трансформации представлений вида конечных автоматов в так называемые семантические конечные автоматы (SFSM, Semantic Finite State Machine). По сравнению с FSM-моделями, SFSM-модель обладает следующей особенностью: каждый переход между состояниями снабжен булевским условием на входные сигналы и переменные состояния (enabling function); функцией вычисления конечного состояния, зависящей от входных сигналов и переменных состояния (update function); функцией вычисления значений выходных сигналов, зависящей от входных сигналов и переменных состояния (action function). Состояния SFSM-модели называются семантическими и определяются наборами переменных состояния исходной FSM-модели. Поскольку SFSM-модель не хранит сведений о конкретных значениях входных сигналов и переменных состояния, размер пространства состояний в ней не превосходит (а в ряде экспериментов оказывается значительно меньше) размера пространства состояний FSM-модели, что потенциально ускоряет её функциональное тестирование на основе обхода графа состояний. Метод построения SFSM-модели по FSM-модели состоит из следующих шагов:

- построение *дерева инструкций* (statement tree);
- построение *семантических состояний*;
- вычисление булевских условий переходов;
- построение переходов SFSM-модели.

Рассмотрим шаги метода более подробно. Построение дерева инструкций проводится посредством обработки *синхронной секции* (synchronous section) FSM-модели, включающей функции вычисления следующего состояния и выходных сигналов. Нетерминальные вершины дерева содержат взаимоисключающие условия на входные сигналы и переменные состояния. Листья дерева содержат функции вычисления переменных состояния и выходных сигналов. На этапе построения семантических состояний анализируются функции вычисления, заданные в листьях дерева инструкций. Если для нескольких листовых вершин соответствующие функции эквивалентны, то листовые вершины объявляются соответствующими одному и тому же семантическому состоянию. Условия переходов вычисляются как конъюнкции условий, заданных в нетерминальных вершинах дерева, для каждого пути в дереве. Построение SFSM-модели завершается тем, что для

каждого семантического состояния определяются допустимые исходящие переходы, посредством проверки их булевских условий. Переходы с противоречивыми условиями отбрасываются.

Приведенные результаты экспериментов показывают, что тестовые наборы, построенные методом обхода графа состояний SFSM-модели, могут обеспечивать более высокое покрытие инструкций исходного кода, чем наборы случайных тестов, даже при условии существенно бо́льшего размера последних.

5. Заключение

В статье сделан обзор существующих методов извлечения моделей из исходного кода описаний цифровой аппаратуры (HDL-описаний). Рассмотренные методы сгруппированы по типам моделей: программные срезы, графы потока, конечные автоматы. Возможна также классификация методов по иному критерию, а именно - по решаемым с их помощью задачам (оптимизация кода ([7], [9], [40], [41], [42], [43]), абстракция ([53], [59], [61]), тестирование ([8], [11], [13], [18], [21], [26], [28], [30], [32], [38], [43], [46], [48], [50], [51], [56])). Ключевыми проблемами существующих подходов можно считать необходимость получения дополнительной информации от пользователя для построения моделей, более адекватно представляющих те или иные аспекты целевой аппаратной системы, а также трудности при работе с HDL-описаниями повышенной сложности. В некоторых случаях программные реализации методов построения моделей являются существенно зависимыми от особенностей языка описания аппаратуры ([13]).

Вместе с тем нельзя не отметить тот факт, что разные типы моделей демонстрируют разную эффективность в решении разных задач. К примеру, в контексте функциональной верификации методы построения программных срезов демонстрируют высокие результаты в задачах генерации нацеленных тестов, в то время как методы построения расширенных конечных автоматов позволяют генерировать тестовые последовательности, обеспечивающие полное покрытие исходного кода. Несомненно, актуальной задачей является разработка программных систем, обеспечивающих интеграцию различных подходов к построению и преобразованию моделей цифровой аппаратуры.

Список литературы

[1]. А.С. Камкин, А.М. Коцыняк, С.А. Смолов, А.А. Сортов, А.Д. Татарников, М.М. Чупилко. Средства функциональной верификации микропроцессоров. Труды ИСП РАН, т. 26, вып. 1, 2014 г., стр. 149-200. doi: 10.15514/ISPRAS-2014-26(1)-5

[2]. Statistical Analysis of Floating Point Flaw: Intel White Paper Section 3. http://www.intel.com/support/processors/pentium/sb/CS-013007.htm. Дата публикации: 08.07.2004.

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

[4]. IEEE Standard for Verilog Hardware Description Language, 1364-2005, IEEE, 2006, 560 p.
[5]. IEEE Standard VHDL Language Reference Manual, 1076-2008, IEEE, 2009, 626 p.
[6]. A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman. Compilers: Principles, Technologies, and Tools (2nd Edition). Addison Wesley, 2006. 1000 p.
[7]. A. Balboni, M. Mastretti, M. Stefanoni. Static Analysis for VHDL Model Evaluation. Proceedings of the European Design Automation Conference (EURO-DAC '94), IEEE Computer Society Press, Los Alamitos, 1994, pp. 586-591.
[8]. L. Baresi, C. Bolchini, D. Sciuto. Software Methodologies for VHDL Code Static Analysis based on Flow Graphs. Proceedings of the European Design Automation Conference (EURO-DAC '94), IEEE Computer Society Press, Los Alamitos, 1994, pp. 406-411.
[9]. D.-C. Peixoto, D. Silva Jr., J.-M. Mata, C.-N. Coelho Jr., A.-O. Fernandes. Translation of hardware description languages to structured representation: a tool for digital system analysis. Proceedings of 13th Symposium on Computer Architecture and High Performance Computing (SBAC - PAD), 2001, Pirenypolis.
[10]. VAUL. A VHDL Analyzer and Utility Library. http://www-dt.e-technick.uin-dortmund.de/~mvo/vaul/ University of Dortmund, Department of Electrical Engineering, AG SIV, 1994.
[11]. M. Zaki, Y. Mokhtari, S. Tahar. A Path Dependency Graph for Verilog Program Analysis. Proceedings of the IEEE Northeast Workshop on Circuits and Systems (NEWCAS'03), Montreal, 2003, pp. 109-112.
[12]. R. Floyd. Assigning meanings to programs. Proceedings of Symposia in Applied Mathematics, vol. 19, 1967, pp. 19-32.
[13]. N. Blanc, D. Kroening. Race Analysis for SystemC using Model Checking. ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 15 issue 3, 2010, pp. 356 – 363. doi: 10.1109/ICCAD.2008.4681598
[14]. IEEE Standard SystemC Language Reference Manual, 1666-2005, IEEE, 2003, 428 p.
[15]. E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Counterexample-Guided Abstraction Refinement. Computer Aided Verification, Lecture Notes in Computer Science vol. 1855, 2000, pp. 154-169.
[16]. Cadence SMV Symbolic Model Checker. http://www.kenmcmil.com/smv.html. Дата обращения: 02.03.2015.
[17]. SCOOT A Tool for the Static Analysis of SystemC. http://www.cprover.org/scoot/. Дата обращения: 04.03.2015.
[18]. L. Liu, S. Vasudevan. Scaling Input Stimulus Generation through Hybrid Static and Dynamic Analysis of RTL. ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 20 issue 1, 2014. doi:10.1145/2676549
[19]. P. Godefroid. Compositional dynamic test generation. Proceedings of the 34th Annual ACM Symposium on Principles of Programming Languages (SIGPLAN-SIGACT), 2007, pp. 47-54. doi: 10.1145/1190216.1190226
[20]. STAR. http://users.crhc.illinois.edu/liu187/. Дата обращения: 04.03.2015.
[21]. S. Hertz, D. Sheridan, S. Vasudevan. Mining Hardware Assertions With Guidance From
[22]. Static Analysis. Computer-Aided Design of Integrated Circuits and Systems, Transactions on IEEE, vol. 32, issue 6, 2013, pp. 952-965. doi: 10.1109/TCAD.2013.2241176
[23]. E. Clarke, O. Grumberg, D. Peled. Model checking. The MIT Press, 1999, 314 p.
[24]. GoldMine. http://goldmine.csl.illinois.edu/. Дата обращения: 04.03.2015.

[25]. Cadence Incisive. http://www.cadence.com/products/fv/iv_kit/pages/default.aspx. Дата обращения: 04.03.2015.
[26]. M. Weiser. Program slicing. IEEE Transactions on Software Engineering, vol. 10, issue 4, 1984, pp. 352–357.
[27]. E.M. Clarke, M. Fujita, S.P Rajan, T. Reps, S. Shankar. Program Slicing of Hardware Description Languages. Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods, pp. 298-302.
[28]. Codesurfer. http://www.grammatech.com/research/technologies/codesurfer. Дата обращения: 02.03.2015
[29]. T. Li, Y. Guo, S.-K. Li. Automatic Circuit Extractor for HDL Description Using Program Slicing. Journal of Computer Science and Technology vol. 19, issue 5, pp. 718-728. doi: 10.1007/BF02945599
[30]. C. Dawson, S.K. Pattanam, D. Roberts. The Verilog Procedural Interface for the Verilog Hardware Description Language. Proceedings of Verilog HDL Conference, 1996, pp. 17-23. doi: 10.1109/IVC.1996.496013
[31]. S. Vasudevan, E. A. Emerson, J. A. Abraham. Efficient Model Checking of Hardware Using Conditioned Slicing. Proceedings of the Fouth International Workshop on Automated Verification of Critical Systems (AVoCS), Vo1. 28, issue 6, 2005, pp. 279–294. doi: 10.1016/j.entcs.2005.04.017
[32]. M. Huth, M. Ryan. Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge University Press, New York, 2004, 440 p.
[33]. A. Tepurov, V. Tihhomirov, M. Jenihhin, J. Raik, G. Bartsch, J.H. Meza Escobar, H. Wuttke. Localization of Bugs in Processor Designs Using ZamiaCAD Framework. Proceedings of 13th International Workshop on Microprocessor Test and Verification (MTV), 2012, pp. 41-47.
[34]. ZamiaCAD. http://zamiacad.sourceforge.net/web/. Дата обращения: 06.03.2015.
[35]. J. Hopcroft, R. Motwani, J. Ullman. Introduction to Automata Theory: Languages, and Computation. Pearson/Addison Wesley, 2007, 535 p.
[36]. M. Yuang. Survey of protocol verification techniques based on finite state machine models. Proceedings of the Computer Networking Symposium, 1988, pp. 164-172.
[37]. R. Obermaisser, C. El-Salloum, B. Huber, H. Kopetz. Modeling and Verification of Distributed Real-Time Systems using Periodic Finite State Machines. Journal of Computer Systems Science & Engineering, vol. 22, No. 6, 2007.
[38]. А.С. Камкин. Метод формальной спецификации аппаратуры с конвейерной организацией и его приложение к задачам функционального тестирования. Труды ИСП РАН, Вып. 16, 2009 г., стр. 107-128.
[39]. T.-H. Wang, T. Edsall. Practical FSM Analysis for Verilog. Proceedings of Verilog HDL Conference and VHDL International Users Forum, 1998, pp. 52-58. doi:10.1109/IVC.1998.660680
[40]. A. Gill. Introduction to the Theory of Finite-state Machines. McGraw-Hill, 1962, 207 p.
[41]. J.-C. Giomi. Method of Extracting Implicit Sequential Behavior from Hardware Description Languages. 5.774.370, USA, 531.996, 18.09.1995, 30.06.1998, pp. 1-44.
[42]. C.-N. Liu, J.-Y. Jou. A FSM Extractor for HDL Description at RTL Level. Proceedings of the 2nd International Symposium on Quality Electronic Design, 2001, p. 372.
[43]. M. E. J. Gilford, G. N. Walker, J. L. Tredinnick, M. W. P. Dane, M. J. Reynolds. Recognition of a State Machine in High-Level Integrated Circuit Description Language Code. 7.152.214 B2, USA, 10/736.967, 15.12.2003, 19.12.2006, pp. 1-32.

[44]. W. Song, J. Garside. Automatic Controller Detection for Large Scale RTL Designs. Euromicro Conference on Digital System Design (DSD), Los Alamitos, CA. 2013, pp. 844-851. doi: 10.1109/DSD.2013.94

[45]. G. D. Micheli. Synthesis and Optimization of Digital Circuits. McGraw-Hill Science/Engineering/Math, 1994, 576 p.

[46]. Asynchronous Verilog Synthesizer. http://wsong83.github.io/index.html. Дата обращения: 05.03.2015.

[47]. A. Höller, C. Preschern, C. Steger, C. Kreiner, A. Krieg, H. Bock, J. Haid. Automatized High-Level Evaluation of Security Properties for RTL Hardware Designs. Proceedings of the Workshop on Embedded Systems Security, No. 6, 2013, pp. 1-8. Doi:10.1145/2527317.2527323

[48]. NuSMV. http://nusmv.fbk.eu/. Дата обращения: 05.03.2015.

[49]. K.-T. Cheng, A.S. Krishnakumar. Automatic Generation of Functional Vectors Using the Extended Finite State Machine Model. ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 1 issue 1, 1996, pp. 57-79. doi: 10.1145/225871.225880

[50]. J-Y. Jou, S. Rothweiler, R. Ernst, S. Sutarwala, A. Prabhu. BESTMAP: Behavioral Synthesis from C. In International Workshop on Logic Synthesis (Research Triangle Park, NC, May), 1998.

[51]. G. D. Guglielmo, L. D. Guglielmo, F. Fummi, G. Pravadelli. Efficient Generation of Stimuli for Functional Verification by Backjumping Across Extended FSMs. Journal of Electronic Testing, vol. 27, issue 2, 2011, pp. 137-162. Doi: 10.1007/s10836-011-5209-8

[52]. G. Di Guglielmo, F. Fummi, G. Pravadelli, S. Soffia, M. Roveri. Semi-formal functional verification by EFSM traversing via NuSMV. IEEE International High Level Design Validation and Test Workshop (HLDVT), 2010, pp. 58-65. doi: 10.1109/HLDVT.2010.5496660

[53]. D. Kim, M. Ciesielski, K. Shim, S. Yang. Temporal Parallel Simulation: A Fast Gate-level HDL Simulation Using Higher Level Models. Proceedings of the Conference on Design, Automation and Test in Europe (DATE), 2011. doi: 10.1109/DATE.2011.5763251

[54]. N. Bombieri, F. Fummi, G. Pravadelli. Automatic Abstraction of RTL IPs into Equivalent TLM Descriptions. IEEE Transactions on Computers, vol. 60, issue 12, 2011, pp. 1730-1743. doi: 10.1109/TC.2010.187

[55]. OSCI TLM-2.0. http://www.accellera.org/resources/videos/tlm20andsubset/. Дата создания: 22.02.2010.

[56]. HIFSuite. https://www.edalab.it/. Дата обращения: 05.05.2015.

[57]. А.С. Камкин, С.А. Смолов. Метод извлечения EFSM-моделей из HDL-описаний: применение к функциональной верификации. Сборник трудов конференции «Проблемы разработки перспективных микро- и наноэлектронных систем» под общ. ред. академика РАН А.Л. Стемпковского, часть 2, 2014, стр. 113-118.

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

[59]. Retrascope. http://forge.ispras.ru/projects/retrascope/. Дата обращения: 05.03.2015.

[60]. D. Moundanos, J.A. Abraham, Y.V. Hoskote. Abstraction Techniques for Validation Coverage Analysis and Test Generation. IEEE Transactions on Computers, IEEE, 1998, vol. 47, issue 1, pp. 2-14. doi:10.1109/12.656068

[61]. T.M. Niermann, J.H. Patel. HITEC: A Test Generation Package for Sequential Circuits. Proceedings of International European Design Automation Conference (EDAC), 1996, pp. 875-884. doi:10.1109/EDAC.1991.206393

[62]. C.-N. Jimmy Liu, J.-Y. Jou. An Efficient Functional Coverage Test for HDL Descriptions at RTL. Proceedings of International Conference on Computer Design (ICCD), Austin, TX, 1999, pp. 325-327. doi:10.1109/ICCD.1999.808561

A Survey of Methods for Model Extraction from HDL Descriptions

S.A. Smolov <smolov@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. In this paper a survey of existing methods of model extraction from hardware system descriptions written in Hardware Description Languages (like Verilog and VHDL) is presented. There are many tasks in hardware and software design where models are applied. The most actual tasks that are mentioned in this paper are: code optimization, logical synthesis optimization, model abstraction, and functional verification. The model categories that are mostly described here are flow or dependency graph models and automata models. As for flow graphs or dependency graphs, the methods of program slices extraction are described in details. Program slices can be characterized as suitable enough for directed test generation. Almost all the described automata models are finite state machine models and extended finite state machine models and so methods of such models extraction are the most popular for logical synthesis optimization and for functional test generation. The tests that can be generated from automata models shows high coverage of the target description. The key problems of existing model extraction methods are: the complexity of an application to industrial hardware descriptions (because of their complex structure), lack of automation (sometimes the hardware designer's knowledge is needed), the absence of open-source implementations. Also it is an actual task to create extendible frameworks for integration of different model extraction and analysis methods. Such framework can help in development of effective hybrid methods for hardware synthesis and verification.

Keywords: hardware description languages; model extraction; static analysis; code optimization; model abstraction; logical synthesis; functional verification; program slicing; flow graphs; dependency graphs; finite state machines; extended finite state machines.

DOI: 10.15514/ISPRAS-2015-27(1)-6

For citation: Smolov S.A. A Survey of Methods for Model Extraction from HDL Descriptions. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 1, 2015, pp. 197-124 (in Russian). DOI: 10.15514/ISPRAS-2015-27(1)-6

References

[1]. A. Kamkin, A. Kotsynyak, S. Smolov, A. Sortov, A. Tatarnikov, M. Chupilko. Sredstva funktsionalnoy verifikatsii mikroprotsessorov [Tools for Functional Verification of Microprocessors]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 26, issue 1, 2014, pp. 149-200. doi: 10.15514/ISPRAS-2014-26(1)-5

[2]. Statistical Analysis of Floating Point Flaw: Intel White Paper Section 3. http://www.intel.com/support/processors/pentium/sb/CS-013007.htm. Access date: 08.07.2004.

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

[4]. IEEE Standard for Verilog Hardware Description Language, 1364-2005, IEEE, 2006, 560 p.

[5]. IEEE Standard VHDL Language Reference Manual, 1076-2008, IEEE, 2009, 626 p.

[6]. A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman. Compilers: Principles, Technologies, and Tools (2nd Edition). Addison Wesley, 2006. 1000 p.

[7]. A. Balboni, M. Mastretti, M. Stefanoni. Static Analysis for VHDL Model Evaluation. Proceedings of the European Design Automation Conference (EURO-DAC '94), IEEE Computer Society Press, Los Alamitos, 1994, pp. 586-591.

[8]. L. Baresi, C. Bolchini, D. Sciuto. Software Methodologies for VHDL Code Static Analysis based on Flow Graphs. Proceedings of the European Design Automation Conference (EURO-DAC '94), IEEE Computer Society Press, Los Alamitos, 1994, pp. 406-411.

[9]. D.-C. Peixoto, D. Silva Jr., J.-M. Mata, C.-N. Coelho Jr., A.-O. Fernandes. Translation of hardware description languages to structured representation: a tool for digital system analysis. Proceedings of 13th Symposium on Computer Architecture and High Performance Computing (SBAC - PAD), 2001, Pirenypolis.

[10]. VAUL. A VHDL Analyzer and Utility Library. http://www-dt.e-technick.uin-dortmund.de/~mvo/vaul/ University of Dortmund, Department of Electrical Engineering, AG SIV, 1994.

[11]. M. Zaki, Y. Mokhtari, S. Tahar. A Path Dependency Graph for Verilog Program Analysis. Proceedings of the IEEE Northeast Workshop on Circuits and Systems (NEWCAS'03), Montreal, 2003, pp. 109-112.

[12]. R. Floyd. Assigning meanings to programs. Proceedings of Symposia in Applied Mathematics, vol. 19, 1967, pp. 19-32.

[13]. N. Blanc, D. Kroening. Race Analysis for SystemC using Model Checking. ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 15 issue 3, 2010, pp. 356 – 363. doi: 10.1109/ICCAD.2008.4681598

[14]. IEEE Standard SystemC Language Reference Manual, 1666-2005, IEEE, 2003, 428 p.

[15]. E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Counterexample-Guided Abstraction Refinement. Computer Aided Verification, Lecture Notes in Computer Science vol. 1855, 2000, pp. 154-169.

[16]. Cadence SMV Symbolic Model Checker. http://www.kenmcmil.com/smv.html. Access date: 02.03.2015.

[17]. SCOOT A Tool for the Static Analysis of SystemC. http://www.cprover.org/scoot/. Access date: 04.03.2015.

[18]. L. Liu, S. Vasudevan. Scaling Input Stimulus Generation through Hybrid Static and Dynamic Analysis of RTL. ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 20 issue 1, 2014. doi:10.1145/2676549

[19]. P. Godefroid. Compositional dynamic test generation. Proceedings of the 34th Annual ACM Symposium on Principles of Programming Languages (SIGPLAN-SIGACT), 2007, pp. 47-54. doi: 10.1145/1190216.1190226
[20]. STAR. http://users.crhc.illinois.edu/liu187/. Access date: 04.03.2015.
[21]. S. Hertz, D. Sheridan, S. Vasudevan. Mining Hardware Assertions With Guidance From
[22]. Static Analysis. Computer-Aided Design of Integrated Circuits and Systems, Transactions on IEEE, vol. 32, issue 6, 2013, pp. 952-965. doi: 10.1109/TCAD.2013.2241176
[23]. E. Clarke, O. Grumberg, D. Peled. Model checking. The MIT Press, 1999, 314 p.
[24]. GoldMine. http://goldmine.csl.illinois.edu/. Access date: 04.03.2015.
[25]. Cadence Incisive. http://www.cadence.com/products/fv/iv_kit/pages/default.aspx. Access date: 04.03.2015.
[26]. M. Weiser. Program slicing. IEEE Transactions on Software Engineering, vol. 10, issue 4, 1984, pp. 352–357.
[27]. E.M. Clarke, M. Fujita, S.P Rajan, T. Reps, S. Shankar. Program Slicing of Hardware Description Languages. Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods, pp. 298-302.
[28]. Codesurfer. http://www.grammatech.com/research/technologies/codesurfer. Access date: 02.03.2015
[29]. T. Li, Y. Guo, S.-K. Li. Automatic Circuit Extractor for HDL Description Using Program Slicing. Journal of Computer Science and Technology vol. 19, issue 5, pp. 718-728. doi: 10.1007/BF02945599
[30]. C. Dawson, S.K. Pattanam, D. Roberts. The Verilog Procedural Interface for the Verilog Hardware Description Language. Proceedings of Verilog HDL Conference, 1996, pp. 17-23. doi: 10.1109/IVC.1996.496013
[31]. S. Vasudevan, E. A. Emerson, J. A. Abraham. Efficient Model Checking of Hardware Using Conditioned Slicing. Proceedings of the Fouth International Workshop on Automated Verification of Critical Systems (AVoCS), Vo1. 28, issue 6, 2005, pp. 279–294. doi: 10.1016/j.entcs.2005.04.017
[32]. M. Huth, M. Ryan. Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge University Press, New York, 2004, 440 p.
[33]. A. Tepurov, V. Tihhomirov, M. Jenihhin, J. Raik, G. Bartsch, J.H. Meza Escobar, H. Wuttke. Localization of Bugs in Processor Designs Using ZamiaCAD Framework. Proceedings of 13th International Workshop on Microprocessor Test and Verification (MTV), 2012, pp. 41-47.
[34]. ZamiaCAD. http://zamiacad.sourceforge.net/web/. Дата обращения: 06.03.2015.
[35]. J. Hopcroft, R. Motwani, J. Ullman. Introduction to Automata Theory: Languages, and Computation. Pearson/Addison Wesley, 2007, 535 p.
[36]. M. Yuang. Survey of protocol verification techniques based on finite state machine models. Proceedings of the Computer Networking Symposium, 1988, pp. 164-172.
[37]. R. Obermaisser, C. El-Salloum, B. Huber, H. Kopetz. Modeling and Verification of Distributed Real-Time Systems using Periodic Finite State Machines. Journal of Computer Systems Science & Engineering, vol. 22, No. 6, 2007.
[38]. A. Kamkin. Metod formalnoy spetsifikatsii apparaturyi s konveyernoy organizatsiey i ego prilozhenie k zadacham funktsionalnogo testirovaniya [A Method of Pipelined Hardware Specification and its Application to Functional Verification]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 16, 2009, pp. 107-128 (in Russian).

[39]. T.-H. Wang, T. Edsall. Practical FSM Analysis for Verilog. Proceedings of Verilog HDL Conference and VHDL International Users Forum, 1998, pp. 52-58. doi:10.1109/IVC.1998.660680
[40]. A. Gill. Introduction to the Theory of Finite-state Machines. McGraw-Hill, 1962, 207 p.
[41]. J.-C. Giomi. Method of Extracting Implicit Sequential Behavior from Hardware Description Languages. 5.774.370, USA, 531.996, 18.09.1995, 30.06.1998, pp. 1-44.
[42]. C.-N. Liu, J.-Y. Jou. A FSM Extractor for HDL Description at RTL Level. Proceedings of the 2nd International Symposium on Quality Electronic Design, 2001, p. 372.
[43]. M. E. J. Gilford, G. N. Walker, J. L. Tredinnick, M. W. P. Dane, M. J. Reynolds. Recognition of a State Machine in High-Level Integrated Circuit Description Language Code. 7.152.214 B2, USA, 10/736.967, 15.12.2003, 19.12.2006, pp. 1-32.
[44]. W. Song, J. Garside. Automatic Controller Detection for Large Scale RTL Designs. Euromicro Conference on Digital System Design (DSD), Los Alamitos, CA. 2013, pp. 844-851. doi: 10.1109/DSD.2013.94
[45]. G. D. Micheli. Synthesis and Optimization of Digital Circuits. McGraw-Hill Science/Engineering/Math, 1994, 576 p.
[46]. Asynchronous Verilog Synthesizer. http://wsong83.github.io/index.html. Access date: 05.03.2015.
[47]. A. Höller, C. Preschern, C. Steger, C. Kreiner, A. Krieg, H. Bock, J. Haid. Automatized High-Level Evaluation of Security Properties for RTL Hardware Designs. Proceedings of the Workshop on Embedded Systems Security, No. 6, 2013, pp. 1-8. Doi:10.1145/2527317.2527323
[48]. NuSMV. http://nusmv.fbk.eu/. Access date: 05.03.2015.
[49]. K.-T. Cheng, A.S. Krishnakumar. Automatic Generation of Functional Vectors Using the Extended Finite State Machine Model. ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 1 issue 1, 1996, pp. 57-79. doi: 10.1145/225871.225880
[50]. J-Y. Jou, S. Rothweiler, R. Ernst, S. Sutarwala, A. Prabhu. BESTMAP: Behavioral Synthesis from C. In International Workshop on Logic Synthesis (Research Triangle Park, NC, May), 1998.
[51]. G. D. Guglielmo, L. D. Guglielmo, F. Fummi, G. Pravadelli. Efficient Generation of Stimuli for Functional Verification by Backjumping Across Extended FSMs. Journal of Electronic Testing, vol. 27, issue 2, 2011, pp. 137-162. Doi: 10.1007/s10836-011-5209-8
[52]. G. Di Guglielmo, F. Fummi, G. Pravadelli, S. Soffia, M. Roveri. Semi-formal functional verification by EFSM traversing via NuSMV. IEEE International High Level Design Validation and Test Workshop (HLDVT), 2010, pp. 58-65. doi: 10.1109/HLDVT.2010.5496660
[53]. D. Kim, M. Ciesielski, K. Shim, S. Yang. Temporal Parallel Simulation: A Fast Gate-level HDL Simulation Using Higher Level Models. Proceedings of the Conference on Design, Automation and Test in Europe (DATE), 2011. doi: 10.1109/DATE.2011.5763251
[54]. N. Bombieri, F. Fummi, G. Pravadelli. Automatic Abstraction of RTL IPs into Equivalent TLM Descriptions. IEEE Transactions on Computers, vol. 60, issue 12, 2011, pp. 1730-1743. doi: 10.1109/TC.2010.187
[55]. OSCI TLM-2.0. http://www.accellera.org/resources/videos/tlm20andsubset/. Access date: 22.02.2010.
[56]. HIFSuite. https://www.edalab.it/. Access date: 05.05.2015.

[57]. A. Kamkin, S. Smolov. The method of EFSM extraction from HDL: application to functional verification. Proceedings of the conference on Problems of Perspective Micro- and Nanoelectronic Systems Development, Part II, 2014, pp. 113-118.

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

[59]. Retrascope. http://forge.ispras.ru/projects/retrascope/. Дата обращения: 05.03.2015.

[60]. D. Moundanos, J.A. Abraham, Y.V. Hoskote. Abstraction Techniques for Validation Coverage Analysis and Test Generation. IEEE Transactions on Computers, IEEE, 1998, vol. 47, issue 1, pp. 2-14. doi:10.1109/12.656068

[61]. T.M. Niermann, J.H. Patel. HITEC: A Test Generation Package for Sequential Circuits. Proceedings of International European Design Automation Conference (EDAC), 1996, pp. 875-884. doi:10.1109/EDAC.1991.206393

[62]. C.-N. Jimmy Liu, J.-Y. Jou. An Efficient Functional Coverage Test for HDL Descriptions at RTL. Proceedings of International Conference on Computer Design (ICCD), Austin, TX, 1999, pp. 325-327. doi:10.1109/ICCD.1999.808561

Сервисные средства интернет для решения бизнес-задач[1]

Е. М. Лаврищева <lavr@ispras.ru>
Л. Е. Карпов <mak@ispras.ru>
А. Н. Томилин <tom11@bk.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. Описываются различные виды сервисов и служб, используемых в современных программных и распределенных системах. Дается характеристика широко распространенных и внедренных в практику систем со спектром системных и функциональных сервисов. Рассмотрены модели сервисной, сервисно-компонентной архитектур и системы сервисной поддержки WCF для представления прикладных систем из готовых сервисов с целью решения бизнес-задач. Приведен пример решения бизнес задачи обработки данных в калькуляторе, реализованном с помощью сервисной службы WCF комплекса ИТК.

Ключевые слова: распределенная система; глобальная сеть; модели сервисных архитектур, сетевой сервис, сетевая служба; протоколы взаимодействия; языки описания сервисов, бизнес задачи.

DOI: 10.15514/ISPRAS-2015-27(1)-7

Для цитирования: Лаврищева Е.М., Карпов Л.Е., Томилин А.Н. Сервисные средства интернет для решения бизнес-задач. Труды ИСП РАН, том 27, вып. 1, 2015 г., стр. 125-150. DOI: 10.15514/ISPRAS-2015-27(1)-7.

1. Введение

Системная поддержка программирования, начавшись со слабоструктурированных наборов полезных последовательностей команд, записанных в долговременную память (например, на магнитный барабан или магнитную ленту), со временем прошла интенсивный путь через развитие библиотек программ, классов и компонентов к возникновению понятия "служба" (часто используется как термин "сервис"). Это понятие стало настолько популярным в среде программистов и пользователей их программ, что уже давно появилось выражение: "Все есть служба". Однако не всякая

[1] Работа поддержана грантами Российского фонда фундаментальных исследований № 14-07-00606 и № 15-07-02355.

системная поддержка, тем более поддержка распределенных программ, работающих в глобальной информационной сети, например, в сети Интернет, представляет собой службу в строгом значении этого термина. К настоящему времени промышленные стандарты требуют, чтобы правильно организованные службы содержали информацию о реализованной системной функции и форме ее применения. При этом службы работают автоматически – программы вызывают другие программы (службы).

Постепенно сформировались службы следующих видов:

- общие системные сервисы, имевшиеся сейчас во всех системных средах для поддержки процессов и функций этих сред (DCE, COM, DCOM, CORBA, MQ Series, J2EE, .NET и множество других), обработки программ и данных (например, службы именования, каталогов, коллекций, безопасности, событий и каналов);
- объектные сервисы, которые управляют объектами, классами и услугами по формированию и обработке объектно-ориентированных систем (например, службы жизненного цикла объектов);
- сетевые сервисы стандартных моделей SOA (Service-oriented Architecture), SCA (Service-Component Architecture), как инструменты представления и обработки ресурсов Интернет, которые реализуют деловые, финансовые, экономические и другие услуги для решения соответствующих задач в среде Интернет (аналогично в стандарте CORBA введен механизм Common Facility, выполняющий функции управления данными, системами, интерфейсами и др.);
- готовые программные ресурсы (reuses, assets, artifacts, resources, компоненты повторного использования), которые предоставляют программные сервисные услуги в некоторой прикладной области.

Системные сервисы необходимы для организации построения и функционирования стандартных функций систем, а также для управления конфигурацией прикладных программ. Некоторые из сервисов стали обязательной частью системной поддержки, другие используются для специальных областей (например, медицина), а некоторые предоставляют необходимые услуги работы с данными, используемыми в языках программирования. В состав общего множества сервисов обязательно или часто могут входить:

1. Служба именования (naming) готовых компонентов для обеспечения их поиска в распределенной среде пространства имен.
2. Служба связи (binding), предназначенная для определения соответствия “имя-объект” применительно к найденным компонентам. По-видимому, эти две первые из перечисленных служб

– единственные, которые с уверенностью можно отнести к обязательному для каждой системы набору служб.

3. Служба транзакций (transaction), которая обеспечивает организацию и управление функционированием совокупности объектов и данных.
4. Служба обмена сообщениями (messaging), необходимая для организации общения между компонентами, как составной элемент в модели асинхронных транзакций. Транзакции могут выполняться как с использованием синхронного способа взаимодействия, так и в асинхронном виде. Однако асинхронное взаимодействие имеет большие преимущества почти во всем, за исключением простоты.

Каждая служба определяется именем, по которому осуществляется поиск в распределенной среде пространства имен через транзакции, устанавливающие соответствие “имя-объект” для организации и управления отдельными сервисными ресурсами глобальной сети, а также с помощью сообщений для визуального общения с требуемыми представителями отдельных ресурсов.
Перечисленные четыре вида сервисов и их служб используются при реализации моделей программной системы (ПС). В настоящее время в Интернете имеется более 100 миллионов сервисных ресурсов. Для их использования при создании программы решении конкретной задачи требуется проводить поиск подходящего сервисного ресурса, его апробацию и встраивание в прикладную программу решения задачи, либо использовать его в динамическом режиме (см. [1-7]).

2. Подходы к представлению и реализации сервисов

Первый вариант достаточно полного набора сервисов, основанный на объектно-ориентированном подходе, был сделан в проекте ***Common Object Request Broker Architecture*** (CORBA, см. [3, 5, 8-9]), который первоначально предполагал, что в состав системы поддержки должны входить компоненты (1994):

- *брокер объектных запросов* (Object Request Broker — ORB), обеспечивающий взаимодействие распределенных объектов;
- *объектная модель* (OM), объекты которой взаимодействуют между собой через интерфейсные объекты –посредники stub для клиента и skeleton для сервера, описываемые в специальном языке IDL (Interface Definition language) и обрабатываемые брокером ORB;
- *общие объектные сервисы* (Common Object Services), предоставляющие услуги всем объектам по управлению данными, изменениями программ и подпроцессов, а также по обработке транзакций и т. п.;
- *общие средства обслуживания* (Common Facilities) или общие

средства, предоставляющие ряд общих прикладных функций для любых приложений (средства печати, управление БД, электронная почта и др.);

- *объектные приложения* (Application Objects), к которым относятся приложения и их компоненты, реализующие задачи пользователя.

Объекты специфицируются средствами языков программирования (Smalltalk, Cobol, Ada-95, Lisp, PL/1, C++, Python, Java, IDLScript и др., см. [5, 8-9]) и могут быть реализованы на разных платформах и средах. Интерфейсы программ-посредников, которые тождественны интерфейсам клиентских и серверных компонентов, описываются в языке IDL (см. [8-9]). Заместитель клиента (stub) выполняет сервисные функции, связанные с преобразованием типов данных клиентских компонентов к стандартным системным типам, а заместитель сервера (skeleton) преобразует стандартное представление данных в типы данных сервера. При этом по сравнению с более ранними системами, ориентированными на работу в рамках процедурной парадигмы, в архитектуре CORBA сделан существенный шаг в сторону системной симметрии: обратные вызовы позволяют серверным компонентам выступать в роли клиентов, а клиентам, обрабатывающим обратные вызовы, обслуживать их, как если бы они были серверными компонентами.
Общие объектные службы предоставляют набор операций для работы с разными категориями объектных приложений. Наиболее известными являются следующие службы:

- именование и поиск объектов по именам, либо по свойствам и атрибутам;
- поддержка жизненного цикла объектов (создание объектов, их копирование и уничтожение);
- поддержка параллелизма обращения к объектам;
- поддержка очередей запросов к объектам;
- поддержка иерархия транзакций;
- обеспечение защиты объектов от несанкционированного доступа, поддержка авторизации и аутентификации клиентов и др.

Система CORBA предоставила типовые методы реализации и диспетчеризации объектов. Они широко используются многими современными операционными системами и средами, а также стали промышленными стандартами и применяются в системах Cloud Computing, Grid и др. Служба транзакций и безопасности CORBA используется в транзакционном мониторе CICS компании IBM, системах управления базами данных Oracle, Sybase, Mybase, сервере приложений J2EE компании Sun Microsystems и др.

2.1 Представление сетевых служб современными средствами

С широким распространением глобальной сети Интернет, интерес к сервисам постоянно растет. Это прежде всего связано с проявившимся пониманием того факта, что без стандартизации адекватное функционирование распределенных приложений, опирающихся на распределенную системную поддержку, просто невозможно. Понятие "сетевой службы" (см. [5]). Не всякая программа, доступная в Интернете, имеет право называться сетевым сервисом. Чтобы получить это право, сервис должен удовлетворять целой серии стандартов, описывающих разные аспекты его состояния и поведения. С общей точки зрения модель сетевой службы имеет типизированную структуру и интерфейс. Модель "клиента" и "поставщика сервиса" сближены, то есть тенденция, наметившаяся в процессе развития более традиционных сервисов, еще более усилена. Для представления разных аспектов описания сервисов в качестве синтаксической основы используется язык XML (см. [10]). К этим аспектам относятся описания структуры и семантики данных, механизмов взаимодействия сетевых служб и функциональных услуг поиска необходимых сервисов (см. [11, 12]).

В качестве базового протокола доступа к объектам в информационной сети, определяющего форматы и методы упаковки информации, используется простой протокол доступа к объектам, который называется SOAP (Simple Object Access Protocol, см. [13, 14]). Этот протокол не детализирует свойства конкретного обмена информацией и задаёт шаблон обобщённого сообщения.

Интерфейс сетевых служб, как правило, описывается на языке IDL, а различные виды взаимодействия описываются с помощью схем XML. При описании сервиса необходимо указывать его адрес (Uniform Resource Identifier – URI) и транспортный протокол (например, HTTP). Средством описания функциональности сервиса является язык WSDL (Web-service description language, см. [15]). Для представления данных, в особенности метаданных, используется модель RDF (см. [16]). Для описания процессов представления и обработки запросов на сервисы в графическом виде предложены языки:

- WSCI (Web Services Choreography Interface, см. [17]),
- WSCL (Web Services Conversation Language, см. [18]),
- BPMN (Business process and model and notation, см. [19]),
- BPEL (Business Process Execution Language for Web Services, см. [20-21]) и другие.

В качестве адреса объектов в сети используются универсальные идентификаторы ресурсов URI и интерфейс, задаваемый для организации связи с другими сервисами с помощью XML-документов.

Таким образом, к основным средствам описания, разработки и взаимодействия систем в глобальных сетях относятся:

- язык XML, предназначенный для разметки и описания структуры и способов взаимодействия компонентов распределённой архитектуры;
- протокол SOAP для определения форматов запросов к сетевым службам;
- язык WSDL, основанный на XML и предназначенный для описания интерфейсов сервисов, типов данных, сообщений, операций, типов портов, моделей взаимодействия и протоколов связи сервисов между собой;
- служба регистрации UDDI (Universal Description, Discovery and Integration, см. [22-25]) для универсального описания, выявления и интеграции служб, обеспечения их хранения, поиска и сопровождения, упорядочения деловой служебной информации в специальном реестре с указателями на конкретные интерфейсы и адреса сервисов;
- язык описания протоколов прикладного уровня ("бизнес-протоколов") BPEL для описания наборов правил, регулирующих взаимодействия сетевых служб ("разговоры");
- графическая нотация языка BPMN для нотации прикладных процессов; описания структуры системы на языке моделирования UML (Unified Modeling Language, см. [26-27]); детализации объектной структуры и последующей разработки текстов прикладных программ; представления процесса выполнения системы в BPEL, включая бизнес-процессы на сервере приложений;
- модель SOA (Service-oriented Architecture) для описания сервисно-ориентированной архитектуры программной системы;
- модель SCA (Service-Component Architecture) для создания сложных систем на основе сервисов и компонентов.

Модели SOA и SCA реализованы в .NET (Microsoft), WebSphere (IBM), Weblogic (BEA, Oracle), Java Enterprise Edition (Sun), SAP, Soap WS и др.

2.2 Модель сервисов SOA (Service-oriented Architecture)

Идея, предлагаемая архитектурой SOA, заключается в группировании на серверной стороне некоторого количество согласованно реализованных сервисов и их служб. Группы должны задавать открытый интерфейс, содержащий описание типов входных/выходных параметров каждого сервиса. Эта информация задается с помощью языка WSDL и описания так называемых портов обмена метаданными (Metadata Exchange Endpoints). Для

работы с языком WSDL созданы компиляторы, позволяющие на основе текстов, написанных на этом языке, серверные и клиентские заместители (прокси-классы). Они учитывают особенности конкретных программных платформ, в том числе языки программирования на этих платформах, которые описывают реализуемые операции.

Данный механизм хорошо отработан и для систем, построенных на традиционных принципах. Он обеспечивает согласованность, языковую независимость и интероперабельность серверной и клиентской частей распределённой системы. От разработчика требуется написать контрактный сервис средствами WCF (Window Communication Foundation) и использовать его в реализации клиентов, написанных в одном из языков – Java, Python, Ruby или др. Клиенты в свою очередь имеют на своей стороне собственные заместители сервера.

Таким образом, на новом витке развития информационных технологий можно наблюдать воспроизведение достижений, полученных еще разработчиками принципов, положенных в основу метода RPC удаленного вызова процедуры: клиент обращается к локальному заместителю сервера (на этот выполненному в виде прокси-класса), затем сетевая служба вызывает метод, реализованный на удаленном сервере. При этом непосредственно к сервису обращается локальный заместитель клиента (прокси-класс).

Модель SOA – это набор принципов и средств создания системного программного обеспечения и прикладных программных систем (ПС) из совместимых и унифицированных сервисов. К принципам SOA относятся:

1. Архитектура, которая не привязана к определённой технологии.

2. Независимость организации системы от используемой вычислительной платформы (платформ) и от применяемых языков программирования.

3. Использование сервисов независимо от конкретных приложений выполняется единообразным интерфейсом и является инструментом для построения систем из сервисов.

Унификация проявляется как типизация функциональности сервиса и его характеристик, а также языков описания сервисов и их взаимодействия. Объект SOA – сервис, который обладает специфицированной функциональностью (Function) и качеством (Quality service). Такие сервисы определены стандартами комитета W3C и имеют следующие уровни коммуникации:

- *транспортный уровень* (transport layer) для обмена данными;
- *уровень коммуникационного сервиса* (service communication layer) для определения высокоуровневых протоколов;
- *уровень описания сервиса* (service description layer) и связанных с ним

интерфейсов;

- *уровень бизнес-процессов* (business process layer) для реализации бизнес-процессов и потоков работ средствами сетевых служб;
- *уровень регистрации сервисов* (service registry layer) в реестрах сетевых служб для их публикации, поиска и вызова с использованием их WSDL интерфейсов.

Технология обеспечения качества сетевых служб имеет следующие уровни:

- *политика* (policy layer) для описания правил и условий применения сервисов;
- *безопасность* (security layer) для описания безопасности служб (авторизация, аутентификация и распределенный доступ);
- *транзакции* (transaction layer) для установки параметров обращения к сервисам и обеспечения надежности их функционирования;
- *управление* (management layer) службами.

Технологический фундамент сетевых служб составляют (рис.1):

- *набор языков* – XML, WSDL, BPREL, BPMN и др. для реализации базовых свойств сетевых сервисов, обеспечения их взаимодействия между собой в соответствующих средах (SOA, SCA и др.);
- *поставщик услуги*, который осуществляет ее реализацию в виде сетевой службы, прием и выполнение запросов пользователей, а также публикацию сведений о сервисе в соответствующем реестре;
- *реестр* (каталог) служб UDDI содержит в себе библиотеку сервисов для пользователей, а также средства их поиска и вызова с помощью запросов, которые поступают от поставщиков сервисов на получение сервисов; реестр служб оформляется как стандартная сетевая служба, адрес и другие параметры доступа к которой заранее известны поставщикам служб и их пользователям;
- пользователь или *потребитель сервиса* (приложение, программный модуль и др.), который осуществляет поиск и вызов необходимого сервиса из реестра описания сервисов, а также использует сервис, предоставленный провайдером в соответствии с заданным интерфейсом.

Связь между поставщиком и потребителем (рис. 1) осуществляется через HTTP и XML-сообщения сетевой среды, которая использует интерфейсы сетевых служб. Посредником между этими сервисами и приложениям является *провайдер*, который обеспечивает взаимодействие между поставщиками и потребителями с помощью средств описания и передачи сервисов WSDL, SOAP, XML.

Для получения сервиса в архитектуре SOA выполняются следующие операции:

1) публикация сервиса WSDL с целью обеспечения доступности (через вызов) пользователю сервиса и его интерфейса;
2) поиск сервиса в реестре с помощью протокола SOAP и заданных критериев;
3) связь с реестром UDDI через описание пользователем необходимого сервиса, который может предоставляться в таких моделях как DCOM, CORBA, DBMS, .NET и т. п.

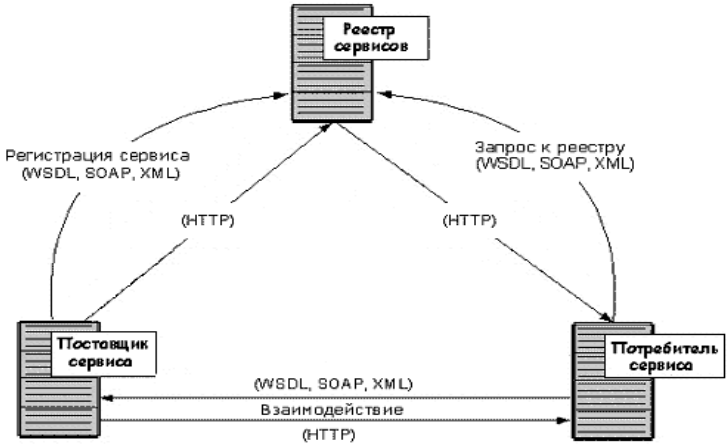


Рис.1. Поставщики и потребители сервисов в сети.

При этом предусматривается, что в реестре архитектуры SOA содержится описание сервиса с форматом запросов пользователя к провайдеру, содержащему в себе перечень описаний сервисов, которые могут быть вызваны соответственно с опубликованным интерфейсом сервиса.

К базовым функциям управления компонентами и службами в операционной среде относятся:

- поиск необходимых ресурсов (компонентов, reuses, assets, artifacts, служб и др.);
- доступ к названным ресурсам;
- организация обмена информации между компонентами, ресурсами и службами;
- динамическое управление функционированием заданной совокупности ресурсов в ПС.

Модель сервисов ПС базируется на унификации и совместимости, что позволяет рассматривать ПС как набор сервисов, функциональности и взаимодействия. Унификация достигается путем:

- типизации функциональности сервиса и их других характеристик;
- применения унифицированных языков для описания сервиса и их взаимодействия;
- использования стандартных базовых технологий.

Отдельный класс средств унификации составляет онтология – модели и словари, которые обеспечивают согласование терминов и понятий языка описания сервисов на уровне семантики. В качестве стандартных базовых технологий при реализации сервисов используются: модель клиент-сервер, унифицированные коммуникационные протоколы, компонентные модели и т. д.

Модель службы сервисов обеспечивает:

- динамическое расширение функциональности ПС за счет поиска и привлечения в сферу обработки новых сервисов;
- повышение масштаба и улучшение возможностей коммуникации между отдельными элементами системы за счет стандартного механизма подключения сервисов через их интерфейсы;
- повышение языкового уровня коммуникации системы с конечными пользователями.

Принципы разработки ПС из готовых компонентов повторного использования (КПИ), сетевых (и не только сетевых) служб, интерфейсов, данных, артефактов и т. п.:

- композиционность систем и ПС из компонентов, интерфейсов и сервисов с их свойствами и характеристикам, а также механизмами их композиции (агрегации) и правилами взаимодействия в интегрированных средах;
- компонентная инженерия (CBSE), как деятельность по созданию ПС из готовых "деталей" КПИ, базированная на системе классификации и каталогизации КПИ, средствах их унификации, стандартизации и интеграции их в ПС с помощью стандартного жизненного цикла (ЖЦ), начиная от процессов инженерии требований, разработки, эксплуатации и уничтожения ПС;
- интероперабельность КПИ и ПС, которая базируется на интерфейсах и стандартных правилах взаимодействия компонентов и сервисов между собой в процессе их интеграции для работы в разных гетерогенных средах;
- вариантность как способность ресурсов КПИ и сервисов служб к изменениям (удаление незавершенных функций или добавление

новых функциональных КПИ в конфигурационную структуру ПС и т. п.).

Сервис SOA – это функция, являющаяся четко определенной, самодостаточной и не зависящей от контекста или состояния других служб. Сервис определяется как единица работы, выполняемая от имени некоторого информационного субъекта-пользователя или программы. Клиент может избирать сервисы от разных поставщиков. Каждая служба может развиваться и быстро доставляться клиенту отдельно от других.

Главная идея сервисного подхода SOA состоит в том, чтобы создавать небольшие независимые компоненты (как сервисы) и собирать их в большой распределенный по глобальной сети комплекс (архитектуру) под задачи конкретного клиента. Функциональные сервисы, как обычные программные модули, могут быть распределены по вычислительным системам и обладать способностью к взаимодействию посредством локальных и/или глобальных сетей. Интерфейс таких модулей не зависит от технологии или платформы, в рамках которой они реализованы.

Бизнес-процесс определяется как набор взаимосвязанных задач, относящихся к деловой деятельности, имеющий начальные и конечные точки для повторения некоторой задачи.

2.3 Модель SCA (Service-Component Architecture) в IBM Sphere

Сервисно-компонентная архитектура (SCA, см. [11-12, 28]) предназначена для работы с компонентами, подчиняющимися самым разным спецификациям, разработанным различными компаниями: с компонентами EJB сервера приложений J2EE компании Sun Microsystems, с сетевыми сервисами, объектами планирования, компонентами доступа к базам данных, к информационной системе предприятия (Enterprise Information System, EIS) и др.

Архитектура SCA обеспечивает доступ к сервисным компонентам и определяет зависимости между ними через аппарат ссылок. Компоненты SCA системы IBM WebSphere Integration Developer (инструментарий для разработки приложений на платформе Eclipse) могут быть упакованы в модуль для выполнения сервисного модуля с WebSphere Process Server – эквивалентного EAR-файлу J2EE и некоторым другим (рис.2). Подмодули J2EE и артефакты упаковываются с модулем SCA. Это позволяет запустить сервис через модель SCA и передавать данные при интеграции.

Механизмы, которые используются для вызова внешнего сервиса, называются импортом и экспортом. Они связаны с другими технологиями, такими как JMS, Enterprise JavaBeans или технологиями сетевых служб. SCA модуль позволяет обратиться к Enterprise JavaBean. Элементы SCA могут компоноваться и обмениваться данными друг с другом, пересылая сервисные объекты данных (Service Data Objects – SDO), подготовленные в необходимом

виде. Этот интерфейс включает определение метода получения и установления свойства данных. В рамках модели SCA сервисы могут собираться в различные образования (хореографии). Они используют архитектуру SOA и/или создают новые сервисы для их комбинирования и конфигурирования.

Ключевые процессы для сложных приложений – разработка КПИ и внедрение SOA-сервисов и SCA. Для банковской сферы в IBM разработан набор компонентов в рамках портала WebSphere Portal (Process Server, WebSphere Service Registry and Repository, WebSphere Enterprise Bus, WebSphere Portlet Factory, WebSphere Application Server), а также соответствующие инструменты разработки новых приложений из сервисных компонентов и служб. Основные особенности предлагаемой технологии – наличие в ней и широкое использование точек изменчивости, вариантов использования и среды выполнения.

Каждое приложение состоит из одного или более компонентов или их объединений, связанных между собой интерфейсами сервисов. Для реализации бизнес-сервисов в конкретное приложение используются определенные бизнес-цели, которые облегчают разработку, применение и повторное использование в них готовых ресурсов.

Характеристики приложения можно согласовывать со специфическими потребностями клиента (то есть проводить конфигурирование приложения), используя такие элементы, как стандарты бизнеса, бизнес-правила, соглашения об уровне бизнес-сервиса и параметры конфигурации. Так, функция Dynamic Profiles WebSphere Portlet Factory обеспечивает высокую динамическую конфигурацию пользовательского интерфейса.

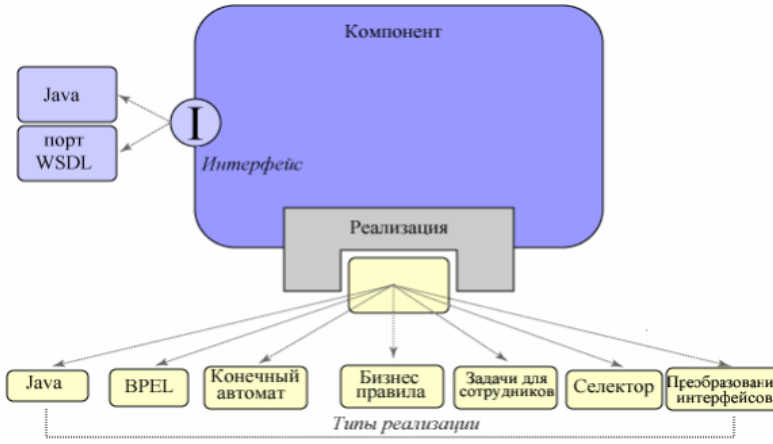


Рис.2. Общая схема сервера SCA IBM.

Сервисы уровня предприятия запускаются на узле сервисов предприятия, который содержит продукты, обеспечивающие сервисы данных, безопасность и другие службы инфраструктур, например, системный реестр служб. Сервер каталогов Tivoli (Tivoli Directory Server) обеспечивает инфраструктуру облегченного протокола доступа к каталогам (Lightweight Directory Access Protocol, LDAP), являющегося основой для управления идентификацией. Реестр WebSphere Service Registry and Repository дает возможность провайдерам регистрироваться, а клиентам – выбирать сервисы.

Модель SCM представляет собой обобщение объектно-компонентной модели семейства программных продуктов (СПП, см. [1]). В ней каждый член является системой удаленных КПИ, которые обмениваются гетерогенными данными и предоставляют сервисы реализации с множеством общих свойств; композицией сетевых сервисов, которые поддерживают некоторый деловой процесс. Данная модель ориентирована на обеспечение адаптивности ПС к переменным условиям использования запросов к функциям и обрабатываемым ими гетерогенных данных. В интересах этой поддержки для программной реализации ПС используются механизмы сервисных объектов данных (SDO) и сервисов доступа к ним (DAS). Они позволяют размежевать код ПС и код доступа к обрабатываемым данным.
Модель SCM для реализации ПС представлена в виде:

$$SCM = \langle N,\ ID,\ MD(N),\ MI(ID),\ SR\rangle,$$

где N – имя ПС,

ID – идентификатор SCA;

$MD(N)$ – подмодель *абстрактных сервисов*, отображающая потребности ПС в функциях/данных на множестве бинарных отношений, которые задают взаимосвязи между абстрактными сервисами;

MI(ID) – *интерфейсная* подмодель на множестве интерфейсов, которые реализуют абстрактные сервисы на множестве бинарных отношений и задают взаимосвязанные интерфейсы;

$SR=\langle R,\ RR\rangle$ – подмодель *сервисных ресурсов* реализации интерфейсов *MI* (*ID*) на множестве R ресурсов и RR бинарных отношений, которые задают взаимосвязи между этими ресурсов. Элементы множества R – сервисы удаленных КПИ, сетевые службы и *композиты* (composites) SCA в виде объектов специального типа, которые соединяют свойства КПИ и сетевые службы.

2.4 Сетевые службы сервера приложений Java Enterprise Edition (Java EE)

Сервер приложений Java Enterprise Edition содержит набор спецификаций на языке Java, которые необходимы при работе с сетевыми программами и средствами. К ним относятся:

- динамическая генерация серверных страниц (Java Server Pages);
- сетевые службы;
- компоненты повторного использования – Enterprise Java Beans;
- служба обмена сообщениями (Java Message Queue) и другие сервисные технологии.

Сетевая служба идентифицируется с помощью универсального ресурсного идентификатора URI, ее ресурсы (свойства и методы) описаны на специальном языке WSDL. Доступ к ресурсам осуществляется через протокол SOAP, который представляет собой XML-запросы, передаваемые посредством интернет-протокола относительно высокого уровня (HTTP, SMTP). Сетевые службы соответствуют объектам объектно-ориентированных языков программирования с некоторыми важными отличиями.

Ключевым понятием сетевой службы является сообщение (message), которое состоит из одной или нескольких переменных. Вместо методов классов в сетевых сервисах используются операции, которые определяются входным и выходным сообщениями. Далее приведен пример запроса, который вызывает операцию сервиса MyService.someMethod.

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:q0 ="http://webservice.demo"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <q0:someMethod>
      <q0:arg0>32.5</q0:arg0>
      <q0:arg1>true</q0:arg1>
    </q0:someMethod>
  </soapenv:Body>
</soapenv:Envelope>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <someMethodResponse xmlns="http://webservice.demo">
      <someMethodReturn>Nothing to see here</someMethodReturn>
    </someMethodResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Для описания общедоступных ресурсов сетевых сервисов в язык WSDL, построенный на синтаксической основе языка разметки XML, введены возможности описания данных следующих типов:

строка (xsd:string),
целые числа (xsd:int, xsd:long, xsd:short, xsd:integer, xsd:decimal),
числа с плавающей запятой (xsd:float, xsd:double),
логический тип (xsd:boolean),
последовательность байтов (xsd:base64Binary, xsd:hexBinary),
дата и время (xsd:time, xsd:date, xsd:g),
объекты (xsd:anySimpleType).

В качестве переменных для сообщений можно использовать последовательности, созданные из фиксированного количества переменных простых типов, причем в начале декларируются типы, которые будут использоваться в службе. Типичный WSDL-файл имеет такую примерную структуру.

```
<wsdl:definitions […]>
  <!-- Декларация типов в сервисе -->
  <wsdl:types>
    <element name="someMethod">
      <complexType>
        <sequence>
          <element name="arg0" type="xsd:double"/>
          <element name="arg1" type="xsd:boolean"/>
        </sequence>
      </complexType>
    </element>
    <element name="someMethodResponse">
      <complexType>
        <sequence>
          <element name="someMethodReturn" type="xsd:
string"/>
        </sequence>
      </complexType>
    </element>
  </wsdl: types>
  <! -- Декларация сообщения -->
    <wsdl: message name="someMethodResponse">
      <wsdl:part element="impl:someMethodResponse"
name="parameters">
      </wsdl:part>
    </wsdl:message>
    <wsdl:message name="someMethodRequest">
      <wsdl:part element="impl:someMethod"
name="parameters">
      </wsdl:part>
    </wsdl:message>
```

```
        <!-- Декларации операций -->
          <wsdl:portType name="MyService">
            <wsdl:operation name="someMethod">
              <!-- Входные сообщения -->
              <wsdl:input message="impl:someMethodRequest"
                          name="someMethodRequest">
              </wsdl:input>
              <!-- Выходное сообщение -->
              <wsdl:output message="impl:someMethodResponse"
                           name="someMethodResponse">
              </wsdl: output>
            </wsdl:operation>
          </wsdl:portType>
        <!-- Декларация связи с SOAP -->
        <wsdl:binding name="MyServiceSoapBinding"
type="impl:MyService">
          <wsdlsoap:binding […]/>
          <wsdl:operation name="someMethod">
            <wsdlsoap:operation soapAction=""/>
            <wsdl:input name="fahrenheitToCelsiusRequest">
              <wsdlsoap:body use="literal"/>
            </wsdl:input>
            <wsdl:output name="fahrenheitToCelsiusResponse">
              <wsdlsoap:body use="literal"/>
            </wsdl:output>
          </wsdl:operation>
        </wsdl:binding>
        <!-- Декларация веб-сервиса -->
        <wsdl:service name="MyServiceService">
          <wsdl:port binding="impl:MyServiceSoapBinding"
name="MyService">
            <wsdlsoap:address location=[…]/>
           </wsdl:port>
        </wsdl:service>
      </wsdl:definitions>
```

Приведенное далее WSDL-описание принадлежит сетевой службе MyService с единственным методом *String someMethod(double arg0, boolean arg1)*. Для вызова метода сгенерированы два типа данных, которые соответствуют входным и исходным аргументам метода. Эти типы применяются в описаниях someMethodRequest и someMethodResponse – входного и выходного сообщения для операции someMethod. Операции декларируются в описании интерфейса службы (декларация wsdl:portType) и описание привязки службы к SOAP (декларация wsdl:binding). Причем во втором случае также оговаривается способ вызова (<wsdlsoap:body use="literal"/>). За счет этого разрешается при вызове операции использовать те же названия параметров, что и в методе класса. В конце WSDL-файла

находится декларация сетевой службы (<wsdl:service>), в которой содержится информация о расположении (параметр location).
На основе WSDL в среде Eclipse имеется возможность описывать клиенты сетевых служб в виде пяти Java-файлов:

- локатор сервиса (service locator), который выполняет нахождение веб-сервиса;
- интерфейс локатора;
- клиентский заместитель для SOAP-связи (SOAP binding stub) , предназначенный для составления и разбора SOAP–сообщения;
- интерфейс сервиса;
- серверный заместитель (прокси-класс), который реализует интерфейс; использующий клиентский заместитель и локатор для доступа к операциям службы.

Для сервиса MyService автоматически создаются классы и интерфейсы - MyServiceServiceLocator, MyServiceService, MyServiceSoapBindingStub, MyService, MyServiceProxy (в порядке их описания). Обращение к операции MyService.someMethod выглядит таким образом:

```
My Service svc = new MyServiceProxy ();
try { System.out.println (svc.someMethod (32.5, true));
}
catch (Remote Exception e) {
      System.err.println("Error occurred while accessing web
service");
      e.printStackTrace();
}
```

2.5 Сервисы IContact WCF MS.NET

Программная среда Windows Communication Foundation (WCF) используется для обмена данными, которые входят в состав среды .NET Framework [29-30]. Она является логическим развитием технологий сетевых служб, .NET Remoting и DCOM.
В основе WCF лежит архитектура SOA. Тем самым предполагается, что на стороне сервера работает некоторое количество сервисов, которые представляют собой группу операций, определенных в некотором интерфейсе, и которые получают абстрактные входные/выходные параметры. Все это описывается на языке WSDL и может быть сделано доступным удаленным клиентским приложениям через порты обмена метаданными (Metadata Exchange Endpoints – mex-endpoints). Это позволяет получить "целевые данные" служб путем подключения к этому интерфейсу, а также получить описания служб и всех их операций через серверные заместители (прокси-классы), специфические для заданного языка или платформы. Клиенты в свою

очередь имеют на своей стороне клиентские заместители, которые содержат ссылки на соответствующие операции на серверной стороне. Таким образом, вызов локальных методов прокси-класса приводит к вызову методов соответствующей службы.

В составе программной среды WCF MS.NET имеется технология сетевых служб, а также фабрика сервисов. Основу технологии составляют схемы, “рецепты”, методы и средства построения фабрик разного назначения. Фабрика сервисных программ в WSF включает набор полезных ресурсов, блоков кода, документации, образцы приложений, автоматизированные инструменты и пакеты Visual Studio Industry Partners (VSIP) для создания на их основе программных комплексов для нестандартных аппаратных устройств. Компания Microsoft создала набор рекомендации, схем и методов, а также стандарты их выполнения разработчиками готовой продукции.

Фабрика сервисов предоставляет рекомендации для использования сервисов при проектировании и конструировании некоторого приложения, которые накапливаются в хранилище Global Bank. К ним относятся ASP.NET для использования в WCF. Данные фабрики программ и сервисов базируются на готовом наборе программных элементов и разного рода сервисов, специфических для MS.NET (см. [1, 2]).

В основе функционирования программной среды WCF лежат так называемые конечные точки (endpoint), что составляет связь "Address – Binding – Contract" ("ABC"). Каждая составляющая играет важную роль в определении конечной точки. Компонент "Address" содержит указание места расположения конечной точки (порта). Адрес может быть как абсолютным, так и относительным (относительно базового адреса). Компонент "Binding" задает привязку и, фактически, определяет транспортный протокол, на основе которого будет происходить взаимодействие. В модели WCF определен ряд классов-привязок, например, BasicHttpBinding (простая привязка на основе HTTP), NetTcpBinding (привязка на основе транспорта TCP) и т.д.

Компонент "Contract" задает контракт, на основе которого будет происходить взаимодействие клиента и сервера. Фактически, определение контракта регламентирует операции для оказания серверной услуги. На основе контракта на стороне клиента строится прокси-класс. Интерфейс *Icontract* содержит описание атрибутов и операций передачи данных от одного сервисного объекта клиента (*Service consumer*) к другому (*Service provider*). Их описание задается в языке XML. Передача интерфейсов между ними выполняет протокол, в котором задаются атрибуты и операции интерфейса. Интерфейс *Icontract* задается сообщениями вида:

1. сервис операций, вызываемых клиентом;
2. фундаментальных данных (int, float, string и др.) для передачи их через сервисные службы;

3. об ошибках, которые могут содержаться при передаче контрактов клиенту и обратно;
4. операций прямого взаимодействия объектов между собой.

Эти сообщения задаются в языке XML с помощью протокола SOAP в конверте (рис.3) следующего вида:

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://www.cbsystematics.com">
        <!-Конверт протокола SOAP->
        <env:Header>
                <!- Заголовок протокола SOAP->
        </env:Header>
        <env:Body>
                <!-Тело протокола SOAP->
        </env:Body>
</env:Envelope>
```

В конверте передаются параметры для обмена данными. Через них происходит взаимодействие между пользователем и провайдером. Параметры могут содержать ошибки, которые приводят к возникновению разного рода конфликтов в сетевой среде следующего вида:

1. Несовместимость переданных типов данных в контрактных интерфейсах (например, тип "целое", а параметр описан как "символьный") или некорректное описание некоторых типов данных в компонентах на ЯП.
2. Различие в порядке задания параметров в протоколе передачи информации между сервисными объектами.
3. Различие в архитектуре платформ объектов клиента и сервера или в конфигурационных файлах взаимодействующих систем.

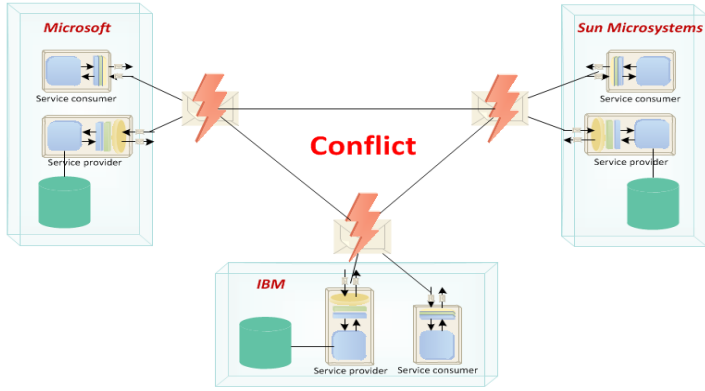


Рис. 3. Схема взаимодействия систем в Интернет и возможность возникновения конфликтов.

Снятие таких конфликтов решается специальными сервисными средствами взаимодействующих систем, расположенных на серверной стороне и отмеченных конечными точками. Клиент в этом случае создает соединение с той конечной точкой, которая требуется именно ему. Пример схемы соединения сервиса и клиентов показан на рис. 4.

Архитектура WCF для удобства построения распределенных приложений следует принципу "слоенного" пирога, когда каждый слой отвечает за свой конкретный уровень абстракции и не знает нижележащие уровни. Инфраструктура WCF состоит из двух главных уровней: уровня серверной модели (Service Model Layer) и канального уровня (Channel Layer). Первый уровень относится и к самому серверу, и к клиенту, он отвечает за преобразование метода и его параметров в сообщение для передачи более низкому канальному уровню. Канальный уровень инкапсулирует в себе множество каналов передачи данных, использующих транспортные протоколы TCP, HTTP, Named Pipes и т. д. Каждый из этих уровней содержит подуровни, в которые может войти любой пользователь.

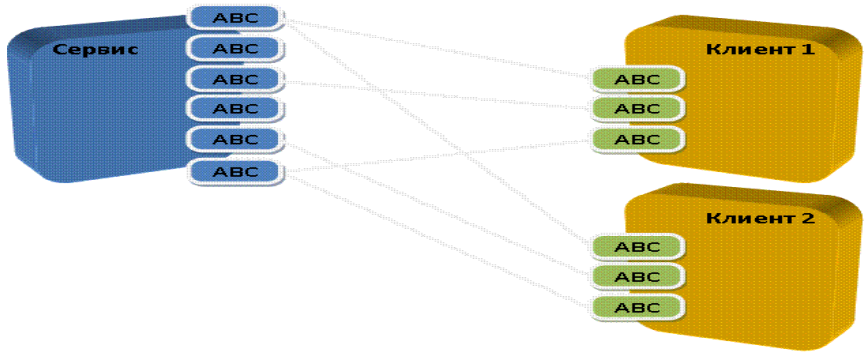


Рис.4. Пример соединения сервиса и клиентов.

Контракты в WCF представляют собой описания сообщений, переданных конечным службам для получения от них ответа. Конечная точка специфицирует операции, которые она может выполнять, и формат ожидаемых данных. Совокупность этих спецификаций и есть контракт.
В WCF содержится три вида контрактов:

- функциональные операции, реализованные сервером. Внутри контракта сервера имеются контракты на операции, которые описывают сервисы, реализующие требуемые функции;
- формат данных, которыми сервисы будут обмениваться, относится как к запросу на сервис, так и к октету сервиса. Если используются примитивные типы данных – целые (int), строки (string) и им подобные, то контракт не требуется, потому что

среда .NET изначально позволяет работать с этими типами данных;

- контроль заголовка SOAP

Чтобы контракты были интероперабельными для широкого диапазона систем, они должны описываться на языке WSDL. Однако на практике контракты описываются на языках WSDL и XSD, а программа обычно работает с типами данных библиотеки CLR, для отображения одной системы типов в требуемую другую. В WCF эта задача решается в три этапа. Сначала при написании кода сервиса поставляется класс, определенный в WCF атрибутами [ServiceContract], [OperationContract], [FaultContract], [MessageContract] и [DataContract]. После создания текста клиентской программы следует запрос к сервису контракта. Утилита svcutil.exe вызывает инфраструктуру конечной точки сервера, чтобы ответить передачей данных, необходимых для генерации WSDL документа с атрибутами. На этапе выполнения метода и определенного в интерфейсе сервиса, WCF сериализует типы CLR и вызов в формате XML, а затем посылает сообщение в сеть для привязки к схеме через WSDL. В этом процессе участвуют четыре конструкции: две со стороны .NET и две со стороны XML. Со стороны .NET имеется тип CLR, который определяет структуры данных и функциональные возможности. После того, как будет создан объект этого типа со стороны XML (XML Instance), сообщение начинает существовать.

Пример создания сервиса WCF в MS. Для наглядности дается пример работы сервиса калькулятора с функциями «+», «-», «*» и «/», выполненного студентом МФТИ (см. [1] и раздел «взаимодействие» на сайте http://sestudy.edu-ua.net).

Использование служб WCF

1. В меню Файл надо выбрать пункт «Создать», а затем команду «Проект».
2. В диалоговом окне «Новый проект» развернуть узел Visual Basic или Visual C# и WCF, Библиотека службы WCF. Нажать кнопку «ОК» для открытия проекта.
3. В обозревателе решений дважды щелкнуть файл IService1.vb или IService1.cs и найти следующий интерфейс IService1.
4. Добавить следующие коды:

```
[OperationContract()]
int Add(int, int b);
[OperationContract()]
int Multiply(int, int b);
[OperationContract()]
int Divide(int, int b);
[OperationContract()]
int Substract(int, int b);
```

5. В обозревателе решений дважды щелкнуть файл Service1.cs и найти Service1 и добавить коды:

```
public int Add(int, int b) {
  return + b;
}
public int Multiply(int, int b) {
  return * b;
}
public int Divide(int, int b) {
  return / b;
}
public int Substract(int, int b) {
  return - b;
}
```

Доступ к службе WCF

1. В меню Файл последовательно выбрать пункты «Добавить» и «Новый проект».
2. В диалоговом окне «Новый проект» развернуть узел Visual C#, выбрать пункт Windows и элемент «Добавление» Windows Forms. Нажать кнопку «ОК» для открытия проекта.
3. Щелкнуть правой кнопкой мыши WindowsApplication1 и выбрать ссылку «Добавить». Появится диалоговое окно и надо добавить ссылку на службу.
4. В диалоговом окне «Добавить» выбрать «Найти», появится Служба1.
5. Нажать кнопку «ОК» для добавления этой ссылки на службу.

Построение клиентского приложения

1. Если конструктор Windows Forms еще не открыт, дважды необходимо щелкнуть файл Form1.cs в обозревателе решений задач и открыть его (рис. 5).
2. На панели элементов перетянуть в форму следующие элементы управления: 2 TextBox, Label и 4 Button. Переименуйте кнопки как показано на рис 4
3. Для каждой кнопки добавить обработчик события Click. Для этого по каждой кнопке щелкнуть два раза.

2.6 Принцип реализации сетевых служб в ИТК

Сетевая служба ИТК [3] идентифицируется программой (сложение и вычитание чисел) из символов URI, свойства и методы которой описаны в языке WSDL [1]. Доступ к ресурсам осуществляется через протокол SOAP,

который представляется XML-запросами, передаваемые HTTP Интернет–протоколом. Сетевые службы близки классам сервера приложений Java Enterprise Edition. Ключевым понятием сетевой службы является сообщение из одной или нескольких переменных. Методы классов задаются операциями с входным и выходным значениями сообщений. После вызова операции переменной входного сообщения протокола SOAP, интерпретируются как параметры соответствующего метода класса, который лежит в основе службы. После завершения работы метода формируется исходное сообщение, которое содержит возвращенное методом значение, после чего оно отправляется клиенту по протоколу SOAP. Сформулированные принципы и понятия создания распределенных приложений с использованием сервисно-компонентных архитектур представлены в комплексе ИТК в виде готовых КПИ и операций их реализации.

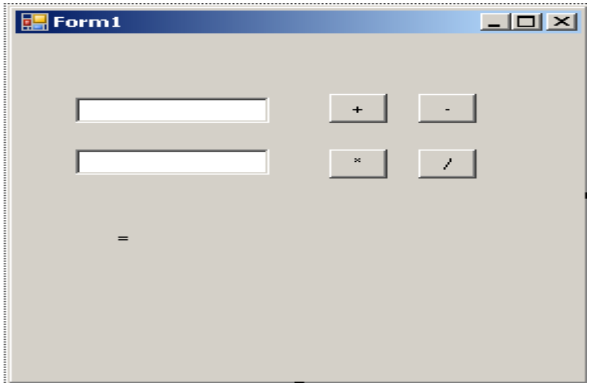


Рис.5. Вид клиентского приложения сервиса калькулятора

Список литературы

[1]. Лаврищева Е. М. Software Engineering компьютерных систем. Парадигмы, Технологии, CASE-средства программирования. – К.: Наук. Думка, 2014 – 284 с.

[2]. Lavrischeva, E.: Formal Fundamentals of Component Interoperability in Programming. In: Cybernetics and Systems Analysis, vol. 46, no. 4, pp. 639–652. Springer, Heidelberg (2010), http://link.springer.com/article/10.1007%2Fs10559-010-9240-z

[3]. Лаврищева Е. М., Зинькович В. М., Куцаченко Л. И. и др. Инструментально-технологический комплекс для разработки и обучения приемам производства программных систем. Госслужба интеллектуальной собственности Украины. – Свидетельство о регистрации №45292 от 27.08.2012. –108 с. (украинский).

[4]. Andrew S. Tanenbaum, Maarten van Steen. "Distributed Systems. Principles and paradigms". Prentice Hall, Inc., 2002 (Таненбаум Э., ван Стеен М.. "Распределённые системы. Принципы и парадигмы". СПб.: Питер, 2003 – 878 с.)

[5]. Gustavo Alonso, Fabio Casati, Harumi Kuno, Vijay Machiraju. Web Services. Concepts, Architectures and Applications. Springer-Verlag, 2004

[6]. Карпов Л. Е. Архитектура распределенных систем программного обеспечения – М., МАКС Пресс, 2007. – 130 с.
[7]. Карпов Л. Е., Юдин В. Н. Обмен данными в распределённой системе поддержки решений. Труды Института системного программирования, т. 19, М., Институт системного программирования РАН, 2010, стр. 71-80, ISBN 978-0-543-57630-9, ISBN 978-5-4221-0085-9, ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), http://www.ispras.ru/ru/proceedings/docs/2010/19/isp_19_2010_71.pdf
[8]. http://www.corba.org/
[9]. Jon Siegel. "Quick CORBA™ 3". Wiley Computer Publishing, John Wiley & Sons, Inc., 2001 (Джон Сигел, "CORBA 3", М., МАЛИП, 2002).
[10]. http://www.w3.org/TR/2008/REC-xml-20081126/
[11]. Гладцын В. А., Кринкин К. В. Яновский В. В. Сервис-ориентированная архитектура: стандарты, алгоритмы, протоколы – Санкт-Петербург: СПб ГЭТУ ЛЭТИ, 2006 – 108 с.
[12]. Papazoglou M. P., Dubray J.-J. A Survey of Web Service Technologies, Technical Report DIT-04-058, Ingegneria e Scienza dell'Informazione, University of Trento, 2004.
[13]. http://www.w3.org/TR/soap/
[14]. http://www.w3.org/TR/soap12-part1/
[15]. http://www.w3.org/TR/wsdl20
[16]. http://www.w3.org/RDF/
[17]. http://www.w3.org/TR/wsci/
[18]. http://www.w3.org/TR/wscl10/
[19]. http://www.omg.org/spec/BPMN
[20]. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
[21]. http://www.oasis-open.org/specs/index.php#wsbpelv2.0
[22]. http://uddi.org/pubs/uddi-v3.0.2-20041019.htm
[23]. http://www.oasis-open.org/specs/index.php#uddiv2
[24]. http://www.oasis-open.org/specs/index.php#uddiv3
[25]. http://www.oasis-open.org/specs/index.php#uddiv3.0.2
[26]. http://www.omg.org/spec/UML/ISO/19505-1/PDF
[27]. http://www.omg.org/spec/UML/ISO/19505-2/PDF
[28]. http://www.ibm.com/developerworks/websphere/techjournal – IBM WebSphere Developer Technical Journal.
[29]. Архитектура IT-ландшафта на базе корпоративных сервисов. Разработка маршрутной карты – 2011 – 16 с., http://www1.sap.com/cis/pdf/ESARoadmap.pdf.
[30]. http://www.ivk.ru/po/upiter/

Internet services for solving business problems

E. Lavrischeva <lavr@ispras.ru>
L. Karpov <mak@ispras.ru>
A. Tomilin <tom11@bk.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn str., Moscow, 109004, Russia.

Abstract. Different types of global information network services, used in the modern distributed software systems are described. Several approaches to service description and to service interaction are shown. Description of wide-spread and inculcated approaches to creation of distributed object interaction is given in the wide practice of the systems with spectrum of system and functional services. Web-service international standards stack (SOAP protocol, WSDL language, UDDI interface and service, XML, BPEL and BPMN notations, CORBA object broker and services, JEE application server) is briefly described. Models of web services, service-oriented approach (SOA), service-component architectures (SCA), and Windows Communication Foundation (WCF) service support application systems are considered for presentation of the business systems by the services ready to the decision of business-tasks. Features that support interoperability and multilanguage interaction are underlined. Principles of distributed software design (service composition, component engineering, interoperability, variance) that are of special care in SOA are listed. Some examples of presenting information using standard notations are given. An additional example of calculator service with functions of the data processing in ITK service-environment (WCF-based system) and the concept of web service implementation are presented.

Keywords: distributed system; worldwide network; models of service architectures; network service; co-operation protocols; languages of services and interconnection; system business.

DOI: 10.15514/ISPRAS-2015-27(1)-7

For citation: Lavrischeva E., Karpov L., Tomilin A. Internet services for solving business problems. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 1, 2015, pp. 125-150 (in Russian). DOI: 10.15514/ISPRAS-2015-27(1)-7

References

[1]. Lavrishcheva E. M. Software Engineering komp'yuternykh sistem. Paradigmy, Tekhnologii, CASE-sredstva programmirovaniya. [Software Engineering for computer systems. Paradigms, Methods, CASE technology]– K.: Nauk. Dumka, 2014 – 284 pp. (in Russian).

[2]. Lavrischeva, E. Formal Fundamentals of Component Interoperability in Programming. In: Cybernetics and Systems Analysis, vol. 46, no. 4, pp. 639–652. Springer, Heidelberg (2010), http://link.springer.com/article/10.1007%2Fs10559-010-9240-z

[3]. Lavrishcheva E. M., Zin'kovich V. M., Kutsachenko L. I. et al. Instrumental'no-tekhnologicheskii kompleks dlya razrabotki i obucheniya priemam proizvodstva programmnykh sistem [Instrumental technology for design and teaching procedures of prgramming system creation]. Gossluzhba intellektual'noi sobstvennosti Ukrainy [Ukraine state service for intellectual property support]. – Svidetel'stvo o registratsii [Registry sertificate] #45292 27.08.2012. – 108 p. (in Ukrainian).

[4]. Andrew S. Tanenbaum, Maarten van Steen. "Distributed Systems. Principles and paradigms". Prentice Hall, Inc., 2002

[5]. Gustavo Alonso, Fabio Casati, Harumi Kuno, Vijay Machiraju. Web Services. Concepts, Architectures and Applications. Springer-Verlag, 2004

[6]. Karpov L. E. Arkhitektura raspredelennykh sistem programmnogo obespecheniya [Distributed software systems architecture] – Moscow, MAKS Press, 2007, 130 p.

[7]. Karpov L. E., Yudin V. N. Obmen dannymi v raspredelennoi sisteme podderzhki reshenii [Data exchange in distributed software system for decision support]. Trudy ISP RAN [The Proceedings of ISP RAS], 2010, vol. 19, pp. 71-80 (in Russian) http://www.ispras.ru/ru/proceedings/docs/2010/19/isp_19_2010_71.pdf
[8]. http://www.corba.org/
[9]. Jon Siegel. "Quick CORBA™ 3". Wiley Computer Publishing, John Wiley & Sons, Inc., 2001.
[10]. http://www.w3.org/TR/2008/REC-xml-20081126/
[11]. Gladtsyn V. A., Krinkin K. V. Yanovskii V. V. Servis-orientirovannaya arkhitektura: standarty, algoritmy, protokoly [Service-oriented architecture: standards, algorithms, protocols] – Sankt-Peterburg: SPb GETU LETI, 2006 – p. 108 (in Russian).
[12]. Papazoglou M. P., Dubray J.-J. A Survey of Web Service Technologies, Technical Report DIT-04-058, Ingegneria e Scienza dell'Informazione, University of Trento, 2004.
[13]. http://www.w3.org/TR/soap/
[14]. http://www.w3.org/TR/soap12-part1/
[15]. http://www.w3.org/TR/wsdl20
[16]. http://www.w3.org/RDF/
[17]. http://www.w3.org/TR/wsci/
[18]. http://www.w3.org/TR/wscl10/
[19]. http://www.omg.org/spec/BPMN
[20]. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
[21]. http://www.oasis-open.org/specs/index.php#wsbpelv2.0
[22]. http://uddi.org/pubs/uddi-v3.0.2-20041019.htm
[23]. http://www.oasis-open.org/specs/index.php#uddiv2
[24]. http://www.oasis-open.org/specs/index.php#uddiv3
[25]. http://www.oasis-open.org/specs/index.php#uddiv3.0.2
[26]. http://www.omg.org/spec/UML/ISO/19505-1/PDF
[27]. http://www.omg.org/spec/UML/ISO/19505-2/PDF
[28]. http://www.ibm.com/developerworks/websphere/techjournal – IBM WebSphere Developer Technical Journal.
[29]. Arkhitektura IT-landshafta na baze korporativnykh servisov. Razrabotka marshrutnoi karty [IT landscape architecture based on corporate services. Building a roadmap] – 2011 – pp. 16, http://www1.sap.com/cis/pdf/ESARoadmap.pdf (in Russian).
[30]. http://www.ivk.ru/po/upiter/

Применение временных рядов в задаче фоновой идентификации пользователей на основе анализа их работы с текстовыми данными[1]

В.Ю. Королёв <bruce27@yandex.ru>
А.Ю. Корчагин <proton.ru@gmail.com>
И.В. Машечкин <mash@cs.msu.su>
М.И. Петровский <michael@cs.msu.su>
Д.В. Царёв <tsarev@cs.msu.su>
Факультет вычислительной математики и кибернетики,
Московский государственный университет имени М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, д. 1с52.

Аннотация. В статье представлен новый подход идентификации пользователя на основе анализа его поведения при работе с текстовой информацией. Для описания поведения пользователя предлагается использовать содержимое текстовых документов, к которым он обращался. Структурированное представление рассматриваемой поведенческой информации осуществляется на основе отображения содержимого электронных документов в тематическое пространство пользователя, формируемое с использованием неотрицательной матричной факторизации. Веса выделенных тематик в документе характеризуют тематическую направленность пользователя во время работы с данным документом. Изменение значений весов тематик во времени формирует многомерный временной ряд, описывающий историю поведения пользователя при работе с текстовыми данными. Построение прогноза такого временного ряда позволит осуществлять идентификацию данного пользователя на основе оценки отклонений наблюдаемой тематической направленности пользователя от спрогнозированных значений. В рамках предложенного подхода был разработан собственный оригинальный метод прогнозирования временных рядов, основанный на ортонормированной неотрицательной матричной факторизации (ОНМФ). Важно отметить, что ранее методы неотрицательной матричной факторизации не использовался для решения задачи прогнозирования временных рядов. Проведённое экспериментальное исследование на примере реальной корпоративной переписки пользователей, сформированной из набора данных Enron,

[1] Работы выполнены при финансовой поддержке Минобрнауки России (Соглашение № 14.604.21.0056 о предоставлении субсидии, Уникальный идентификатор прикладных научных исследований RFMEFI60414X0056).

показало применимость предложенного подхода идентификации пользователя. Кроме того, эксперименты с применением других популярных на сегодняшний день методами прогнозирования показали превосходство разработанного метода на основе ОНМФ по качеству классификации тематических характеристик пользователя. Также в работе исследовались два различных подхода оценки отклонений: абсолютная оценка и оценка p-значения. Эксперименты показали, что оба рассмотренные подхода расчёта оценки отклонения временной точки от прогноза применимы в предложенном подходе идентификации пользователя.

Ключевые слова: компьютерная безопасность; идентификация пользователя; тематическое моделирование; ортонормированная неотрицательная матричная факторизация; прогнозирование временных рядов

DOI: 10.15514/ISPRAS-2015-27(1)-8

Для цитирования: Королёв В.Ю., Корчагин А.Ю., Машечкин И.В., Петровский М.И., Царёв Д.В. Применение временных рядов в задаче фоновой идентификации пользователей на основе анализа их работы с текстовыми данными. Труды ИСП РАН, том 27, вып. 1, 2015 г., стр. 151-172. DOI: 10.15514/ISPRAS-2015-27(1)-8.

1. Введение

Актуальность проблемы защиты компьютерной информации в настоящее время не вызывает сомнений. Использование стандартных средств защиты информации, основанных на разграничении прав доступа, контроле целостности, аутентификации пользователей с использованием паролей, ключей или цифровых подписей, а также применение систем контроля работы пользователей, основанных на предопределенных регламентах, политиках, правилах и использовании сигнатурных методов обнаружения вторжений, не дают надежной защиты.

По мнению ведущих специалистов по компьютерной безопасности, перспективным инструментом в настоящей предметной области являются подходы на основе анализа поведенческой биометрии пользователей с использованием статистических методов и методов машинного обучения [1]. В общем случае биометрия включает совокупность процедур и методов распознавания людей по одной или нескольким физиологическим или поведенческим чертам. Поведенческие признаки относятся к индивидуальным особенностям поведения человека, сформированным в результате его уникального опыта, навыков и знаний, которые не являются секретной информацией, но их невозможно абсолютно точно передать или скопировать, а также достаточно тяжело подделать. Например, графическая подпись, стилистические и морфологические особенности устной и письменной речи, динамика работы с устройствами ввода-вывода и т.д. К поведенческим признакам также относятся особенности потребляемой и создаваемой пользователем текстовой информации (документы, электронные сообщения, почта).

Настоящая статья посвящена одной из важнейших задач защиты компьютерной информации — задаче идентификации пользователей, т.е. постоянной (или периодической) оценки достоверности того, что пользователь, работающий с защищаемой компьютерной системой, является действительно тем, от имени кого он авторизовался. В отличие от задачи аутентификации при идентификации не подразумевается явных процедур проверки, требующих интерактивных действий от пользователя.

В статье рассматриваются методы машинного обучения и математической статистики для построения и применения поведенческих моделей с целью решения задач постоянной фоновой идентификации пользователей на основе его работы с текстовыми данными. Идея предлагаемого подхода состоит в тематическом анализе сложившихся в прошлом тенденций работы (поведения) пользователя с текстовым контентом различных (в том числе конфиденциальных) категорий и прогнозировании его дальнейшего поведения. Тематический анализ работы пользователя предполагает определение основных тематик его текстового контента и расчёт соответствующих им весов в заданные интервалы времени. На основе отклонений поведения в работе пользователя с контентом от прогноза можно выявить интервалы времени, когда:

- велась работа с документами несвойственных категорий;
- работа с документами той или иной категории отличается от обычной (исторической).

Настоящая статья имеет следующую структуру. В разделе 2 приведено описание процедуры выделения тематических характеристик из текстовых данных пользователя и формирования тематических временных рядов. В Разделе 3 рассматриваются популярные подходы прогнозирования временных рядов и представлен собственный оригинальный метод прогнозирования на основе ортонормированной неотрицательной матричной факторизации. Раздел 4 посвящен экспериментальному исследованию предложенного подхода идентификации пользователя на примере реальной корпоративной переписки, сформированной из набора электронных писем Enron. Также в данном разделе исследуются два различных подхода оценки отклонений: абсолютная оценка по всем тематикам и оценка p-значения критерия согласия Хи-квадрат. В Разделе 5 делаются основные выводы и приводится заключение.

2. Выделение тематических характеристик из текстовых данных пользователя

В основе предлагаемого подхода идентификации пользователей лежит тематическое моделирование текстовых данных, с которыми работал

пользователь за заданное модельное время. Модельное время разбивается на последовательно измеренные через некоторые (зачастую равные) промежутки времени интервалы, например, в качестве промежутка времени (шага) может быть выбран час, день или время, за которое происходит заданное число событий. (рис. 1) [2]. С помощью тематического моделирования выделяются основные тематики текстового контента пользователя и соответствующие им веса в каждом временном интервале модельного времени.

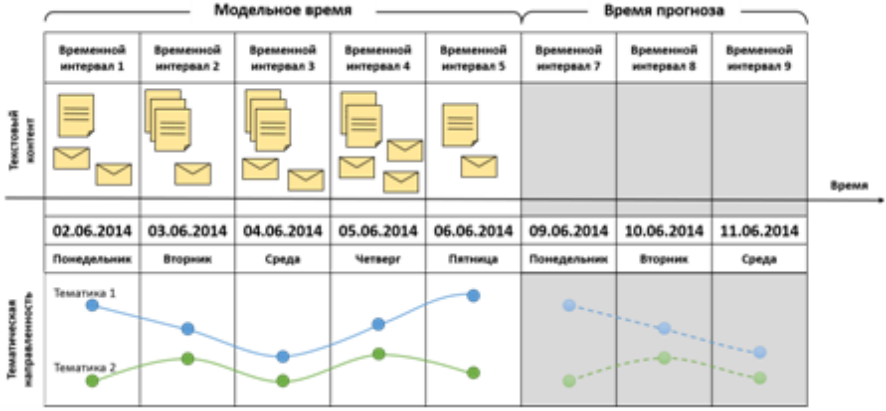


Рис. 1. Формирование временных рядов тематической направленности пользователя.

Веса тематик во временном интервале характеризуют тематическую направленность пользователя, на их основе формируются временные ряды изменения его тематической направленности для каждой из тематик. Далее по сформированным временным рядам строятся прогнозы (рис. 1). На основе значений отклонений тематической направленности от спрогнозированных данных определяются временные интервалы с несвойственной активностью пользователя с контентом.

Исходя из предыдущих работ авторов [3-7] в качестве методов тематического моделирования были выбраны методы, основанные на неотрицательной матричной факторизации. Методы неотрицательной матричной факторизации работают с векторным представлением текста типа «мешок слов» (англ. «bag-of-words») [8]. В нашем случае в качестве текстов выступают текстовые данные каждого временного интервала. Далее под термином временной интервал в зависимости от контекста будет пониматься либо совокупность текстовых данных анализируемого пользователя за рассматриваемый временной интервал, либо непосредственно интервал времени.

Формально опишем модель построения тематических временных рядов для n временных интервалов модельного времени. Каждый временной интервал j ($1 \le j \le n$) отображается в числовой вектор $A_j=[a_{1,j}, a_{2,j}, \ldots, a_{m,j}]^T$ фиксированной размерности m, где m — число признаков текстовых данный за модельное

время, а *i*-ая компонента вектора определяет вес *i*-го признака в *j*-ом временном интервале.

В качестве признаков в модели «мешка слов» используются лексемы, входящие в текст, а размерность признакового пространства равна размерности словаря лексем. Под лексемами в общем случае понимаются все различные слова текста. Однако обычно применяются некоторые меры по предварительной обработке текста с целью получения более «информативного» признакового пространства [8]: удаление стоп-слов, приведение слов к нормализованной форме (стемминг) и т.д. Цель предварительной обработки текста — оставить только те признаки, которые наиболее информативны, т.е. наиболее сильно характеризуют текст. К тому же уменьшение анализируемых признаков приводит к уменьшению использования вычислительных ресурсов. В интеллектуальном анализе текстовых данных для обозначения признака текста принято использовать термин «терм».

Вес *i*-го терма в векторном представлении *j*-го временного интервала определяется как $a_{i,j}=L_{i,j}G_i$. $L_{i,j}$ — локальный вес терма *i* во временном интервале *j*, G_i — глобальный вес терма *i* во всех временных интервалах. Т.к. для вычисления отклонений от спрогнозированных значений будут использоваться новые временные интервалы, не вошедшие в модельное время, то заранее определить использование того или иного терма в будущих временных интервалах невозможно, поэтому использование глобального веса исключается. В ходе экспериментов, проводимых в Разделе 4, наилучшие результаты были получены при использовании логарифмического веса в качестве локального: $L_{i,j} = 1 + \log(t_{i,j})$, где $t_{i,j}$ — число появлений терма *i* во временном интервале *j* [7, 8].

Таким образом, текстовый контент пользователя за модельное время представляется в виде числовой матрицы, строки которой соответствуют термам, а столбцы текстам каждого временного интервала. Объединение термов в тематики и представление временных интервалов в пространстве тематик осуществляется путём применения к данной матрице неотрицательной матричной факторизации.

Матрица модельных временных интервалов $A \in \Re^{m \times n}$, где *m* — число различных термов, *n* — число временных интервалов. Элементы матрицы *A* принимают неотрицательные значения, т.к. являются весами соответствующих термов во временных интервалах. Тогда цель неотрицательной матричной факторизации состоит в нахождении матриц $W_k \in \Re^{m \times k}$ и $H_k \in \Re^{k \times n}$ с неотрицательными элементами, которые минимизируют целевую функцию [9]:

$$f(W_k, H_k) = \frac{1}{2}\left\|A - W_k H_k\right\|_F^2, \; k << min(m,n).$$

Матрица $W_k=[w_{ij}]$ задает отображение пространства k тематик в пространство m термов, матрица $H_k=[h_{ij}]$ соответствует представлению временных интервалов в пространстве тематик, т.е. элемент h_{ij} соответствует представлению i-ой тематики в j-ом временном интервале. В связи с тем, что элементы матрицы H_k неотрицательны, то их можно рассматривать как вклад(вес) тематики во временной интервал. Чем больше значение элемента h_{ij} по сравнению с другими элементами j-го временного интервала, тем более характерна i-ая тематика для текста данного временного интервала. На этом свойстве основаны алгоритмы кластеризации, использующие неотрицательную матричную факторизацию [10, 11]. Аналогично и для матрицы W_k, чем больше значение элемента w_{ij} по сравнению с другими элементами j-го столбца (j-ой тематике), тем более характерен i-ый терм для данной тематики [12].

Исходя из описанных свойств неотрицательной матричной факторизации, временной ряд изменения каждой из выделенных k тематик формируется из значений элементов соответствующей строки матрицы H_k (рис. 2).

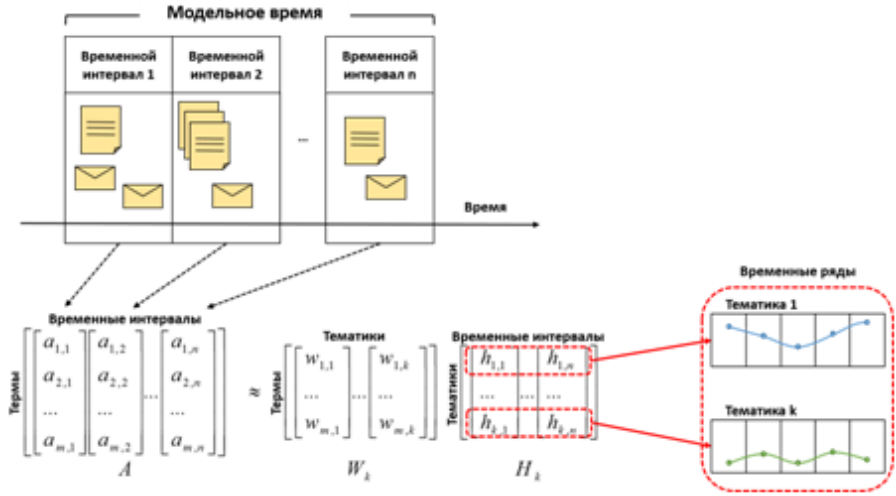


Рис. 2. Построение временных рядов на основе неотрицательной матричной факторизации.

Существуют различные методы реализации неотрицательной матричной факторизации [9, 11, 12, 13]. Однако для реализации предлагаемого подхода, необходимо иметь возможность отображать новые, не вошедшие в модельное время, временные интервалы в уже построенное пространство тематик. Для реализации данного функционала необходимо наложить дополнительное условие ортонормированности матрицы W_k: $W_k^T \cdot W_k = I$. Тогда для отображения матрицы временных интервалов времени прогноза A_{new} в

пространство тематик модельного времени достаточно A_{new} слева умножить на W_k^T : $W_k^T \cdot A_{new} = H_{k\_new}$.

Авторы статьи исследовали применение множества популярных алгоритмов, реализующих ортонормированную неотрицательную матричную факторизацию [9, 11, 13], для использования в предлагаемом подходе. На основе экспериментальных исследований, оставшихся за рамками данной статьи, был выбран алгоритм минимизации целевой функции $f(W_k, H_k) = \frac{1}{2}\|A - W_k H_k\|_F^2 + \frac{\alpha}{2}\left\|W_k^T W_k - I\right\|_F^2$, описанный в [9] и позволяющий задавать баланс между точностью приближения исходной матрицы и ортонормированностью получаемых тематик с помощью параметра α.

1. Элементы матриц $W^1 \in \Re_+^{m \times k}$ и $H^1 \in \Re_+^{k \times n}$ инициализируются случайными неотрицательными числами;
2. В цикле p раз выполняются итерационные формулы для вычисления матриц W и H:

   a. $H_{b,j}^{p+1} = H_{b,j}^p \frac{((W^p)^T A)_{b,j}}{((W^p)^T W^p H^p)_{b,j}}, \forall b, j : 1 \le b \le k, 1 \le j \le n$,

   b. $W_{i,a}^{p+1} = W_{i,a}^p \frac{(A(H^{p+1})^T + \alpha W^p)_{i,a}}{(W^p H^{p+1}(H^{p+1})^T + \alpha W^p (W^p)^T W^p)_{i,a}}, \forall i, a : 1 \le i \le m, 1 \le a \le k$.

Для построения прогнозов k тематических временных рядов, заданных матрицей H_k, в предлагаемом подходе идентификации пользователя применялись методы, описанные в Разделе 3. После построения прогнозов тематических рядов вычисляются отклонения тематической направленности пользователя за время прогноза от спрогнозированных значений. Вычисленные отклонения используются для идентификации временных интервалов с несвойственной тематической направленностью пользователя (Рисунок 3).

Отдельно отметим, что матрица W_k, построенная по модельным временным интервалам пользователя, описывает основные тематики пользовательского контента и служит для отображения любых текстовых данных (не обязательно временных интервалов) в пространство тематик данного пользователя, т.е. матрица W_k «характеризует» пользователя с точки зрения его тематических предпочтений в контенте. Поэтому для обозначения матрицы W_k нами также будет использоваться термин тематический «портрет» пользователя. Получаем, что в общем случае временные интервалы одного пользователя можно проецировать в тематическое пространство другого пользователя, и на основе полученных представлений также анализировать временные ряды. Данная возможность позволяет определять временные интервалы, когда один пользователь активно интересовался материалами характерными для другого

пользователя. Также возможно сформировать подобную матрицу W_k вообще не на основе временных интервалов пользователей, а на основе заранее сформированного набора документов организации (тренировочный набор), чьи тематики представляют наибольший интерес для анализа работы пользователей, получив таким образом тематический «портрет» набора документов и уже на его основе строить временные ряды для пользователя.

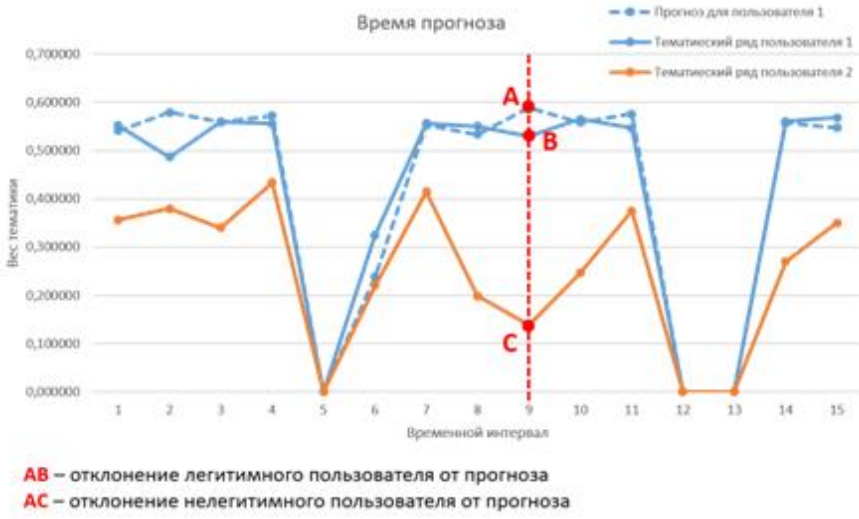


Рис. 3. Пример отклонений тематических временных рядов от прогноза.

3. Прогнозирование временных рядов

Прогнозирование временных рядов заключается в построении модели для предсказания будущих событий, основываясь на известных событиях прошлого [1]. Для построения прогноза временных рядов в статье использовались следующие модели: линейная модель авторегрессии, авторегрессионная модель дерева и предложенная оригинальная модель на основе ортонормированной неотрицательной матричной факторизации.

3.1 Линейная модель авторегрессии и авторегрессионная модель дерева

В линейной модели авторегрессии временных рядов значение временного ряда в данный момент линейно зависят от предыдущих значений этого же ряда. Формально авторегрессионную модель порядка p, которую обычно обозначают, как AR(p), определяют следующим образом:

1. $X_t = c + \sum_{i=1}^{p} \varphi_i X_{t-i} + \varepsilon_t$, где φ_i — параметры авторегрессионной модели, c — константа (для простоты константу, как правило, опускают), ε_t — белый шум;
2. Форма записи с помощью оператора задержки L ($Lx_t = x_{t-1}$): $c + \varepsilon_t = \left(1 - \sum_{i=1}^{p} \varphi_i L_i\right) X_t$.

Авторегрессионная модель дерева (англ. AutoRegressive Tree Model, ART) — модель дерева принятия решений, в «листьях» которого располагаются авторегрессионные модели (AR). Для реализации данной модели в статье используется алгоритм ARTXP, разработанный Microsoft, который основан на их реализации алгоритма дерева принятия решений. Алгоритм ARTXP устанавливает соотношение между переменным количеством предыдущих элементов и каждым текущим элементом, для которого выполняется прогноз [14].

Отметим основные особенности алгоритма Microsoft ARTXP [15]:

- алгоритм ARTXP поддерживает учёт корреляций между несколькими анализируемыми рядами, т.е. поддерживает перекрестное прогнозирование;
- алгоритм ARTXP используется для прогнозирования ближайших значений временного ряда (порядка 5-7 временных шагов) и даёт существенно менее точный долгосрочный прогноз.

3.2 Модель прогнозирования на основе ортонормированной неотрицательной матричной факторизации

Анализируемый временной ряд можно представить в виде вектора, элементами которого являются рассчитанные значения в соответствующих временных точках, например, строки матрицы H_k (Раздел 2). Для того чтобы представить вектор временного ряда в матричной форме введём понятие порядка модели, т.е. количество подряд идущих значений временного ряда по которым определяются основные взаимосвязи, аналогично линейной модели авторегрессии. Тогда вектор временного ряда размерности n при заданном порядке модели p можно представить в виде матрицы размерности $p \times (n-p+1)$, чьи столбцы соответствуют всевозможным подпоследовательностям длины p подряд идущих временных точек анализируемого ряда (рис. 4).

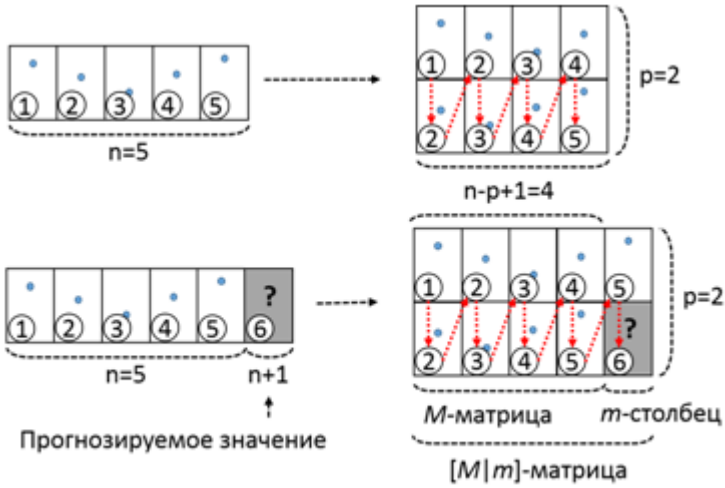


Рис. 4. Отображение вектора временного ряда в матрицу.

Задача прогнозирования следующего (n+1)-го значения (рис. 4) сводится к задаче аппроксимации пропущенных значений в матрице временного ряда (матрица [M|m]) на основе уже заполненных значений (матрица M) — т.н. задача подстановки пропущенных значений (англ. Missing Value Imputation) [16-18].

Наиболее распространенным методами матричного разложения к решению задачи подстановки пропущенных значений являются подходы на основе сингулярного разложения (Singular Value Decomposition, SVD) [16-18]. Однако, авторами была исследована возможность применения методов неотрицательной матричной факторизации для решения задачи подстановки пропущенных значений [18], на основе полученных результатов в рамках настоящей статьи будет описан алгоритм подстановки пропущенных значений с использованием ортонормированной неотрицательной матричной факторизации.

Рассмотрим ортонормированную неотрицательную матричную факторизацию матрицы M: $M \approx M_k = Wm_k \cdot Hm_k$, $Wm_k^T \cdot Wm_k = I$, где M_k является аппроксимацией исходной матрицы M, k — число «латентных» признаков, при этом $k << min(p, n\text{-}p+1)$ [6, 18, 19]. Матрица Wm_k задает отображение между пространством «латентных» признаков размерности k и пространством позиций элементов (от 1 до p) в сформированных подпоследовательностях длины p. Таким образом, выделенные «латентные» признаки содержат только наиболее значимую информацию о взаимосвязях между позициями элементов среди всех подпоследовательностей (см Раздел 2). Каждый столбец матрицы Hm_k описывает соответствующую подпоследовательность в виде вектор-столбца с весами соответствующих базисных «латентных» признаков.

Было предложено вычисление пропущенных значений в матрице $[M|m]$ на основе значений матрицы M следующим итерационным способом (по аналогии с решением в [16]):

- *Шаг 0*. Рассчитать ортонормированную неотрицательную матричную факторизацию матрицы M (с изначально заполненными элементами): $M \approx M_k = Wm_k \cdot Hm_k$, $Wm_k^T \cdot Wm_k = I$.
- *Шаг 1*. Инициализировать пропущенное значения в столбце m. Традиционно в литературе пропущенные значения инициализируются средним значением по столбцу или всему вектору временного ряда. Если известны данные о сезонности временного ряда, то при расчёте средних значений можно учитывать и шаг сезонности. На выходе получается полностью заполненный столбец m^i, где $i = 0$.
- *Шаг 2*. Рассчитать аппроксимацию столбца m с помощью полученной на *шаге 0* модели ортонормированную неотрицательную матричную факторизацию (ONMF-модель): $m_{approx} = (Wm_k \cdot Wm_k^T) \cdot m^i$. После чего сформировать m^{i+1} путем замены пропущенного значения в исходном столбце m соответствующим полученным значением в m_{approx}.
- *Шаг 3*. До тех пор пока значение $\left\| m^i - m^{i+1} \right\| / \left\| m^i \right\|$ меньше заданного порога (как правило, значение порога берут равным 10^{-6}), установить $i = i+1$ и перейти на шаг 2. На практике обычно достаточно 5-6 шагов для сходимости алгоритма.

Заметим, что в приведённом алгоритме вместо ортонормированной неотрицательной матричной факторизации можно использовать и традиционное сингулярное разложение [16].

При прогнозировании многомерных временных рядов можно их прогнозировать как по отдельности, так и с поддержкой перекрестного прогнозирования. Для этого совокупность из m временных рядов нужно объединить в одну матрицу (рис. 5). Тогда получаемые «латентные» признаки (матрица Wm_k) будут содержать наиболее значимую информацию о взаимосвязях между позициями элементов всех временных рядов среди всех подпоследовательностей, таким образом будет учитываться влияние всех анализируемых временных рядов друг на друга.

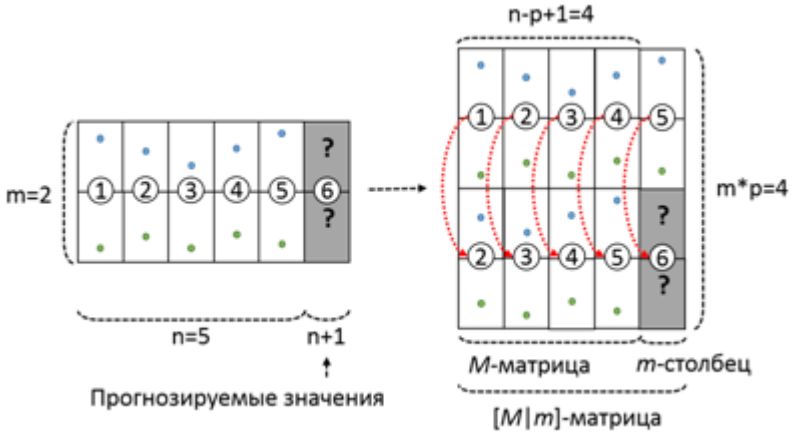


Рис. 5. Формирование матрицы прогнозирования для совокупности временных рядов.

4. Экспериментальные исследования

Первоочередной задачей при проведении экспериментальных исследований предлагаемого подхода анализа работы пользователя с текстовой информацией являлся выбор тестового набора данных. Основными критериями выбора тестового наборы были:

1. контент из корпоративной среды;
2. большое количество текстовых данных;
3. возможность разделения контента по пользователям и времени.

На основе сформулированных выше требований для экспериментального исследования предлагаемого подхода был выбран набор Enron [20]. Набор Enron содержит электронную почту 150 сотрудников (главным образом из высшего руководства) американской энергетической компании, обанкротившейся в конце 2001 года. Кроме того, данный набор широко распространён в работах, посвящённых тематическому анализу текстовых данных [12].

Для экспериментального исследования предлагаемого подхода из набора Enron были выбраны три сотрудника с наибольшим количеством писем: «dasovich-j», «kaminski-v» и «kean-s». В качестве ограничения по времени было выбрано первое полугодие 2001 года, т.к. в это время велась самая активная переписка данных пользователей. Выбранные шесть месяцев были разбиты на пересекающиеся интервалы по пять недель с шагом одна неделя, при этом первые четыре недели каждого интервала использовались в качестве модельного времени, а следующая неделя для прогноза. Таким образом всё рассматриваемое время было разбито на 21 анализируемый временной интервал. В табл. 1 приведена статистика распределения общих писем между

выбранными пользователями. Из представленной в табл. 1 статистике распределения общих писем между выбранными пользователями видно, что пользователь «kean-s» имеет существенное количество (более 25%) общих писем с «dasovich-j», что должно сказаться на качестве классификации.

Табл. 1. Статистика распределения общих писем между пользователями.

Письма пользователя	Общие письма с пользователем		
	dasovich-j	kaminski-v	kean-s
dasovich-j	11480	56	1730
kaminski-v	16	8757	36
kean-s	2628	91	10043

Для демонстрации предлагаемого подхода идентификации работы пользователя рассматривалась следующая задача бинарной классификации: нужно определить временные точки работы с письмами пользователя «dasovich-j» от временных точек работы пользователей «kaminski-v» и «kean-s» на каждой неделе прогноза каждого из 21-го анализируемого временного интервала.

Для каждого из 21-го анализируемого временного интервала производилась следующая процедура, состоящая из 5 шагов:

1. Для каждого пользователя в качестве шага временной точки выбирался не фиксированный шаг времени, а время, за которое пользователь успевал обработать 50 писем. Таким образом, каждая точка временных рядов пользователей представляет конкатенацию из 50 их писем (рис. 6). Текстовые данные в наборе Enron являются англоязычными, поэтому для формирования словаря термов использовались такие методы предварительной обработки текста, как удаление стоп-слов и приведение слов к нормализованной форме на основе семантической сети WordNet [21]. Для вычисления весов термов использовался только локальный логарифмический вес. Векторы временных интервалов нормализовались по евклидовой норме.
2. К сформированной матрице модельных временных интервалов (точек) пользователя «dasovich-j» применялось тематическое моделирование на основе ортонормированной неотрицательной матричной факторизации для получения матрицы «портрета» пользователя (W_k) и матрицы представления временных интервалов в пространстве тематик (H_k). В итоге для пользователя «dasovich-j» получаем k тематических временных рядов для модельного времени (в проводимых экспериментах k=3).
3. Отображения векторов временных точек времени прогноза для всех пользователей осуществлялось с использованием матрицы «портрета» пользователя «dasovich-j» (W_k). Таким образом, получаем реальные

тематические данные всех пользователей для временных точек прогноза (рис. 6).

4. С помощью каждого метода прогнозирования (Раздел 3) строились прогнозы тематических временных рядов пользователя «dasovich-j» за модельное время (рис. 6). Далее для обозначения методов прогнозирования используются сокращения: метод линейной авторегрессии — AR, метод на основе авторегрессионной модели дерева решений — MS_ARTXP, предложенный метод прогнозирования на основе ортонормированной неотрицательной матричной факторизации — ONMF.
5. Для каждого метода прогнозирования рассчитывалась оценка отклонения каждой временной точки времени прогноза всех пользователей от спрогнозированных значений. Авторами исследовались два подхода к расчёту оценки отклонения временной точки от прогноза:
    a. *абсолютная оценка*: использование суммарного по всем k тематикам абсолютного отклонения реальных значений весов тематик от спрогнозированных (рис. 3);
    b. *оценка p-значения*: использование p-значения критерия согласия Хи-квадрат [22] для реальных (наблюдаемых) значений весов тематик и прогнозируемых, при этом веса тематик рассматриваются как вероятностное распределение, где вес каждой тематики фактически задает вероятность того, что указанный текст принадлежит данной тематике. Далее спрогнозированные веса рассматривались как ожидаемое распределение, а реальные веса — как наблюдаемое распределение вероятности, что позволяет определить уровень согласия по критерию Хи-квадрат. И чем он ниже, тем более вероятно, что произошла подмена пользователя.

Рис. 6. Временные ряды анализируемых пользователей для одной из тематик.

После проведения вышеописанной процедуры с каждым из 21-го анализируемого временного интервала для каждого метода прогнозирования получаем, что всем прогнозируемым временным точкам (96 точек прогноза для каждого пользователя, всего 288) всех пользователей сопоставлены две оценки отклонения.

Фиксируя значение порога допустимого отклонения от прогноза пользователя «dasovich-j» получаем бинарную классификацию для всех прогнозируемых временных интервалов. Для оценки качества классификации обычно используют ROC-кривые — графическая характеристика качества бинарного классификатора, зависимость доли верных положительных классификаций от доли ложных положительных классификаций при варьировании порога решающего правила (отклонения) [23]. Для сравнения нескольких моделей классификации будем использовать значение AUC (англ. Area Under Curve), которое вычисляется как площадь под ROC-кривой и является агрегированной характеристикой качества классификации, не зависящей от соотношения цен ошибок [23]. Чем больше значение AUC, тем «лучше» модель классификации. Полученные ROC-кривые и значения AUC для рассмотренных методов классификации и подходов расчёта оценки отклонения приведены на рис. 7, рис. 8 и табл. 2.

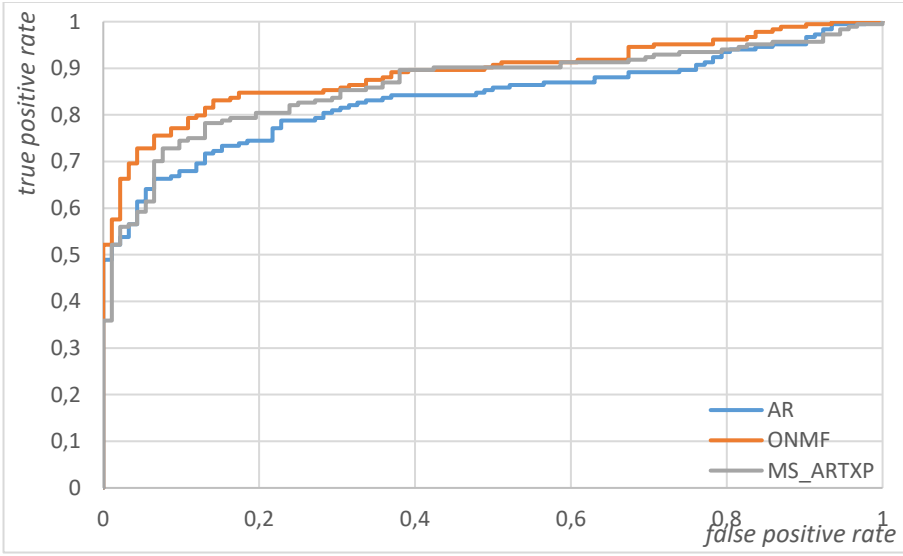


Рис. 7. ROC-кривые методов прогнозирования при абсолютной оценки отклонения.

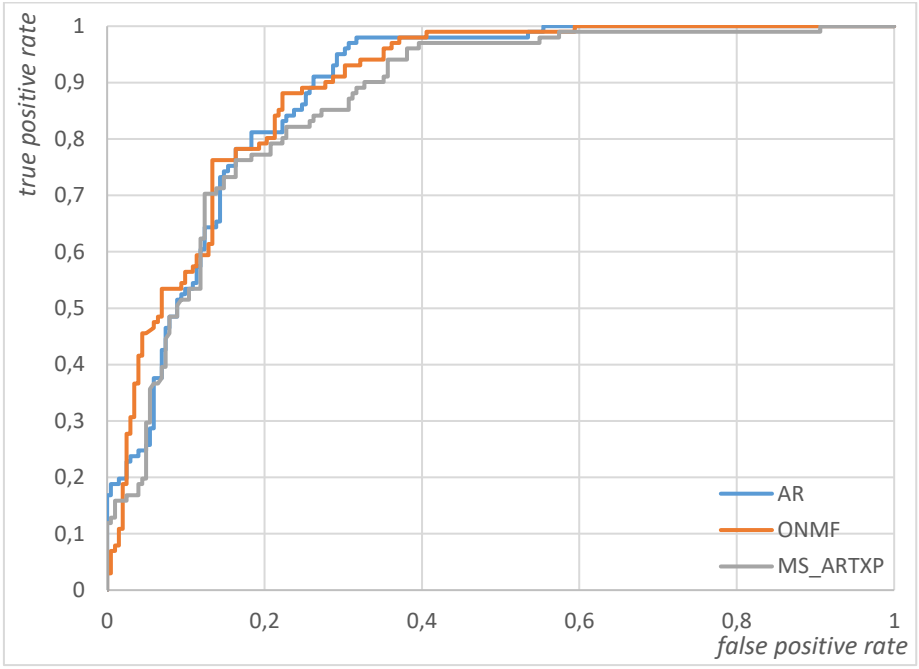


Рис. 8. ROC-кривые методов прогнозирования при оценке p-значения в качестве отклонения.

Табл. 2. Значения AUC для методов прогнозирования и подходов расчёта оценки отклонения.

	AR	MS_ARTXP	ONMF
Абсолютная оценка	0,835539	0,864367	0,889887
Оценка p-значения	0,883884	0,864719	0,889325

Из табл. 2 следует, что использование оценки p-значения показывает более стабильный результат классификации среди набора рассмотренных методов прогнозирования, так значение AUC не опускается ниже 0.86. Поэтому в дополнение приведём табл. 3 с сравнением средних p-значений критерия Хи-квадрат для пользователя «dasovich-j» и рассмотренных методов прогнозирования. При этом для достоверности рассматривались только те эксперименты, в которых число точек прогноз превышало 3, поэтому в табл. 3 приведены результаты 16 экспериментов вместо 21.

Табл. 3 Сравнение средних p-значений для пользователя «dasovich-j».

		Среднее p-значение		
Эксперимент	Число точек прогноза	AR	MS_ARTXP	ONMF
0	5	0.9475	0.8904	**0.9515**
2	7	0.8740	**0.888**	0.869
4	6	0.7773	0.7547	**0.8007**
5	6	0.9129	0.9037	**0.923**
6	6	0.8388	0.8285	**0.8526**
7	7	0.7932	**0.8528**	0.8336
8	4	**0.9462**	0.9038	0.9368
9	7	0.9718	0.9654	**0.9743**
10	7	**0.7456**	**0.7456**	0.6726
11	5	**0.8959**	0.8662	**0.8708**
12	7	0.7696	0.7492	**0.7985**
13	6	0.8609	0.5805	**0.8912**
14	4	**0.9288**	0.8555	0.9055
18	5	**0.8870**	0.5332	0.8639
19	4	0.9766	**0.9958**	0.9899
20	5	0.7365	**0.7409**	0.7267

Данные табл. 2 также согласуются с данными табл. 1 тем, что метод авторегрессии с использованием оценки p-значения продемонстрировал высокое значение AUC. Так в сравнении с остальными методами прогнозирования только метод авторегрессии во всех экспериментах показал средние p-значения большие 0.7.

Из приведённых данных следует, что:

- Предложенный подход идентификации работы пользователя на основе отклонений его тематической направленности от спрогнозированных данных показывает высокое качество идентификации даже при использовании стандартных методов прогнозирования;
- Предложенный в рамках данной статьи алгоритм прогнозирования временных рядов, основанный на ортонормированной неотрицательной матричной факторизации, показал высокое качество прогнозирования и свою применимость в рассмотренном подходе идентификации работы пользователя;

- Оба рассмотренные подхода расчёта оценки отклонения временной точки от прогноза применимы в предложенном подходе идентификации пользователя. Однако использование оценки p-значения показывает более стабильный результат классификации среди набора рассмотренных методов прогнозирования.

5. Заключение

В рамках проведенных исследований изучался вопрос возможности идентификации пользователя на основе анализа отклонений его тематической направленности при работе с текстовой информацией. Для решения указанной задачи был предложен подход, состоящий в тематическом анализе сложившихся в прошлом тенденций работы (поведения) пользователя с текстовым контентом различных (в том числе конфиденциальных) категорий и прогнозировании его дальнейшего поведения. Тематический анализ работы пользователя предполагает определение основных тематик его текстового контента и расчёт соответствующих им весов в заданные интервалы времени. На основе отклонений поведения в работе пользователя с контентом от прогноза осуществляется идентификация данного пользователя.

Для реализации тематического моделирования в предложенном подходе используется ортонормированная неотрицательная матричная факторизация (ОНМФ). В ходе дальнейших исследований был предложен собственный оригинальный метод прогнозирования, который также основан ОНМФ. Важно отметить, что ранее метод ОНМФ не использовался для решения задачи прогнозирования временных рядов.

Экспериментальное исследование предложенного подхода идентификации пользователя проводилось на примере реальной корпоративной переписки пользователей, сформированной из набора данных Enron. В ходе экспериментов было продемонстрировано, что использование разработанного метода прогнозирования на основе ОНМФ, показывает высокое качество классификации тематических характеристик пользователя по сравнению с другими популярны на сегодняшний день методами и свою применимость в рассмотренном подходе идентификации. Кроме того, в работе исследовались два различных подхода оценки отклонений: абсолютная оценка и оценка p-значения. Эксперименты показали, что оба рассмотренные подхода расчёта оценки отклонения временной точки от прогноза применимы в предложенном подходе идентификации пользователя.

В будущем планируется использовать предложенный подход идентификации пользователя при разработке программных средств анализа индивидуальных особенностей поведения пользователей компьютерных систем (поведенческой биометрии) при работе в рамках стандартного человеко-машинного интерфейса.

Список литературы

[1]. R.V. Yampolskiy, V. Govindaraju, Behavioural biometrics: a survey and classification. International Journal of Biometrics (IJBM), Vol. 1, No. 1, 2008.

[2]. Временной ряд (Time Series). March 24 2015. (http://www.machinelearning.ru/wiki/index.php?title=Временной_ряд)

[3]. И.В. Машечкин, М.И. Петровский, Д.В. Царёв. Методы вычисления релевантности фрагментов текста на основе тематических моделей в задаче автоматического аннотирования. Вычислительные методы и программирование. Том 14, 2013. 91-102.

[4]. I.V. Mashechkin, M.I. Petrovskiy, D.S. Popov, D.V. Tsarev. Automatic text summarization using latent semantic analysis. Programming and Computer Software, 2011, pp. 299-305.

[5]. D.V. Tsarev, M.I. Petrovskiy, I.V. Mashechkin. Using NMF-based text summarization to improve supervised and unsupervised classification. 11th International Conference on Hybrid Intelligent Systems (HIS), 2011. Malacca, MALAYSIA. P. 185-189.

[6]. D.V. Tsarev, M.I. Petrovskiy I.V. Mashechkin. Supervised and Unsupervised Text Classification via Generic Summarization. International Journal of Computer Information Systems and Industrial Management Applications. MIR Labs, Volume 5, 2013, pp. 509-515.

[7]. I.V. Mashechkin, M.I. Petrovskiy, D.S. Popov, D.V. Tsarev. Applying Text Mining Methods for Data Loss Prevention. Programming and Computer Software. January 2015, Volume 41, Issue 1, pp 23-30.

[8]. C.D. Manning, P. Raghavan, H. Schutze. Introduction to Information Retrieval. Cambridge University Press, 2008.

[9]. A. Mirzal. Converged Algorithms for Orthogonal Nonnegative Matrix Factorizations. CoRR abs/1010.5290, 2010.

[10]. Wei Xu, Xin Liu, Yihong Gong. Document clustering based on non-negative matrix factorization. Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval, Toronto, Canada, 2003.

[11]. Chris Ding, Tao Li, Wei Peng, Haesun Park. Orthogonal Nonnegative Matrix Tri-Factorizations for Clustering. SIGKDD, 2006.

[12]. M.W. Berry, M. Browne, A.N. Langville, V.P. Pauca, R.J. Plemmons. Algorithms and applications for approximate nonnegative matrix factorization. Computational Statistics and Data Analysis, pp. 155-173, 2007.

[13]. J. Yoo, S. Choi. Orthogonal Nonnegative Matrix Factorization: Multiplicative Updates on Stiefel Manifolds. Intelligent Data Engineering and Automated Learning – IDEAL 2008, vol. 5326 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 140–147.

[14]. C. Meek, D.M. Chickering, D. Heckerman. Autoregressive Tree Models for Time-Series Analysis, 2002. (http://go.microsoft.com/fwlink/?LinkId=45966)

[15]. Технический справочник по алгоритму временных рядов (Майкрософт). (http://msdn.microsoft.com/ru-ru/library/bb677216.aspx)

[16]. T. Hastie, R. Tibshirani, G. Sherlock, M. Eisen, P. Brown, D. Botstein. Imputing Missing Data for Gene Expression Arrays. Technical report, Stanford Statistics Department 1999.

[17]. O. Troyanskaya. Missing value estimation methods for DNA microarrays. Bioinformatics, , vol. 17, no. 6, 2001. pp. 520-525.

[18]. D.V. Tsarev, R.V. Kurynin, M.I. Petrovskiy, I.V. Mashechkin. Applying non-negative matrix factorization methods to discover user's resource access patterns for computer security tasks. Proceedings of the 2014 International Conference on Hybrid Intelligent Systems (HIS 2014). IEEE Computer Society [New York], United States, 2014. pp. 43–48.
[19]. D. Lee, S. Seung. Learning the parts of objects by non-negative matrix factorization. Nature, 401, 1999. pp. 788-791.
[20]. Enron Email Dataset. March 24 2015. (http://www.cs.cmu.edu/~./enron/)
[21]. Natural Language Toolkit (NLTK). March 24 2015. (http://www.nltk.org)
[22]. М. Кендалл, А. Стьюарт. Статистические выводы и связи. М.: Наука, 1973.
[23]. Кривая ошибок (Receiver Operating Characteristic, ROC curve). March 24 2015. (http://www.machinelearning.ru/wiki/index.php?title=ROC-кривая)

Applying Time Series to The Task of Background User Identification Based on Their Text Data Analysis[2]

D.V. Tsarev <tsarev@cs.msu.su>
M.I. Petrovskiy <michael@cs.msu.su>
I.V. Mashechkin <mash@cs.msu.su>
A.Y. Korchagin <proton.ru@gmail.com>
V.Y. Korolev <bruce27@yandex.ru>
Department of Computational Mathematics and Cybernetics,
Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia.

Abstract. The paper presents the novel approach of user identification based on behavior analytics of user operations with a text information. It is offered to describe user behavior by content of his text documents. The structured representation of the considered behavioral information is carried out based on representation of documents text content in the user topic space, which is created by non-negative matrix factorization. The topic weights in the document characterize the user's topic trend during an operating time with this document. The time variation of the topic weight values creates multidimensional time series that describe the history of user behavior when working with text data. Forecasting of such time series will allow for user identification based on estimated deviation of observed topic trend from the predicted topic weight values. This paper also presents the new time series forecasting method based on orthogonal nonnegative matrix factorization (ONMF) which is

[2] The production of this publication has been made possible through the financial support of the Ministry of Education and Science of the Russian Federation (the subsidy agreement #14.604.21.0056, Unique project identifier RFMEFI60414X0056).

used within proposed user identification approach. It is worth noting that nonnegative matrix factorization methods were not used before for the time series forecasting task. The proposed user identification approach has been experimentally verified on the example of real corporate email correspondence created from the Enron dataset. In addition, experiments with other today popular forecasting methods have shown the superiority of proposed forecasting method in quality of user's topic weights classification. Also we investigated two different approaches to estimates of the deviation of a time series point from the predicted value: absolute deviation and p-value estimation. Experiments have shown that both discussed approaches of deviation estimates are applicable in the proposed user identification approach.

Keywords: computer security; user identification; topic modeling; orthogonal nonnegative matrix factorization; time series forecasting.

DOI: 10.15514/ISPRAS-2015-27(5)-8

For citation: Tsarev D.V., Petrovskiy M.I., Mashechkin I.V., Korchagin A.Y., Korolev V.Y. Applying Time Series to The Task of Background User Identification Based on Their Text Data Analysis. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 1, 2015, pp. 173-192 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)-8.

References

[1]. R.V. Yampolskiy, V. Govindaraju, Behavioural biometrics: a survey and classification. International Journal of Biometrics (IJBM), Vol. 1, No. 1, 2008.

[2]. Vremennoi ryad [Time Series]. March 24 2015. (http://www.machinelearning.ru/wiki/index.php?title=Временной_ряд) (in Russian)

[3]. I.V. Mashechkin, M.I. Petrovskiy, D.V.Tsarev. Metody vychislenija relevantnosti fragmentov teksta na osnove tematicheskix modelej v zadache avtomaticheskogo annotirovanija [Methods of text fragment relevance estimation based on the topic model analysis in the text summarization problem]. Vychislitel'nye Metody i Programmirovanie [Numerical Methods and Programming], 2013, vol. 14, pp. 91–102. (in Russian).

[4]. I.V. Mashechkin, M.I. Petrovskiy, D.S. Popov, D.V. Tsarev. Automatic text summarization using latent semantic analysis. Programming and Computer Software, 2011, pp. 299-305.

[5]. D.V. Tsarev, M.I. Petrovskiy, I.V. Mashechkin. Using NMF-based text summarization to improve supervised and unsupervised classification. 11th International Conference on Hybrid Intelligent Systems (HIS), 2011. Malacca, MALAYSIA. P. 185-189.

[6]. D.V. Tsarev, M.I. Petrovskiy I.V. Mashechkin. Supervised and Unsupervised Text Classification via Generic Summarization. International Journal of Computer Information Systems and Industrial Management Applications. MIR Labs, Volume 5, 2013, pp. 509-515.

[7]. I.V. Mashechkin, M.I. Petrovskiy, D.S. Popov, D.V. Tsarev. Applying Text Mining Methods for Data Loss Prevention. Programming and Computer Software. January 2015, Volume 41, Issue 1, pp 23-30.

[8]. C.D. Manning, P. Raghavan, H. Schutze. Introduction to Information Retrieval. Cambridge University Press, 2008.

[9]. A. Mirzal. Converged Algorithms for Orthogonal Nonnegative Matrix Factorizations. CoRR abs/1010.5290, 2010.
[10]. Wei Xu, Xin Liu, Yihong Gong. Document clustering based on non-negative matrix factorization. Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval, Toronto, Canada, 2003.
[11]. Chris Ding, Tao Li, Wei Peng, Haesun Park. Orthogonal Nonnegative Matrix Tri-Factorizations for Clustering. SIGKDD, 2006.
[12]. M.W. Berry, M. Browne, A.N. Langville, V.P. Pauca, R.J. Plemmons. Algorithms and applications for approximate nonnegative matrix factorization. Computational Statistics and Data Analysis, pp. 155-173, 2007.
[13]. J. Yoo, S. Choi. Orthogonal Nonnegative Matrix Factorization: Multiplicative Updates on Stiefel Manifolds. Intelligent Data Engineering and Automated Learning – IDEAL 2008, vol. 5326 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 140–147.
[14]. C. Meek, D.M. Chickering, D. Heckerman. Autoregressive Tree Models for Time-Series Analysis, 2002. (http://go.microsoft.com/fwlink/?LinkId=45966)
[15]. Tekhnicheskii spravochnik po algoritmu vremennykh ryadov (Microsoft) [Microsoft Time Series Algorithm Technical Reference]. (http://msdn.microsoft.com/ru-ru/library/bb677216.aspx) (in Russian)
[16]. T. Hastie, R. Tibshirani, G. Sherlock, M. Eisen, P. Brown, D. Botstein. Imputing Missing Data for Gene Expression Arrays. Technical report, Stanford Statistics Department 1999.
[17]. O. Troyanskaya. Missing value estimation methods for DNA microarrays. Bioinformatics, , vol. 17, no. 6, 2001. pp. 520-525.
[18]. D.V. Tsarev, R.V. Kurynin, M.I. Petrovskiy, I.V. Mashechkin. Applying non-negative matrix factorization methods to discover user's resource access patterns for computer security tasks. Proceedings of the 2014 International Conference on Hybrid Intelligent Systems (HIS 2014). IEEE Computer Society [New York], United States, 2014. pp. 43–48.
[19]. D. Lee, S. Seung. Learning the parts of objects by non-negative matrix factorization. Nature, 401, 1999. pp. 788-791.
[20]. Enron Email Dataset. March 24 2015. (http://www.cs.cmu.edu/~./enron/)
[21]. Natural Language Toolkit (NLTK). March 24 2015. (http://www.nltk.org)
[22]. M. Kendall, A. Stuart. Statisticheskie vyvody i svyazi [Statistical derivations and associations.]. M.: Nauka, 1973 (In Russian).
[23]. Krivaya oshibok [Receiver Operating Characteristic, ROC curve]. March 24 2015. (http://www.machinelearning.ru/wiki/index.php?title=ROC-кривая) (In Russian)

Объектные модели ODMG и SQL десять лет спустя: нет противоречий

С. Д. Кузнецов <kuzloc@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25

Аннотация. В 2005 г. я написал статью, в которой приводил наиболее существенные черты стандартов ODMG 3.0 (объектная модель ODMG) и SQL:2003 (модель данных SQL) и убедительно (как мне тогда казалось) доказывал, что сходство между объектной моделью и объектными расширениями SQL является чисто внешним, что за близкими на вид синтаксическими конструкциями скрываются глубинные различия модельного уровня. Примерами таких различий являются фоннеймановское разыменование объектных идентификаторов в модели ODMG по сравнению с ассоциативным разыменованием ссылочных значений в модели SQL, раздельное и независимое хранение объектов одного объектного типа в модели ODMG по сравнению с хранением всех строк типизированной таблицы в одной этой таблице в модели SQL, хранение объектных идентификаторов в экстенте в модели ODMG и хранение в аналоге экстента самих объектов в модели SQL и т.д. С тех пор прошло много лет, за которые я понял многие вещи, неправильно или недостаточно правильно понимавшиеся мной тогда, и постепенно пришел к выводам, что:

1. различия, которые мне казались глубинными, таковыми не являются, да и вообще не являются различиями уровня модели;
2. объектные расширения SQL обеспечивают не меньшие (а скорее большие) возможности, чем объектная модель ODMG;
3. при разумном (с позиций сообщества баз данных) использовании СУБД, основанной на модели ODMG, будут создаваться базы данных и средства манипулирования ими, близкие к тем, которые предписывает модель данных SQL.

Ключевые слова: модели данных, объектная модель, ODMG 3.0, SQL, разыменование объектных ссылок, размещение объектов.

DOI: 10.15514/ISPRAS-2015-27(1)-9

Для цитирования: Кузнецов С.Д. Объектные модели ODMG и SQL десять лет спустя: нет противоречий. Труды ИСП РАН, том 27, вып. 1, 2015 г., стр. 173-192. DOI: 10.15514/ISPRAS-2015-27(1)-9.

1. Введение

В 2005 г. я написал статью [1], в которой приводил наиболее существенные черты стандартов ODMG 3.0 [2] (объектная модель ODMG) и SQL:2003 [3-4] (модель данных SQL) и убедительно (как мне тогда казалось) доказывал, что сходство между объектной моделью и объектными расширениями SQL является чисто внешним, что за близкими на вид синтаксическими конструкциями скрываются глубинные различия модельного уровня. Вот цитата из [1], которая достаточно точно характеризует мои тогдашние взгляды: «Единственное, что объединяет модели данных SQL и ODMG, – развитая система типов с возможностью определения новых типов с произвольно сложной структурой значений. “Объектные” же расширения SQL, по сути, представляют собой дальнейшую синтаксическую маскировку операций естественного соединения.»

С тех пор прошло 10 лет, за эти годы я понял многие вещи, неправильно или недостаточно правильно понимавшиеся мной тогда, и постепенно пришел к выводам, что:

1. различия, которые мне казались глубинными, таковыми не являются, да и вообще не являются различиями уровня модели;
2. объектные расширения SQL обеспечивают не меньшие (а скорее большие) возможности, чем объектная модель ODMG;
3. при разумном (с позиций сообщества баз данных) использовании СУБД, основанной на модели ODMG, будут создаваться базы данных и использоваться средства манипулирования ими, близкие к тем, которые предписывает модель данных SQL.

Данная статья имеет следующую структуру. Во втором разделе кратко (и критически) обсуждаются основные черты модели ODMG и «объектной части» модели SQL, требуемые для понимания сути статьи. В третьем разделе рассматриваются два различия моделей ODMG и SQL, которые полагались фундаментальными в [1], и с новых позиций обосновывается реальная незначительность этих различий. В четвертом разделе демонстрируется, что «объектная часть» модели данных SQL содержит не меньше (а реально больше) «объектных» возможностей, чем модель ODMG, а при разумном использовании СУБД, основанных на модели ODMG, будут применяться методы и средства, близкие к тем, которые предписываются моделью SQL. В последнем разделе статьи приводятся заключительные замечания.

2. Основные черты объектной модели ODMG 3.0 и «объектной части» модели SQL

Много раз и по разным поводам мне приходилось писать о моделях данных ODMG и SQL. Наиболее кратко об этом говорилось в [1], наиболее полно – в [4], умеренно подробно – в [5]. В принципе, здесь можно было бы ограничиться ссылками на эти прошлые публикации, но мне кажется, что без

краткого критического повторения основных черт этих объектных моделей было бы трудно уловить основной смысл статьи. Естественно, далее предполагается, что читатели имеют общее представление о стандартах ODMG 3.0 и SQL:2003.

Общей характеристикой моделей данных ODMG и SQL является то, что обе они являются полнотиповыми. Говоря неформально, это означает, что в обеих моделях поддерживаются практически неограниченные возможности определения новых типов данных, и при определении любого нового типа данных можно использовать любой ранее определенный тип.

2.1 ODMG 3.0

В объектной модели ODMG имеются две категории типов: литеральные и объектные типы. Понятие литерального типа данных совпадает с традиционным понятием типа: тройка {множество значений, набор операций, множество литералов}. В состав литеральных типов входят атомарные и конструируемые типы. Атомарные литеральные типы включают традиционный набор типов целых чисел и чисел с плавающей точкой, типы символьных строк и булевский тип. Булевский тип в модели ODMG обычный, двузначный, поскольку в этой модели в явном виде неопределенные значения отсутствуют (неявно в разных местах стандарта они упоминаются, и это можно отнести к числу небрежностей, которых в стандарте предостаточно).

К конструируемым литеральным относятся структурные литеральные типы и типы коллекций. Структурные литеральные типы – это прямые аналоги типов записи в языках C/C++ (именованное множество именованных типизированных атрибутов). При определении структурных литеральных типов невозможно использовать наследование и определять операции. Несколько сбивает с толку то, что типом атрибута структурного литерального типа может являться не только любой ранее определенный литеральный тип, но и любой ранее определенный объектный тип. В последнем случае (имея в виду, что элементами объектного типа являются объекты) можно представить, что «значениями» такого атрибута являются объекты, что невозможно, поскольку объекты являются не значениями, а контейнерами, содержащими значения. Конечно, такое представление противоречит здравому смыслу и недопустимо. Но с этой (с моей точки зрения) небрежностью стандарта мы разберемся немного позже.

Наконец, конструируемые литеральные типы коллекций включают типы множеств, мультимножеств, списков и словарей (dictionary – множества пар <ключ, значение>, причем все ключи в этих парах должны быть различными). Типом элемента коллекции может быть любой ранее определенный литеральный или объектный тип. После многолетнего общения с моделью SQL типы мультимножеств кажутся мне подозрительными и не слишком нужными, упорядоченных типов данных я опасаюсь на основе многолетнего общения с XQuery [7] в контексте XML-ориентированных СУБД; наконец,

мне неизвестно, каким образом можно здраво использовать тип справочников в среде полноценной базы данных. Поэтому разумным (и действительно нужным) литеральным типом коллекций в модели ODMG считаю тип множества.

Объектные типы данных делятся на атомарные типы (определяемые пользователями объектные типы, user-defined types, UDT) и типы коллекций. Наиболее существенными в контексте объектно-ориентированных баз данных являются UDT, хотя, по моему мнению, их невозможно здравым образом использовать, не используя также и объектные типы коллекций.

Объектный тип служат для создания объектов с внутренней структурой и набором операций, специфицированными в определении данного объектного типа. В отличие от традиционных типов данных у любого объектного типа имеется операция создания и инициализации объекта, выделяющая память для нового объекта, присваивающая его атрибутам начальные значения и возвращающая уникальный идентификатор объекта (Object Identifier, OID), который в дальнейшем можно разыменовывать для доступа к атрибутам объекта и вызова его операций. Очевидно, что в соответствии со своими свойствами OID является *ссылкой на объект* (или абстрактным *адресом* объекта).

Мне кажется столь же очевидным, что понятие *объекта* в модели ODMG (да и в большинстве объектно-ориентированных языков программирования) практически (с точностью до различия в терминологии) не отличается от понятия *переменной* в более традиционных моделях данных и языках программирования. Действительно, переменная является типизированным контейнером, который может содержать значения соответствующего типа данных. Динамические переменные создаются операцией new, которая выделяет память для новой переменной и возвращает типизированную ссылку на соответствующий контейнер. Типом этой ссылки является *ссылочный тип*, парный типу значений, которые может содержать переменная. Так, например, в языках C/C++ парным ссылочным типом для типа integer является тип *integer.

Видимо, основным источником недоразумений, связанных с использованием терминов *объект, атрибут объектного типа* и т.д. является отсутствие в модели ODMG понятия ссылочного типа. В определении объектного типа под одним именем склеены определения типа значений объектов и ссылочного типа для соответствующих объектов (можно сказать, типа OID'ов объектов данного объектного типа). Везде, где объектный тип используется для типизации атрибутов, элементов коллекций и т.д., в действительности имеется в виду ссылочный тип, значениями которого являются OID.

Основными компонентами определения UDT являются определения атрибутов, связей и операций. В этой статье несущественны особенности средств определения операций в обсуждаемых моделях данных, поэтому этот аспект обсуждаться не будет.

Раздел определения атрибутов не отличается от соответствующего раздела определения структурного литерального типа: атрибуты именуются и типизируются, причем типом атрибута может являться любой ранее определенный литеральный или объектный тип. Разыменовывая значение атрибута объектного типа (OID), можно «перейти» к соответствующему объекту, т.е. получить доступ к его атрибутам и операциям. Если определить атрибут литерального типа множества с элементами некоторого объектного типа (типа множества OID'ов заданного объектного типа), то, перебирая элементы этого множества, можно последовательно «получить доступ» ко всем соответствующим объектам. Однако такие «ассоциации» являются однонаправленными: в объекте, на который указывают подобные ссылки, отсутствует какая-либо информация об их наличии.

Для организации двунаправленных «ассоциаций» используется механизм связей. Бинарные связи (а других в модели ODMG и нет) определяются сразу для двух объектных типов: в каждом из них задается имя связи, объектный тип, к которому она ведет, и имя парной связи в этом объектном типе. Такой способ определения двунаправленных бинарных ассоциаций в модели ODMG приходится применять по той причине, что OID представляет собой аналог абстрактного «фоннеймановского» адреса, при разыменовании OID происходит простой «переход по адресу». В этом состоит отличие от реляционной модели данных и модели данных SQL, где для организации связей видов 1:1 и 1:n используется механизм внешних ключей, для разыменования «ссылки» используется коммутативная операция соединения, и «связи» автоматически получаются двунаправленными. Это различие между моделями ODMG и SQL в [1] ошибочно казалось мне фундаментальным, и в третьем разделе статьи я постараюсь развеять это заблуждение.

Чтобы завершить обсуждение особенностей атомарного объектного типа, требуется сначала немного поговорить об объектных типах коллекций. Такие типы, как и литеральные типы коллекций, определяются с использованием конструктора типа, которому сообщается вид коллекции (множество, мультимножество, список или справочник) и тип элементов коллекции (любой ранее определенный литеральный или объектный тип). Объекты типа коллекции, как и объекты атомарных объектных типов, имеют OID и представляют собой контейнеры, хранящие соответствующие литеральные коллекции. Набор операций объектных типов коллекций предопределен и содержит, в частности, операцию, позволяющую перебирать элементы соответствующей коллекции.

По моему мнению, как и для литеральных типов коллекций, практическую ценность в контексте объектных баз данных имеют только объектные типы множеств. Более того, как я показываю в четвертом разделе статьи, наибольшей значимостью обладает одно «самоприменение» объектных типов множества внутри модели ODMG. Речь идет о понятии *экстента* атомарного объектного типа.

При определении атомарного объектного типа, помимо других свойств, несущественных для целей этой статьи, можно указать, что для этого объектного типа требуется автоматически поддерживать экстент – объект типа коллекции (по умолчанию, множества), типом элементов которого является определяемый атомарный объектный тип. Сам этот тип коллекции определяется неявным образом.

Как говорится в стандарте ODMG 3.0, экстент атомарного объектного типа в любой момент времени содержит все существующие объекты этого типа (как говорилось выше, имеются в виду OID-ы объектов). Наличие такого регулярного хранилища однотипных объектов позволяет использовать ненавигационные средства языка запросов OQL, определенного в модели данных ODMG.

При определении атомарных объектных типов можно использовать механизм наследования. В этой статье для нас несущественны особенности этого механизма, главное, что он поддерживает традиционную *семантику включения*: объекты, относящиеся к любому подтипу некоторого супертипа, являются и объектами этого супертипа. Семантика включения естественным образом поддерживается и для экстентов: значение экстента любого подтипа некоторого супертипа является подмножеством значения экстента этого супертипа.

Экстент атомарного объектного типа получает имя, совпадающее с именем этого типа. Экстенты являются единственным видом объектов, которые получают имена автоматически, без потребности дополнительных взаимодействий с системой.

Конечно, наличие в модели ODMG возможности явного конструирования объектных типов коллекций позволяет образовывать произвольное число регулярных хранилищ объектов (OID-ов) для одного и того же атомарного объектного типа. Но у экстента имеется ряд преимуществ: простота определения, наличие неявно присвоенного имени, автоматическое отслеживание содержимого (добавление OID при создании новых объектов и удаление OID при уничтожении объектов).

Можно заметить, что экстент атомарного объектного типа в модели ODMG напоминает таблицу в модели SQL. В [1] утверждалось, что при внешнем сходстве у этих двух понятий имеется принципиальное различие: в экстенте размещаются не сами объекты, а их OID-ы, а таблица является в некотором смысле контейнером, хранящем именно сами строки таблицы. В третьем разделе я покажу, что и это различие принципиальным не является.

2.2 Объектная часть модели SQL

В [1] при обсуждении модели данных SQL я ссылался на стандарты SQL:1999 и SQL:2003, хотя следовало сослаться еще и на стандарт SQL:1992 [8], в котором были заложены основы традиционной части модели SQL. За прошедшие годы были выпущены два новых стандарта: SQL:2008 и SQL:2011

[9]. В этих стандартах имеется ряд новшеств, влияющих на представление модели данных SQL в целом (в частности, темпоральные возможности [10]). Но на объектную часть модели SQL эти новшества не влияют, и я продолжаю опираться и в данной статье на стандарт SQL:1999 и его расширения в стандарте SQL:2003.

Как и в модели ODMG, фундаментом объектной части модели SQL является система типов. В модели SQL используется только традиционное понятие типа данных, никаких аналогов объектных типов модели ODMG в модели SQL нет. Имеются следующие основные категории типов:

1. точные числовые типы;
2. приближенные числовые типы;
3. типы символьных строк;
4. типы битовых строк;
5. типы даты и времени;
6. типы временных интервалов;
7. булевский тип;
8. типы коллекций;
9. анонимные строчные типы;
10. типы, определяемые пользователем (User Defined Type, UDT);
11. ссылочные типы.

Особенности первых шести категорий типов для этой статьи несущественны. Булевский тип в SQL трехзначный, включающий значения *true*, *false* и *unknown*, причем в контексте булевского типа значение *unknown* считается тем же самым, что и неопределенное «значение» NULL (это одно из самых неудачных мест стандарта SQL, но и оно для данной статьи несущественно). Коллекции в модели SQL могут быть массивами, множествами и мультимножествами, причем типом элемента коллекции может быть любой ранее определенный тип данных. Анонимные строчные типы (безымянные типы записи), главным образом, нужны для обеспечения возможности определять вложенные таблицы (таблицы, у которых хотя бы один столбец содержит значения-таблицы). Действительно существенными для объектной части модели данных SQL являются UDT – типы данных, определяемые пользователями (да и то не все) и ссылочные типы.

В категории UDT для объектной части модели SQL не существенны индивидуальные (distinct) типы, которые позволяют контролируемым образом учитывать в базах данных SQL семантику данных, иным образом представляемых с использованием базовых типов данных. Зато структурные UDT играют в объектной части модели решающую роль.

Внешне определение структурного UDT в модели SQL напоминает определение атомарного объектного типа в модели ODMG: набор типизированных атрибутов (типом атрибута может быть любой ранее определенный тип данных) и набор операций. При определении структурного UDT можно использовать достаточно простой механизм одиночного

наследования. Но, в отличие от атомарного объектного типа, структурный UDT в SQL является типом данных в традиционном смысле (т.е. множеством структурных значений) и поэтому ничего похожего на спецификацию связей в определении структурного UDT не содержится.

После определения структурного UDT его можно использовать (как и любой определенный тип данных) для определения новых типов данных (типов коллекций, анонимных строчных типов или UDT) или типизации столбцов таблиц. В последнем качестве структурные UDT объектную часть модели SQL не затрагивают. Важнейшими компонентами, вместе с которыми структурные UDT образуют объектную модель, являются типизированные таблицы и ссылочные типы.

Про ссылочные типы удобнее говорить после введения понятия типизированной таблицы. Формально типизированная таблица – это таблица, создаваемая с использованием конструкции CREATE table_name OF type_name, где type_name – имя ранее определенного структурного UDT. После выполнения такой операции в базе данных создается базовая таблица с именем table_name, которая при традиционной (не объектной) трактовке содержит n+1 столбец, имена и типы данных последних n которых совпадают с именами и типами атрибутов структурного UDT type_name. Если типизированная таблица используется исключительно в традиционной трактовке, содержимое первого столбца не существенно.

При объектной трактовке типизированной таблицы она представляется как контейнер объектов (отдельному объекту соответствует строка таблицы). Первый столбец типизированной таблицы в этом случае содержит так называемые ссылочные значения, уникальные типизированные значения, типом которых является ссылочный тип, парный структурному UDT, на котором определена типизированная таблица.

В модели SQL ссылочный тип может быть определен (вернее, объявлен, поскольку практически никакие специальные определения не требуются) для любого структурного UDT. При объявлении ссылочного типа требуется указать один из трех возможных способов генерации ссылочных значений: генерируется системой, задается пользователем или является комбинацией значений других атрибутов структурного UDT. Поскольку от ссылочного значения требуется уникальность (вообще говоря, полная уникальность; т.е. любое ссылочное значение одного ссылочного типа должно отличаться от любого другого ссылочного значения), реально работает только первый вариант – ссылочные значения, генерируемые системой.

Заметим, что с позиций реляционной модели данных ссылочные значения, генерируемые системой, больше всего похожи на суррогатные возможные ключи, которые, по всей видимости, впервые были предложены в [12], а более внятно были описаны в [11]. Интересные рассуждения относительно логических отличий суррогатных ключей в реляционной модели от объектных идентификаторов в объектных моделях приводятся в [13]. У теоретиков

реляционной модели отношение к суррогатным ключам варьируется от умеренно положительного до резко отрицательного (более правильным считается использование возможных ключей, отражающих семантику предметной области). Но если суррогатные ключи в таблицах базы данных используются, то основное их назначение состоит в обеспечении связок «возможный_ключ-внешний_ключ» для выполнения операций естественного соединения.

Ссылочное значение в объектной части модели SQL является аналогом OID в модели ODMG. В предыдущем подразделе уже отмечалось, что отсутствие понятия ссылочного типа является явным недостатком модели ODMG, приводящим к путанице при описании модели и затруднениям при ее понимании. В модели SQL все гораздо понятнее. Ссылочное значение (ссылка на объект, или уникальный идентификатор объекта) образуется при создании нового объекта (вставке новой строки в типизированную таблицу), сохраняется в нулевом столбце соответствующей строки (в стандарте SQL этот столбец называется «самоссылающимся» – self-referencing) и может сохраняться, в частности, в любом столбце любой таблицы (в частности, типизированной), если этот столбец определен на том же ссылочном типе.

Впоследствии при выполнении, например, запроса к базе данных может быть выполнена операция разыменования ссылочного значения и обеспечен доступ к атрибутам и операциям объекта. Как говорилось в предыдущем подразделе, в [1] меня сильно смущала природа этой операции «разыменования», которая из-за реляционного происхождения модели SQL является не фоннеймановской, а ассоциативной, основанной на использовании операции естественного соединения. В третьем разделе я поясню, почему считаю заблуждением свою тогдашнюю точку зрения, что специфика разыменования в модели SQL противоречит принципам объектно-ориентированной модели данных.

Наконец, поговорим еще немного о типизированных таблицах. В модели SQL можно представлять, что любая типизированная таблица является контейнером объектов, поскольку с точки зрения пользователя новый объект создается (и инициализируется) путем вставки в типизированную таблицу новой строки. На основе одного структурного UDT можно определить несколько типизированных таблиц, и все они будут контейнерами, сохраняющими однотипные объекты.

Если типизированная таблица A определена на структурном UDT T, и структурный UDT T_1 является непосредственным подтипом типа T, то на типе T_1 можно определить типизированную таблицу A_1, являющуюся подтаблицей таблицы A. Для супертаблиц и подтаблиц поддерживается семантика включения: любой объект, созданный в таблице A_1, считается входящим и в таблицу A.

Заметим, что понятие типизированной таблицы в объектной части модели SQL напоминает понятие экстента в модели ODMG. При поверхностном

рассмотрении и типизированная таблица, и экстент являются контейнерами однотипных объектов, к которым можно адресовать декларативные запросы на выборку объектов. Однако при более внимательном взгляде между типизированными таблицами и экстентами заметно различие, которое в [1] мне также казалось фундаментальным.

Основным различием казалось то, что в модели ODMG экстент объектного типа сохраняет не сами объекты, а их OID'ы. Операция создания объекта (т.е. в основном выделения памяти под объект) – это операция объектного типа, а не экстента. Природа экстента является такой же, как у любой другой «коллекции объектов», вручную поддерживаемой пользователем объектной базы данных: все такие коллекции содержат OID'ы объектов, а сами объекты располагаются независимо один от другого где-то в «куче» базы данных.

В модели SQL операция создания объекта является операцией типизированной таблицы (правильнее сказать, над типизированной таблицей). Объект создается путем выполнения операции вставки в таблицу новой строки, и естественно представлять себе, что эта строка является частью таблицы. Кстати, здесь полезен некоторый дуализм в понимании того, что такое таблица в модели данных SQL. В теоретически правильном понимании (следуя, в частности, представлениям Дейта и Дарвена в Третьем манифесте, см., например, [6]) таблица, это переменная базы данных, тип которой определяется заголовком таблицы, а значениями являются мультимножества строк, соответствующих этому заголовку. При таком понимании, конечно, некорректно считать, что у отдельной строки таблицы имеется какое-то собственное пространство хранения.

Однако возможно и дуальное понимание таблицы как типизированного контейнера (вообще говоря, изменяемого размера), в котором распределяется память для хранения однотипных строк. Можно утверждать, что эти два понимания не противоречат одно другому, и при втором понимании у каждой строки, входящей в таблицу имеется свое собственное пространство хранения, прямой адрес которого пользователям не сообщается (в следующем разделе мы увидим, что и это не всегда так).

Как и в модели ODMG, в модели SQL можно сохранять в базе данных дополнительные коллекции ссылочных значений существующих однотипных объектов, однако такие коллекции принципиально (как мне казалось в [1]) отличаются от коллекции объектов, сохраняемой в типизированной таблице: типизированная таблица хранит сами объекты плюс их ссылочные значения, а упомянутые дополнительные коллекции объектов – только ссылочные значения.

Перейдем теперь к критике взглядов автора десятилетней давности.

3. Действительно ли существенны различия моделей ODMG и SQL?

После публикации статьи [1] прошло много лет. За эти годы я много раз рассказывал студентам об особенностях моделей ODMG и SQL, много размышлял об этом и постепенно изменял свое мнение о существенности различий между этими моделями.

3.1 Разные способы адресации объектов

В модели ODMG объекты адресуются своими OID'ами. OID – это абстрактный аналог однонаправленного фоннеймановского адреса. OID хранится только на стороне ссылающейся сущности. При выполнении операции создания нового объекта его OID определяется системой, является обратным параметром операции создания и, вообще говоря, не сохраняется внутри создаваемого объекта. Из-за однонаправленности объектной ссылки в модели ODMG требуется дополнительное понятие связи, представляемой контролируемой парой однонаправленных ссылок.

В объектной части модели SQL объекты адресуются ссылочными значениями, которые с позиций реляционной модели данных являются суррогатными ключами. Ссылочное значение объекта, вообще говоря, хранится внутри объекта и на стороне любой ссылающейся на него сущности. Операция разыменования ссылочного значения (доступа по ссылке) производится на основе коммутативной реляционной операции естественного соединения, которая по своей сути является ассоциативной. В частности, по этой причине в модели SQL не требуется дополнительное понятие связи (все ссылки являются двунаправленными).

Действительно ли эти различия являются фундаментальными различиями моделей ODMG и SQL? Первое и основное соображение, которое заставило меня в этом усомниться, основано на следующем наблюдении.

Принято считать, что подавляющее большинство программируемых компьютеров, начиная от времени их появления в середине 1940-х гг., и до наших дней, основано на архитектуре фон Неймана. В соответствии с принципами фон Неймана, программа и обрабатываемые ей данные хранятся в однородной прямоадресуемой памяти. Программы выполняются последовательно команда за командой, пока не встретится команда перехода, т.е. насильственного изменения счетчика команд. Команды могут вырабатывать прямые адреса памяти, из которых выбираются и в которые записываются данные.

Однако при этом никого не смущает, что в этих компьютерах с 1970-х гг. между аппаратурой управления виртуальной памятью и аппаратурой управления основной памятью присутствует поддерживаемый аппаратурой дополнительный уровень сверхбыстродействующей ассоциативной памяти – кэш. Если не стремиться к полной строгости, можно считать, что кэш – это

буферная память, состоящая из блоков одного и того же небольшого размера (часто 64 байта). При каждой выборке команд или данных из основной памяти соответствующий блок памяти буферизуется в одном из блоков кэша, причем в этом же блоке запоминается виртуальный адрес, обращение по которому привело к этой выборке.

При выработке некоторой командой виртуального адреса для чтения команд или данных аппаратура управления виртуальной памятью до преобразования виртуального адреса в физический обращается к кэшу, в котором происходит параллельный ассоциативный поиск этого виртуального адреса во всех блоках. При удачном завершении поиска нужные команды или данные передаются в процессор, а при неудачном завершении продолжается работа аппаратуры управления виртуальной памятью.

Запись по виртуальному адресу работает аналогично, но после размещения блока данных в кэше нужные данные обязательно проталкиваются и в основную память. Естественно, аппаратура управления кэш-памятью производит в случае надобности замещение блоков кэша.

Так вот, операция «разыменования» виртуального адреса с использованием кэша больше всего напоминает операцию естественного соединения из реляционной модели данных: коммутативную операцию, выполняемую в параллельной и ассоциативной манере. Никого не смущает, что фоннеймановский виртуальный адрес сначала «разыменовывается» в компьютерах совершенно не фоннеймановским образом. По всей видимости, различие к технике разыменования ссылок на объекты не должно быть серьезным модельным различием и при сравнении моделей ODMG и SQL.

Стоит сделать еще одно замечание, показывающее, насколько не обязательно особенности моделей данных прямо влияют на реализации. Десять лет тому назад, когда писалась статья [1], мне казалось очевидным, что единственным разумным способом выполнения операции разыменования ссылочных значений в объектной части модели SQL является применение естественного соединения. Действительно, в первом столбце типизированной таблицы хранятся суррогатные значения возможного ключа, а в столбцах таблиц, ссылающихся на эту типизированную таблицу, сохраняются соответствующие значения внешнего ключа. Чтобы перейти от строки ссылающейся таблицы к объекту типизированной таблицы нужно выполнить, строго говоря, естественное полусоединение этих таблиц.

Всем известно, что любая операция соединения является сравнительно дорогостоящей. Поэтому я был удивлен, когда вскоре после написания [1] в одной из статей, посвященной некоторым деталям реализации объектных расширений в Oracle, прочитал, что в этой системе запросы, написанные на «объектном диалекте» SQL с применением синтаксических конструкций навигации по объектным ссылкам, выполняются эффективнее, чем эквивалентные запросы с соединениями. Я тогда действительно считал, что этот «объектный диалект» SQL является тем, что принято называть

«синтаксическим сахаром»: подслащиванием SQL в угоду любителям объектно-ориентированных конструкций.

Ан был неправ, что отметил уже в 2007 г. в [14]. Всем известно, что, начиная с System R [15], практически во всех SQL-ориентированных СУБД с построчным хранением таблиц для каждой строки таблицы на физическом (обычно скрытом от пользователя) уровне поддерживается уникальный идентификатор (в System R он назывался tid – tuple identifier), который обеспечивает «почти прямой» доступ к строке во внешней памяти. В Oracle аналог такого идентификатора называется rowid и, хорошо ли это или плохо, он открыт для пользователей баз данных. Можно объявить любую базовую (реально хранимую в базе данных) таблицу как содержащую столбец rowid (реально этот столбец в базе данных не хранится, поскольку система и так знает значение rowid для любой строки любой таблицы). Эти значения можно читать, сохранять (явно) в столбцах других таблиц и использовать в запросах для прямого (совсем не реляционного) доступа к строкам.

Тот же фокус использовала Oracle при организации доступа к объектам, сохраняемым в «объектных таблицах» (аналог типизированной таблицы в модели SQL). В Oracle ссылочные значения, генерируемые системой, в действительности являются rowid. Поэтому самоссылающийся столбец явным образом в объектной таблице не хранится, но его значения известны системе и доступны пользователям в качестве ссылочных значений объектов.

С одной стороны, Oracle не нарушает предписаний объектной части модели SQL. Если не присматриваться к деталям представления ссылочных значений, то можно представлять, что операция их разыменования основана на естественном соединении. Но в действительности ссылка трактуется как прямой однонаправленный адрес.

С моей точки зрения, приведенные замечания и наблюдения убеждают в том, что различия в способе разыменования ссылок на объекты не являются принципиальными различиями моделей ODMG и SQL.

3.2 Разные подходы к размещению объектов в памяти базы данных

В модели ODMG операция создания и инициализации объекта является операцией соответствующего атомарного объектного типа. Если при определении объектного типа не специфицировано требование наличия экстента этого типа, то любой заново созданный объект существует изолированно от других объектов данного типа, если не будет выполнена явная операция помещения OID созданного объекта в какой-либо контейнер (объект типа коллекции, скорее всего, множества). Если экстент в определении атомарного объектного типа специфицирован, то при создании любого нового объекта данного типа его OID автоматически помещается в этот экстент (объект типа коллекции, скорее всего, множества).

С моей точки зрения, база данных, в которой поддерживаются изолированные объекты, не слишком осмысленна, поскольку, в частности, к ней невозможно адресовать декларативные запросы, поддержка коллекций OID вручную обременительна и не очень полезна, и поэтому наиболее естественным образом объектная база данных в модели ODMG представляется в виде набора экстентов, именованных объектов типа множества, элементами которого являются OID'ы атомарного объектного типа (более подробно и конкретно остановимся на этом вопросе в четвертом разделе статьи).

В объектной части модели SQL операция создания и инициализации нового объекта является операцией типизированной таблицы (над типизированной таблицей) – вставка новой строки. Поскольку для типизированных таблиц в модели SQL допускаются как объектное представление (типизированная таблица как контейнер объектов), так и традиционное (унаследованное от реляционной модели данных) представление (типизированная таблица как переменная со значениями-множествами строк), на уровне модели данных приходится считать, что объекты, созданные в данной типизированной таблице, в ней и располагаются.

Изолированных объектов в модели SQL не бывает, любой существующий объект входит к какую-то типизированную таблицу. Как и в модели ODMG, никто не мешает создавать дополнительные объекты, сохраняющие коллекции ссылочных значений заданного типа, вручную контролировать их содержимое и производить в них поиск (как и в случае ODMG, я не уверен, что эта потенциальная возможность полезна). Тем самым, в модели SQL объектная база наиболее естественным образом представляется в виде набора типизированных таблиц, именованных объектов типа множества, элементами которого являются объекты, структура и поведение которых соответствует структурному UDT типизированной таблицы.

Так что в этом отношении модели ODMG и SQL различаются только тем, что в экстентах хранятся OID'ы объектов, а в типизированных таблицах – сами объекты. В [1] я считал и это различие фундаментальным. Теперь мне так не кажется.

Во-первых, в стандарте ODMG недаром почти всегда говорится «объект» в тех случаях, когда имеется в виду OID: объекты как значения атрибутов объектных типов, объекты как элементы коллекций и т.д. С одной стороны, как отмечалось выше, это вносит путаницу, поскольку понятно, что один объект не может быть значением, например, нескольких атрибутов. Но с другой стороны, это делается умышленно для простоты и удобства пользователей объектных баз данных, которым совершенно ни к чему думать о вспомогательных операциях разыменования.

Во-вторых, хотя в модели SQL действительно разный статус имеют типизированная таблица (реальный контейнер объектов) и любая коллекция ссылочных значений (контейнер ссылок на объект), на практике более

естественно использовать именно типизированные таблицы, и в этом случае отсутствие операции разыменования для пользователей может пройти так же незаметно, как наличие этой операции при использовании экстентов в модели ODMG.

Наконец, таблица в модели SQL – это понятие более высокого логического уровня, чем экстент в модели ODMG. На физическом уровне строки таблиц хранятся в блоках внешней памяти независимо одна от другой. В таблицу они связываются благодаря наличию идентификатора таблицы во всех ее хранимых строках или за счет использования индексов. Так что в действительности на уровне хранения данных типизированная таблица вовсе не является контейнером объектов.

В целом, материал этого раздела демонстрирует, что фундаментальных различий между моделью ODMG и объектной частью модели SQL нет.

4. «Объектная мощь» SQL и «реляционная натура» ODMG

Сравнивать модели данных, описанные совершенно в разных стилях и с использованием разной (и не очень строгой) терминологии, дело трудное и неблагодарное. Однако, как мне кажется, в целом мне удалось сопоставить объектную модель ODMG и объектную часть модели SQL. Это сопоставление позволило мне прийти к трем основным выводам, первый из которых содержится в конце предыдущего раздела.

Второй вывод состоит в том, что **«объектные» возможности модели SQL не меньше тех, которые предоставляются моделью ODMG**. Действительно, системы типов моделей ODMG и SQL сопоставимы. В SQL система типов богаче и строже (более адекватный булевский тип, имеются явные ссылочные типы, отсутствуют смущающие ум объектные типы и т.д.), но по сути обеспечиваются те же возможности, что в ODMG. Наследование и семантика включения поддерживаются почти на равных в обеих моделях. Инкапсуляция объектов в обеих моделях фактически отсутствует (для сокрытия отсутствия инкапсуляции состояния объектов используются старые фокусы с неявной поддержкой автоматически определяемых методов *observer* и *mutator* для всех атрибутов объектов [12]). Язык запросов OQL из модели данных ODMG немногим отличается (не только синтаксически, но и по своей семантике) от «объектного диалекта» языка запросов SQL.

Проблема SQL (и в том числе объектной части этой модели данных) состоит не в том, что в ней чего-то не хватает, а в громадном переизбытке возможностей. В частности, объектные средства SQL во многом трудно оценить по той причине, что их можно использовать практически в произвольной комбинации с бесчисленными «традиционными» возможностями. Например, с использованием одного и того же структурного UDT с *n* атрибутами можно определить «объектную» типизированную таблицу с *n+1* атрибутом, но допускается и определение «традиционной»

таблицы с одним столбцом, типом которого является данный структурный UDT.

Наконец, третий, заключительный вывод формулируется следующим образом: **при разумном использовании любой СУБД, основанной на модели ODMG, будут использоваться базы данных и средства манипулирования ими, близкие к тем, которые предписываются моделью данных SQL**.

На самом деле, язык OQL, описанный в [2] содержит набор средств, которые обеспечивают возможности явной навигации по графам объектов, связанных объектными ссылками, а также возможности, близкие к SQL, для формирования декларативных запросов над коллекциями OID. С точностью до терминологии и синтаксических обозначений можно считать, что OQL входит в современный SQL (начиная с SQL:2003 [4]).

Чтобы иметь возможность разумно и эффективно пользоваться этими возможностями, наиболее естественно представлять базу данных в виде набора экстентов объектных типов, определенных в схеме базы данных. Такое представление базы данных является естественным, поскольку средства автоматической поддержки экстентов предопределены в модели ODMG, и экстенты являются единственными сущностями объектной базы данных, которые автоматически получают имена (совпадающие с именами соответствующих атомарных объектных типов). Поддержка дополнительных коллекций OID'ов кажется мне обременительной, ненадежной и избыточной (помимо необходимости явно отслеживать состав таких коллекций, им нужно еще и нетривиальным образом присваивать имена).

Легко видеть, что в этом случае схема объектной базы данных становится очень похожей на схему SQL-ориентированной базы данных, построенной в объектном стиле. Атомарные объектные типы (если не обращать внимания на их странности) напоминают структурные UDT SQL, а экстент – это почти то же, что и типизированная таблица (в конце концов, что представляет собой внутренняя структура объекта в модели ODMG как не строку в смысле SQL). Эти соображения в целом обосновывают приведенный выше третий вывод.

Но можно сказать больше. В SQL-ориентированных базах данных никто не запрещает обращаться с типизированными таблицами как с таблицами традиционными: выполнять проецирование, соединения таких таблиц и т.д. Конечно, результирующие таблицы типизированными уже не будут, но в мире SQL это не должно никого смущать, поскольку в одной базе данных могут содержаться и типизированные, и традиционные таблицы. Поскольку экстент базы данных ODMG является аналогом типизированной таблицы SQL, то в принципе было бы достаточно логичным допустить над ними операции, доступные для традиционных таблиц SQL.

И здесь мы интересным образом отбрасываемся на двадцать лет в прошлое, когда я пытался придумать аналог реляционной алгебры [16]. Не буду здесь воспроизводить суть этой старой статьи, которая неожиданно снова стала актуальной. Но хочу заметить следующее. Модель данных SQL эклектична и

из-за этого трудно постижима, но зато она не ограничивает пользователей по чисто идеологическим причинам, для которых отсутствуют технические подтверждения. Кто знает, не стоит ли (или не стоило ли?) временами жертвовать идеологическими принципами?

5. Заключение

Основная цель этой статьи состоит в том, чтобы исправить ошибки, сделанные автором в публикации 10-летней давности. Думаю, что здесь речь идет не только об установлении более справедливого отношения к моделям ODMG и SQL. Объектно-ориентированные СУБД снова начинают пользоваться спросом, единственной общепринятой объектной моделью является ODMG 3.0, и эта статья показывает, что (а) эта модель не так уж плоха, как может показаться при чтении стандарта [2], и (б) объектная часть модели SQL фактически ничем не отличается от ODMG. Возможно, эти заключения принесут пользу разработчикам и пользователям баз данных.

Десять лет тому назад я выступил на семинаре Московской секции ACM SIGMOD с докладом [17], по мотивам которого была написана статья [1]. Доклад был с пониманием встречен аудиторией, которая в целом согласилась с моими тогдашними доводами. В конце 2014 г. я выступил на том же семинаре с докладом [18], по мотивам которого написана эта статья. И этот доклад тоже был встречен аудиторией вполне лояльно, и с моими теперешними доводами никто всерьез не спорил. Поэтому в конце доклада я пообещал собравшимся, что если в мире моделей данных не произойдет что-либо непредвиденное, то следующий раз я выступлю с докладом на близкую тему не раньше, чем через 10 лет. Думаю, что то же следует сказать и насчет статей по этой тематике.

Список литературы

[1]. Сергей Кузнецов. "Объектны" ли объектные расширения языка SQL?, 2005. http://citforum.ru/database/articles/sql_odmg/

[2]. The Object Data Standard: ODMG 3.0. Edited by R.G.G. Cattel, Douglas K. Barry. Morgan Kauffmann Publishers, 2000.

[3]. Jim Melton. "Advanced SQL:1999. Understanding Object-Relational and Other Advanced Features". Morgan Kaufmann Publishers, 2003.

[4]. Сергей Кузнецов. Наиболее интересные новшества в стандарте SQL:2003, 2004. http://citforum.ru/database/sql/sql2003/

[5]. С.Д. Кузнецов. Базы данных: языки и модели. Москва, Бином, 2008.

[6]. С.Д. Кузнецов. Три манифеста баз данных: ретроспектива и перспективы. Базы данных и информационные технологии XXI века. Материалы международной научной конференции. Москва, 29-30 сентября 2003 г. Москва, РГГУ, 2004, стр. 52-229. http://citforum.ru/database/articles/manifests/

[7]. Дон Чемберлин. XQuery: язык запросов XML. Открытые системы, № 1, 2003, стр. 61-72. Оригинал: D. Chamberlin. XQuery: An XML query language. IBM SYSTEMS JOURNAL, VOL 41, NO 4, 2002, pp. 597-615.

[8]. C.J. Date. A Guide to SQL Standard (4th Edition). Addison-Wesley Professional, 1996)
[9]. Fred Zemke. What's new in SQL:2011. SIGMOD Record, Volume 41, Number 1, March 2012, pp. 67-73.
[10]. Krishna Kulkarni and Jan-Eike Michels. Temporal Features in SQL:2011. SIGMOD Record, Volume 41, Number 3, September 2012, pp. 34-43.
[11]. Э. Ф. Кодд. Расширение реляционной модели для лучшего отражения семантики. Системы управления базами данных № 5, 1996, издательский дом «Открытые системы», новая редакция: 2009 г., http://citforum.ru/database/classics/codd_2/. Оригинал: E.F. Codd. Extending the Database Relational Model to Capture More Meaning. ACM Transactions on Database Systems, Vol. 4, № 4, December 1979.
[12]. Hall, P., Owlett, J., and Todd, S. Relations and Entities. In Modelling in Data Base Management Systems, G.M.Nijssen, Ed., North-Holland Pub. Co., Amsterdam, 1976.
[13]. Date, C. J. Relational Database Writings 1994–1997. Chapter 12. Object Identifiers vs. Relational Keys. Addison-Wesley, 1998.
[14]. С.Д. Кузнецов. Объектно-реляционные базы данных: прошедший этап или недооцененные возможности? Труды Института системного программирования, т. 13, часть 2, М., ИСП РАН, 2007, стр. 115-140.
http://ispras.ru/ru/proceedings/docs/2007/13/2/isp_2007_13_2_115.pdf (pdf)
http://citforum.ru/database/articles/ordbms10/ (html)
[15]. С.Д. Кузнецов. Развитие идей и приложений реляционной СУБД System R. Москва, ВИНИТИ, 1989, Тем. изд. "Итоги науки и техники. Вычислительные науки". Т.1. Стр. 3-75. http://www.citforum.ru/database/articles/art_27.shtml
[16]. S. Kuznetsov. OODMBS's Query and Programming Languages: What Do They Provide and What Do We Need. Extended Information Systems Technology. Proceedings of the International East/West Database Workshop, Klagenfurt, Austria, 25-28 Sept. 1994, J.Eder, L.A.Kalinichenko (Eds), Springer-Verlag, 1995, 138-147. DOI: 10.1007/978-1-4471-3577-7_10.
[17]. С.Д. Кузнецов. Объектно-ориентированные базы данных и объектные расширения языка SQL. Семинар Московской секции ACM SIGMOD, 23 декабря 2004 г. http://synthesis.ipi.ac.ru/sigmod/seminar/s20041223
[18]. С.Д. Кузнецов. Объектные расширения SQL «объектны»! Семинар Московской секции ACM SIGMOD, 25 декабря 2014 г.
http://synthesis.ipi.ac.ru/sigmod/seminar/s20141225

ODMG and SQL object models ten years later: there are no contradictions

S.D. Kuznetsov <kuzloc@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, Russia, 109004.

Abstract. In 2005, I wrote an article in which I discussed the most important features of the standards ODMG 3.0 (ODMG object model) and the SQL:2003 (SQL data model) and convincingly (as it seemed to me) argued that the similarity between the object model and object extensions to SQL is purely external, that close in mind syntactic constructions hide

deep differences at the model level. Examples of such differences include von Neumann-style dereference of ObjectIDs in the ODMG model vs join-style dereference of reference values in the SQL model, separate and independent store of objects of one and the same object type in the ODMG model vs store of all raws of a typed table (SQL analogy of object) within this table, store of ObjectISs within extents in the ODMG model vs store within analogy of extent of objects their self in the SQL model, etc. Since then, it took many years for which I understood many things that were wrongly or insufficient correctly understood by me then, and gradually came to the conclusions that:

1. differences that seemed to me deep, are not such, and indeed are not differences at the model level;
2. the object extensions of SQL provide no less (and rather large) capabilities than the ODMG object model;
3. reasonably (from the standpoint of the database community) used DBMSes based on the ODMG data model, one will create databases and tools to manipulate them similar to those prescribed by the SQL data model.

Keywords: data model, object model, ODMG 3.0, SQL, dereferencing, object relocation.

DOI: 10.15514/ISPRAS-2015-27(1)-9

For citation: Kuznetsov S.D. ODMG and SQL object models ten years later: there are no contradictions. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 1, 2015, pp. 173-192 (in Russian). DOI: 10.15514/ISPRAS-2015-27(1)-9

References.

[1]. Sergey Kuznetsov. "Objektny" li objektnye rasshirenija jazyka SQL? [Whether are "object" the object extensions of SQL?], 2005 (in Russian). http://citforum.ru/database/articles/sql_odmg/

[2]. The Object Data Standard: ODMG 3.0. Edited by R.G.G. Cattel, Douglas K. Barry. Morgan Kauffmann Publishers, 2000.

[3]. Jim Melton. "Advanced SQL:1999. Understanding Object-Relational and Other Advanced Features". Morgan Kaufmann Publishers, 2003.

[4]. Sergey Kuznetsov. Naibolee interesnye novshestva v standarte SQL:2003 [The most interesting innovations in the SQL:2003 standard], 2004 (in Russian). http://citforum.ru/database/sql/sql2003/

[5]. S.D. Kuznetsov. Bazy dannyx: jazyki i modeli [Databases: Languages and Models], Moscow, Binom, 2008 (in Russian).

[6]. S.D. Kuznetsov. Tri manifesta baz dannyx: retrospektiva i perspektivy [Three manifests of databases: Retrospect and Prospects]. In Databases and Information Technology in XXI Century. Proceedings of the International Conference. Moscow, RSGU, 2004, pp. 52-229 (in Russian). http://citforum.ru/database/articles/manifests/

[7]. D. Chamberlin. XQuery: An XML query language. IBM SYSTEMS JOURNAL, VOL 41, NO 4, 2002, pp. 597-615.

[8]. C.J. Date. A Guide to SQL Standard (4th Edition). Addison-Wesley Professional, 1996)

[9]. Fred Zemke. What's new in SQL:2011. SIGMOD Record, Volume 41, Number 1, March 2012, pp. 67-73.

[10]. Krishna Kulkarni and Jan-Eike Michels. Temporal Features in SQL:2011. SIGMOD Record, Volume 41, Number 3, September 2012, pp. 34-43.

[11]. E.F. Codd. Extending the Database Relational Model to Capture More Meaning. ACM Transactions on Database Systems, Vol. 4, № 4, December 1979.

[12]. Hall, P., Owlett, J., and Todd, S. Relations and Entities. In Modelling in Data Base Management Systems, G.M.Nijssen, Ed., North-Holland Pub. Co., Amsterdam, 1976.

[13]. Date, C. J. Relational Database Writings 1994–1997. Chapter 12. Object Identifiers vs. Relational Keys. Addison-Wesley, 1998.

[14]. S.D. Kuznetsov. Objektno-reljacionnye bazy dannyx: proshedshij ehtap ili nedoocenennye vozmozhnosti? [Object-relational databases: the last stage or undervalued opportunities?]. Trudy ISP RAN [Proceedings of ISP RAS], v. 13, part 2, M., ISP RAS, 2007, pp. 115-140. (in Russian)
http://ispras.ru/ru/proceedings/docs/2007/13/2/isp_2007_13_2_115.pdf (pdf)
http://citforum.ru/database/articles/ordbms10/ (html)

[15]. S.D. Kuznetsov. Razvitie idej i prilozhenij reljacionnoj SUBD System R [Evolution of ideas and applications of the relational DBMS System R]. Moscow, VINITI, 1989, " Itogi nauki i texniki. Vychislitelqnye nauki [The results of science and technology. Computer Science]". V.1. Pp. 3-75. http://www.citforum.ru/database/articles/art_27.shtml

[16]. S. Kuznetsov. OODMBS's Query and Programming Languages: What Do They Provide and What Do We Need. Extended Information Systems Technology. Proceedings of the International East/West Database Workshop, Klagenfurt, Austria, 25-28 Sept. 1994, J.Eder, L.A.Kalinichenko (Eds), Springer-Verlag, 1995, 138-147. DOI: 10.1007/978-1-4471-3577-7_10

[17]. S.D. Kuznetsov. Objektno-orientirovannye bazy dannyx i objektnye rasshirenija jazyka SQL [Object-oriented databases and object extensions of the SQL query language]. Seminar Moskovskoj sekcii ACM SIGMOD, 23 dekabrja 2004 g. [Seminar of the Moscow ACM SIGMOD chapter, December 24th, 2004] (in Russian),
http://synthesis.ipi.ac.ru/sigmod/seminar/s20041223

[18]. S.D. Kuznetsov. Objektnye rasshirenija SQL «objektny»! [Object extensions of SQL are «object»!]. [Seminar of the Moscow ACM SIGMOD chapter, December 25th, 2014] (in Russian), http://synthesis.ipi.ac.ru/sigmod/seminar/s20141225.