ИСП Российская Академия наук
Институт Системного Программирования

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

Труды Института Системного Программирования РАН

Proceedings of the Institute for System Programming of the RAS

Том (Volume) 26

выпуск (issue) 2

Москва 2014

ИСП **Российская Академия наук**
Институт Системного Программирования

Труды Института Системного Программирования РАН

Proceedings of the Institute for System Programming of the RAS

Том (Volume) 26
Выпуск (issue) 2

Под редакцией
академика РАН В.П. Иванникова
Edited by
Academician V.P. Ivannikov

Москва 2014

УДК004.45

Труды Института системного программирования: Том 26, выпуск 2. /Под ред. Академика РАН В.П. Иванникова/ – М.: ИСП РАН, 2014.

Proceedings of the Institute for System Programming: Volume 26, issue 2. /Edited by Academician V.P. Ivannikov/ – M.: ISP RAS, 2014.

Во второй части двадцать шестом тома Трудов Института системного программирования публикуются статьи, содержащие результаты работ, проводимых в Институте в 2014 году.

The second issue of the 26th volume of the Proceedings of the Institute for System Programming of the RAS contains articles containing the results of researches carried out at the Institute in 2014.

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

Содержание

Конфигурируемая система статической верификации модулей ядра операционных систем

И.С. Захаров, М.У. Мандрыкин, В.С. Мутилин, Е.М. Новиков, А.К. Петренко, А.В. Хорошилов
{ilja.zakharov, mandrykin, mutilin, novikov, petrenko, khoroshilov}@ispras.ru

Аннотация. Ядро операционной системы (ОС) представляет собой критичную в отношении надежности и производительности программную систему. Качество ядра современных ОС уже находится на достаточно высоком уровне. Иначе обстоит дело с модулями ядра, например, драйверами устройств, которые по ряду причин имеют существенно более низкий уровень качества. Одними из наиболее критичных и распространенных ошибок в модулях ядра являются нарушения правил корректного использования программного интерфейса ядра ОС. Выявить все такие нарушения в модулях или доказать их корректность потенциально можно с помощью инструментов статической верификации, которым для проведения анализа необходимо предоставить контрактные спецификации, описывающие формальным образом обязательства ядра и модулей по отношению друг к другу. В статье рассматриваются существующие методы и системы статической верификации модулей ядра различных ОС. Предлагается новый метод статической верификации модулей ядра ОС Linux, который позволяет конфигурировать процесс проверки на каждом из его этапов. Показывается, каким образом данный метод может быть адаптирован для проверки компонентов ядра других ОС. Описывается архитектура конфигурируемой системы статической верификации модулей ядра ОС Linux, реализующей предложенный метод, и демонстрируются результаты ее практического применения. В заключении рассматриваются направления дальнейшего развития.

Ключевые слова: ядро операционной системы; модуль ядра; качество программной системы; статическая верификация; контрактная спецификация; модель окружения; спецификация правила корректного использования программного интерфейса.

1. Введение

Надежность и производительность ядра операционной системы (ОС) являются важными характеристиками его качества, поскольку ядро лежит в основе ОС и на результаты его работы во многом полагаются все пользовательские приложения. В ядре большинства современных ОС реализуется только основная функциональность, например, планирование процессорного времени, управление памятью и межпроцессным взаимодействием. Благодаря

этому ядро, с одной стороны, является достаточно небольшим по объему, а с другой стороны, представляет собой программную систему, высокое качество которой формировалось в течение длительного периода времени в разнообразных сценариях использования.

В большинстве ОС набор функций ядра можно расширить путем динамической загрузки модулей. Типичным примером модуля ядра служит драйвер устройства, необходимость в котором может возникнуть при подключении данного устройства. Также в виде модулей часто реализуют файловые системы, сетевые протоколы, аудиокодеки и т.д. Объем исходного кода модулей, поставляемых вместе с ядром ОС, может существенно превышать объем последнего, например, для ядра ОС Linux примерно в 8 раз. Кроме того, многие разработчики по разным причинам не предоставляют доступ к исходному коду некоторых из своих модулей, а распространяют их в виде бинарного кода, что существенно затрудняет применение некоторых подходов обеспечения качества. Не все модули ядра могут использоваться активно, например, если это драйверы специфичных устройств. Все это приводит к тому, что модули ядра различных ОС имеют существенно меньший уровень качества по сравнению с самим ядром. Это подтверждается исследованиями, которые показали, что в тех модулях, чей исходный код доступен для анализа, содержится примерно в 7 раз больше ошибок, чем в ядре ОС [1-3].

Высокая производительность модулей ядра ОС обеспечивается за счет того, что большинство из них работают в том же адресном пространстве и с тем же уровнем привилегий, что и ядро. Из-за этого ошибки в модулях могут привести к ненадежной работе и снижению производительности ядра, а также и ОС в целом. Анализ изменений в модулях ядра ОС Linux за год разработки показал, что нарушения правил корректного использования программного интерфейса ядра в модулях являются источником около половины ошибок, которые не связаны с нарушениями спецификаций аппаратного обеспечения, сетевых протоколов, аудиокодеков и т.д. [4]. Для других ОС авторам недоступна подобная статистика, но существующие направления исследований также демонстрируют достаточно высокий интерес к ошибкам данного типа [5].

1.1. Подходы к обнаружению нарушений правил корректного использования программного интерфейса ядра ОС в модулях

Выявлять нарушения правил корректного использования программного интерфейса ядра ОС в модулях можно различными способами. На практике наиболее часто используются экспертиза кода и тестирование. Благодаря использованию данных подходов удается обнаружить и исправить достаточно большое количество ошибок в модулях. Однако они не позволяют выявить все возможные ошибки [6]. Тщательная экспертиза кода требует больших

трудозатрат, а потому она в полной мере проводится только для ядра ОС, но не для такой большой, сложной и динамично развивающейся программной системы, как модули ядра. Тестирование обычно проводится автоматизированным образом, что позволяет сократить трудозатраты по сравнению с экспертизой кода. Однако данный подход требует подготовки тестового окружения, что, например, в случае драйверов устройств бывает достаточно затруднительно. Кроме того, с помощью тестирования можно тщательно проверить только небольшие программные системы.

В последнее время основной тенденцией в проверке программных систем является применение методов и инструментов статического анализа кода, т.е. анализа кода без его реального выполнения (как правило, анализируется исходный код программных систем). На практике преимущественно применяются так называемые легковесные подходы, которые благодаря использованию ряда эвристик позволяют проверять код больших программных систем за время, сравнимое по порядку со временем компиляции. Негативными последствиями использования эвристик является то, что инструменты, которые реализуют данные подходы статического анализа кода, с одной стороны, пропускают ошибки, а с другой стороны, выдают большое количество ложных сообщений об ошибках. Эти инструменты преимущественно применяются для поиска нарушений общих правил безопасного программирования, таких как разыменования нулевых указателей, выход за границу массива и т.д. [7, 8]. Некоторые инструменты были использованы для проверки правил корректного использования программного интерфейса ядра ОС Linux в модулях [9].

Выявить все ошибки искомого вида в программных системах или доказать их корректность потенциально можно с помощью тщательного статического анализа – *статической верификации*. Современные инструменты статической верификации, которые реализуют, например, метод уточнения абстракции по контрпримерам [10], уже позволяют доказывать выполнимость специфицированных свойств для средних по размеру программных систем за приемлемое время. В частности, с помощью данных инструментов можно верифицировать отдельные компоненты ядра ОС, такие как модули ядра (для ядра ОС Linux размер большинства модулей составляет несколько тысяч строк кода).

Сами по себе инструменты статической верификации не способны искать нарушения правил корректного использования программного интерфейса ядра ОС в модулях – они позволяют решать *задачу достижимости*. Как правило, инструменты позволяют определить возможность достижимости некоторого оператора, помеченного заданной меткой, от указанной точки входа. Поэтому необходимо некоторым образом свести задачу обнаружения нарушений проверяемых правил к задачам достижимости. Для этого требуется разработать *спецификации правил корректного использования программного интерфейса ядра ОС*, которые устанавливают соответствие между

нарушениями правил и достижимостью оператора, помеченного заданной меткой. Кроме того, исследования показали, что для того, чтобы получить приемлемые результаты статической верификации модулей ядра (найти ошибки искомого вида при умеренном количестве ложных сообщений об ошибках), инструментам требуется достаточно точная *модель окружения*, которая должна описывать те же сценарии взаимодействия ядра и модулей, которые возможны при их работе в реальном окружении [11].

Таким образом, инструментам статической верификации для поиска нарушений правил корректного использования программного интерфейса ядра ОС в модулях необходимо предоставить *контрактные спецификации*, которые формальным образом описывают обязательства ядра и модулей по отношению друг к другу. Со стороны ядра контрактные спецификации должны задавать множество возможных сценариев взаимодействий с модулями корректным и полным образом, а также предоставлять модель программного интерфейса ядра, используемого модулями. Со стороны модулей контрактные спецификации должны задавать, какие обращения модулей к ядру являются корректными, а какие – нет.

В настоящее время инструменты статической верификации активно развиваются в университетах и научно-исследовательских институтах по всему миру. С каждым годом список инструментов пополняется [12-14]. Инструменты статической верификации реализуют различные подходы, что позволяет применять их для доказательства выполнимости различных специфицированных свойств. Среди этих инструментов нет однозначного лидера, так как они используют разные техники и нацелены на разные классы ошибок. Поэтому в долгосрочной перспективе важно иметь возможность использовать различные инструменты статической верификации.

В данной статье будут рассматриваться только те инструменты статической верификации, которые позволяют проверять программные системы на языке Си, поскольку модули ядра большинства ОС разрабатываются на языке Си.

1.2. Особенности процесса разработки ядра различных ОС

При разработке контрактных спецификаций и проведении статической верификации модулей необходимо учитывать особенности процесса разработки ядра различных ОС.

Ядро ОС Microsoft Windows разрабатывается централизовано. При этом выделяются большие ресурсы на развитие и применение новых технологий, в частности, для обеспечения качества посредством статической верификации. Большое количество модулей ядра ОС Microsoft Windows разрабатываются и поддерживаются исключительно производителями соответствующего аппаратного обеспечения. Это в том числе означает, что только разработчики имеют доступ к исходному коду модулей, необходимому для проведения статической верификации. В связи с этим часто сами разработчики выполняют статическую верификацию, анализируют получающиеся результаты и при

необходимости могут провести доработку контрактных спецификаций. Программный интерфейс между модулями и ядром ОС Microsoft Windows является стабильным[1], что позволяет использовать одни и те же контрактные спецификации для разных версий ядра.

Ядро ОС Linux поставляется вместе с большим количеством модулей (около 4 тыс. в последних версиях) в виде исходного кода. В подготовке всех новых версий ядра принимают участие более 1 000 разработчиков из более 200 организаций, рассредоточенных по всему миру [15]. При этом разработчики сами не пишут контрактные спецификации. Программный интерфейс между модулями и ядром ОС Linux не является стабильным[2] – это осложняет задачу разработки и поддержки контрактных спецификаций, поскольку при выходе новых версий ядра может потребоваться их переработка.

Особенности разработки ядра других ОС не рассматриваются в рамках данной статьи, поскольку авторам неизвестно об опыте применения инструментов статической верификации для них.

1.3. Существующие системы статической верификации модулей ядра различных ОС

Для того чтобы автоматизировать процесс статической верификации модулей ядра различных ОС, разрабатываются *системы статической верификации*. На сегодняшний день до уровня промышленного использования доросла единственная система статической верификации Static Driver Verifier (SDV) [16], которая была разработана в компании Microsoft. Данная система позволяет проверять модули ядра ОС Microsoft Windows различных версий на предмет выполнения правил корректного использования программного интерфейса ядра с помощью инструмента статической верификации SLAM [17].

Информацию о составе и опциях сборки модулей SDV получает автоматически на основе оригинальных скриптов, описывающих сборку. В данной системе статической верификации часть модели окружения, описывающая возможные сценарии взаимодействия ядра и модулей, генерируется автоматически на основе спецификаций, которые разработчики SDV задали для всех типов модулей, и аннотаций к модулям, которые пишутся вручную. Модель программного интерфейса ядра ОС Microsoft Windows, используемого модулями, входит в состав SDV. Спецификации правил корректного использования программного интерфейса ядра ОС пишутся на языке Specification Language for Interface Checking (SLIC) [18]. В настоящее время SDV поставляется с набором, включающим около 200

[1] Документация по Windows Driver Frameworks:
http://msdn.microsoft.com/en-us/Library/Windows/Hardware/ff557565%28v=vs.85%29.aspx.

[2] Программный интерфейс ядра ОС Linux для драйверов:
http://www.kernel.org/doc/Documentation/stable_api_nonsense.txt.

спецификаций правил. Исследовательская версия SDV [19] позволяет добавлять свои спецификации правил, а также использовать инструмент статической верификации Yogi. Большое внимание разработчики SDV уделили автоматизации запуска процесса статической верификации модулей ядра и поддержке анализа результатов статической верификации. Пользователям показываются как суммарные результаты статической верификации для анализируемых модулей по всем проверяемым правилам, так и визуализированные *трассы ошибок* (пути в исходном коде, на которых возможны нарушения проверяемых правил). По состоянию на 2010 год с помощью SDV было выявлено 270 ошибок в модулях ядра, входящих в поставку ОС Microsoft Windows.

Для модулей ядра ОС Linux были разработаны две системы статической верификации: DDVerify (Университет Карнеги-Меллон, Питтсбург, США) [20] и Avinux (Университет Эберхарда и Карла, Тюбинген, Германия) [21].

DDVerify использует собственные скрипты сборки для извлечения информации о составе и опциях сборки анализируемых модулей. Контрактные спецификации в этой системе статической верификации задаются полностью вручную на языке программирования Си. Разработчики DDVerify сделали модель окружения для четырех типов модулей, а также для обработчиков аппаратных прерываний, таймеров и т.д. Кроме того, непосредственно в коде модели окружения они задали восемь спецификаций правил корректного использования примитивов синхронизации и правил корректной инициализации переменных до их использования. DDVerify позволяет проверять модули с помощью двух инструментов статической верификации CBMC [22] и SATABS [23]. Для упрощения анализа трасс ошибок в состав системы статической верификации входит плагин для интегрированной среды разработки Eclipse.

Система статической верификации Avinux предоставляет возможность автоматизированным образом проверять единичные препроцессированные файлы модулей ядра, для чего в ней осуществляется встраивание в процесс сборки ядра путем модификации оригинальных скриптов, описывающих сборку. Модель окружения в Avinux задается практически полностью вручную (автоматически вызываются только экспортируемые функции модулей, для параметров которых генерируется код инициализации). Спецификации правил корректного использования программного интерфейса ядра ОС пишутся на расширении языка SLIC. В Avinux поддерживается проверка правил корректного использования примитивов синхронизации и правил корректной работы с памятью. Система статической верификации была интегрирована с единственным инструментом статической верификации CBMC. Avinux реализован в виде плагина для Eclipse, что позволяет запускать данную систему автоматизированным образом. Но пользователю выдаются трассы ошибок в формате CBMC, что существенно затрудняет их анализ.

Ни одна из рассмотренных систем статической верификации не учитывает особенности процесса разработки ядра ОС Linux в полной мере. Система SDV предназначена только для статической верификации модулей ядра ОС Microsoft Windows, программный интерфейс которого не изменяется с течением времени. DDVerify требует переработки собственного процесса сборки, заголовочных файлов ядра и контрактных спецификаций для каждой новой версии ядра. Кроме того, в данной системе статической верификации была реализована модель окружения только для четырех типов модулей из нескольких сотен. Avinux не поддерживает автоматическую проверку модулей, состоящих из нескольких файлов, требует задавать большую часть модели окружения вручную и также вручную следить за актуальностью спецификаций правил. Обе системы статической верификации модулей ядра ОС Linux не продемонстрировали значительных результатов на практике, и их поддержка была прекращена несколько лет назад.

Ни одна из рассмотренных систем статической верификации не поддерживает интеграцию сторонних инструментов статической верификации. Это особенно важно при проверке модулей ядра ОС Linux, поскольку, с одной стороны, для проведения анализа доступно большое количество инструментов статической верификации, реализующих принципиально различные подходы, а с другой стороны, в отличие от Microsoft SDV у разработчиков систем статической верификации нет достаточного количества ресурсов на то, чтобы одновременно вести полноценную поддержку какого-то выделенного инструмента статической верификации.

Существующие системы статической верификации не предоставляют средств для сравнительного анализа результатов верификации и автоматизированной разметки сообщений об ошибках, что важно, поскольку проверяемых модулей ядра ОС Linux, спецификаций правил, инструментов статической верификации и их конфигураций может быть достаточно много, а кроме того, со временем они развиваются.

В связи с этим разработка нового метода и системы статической верификации модулей ядра ОС Linux является актуальной задачей.

1.4. План статьи

Во втором разделе данной статьи представлен метод статической верификации модулей ядра ОС Linux. Показывается, что метод позволяет конфигурировать процесс проверки на каждом из его этапов. Кроме того, описываются модификации метода, необходимые для его использования с целью верификации компонентов ядра других ОС. Архитектура разработанной конфигурируемой системы статической верификации модулей ядра ОС Linux рассмотрена в третьем разделе. Четвертый раздел посвящен результатам практического применения данной системы. В данном разделе анализируются выявленные ошибки в модулях ядра ОС Linux, причины ложных сообщений об ошибках и пропуска ошибок, время проведения

статической верификации и причины неуспешного завершения работы системы статической верификации. Помимо этого, в четвертом разделе рассматриваются возможности разработанной конфигурируемой системы статической верификации для сравнения инструментов статической верификации и их конфигураций. В заключении подводятся итоги, а также представляются направления дальнейшего развития.

2. Метод статической верификации модулей ядра ОС Linux

Предлагаемый в данной статье метод статической верификации модулей ядра ОС Linux состоит из нескольких шагов, которые описаны в подразделах 2.1-2.5. В подразделе 2.6 описываются модификации метода, которые позволяют применять его для верификации компонентов ядра других ОС.

2.1. Первоначальная подготовка исходного кода к статической верификации

На первом шаге метода осуществляется первоначальная подготовка исходного кода ядра и модулей ОС Linux к статической верификации. В подразделе 1.1 было отмечено, что современные инструменты статической верификации не позволяют проверять ядро ОС целиком. В связи с этим возникает задача разделить исходный код ядра и модулей ОС Linux и соответствующие им команды компиляции и компоновки, описывающие правила сборки, таким образом, чтобы размер получаемых *объектов верификации* был ограничен несколькими тысячами или десятками тысяч строк, но при этом данные объекты включали в себя как можно больше кода, участвующего в реализации функциональности соответствующих модулей. Например, можно выделять в отдельный объект верификации файлы с исходным кодом, которые непосредственно составляют анализируемый модуль, и соответствующие им команды сборки[3]. Модуль ядра может вызывать функции ядра, а также функции из других модулей, поэтому соответствующий объект можно дополнить кодом этих функций.

В предлагаемом методе объекты верификации готовятся автоматически на основе оригинальных скриптов сборки ядра ОС Linux, для чего они модифицируются таким образом, чтобы наряду с собственно сборкой ядра и модулей выводилась информация о соответствующих командах компиляции и компоновки.

[3] Это можно сделать на основе команд компоновки, которые соответствуют модулям, поскольку выходные файлы команд компиляции файлов с исходным кодом являются входными файлами для команд компоновки, в том числе, команд компоновки модулей.

2.2. Построение модели окружения

Для того чтобы проверить получаемые на первом шаге объекты верификации на предмет выполнения правил корректного использования программного интерфейса ядра ОС Linux, на втором шаге метода для этих объектов строится модель окружения и задается точка входа для инструментов статической верификации.

В типичном модуле ядра ОС Linux определяются функция инициализации, обработчики (обработчики инициализации и отключения устройств, обработчики открытия, чтения, записи и закрытия файлов, обработчики аппаратных прерываний и т.д.) и функция выхода. Функция инициализации вызывается при загрузке модуля ядром. Данная функция регистрирует обработчики модуля, которые затем вызываются ядром при необходимости для обработки системных вызовов от пользовательских приложений, аппаратных прерываний и внутренних событий ядра. Перед выгрузкой модуля вызывается функция выхода, в которой происходит освобождение выделенных для модуля ресурсов и дерегистрация обработчиков модуля.

В предлагаемом методе статической верификации модулей ядра ОС Linux множество сценариев взаимодействия ядра и модулей, которые возможны при их работе в реальном окружении, описывается с помощью π-процессов [24, 25]. Это позволяет задавать модель окружения как на детальном уровне для определенных групп обработчиков модулей, так и в виде шаблонов, под которые попадает большинство оставшихся групп. В шаблонах может быть описано, например, что обработчики модулей могут вызываться только после успешного вызова функции инициализации, что отключение устройства может быть выполнено только после его успешной инициализации, а читать из файла можно только после его открытия. Благодаря этому спецификацию π-модели окружения для всех типов модулей ядра ОС Linux удается задать достаточно компактным образом. Кроме того, одна и та же спецификация π-модели окружения может быть использована для различных версий ядра, поскольку в шаблонах не указываются ни конкретные обработчики модулей, ни даже конкретные группы обработчиков. В качестве примера на рис. 1 показано схематичное описание модели окружения для драйвера USB-устройства.

Пользователь может дополнить спецификацию π-модели окружения, например, описать точно определенную группу обработчиков модулей, для которой модель окружения строилась на основе шаблона, если выяснится, что некоторая известная ошибка пропускается из-за того, что модель задает не все возможные сценарии взаимодействия ядра и модулей, или выдается ложное сообщение об ошибке вследствие того, что модель допускает сценарии невозможные в реальном окружении.

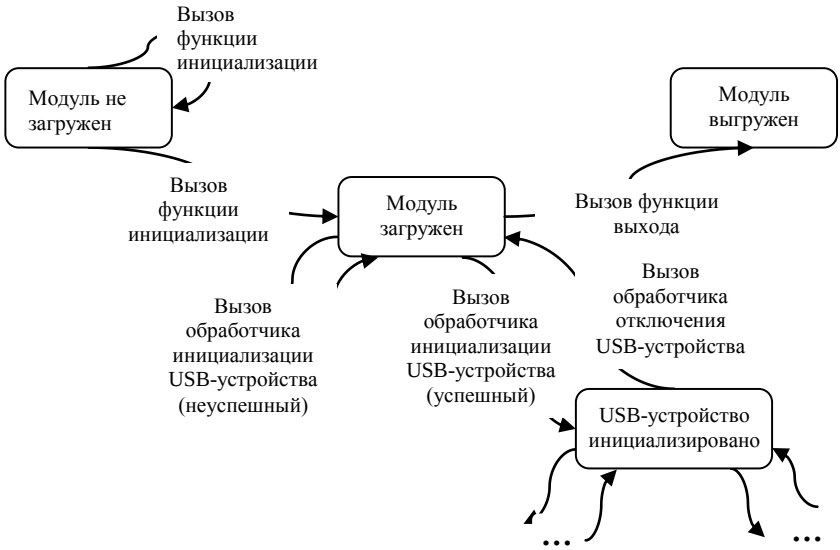


Рис. 1. Схематичное описание модели окружения для драйвера USB-устройства.

Подбирать и заполнять шаблоны из спецификации π-модели окружения предлагается автоматически на основе анализа исходного кода, составляющего полученные на предыдущем шаге объекты верификации. Для каждого объекта может быть построено несколько моделей окружения, например, для разных групп обработчиков модулей. С целью того, чтобы впоследствии использовать различные инструменты статической верификации, каждая полученная модель окружения транслируется в функцию на языке Си, из которой вызываются обработчики объекта верификации в том виде и в той последовательности, в которых это происходит при его работе в реальном окружении. Эта функция добавляется к исходному коду объекта верификации и в дальнейшем используется, как точка входа для инструментов статической верификации.

2.3. Построение спецификаций правил корректного использования программного интерфейса ядра ОС Linux

На третьем шаге метода для тех элементов программного интерфейса ядра ОС Linux, которые релевантны специфицируемым правилам, предлагается разработать модель[4], а на основании состояния этой модели и этих правил – задать предусловия данных элементов программного интерфейса.

[4] Эта модель является частью модели окружения для модулей ядра ОС Linux.

```
#include <linux/usb.h>
#include <verifier/rcv.h>

/* Множество указателей на структуры
urb, под которые для модуля была
выделена память. Тип данных множество
set, а также операции с объектами данного
типа определены в заголовочном файле
verifier/rcv.h. */
set URBS = empty;

/* Модельные функции для выделения и
освобождения памяти под структуры urb.
Модельная функция выделения памяти
ldv_alloc определена в заголовочном файле
verifier/rcv.h. */
struct urb *
ldv_usb_alloc_urb(void)
{
  void *urb;
  urb = ldv_alloc();
  if (urb) {
    add(URBS, urb);
  }
  return urb;
}
void
ldv_usb_free_urb(struct urb
*urb)
{
  if (urb) {
    remove(URBS, urb);
  }
}
```

```
/* Описание множества точек использования
элементов программного интерфейса ядра. */
pointcut USB_ALLOC_URB: call(
  struct urb *usb_alloc_urb(int,
gfp_t))
pointcut USB_FREE_URB: call(
  void usb_free_urb(struct urb *))

/* Привязка вызовов модельных функций к
точкам использования элементов программного
интерфейса. */
around: USB_ALLOC_URB {
  return ldv_usb_alloc_urb();
}
around: USB_FREE_URB {
  ldv_usb_free_urb($arg1);
}

/* Предусловия элементов программного
интерфейса ядра. Макрофункция ldv_assert
определяется в заголовочном файле verifier/rcv.h
следующим образом:
#define ldv_assert(e) (e ? 0 :
ldv_error())
static inline void ldv_error(void) {
  LDV_ERROR: goto LDV_ERROR;
}
LDV_ERROR является меткой, достижимость
которой равнозначна нарушению проверяемых
правил.
*/
before: USB_FREE_URB {
  ldv_assert(contains(URBS, $arg1));
}
after: MODULE_EXIT {
  ldv_assert(is_empty(URBS));
}
```

Рис. 2. Пример спецификации правил корректного выделения и освобождения блоков-запросов для USB-устройств.

Подробно процесс построения спецификаций правил корректного использования программного интерфейса ядра ОС Linux описан в диссертации [26]. Там же представлено аспектно-ориентированное расширение языка программирования Си, которое используется для задания спецификаций. На рис. 2 представлен пример спецификации правил корректного выделения и освобождения блоков-запросов для USB-устройств (USB Request Blocks – URB), который наглядным образом показывает составные части спецификаций и их взаимосвязь друг с другом и кодом анализируемых модулей.

Следует отметить, что в предлагаемом методе статической верификации модулей ядра ОС Linux сигнатуры элементов программного интерфейса ядра, для которых разрабатывается модель и задаются предусловия, описываются

полностью (см. определения *USB_ALLOC_URB* и *USB_FREE_URB* на рис. 2). Благодаря этому возможно автоматизированным образом поддерживать согласованность спецификаций правил с программным интерфейсом ядра и его реализацией, поскольку при существенных изменениях в реализации ядра ОС Linux разработчики, как правило, изменяют соответствующие элементы программного интерфейса.

Пользователь может дорабатывать спецификации правил корректного использования программного интерфейса ядра ОС Linux, если окажется, что они недостаточно точные и приводят к пропуску известных ошибок или к ложным сообщениям об ошибках. Также пользовать может предоставить собственные спецификации правил.

На основе спецификаций правил в методе предлагается инструментировать исходный код, полученный на втором шаге, таким образом, чтобы нарушения правил корректного использования программного интерфейса ядра ОС Linux соответствовали достижимости метки *LDV_ERROR* от заданных точек входа. В результате этого получаются *верификационные задачи* – объекты верификации с поставленными на них задачами достижимости.

2.4. Запуск инструментов статической верификации

На четвертом шаге метода на полученных верификационных задачах запускаются инструменты статической верификации для того, чтобы выявить нарушения проверяемых правил или доказать корректность соответствующих модулей относительно данных правил. В методе предлагается запускать инструменты посредством адаптеров. Адаптер для некоторого инструмента статической верификации выполняет следующее:

- Готовит составляющие объекты верификации Си-файлы, например, применяет к каждому из них стандартный препроцессор языка Си, обрабатывает файлы с помощью инструмента трансформации кода CIL [27], или соединяет все файлы в один с помощью того же CIL (все инструменты статической верификации принимают на вход препроцессированные файлы с исходным кодом, а некоторые инструменты могут анализировать только один файл за раз).

- Запускает инструмент статической верификации, передавая подготовленные Си-файлы и необходимую конфигурацию и ограничивая потребляемые инструментом ресурсы, такие как процессорное время и оперативная память (это необходимо, поскольку инструменты статической верификации могут работать неприемлемо долго и/или потреблять чрезмерно большое количество памяти).

- Обрабатывает статус завершения работы инструмента статической верификации (например, завершен успешно, или завершен по сигналу в связи с превышением ограничения по памяти, или завершен

некорректно из-за ошибки в инструменте) и подсчитывает потребленные инструментом ресурсы.

- При неуспешном завершении инструмент статической верификации не выносит определенного вердикта, поэтому в данном случае адаптер должен сгенерировать вердикт *Unknown* и указать причины неуспешного завершения работы, которые он может определить на основе анализа вывода инструмента и информации о статусе завершения работы инструмента.
- В случае успешного завершения инструмент статической верификации либо гарантирует отсутствие нарушений проверяемых правил (выносит вердикт *Safe*), либо говорит об обнаружении возможного нарушения правила (выносит вердикт *Unsafe*) – адаптер должен определить соответствующий вердикт на основе вывода инструмента.
- При вынесении вердикта *Unsafe* вывод инструмента статической верификации сопровождается трассой ошибки, формат которой у разных инструментов может существенно отличаться, поэтому с целью упрощения последующего анализа результатов статической верификации адаптер должен преобразовывать все трассы ошибок к общему формату (см. следующий подраздел).

Для интеграции сторонних инструментов статической верификации пользователю необходимо разработать соответствующий адаптер. При этом можно использовать части адаптеров для других инструментов, поскольку они могут быть устроены схожим образом.

2.5. Анализ результатов статической верификации

На заключительном шаге рассматриваемого метода предлагается автоматизировать анализ результатов статической верификации модулей ядра ОС Linux в нескольких направлениях.

Во-первых, упростить анализ трасс ошибок, выдаваемых инструментами статической верификации, – данный вид анализа необходимо выполнять для того, чтобы разобраться в причинах выявленных ошибок, а также, чтобы определить, какие сообщения об ошибках являются ложными. С этой целью предлагается сначала преобразовывать все трассы ошибок к общему формату (это делается адаптером для соответствующего инструмента статической верификации), а затем единообразно визуализировать трассы, представленные в общем формате, с привязкой к соответствующему исходному коду анализируемых модулей и ядра ОС Linux.

Во-вторых, автоматизировано размечать сообщения об ошибках (указывать была ли найдена ошибка или сообщение об ошибке является ложным), если соответствующие трассы ошибок оказываются похожими на уже

проанализированные трассы ошибок, например, имеют одинаковые деревья или стеки вызываемых функций. Благодаря этому можно существенно сократить трудозатраты на анализ результатов. Например, если инструмент статической верификации выдает ложное сообщение об ошибке при проверке модуля некоторой версии ядра ОС Linux, то, как правило, он выдает достаточно похожую трассу ошибки при проверке данного модуля в следующей версии ядра, и благодаря автоматизированной разметке сообщений об ошибках можно не проводить анализ повторно.

В-третьих, предоставить статистику и возможность сравнения результатов статической верификации, что необходимо, поскольку проверяемых модулей ядра ОС Linux, спецификаций правил, инструментов статической верификации и их конфигураций достаточно много, а кроме того, со временем они развиваются.

В-четвертых, обеспечить возможность совместного анализа результатов статической верификации без существенных трудозатрат на организацию взаимодействия.

2.6. Адаптация метода с целью верификации компонентов ядра других ОС

Предложенный в данном разделе метод может быть применен для верификации компонентов ядра других ОС. С этой целью для целевой ОС необходимо:

- Предложить подход к подготовке объектов верификации. Получать информацию о составе и опциях сборки компонентов ядра можно на основе оригинальных скриптов сборки, также как и для модулей ядра ОС Linux.

- Построить модель окружения для анализируемых компонентов ядра ОС. Для этого нужно изучить особенности взаимодействия компонентов между собой и с окружением ОС и сформулировать ограничения на множество сценариев взаимодействия между ними. Строить модель окружения, опираясь на данные ограничения, можно вручную либо на основе некоторого формального описания.

- Задать спецификации правил, соответствующих ошибкам искомого вида. Например, для поиска нарушений правил корректного использования программного интерфейса ядра ОС сначала нужно определить релевантные правилам элементы программного интерфейса, а затем разработать модель и задать предусловия данных элементов программного интерфейса.

Дальнейшие шаги, интеграция и запуск инструментов статической верификации и анализ результатов статической верификации, остаются такими же, как и для модулей ядра ОС Linux.

2.7. Выводы

В предложенном методе статической верификации модулей ядра ОС Linux можно конфигурировать процесс проверки на каждом из его этапов:

- При подготовке объектов верификации можно выбрать, каким образом разделить исходный код модулей и ядра и соответствующие им команды сборки.
- Можно дополнить спецификацию π-модели окружения.
- Можно уточнить существующие спецификации правил корректного использования программного интерфейса ядра, а также добавить собственные спецификации.
- Для проведения анализа можно задать особые конфигурации и ограничения на потребляемые ресурсы для интегрированных инструментов статической верификации, а кроме того, можно интегрировать сторонние инструменты статической верификации.

Возможность конфигурируемости послужила одной из причин, которая позволила довести реализующую метод систему до достаточно зрелого уровня. В настоящее время она продолжает активно развиваться, а с ее помощью уже было выявлено более 150 критичных ошибок в модулях ядра ОС Linux.

3. Конфигурируемая система статической верификации Linux Driver Verification Tools

Предложенный метод статической верификации модулей ядра ОС Linux был реализован в конфигурируемой системе статической верификации Linux Driver Verification Tools (LDV Tools) [28].

3.1. Архитектура Linux Driver Verification Tools

Архитектура LDV Tools представлена на рис. 3. Компоненты и подкомпоненты LDV Tools, а также инструменты статической верификации изображены в центре в порядке их вызова. Слева изображены входные данные для конфигурируемой системы статической верификации. Справа показан порядок формирования отчета по результатам статической верификации, загрузка отчета в базу данных LDV и компонент LDV Analytics Center, предназначенный для автоматизации анализа результатов статической верификации.

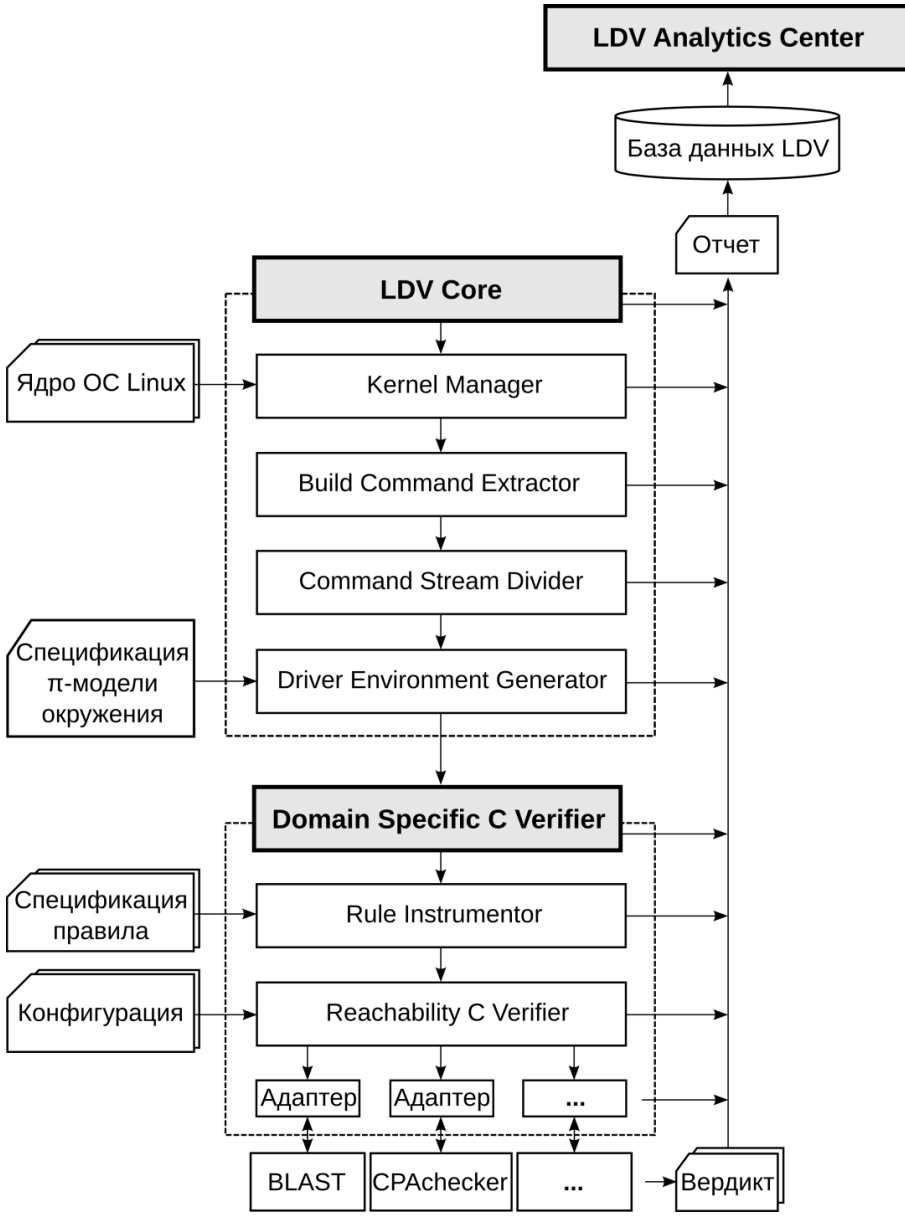


Рис. 3. Архитектура системы статической верификации Linux Driver Verification Tools.

3.2. Архитектура Linux Driver Verification Tools

Архитектура LDV Tools представлена на рис. 3. Компоненты и подкомпоненты LDV Tools, а также инструменты статической верификации изображены в центре в порядке их вызова. Слева изображены входные данные для конфигурируемой системы статической верификации. Справа показан порядок формирования отчета по результатам статической верификации, загрузка отчета в базу данных LDV и компонент LDV Analytics Center, предназначенный для автоматизации анализа результатов статической верификации.

3.3. LDV Core

Процесс статической верификации модулей ядра ОС Linux начинается с запуска компонента LDV Core. Данный компонент вызывает подкомпонент Kernel Manager, который создает на диске копию ядра ОС Linux, предоставленного пользователем в виде архива, директории или Git-репозитория. В дальнейшем все компоненты и подкомпоненты LDV Tools работают только с копией ядра. Kernel Manager модифицирует оригинальные скрипты сборки ядра, чтобы впоследствии получить информацию о составе модулей и опциях их сборки.

Далее LDV Core вызывает Build Command Extractor, который запускает сборку ядра ОС Linux. По мере выполнения сборки данный подкомпонент перехватывает поток команд компиляции и компоновки файлов ядра. По этим командам Build Command Extractor выделяет те файлы с исходным кодом, которые относятся к верифицируемым модулям.

Получаемые команды компиляции и компоновки передаются подкомпоненту Command Stream Divider, который формирует на их основе объекты верификации. В настоящее время каждый объект верификации соответствует ровно одному модулю ядра, благодаря чему в дальнейшем обрабатываются и анализируются все файлы с исходным кодом, которые составляют модули. В будущем планируется составлять объекты для групп взаимосвязанных модулей, а также дополнять их тем кодом ядра, от которого зависят анализируемые модули. Пользователю будет предоставлена возможность выбирать ту или иную стратегию разделения исходного кода ядра и модулей ОС Linux и соответствующих им команд сборки на объекты верификации.

Каждый из подготовленных объектов верификации LDV Core подает на вход подкомпоненту Driver Environment Generator [29]. Также данному подкомпоненту на вход поступает спецификация π-модели окружения, входящая в состав LDV Tools.. При необходимости пользователь может дополнить данную спецификацию.

Driver Environment Generator определяет функцию инициализации, обработчики и функцию выхода по исходному коду, входящему в объекты верификации. Затем, на основе этих данных и спецификации π-модели

окружения для каждого объекта строится промежуточное представление модели окружения, которое Driver Environment Generator транслирует в исходный код на языке Си, который добавляется к коду объектов верификации.

3.4. Domain Specific C Verifier

Компонент Domain Specific C Verifier связывает объекты верификации с указанными спецификациями правил и запускает на них указанный инструмент статической верификации в заданной конфигурации.

Спецификации правил, доступные для проверки, хранятся в базе, где каждой спецификации соответствует уникальный идентификатор. При необходимости пользователь может использовать свою базу спецификаций правил, добавить в существующую базу свои собственные спецификации правил либо модифицировать существующие спецификации.

Для каждого из указанных идентификаторов спецификаций правил Domain Specific C Verifier вызывает Rule Instrumentor. Данный подкомпонент по идентификатору извлекает из базы информацию о проверяемой спецификации, на основе которой он инструментирует файлы, входящие в объект верификации, так, чтобы была поставлена задача достижимости для инструментов статической верификации (нахождение решения данной задачи соответствует возможному нарушению проверяемых правил).

Для проведения статической верификации используются либо те инструмент и конфигурация, которые указаны пользователем, либо, при отсутствии данных указаний, те, которые используются в LDV Tools по умолчанию. Взаимодействие с инструментом статической верификации осуществляет подкомпонент Reachability C Verifier, который вызывает инструмент статической верификации посредством соответствующего адаптера. Данному адаптеру передаются верификационная задача, конфигурация и ограничения на ресурсы для инструмента статической верификации.

На сегодняшний день в состав конфигурируемой системы статической верификации LDV Tools входят адаптеры для таких инструментов статической верификации, как BLAST [30] (используется по умолчанию), CPAchecker [31], UFO [32] и CBMC [22]. Кроме того, пользователь может интегрировать в LDV Tools сторонние инструменты статической верификации путем реализации адаптеров для этих инструментов.

3.5. Формирование итогового отчета и его загрузка в базу данных LDV

Вердикты, которые выдает инструмент статической верификации, обрабатываются всеми компонентами и подкомпонентами LDV Tools в обратном порядке (рис. 3). При этом на каждом этапе промежуточный отчет дополняется информацией о работе соответствующих компонентов и подкомпонентов. В итоге формируется финальный отчет с результатами

статической верификации модулей ядра ОС Linux, который может быть загружен в базу данных LDV.

#	Task	Kernel	Rule	Total	Safe	Unsafe	Unknown	Verdicts		
								True	False	?
1	Task description BLAST, 15Gb	linux-3.13-rc1	08_1a	5994	5004	92	898	2	79	11
2			101_1a	5994	5112	1	881	-	1	-
3			106_1a	5994	5008	23	963	2	20	1
4			10_1a	5994	5175	8	811	-	8	-
5			118_1a	5994	5115	13	866	-	12	1
6			129_1a	5994	5028	7	959	2	5	-
7			132_1a	5994	5055	42	897	13	14	15
8			134_1a	5994	5074	3	917	3	-	-
9			146_1a	5994	5028	11	955	-	1	10
10			147_1a	5994	5017	21	956	3	1	17
11			150_1a	5994	5119	3	872	1	2	-
12			32_7a	5994	5037	81	876	8	70	3
13			39_7a	5994	5050	76	868	5	58	13
14			68_1	5994	4867	119	1008	4	115	-
15			77_1a	5994	5184	1	809	1	-	-

Рис. 4. Результаты статической верификации, сгруппированные по версии ядра ОС Linux (3.13-rc1) и по идентификаторам спецификаций правил.

3.6. LDV Analytics Center

LDV Analytics Center – это компонент конфигурируемой системы статической верификации LDV Tools, который позволяет автоматизированным образом проводить несколько видов анализа результатов статической верификации, загруженных в базу данных LDV:

- Статистический анализ (можно сгруппировать результаты статической верификации, например, по версии ядра ОС Linux, идентификаторам спецификаций правил и т.д. – рис. 4).
- Анализ результатов статической верификации, полученных для конкретных спецификаций, модулей и т.д. – рис. 5.

Task	Kernel	Rule	Module	Entry point	Verifier	Error trace	KB Verdict	KB Tags
Task description BLAST, 15Gb	linux-3.12-rc1	106_1a	drivers/base/firmware_class.ko	ldv_main0	blast	...	False positive	env_gen
			drivers/char/ppdev.ko	ldv_main0	blast	...	False positive	multi_module
			drivers/hid/hid.ko	ldv_main1	blast	...	False positive	kernel_model
			drivers/hsi/clients/hsi_char.ko	ldv_main0	blast	...	False positive	env_gen
			drivers/iio/industrialio.ko	ldv_main0	blast	...	False positive	rule_model
			drivers/infiniband/core/ib_ucm.ko	ldv_main0	blast	...	False positive	init_global_var
			drivers/media/pci/ddbridge/ddbridge.ko	ldv_main0	blast	...	False positive	multi_module
			drivers/media/usb/pvrusb2/pvrusb2.ko	ldv_main5	blast	...	True positive	obsolete
			drivers/mtd/ubi/ubi.ko	ldv_main3	blast	...	False positive	pointer_analysis
				ldv_main4	blast	...	False positive	env_gen
			drivers/net/wireless/mac80211_hwsim.ko	ldv_main0	blast	...	True positive	new
			drivers/scsi/dpt_i2o.ko	ldv_main0	blast	...	False positive	init_global_var
			drivers/scsi/scsi_mod.ko	ldv_main0	blast	...	False positive	rule_model

Рис. 5. Анализ сообщений об ошибках, выдаваемых инструментом статической верификации BLAST при проверке спецификации правил, которые описывают корректную регистрацию USB-устройств класса Gadget (справа показаны результаты разметки сообщений об ошибках: True positive – выявлена ошибка, False positive – ложное сообщение об ошибке).

- Сравнительный анализ (например, можно сравнить результаты статической верификации, полученные для разных версий ядра ОС Linux, – рис. 6).

Kernel	Rule	Total changes	Safe → Unknown	Unsafe → Safe	Unsafe → Unknown	Unknown → Safe	Unknown → Unsafe
linux-3.12-rc1 → linux-3.13-rc1	39_7a	310	23	1	2	75	10

Рис. 6. Сравнение результатов статической верификации модулей ядра ОС Linux версий 3.12-rc1 и 3.13-rc1 (часть ошибок была исправлена – переходы из Unsafe, для большего количества модулей удалось доказать корректность – переходы в Safe или найти возможные ошибки – переходы в Unsafe).

С целью упростить анализ трасс ошибок LDV Analytics Center использует подкомпонент Error Trace Visualizer [33]. Данный подкомпонент визуализирует трассы ошибок, представленные в общем формате, в виде веб-страниц. Пользователю предоставляется удобная навигация по трассам ошибок и соответствующему им исходному коду модулей, ядра и

контрактных спецификаций. Пример визуализированной трассы ошибки приведен на рис. 7.

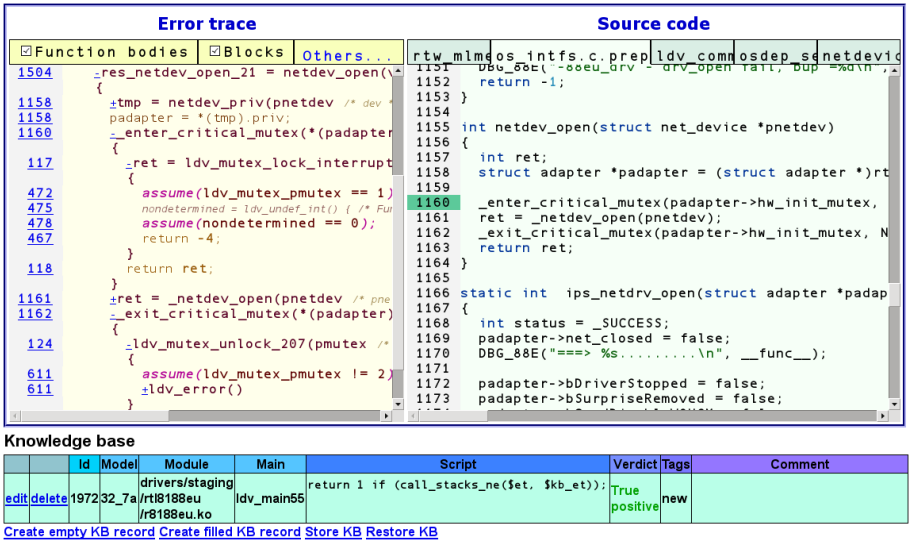


Рис. 7. Визуализированная трасса ошибки (выявлено нарушение спецификации правил корректного использования мьютексов в одном потоке).

Также на рис. 7 показан веб-интерфейс Knowledge Base – базы знаний LDV, еще одного подкомпонента LDV Analytics Center. Данный веб-интерфейс позволяет сохранять результаты анализа трасс ошибок инструментов статической верификации. Скрипты, входящие в Knowledge Base, автоматически размечают все новые загружаемые трассы, если они оказываются похожими по тем или иным критериям на ранее сохраненные трассы ошибок. Например, для трассы ошибки, приведенной на рис. 7, вердикт был вынесен автоматически на основе сравнения по идентификатору спецификации правила (*32_7a*), по модулю (*drivers/staging/rtl8188eu/r8188eu.ko*), по точке входа (*ldv_main55*) и по стеку вызываемых функций (*call_stacks_ne*) с трассой ошибкой, проанализированной для ядра ОС Linux более ранней версии.

Сравнение трасс по стеку вызываемых функций в настоящее время используется в Knowledge Base по умолчанию. Пользователь также может сравнивать трассы ошибок по деревьям вызываемых функций или задать свой собственный скрипт сравнения.

Для проведения анализа LDV Analytics Center предоставляет веб-интерфейс, что позволяет использовать его совместно нескольким пользователям.

Благодаря этому удается существенно сократить трудозатраты на анализ получаемых результатов статической верификации.

3.7. Выводы

Разработанная система статической верификации LDV Tools допускает конфигурирование на каждом из этапов своей работы. Пользователь может выбрать, каким образом разделить исходный код ядра и модулей и соответствующие ему команды сборки на объекты верификации; дополнить спецификацию π-модели окружения; модифицировать существующие и предоставить собственные спецификации правил; а также указать инструмент статической верификации, его конфигурацию и ограничения на ресурсы, которые он может использовать при проверке модулей ядра ОС Linux. Благодаря этому система статической верификации LDV Tools в полной мере реализует метод, предложенный в разделе 2.

4. Результаты практического применения Linux Driver Verification Tools

Конфигурируемая система статической верификации LDV Tools применяется на практике более четырех лет. За это время разработчикам LDV Tools удалось выявить более 150 ошибок в модулях ядра ОС Linux, а также определить наиболее существенные проблемы в системе статической верификации и используемых инструментах статической верификации.

4.1. Анализ выявленных ошибок в модулях ядра ОС Linux

На сегодняшний день с помощью LDV Tools было выявлено 150 ошибок, которые были признаны разработчиками ядра[5] [34]. На рис. 8 представлен график, который демонстрирует, как зависит количество исправлений данных ошибок в модулях ядра ОС Linux от версии ядра. В среднем для ядра версий от 2.6.31 (выпущено 9 сентября 2009) до 3.15-rc2 (выпущено 20 апреля 2014) в каждой версии было исправлено около 6 ошибок. При этом с течением времени количество исправлений ошибок увеличивается. Объясняется это тем, что за четыре года конфигурируемая система статической верификации LDV Tools достигла такого уровня развития, что разработчики применяют ее на практике все более и более активно, а также тем, что постепенно расширяется список проверяемых правил. Сейчас LDV Tools позволяет проверять более 40 правил корректного использования программного интерфейса ядра ОС Linux.

[5] Кроме ошибок, признанных разработчиками ядра ОС Linux, были выявлены ошибки, которые уже были исправлены к моменту обнаружения, а также ошибки в неподдерживаемых модулях ядра. Эти ошибки не входят в статистику, приводимую в данной статье.

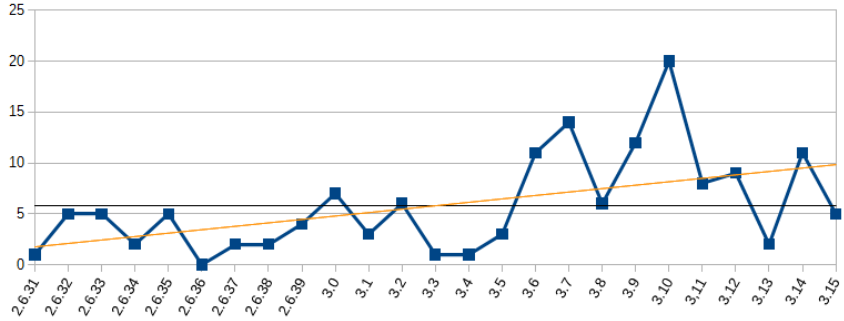


Рис. 8. Зависимость количества исправлений выявленных с помощью LDV Tools ошибок в модулях ядра ОС Linux от версии ядра (горизонтальная линия – среднее число, прямая наклонная линия – линейная регрессия).

Стоит отметить, что последние 1-2 года с помощью LDV Tools выявляется больше ошибок в модулях ядра ОС Linux, чем разработчики LDV Tools успевают проанализировать и сообщить авторам соответствующих модулей. Это явно свидетельствует о том, что разработанная конфигурируемая система статической верификации имеет достаточно высокий потенциал для выявления новых проблем в модулях.

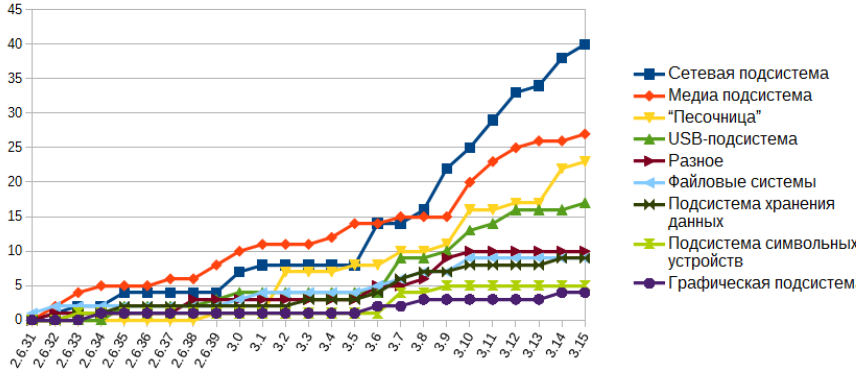


Рис. 9. Зависимость количества исправлений выявленных с помощью LDV Tools ошибок в модулях ядра ОС Linux от версии ядра для различных подсистем ядра.

На рис. 9 методом нарастающего итога представлена зависимость количества исправлений ошибок в модулях ядра ОС Linux от версии ядра для различных

подсистем ядра. Представленные графики демонстрируют, что в целом количество исправляемых ошибок в подсистемах сохраняется пропорциональным. Скачки на графиках объясняются изменением анализируемой кодовой базы. Например, для сетевой подсистемы на ядре версии 3.6 скачок произошел, потому что с помощью LDV Tools помимо модулей, которые представляют драйверы сетевых устройств, стали также верифицировать модули, входящие в основную сетевую подсистему ядра. В подсистему «песочница» попадают новые модули ядра, которые уже имеют достаточно высокий уровень качества, но по тем или иным техническим причинам пока не могут быть перенесены в одну из основных подсистем. Поскольку в новых модулях может выявляться большое количество ошибок, поведение соответствующего графика достаточно непредсказуемое.

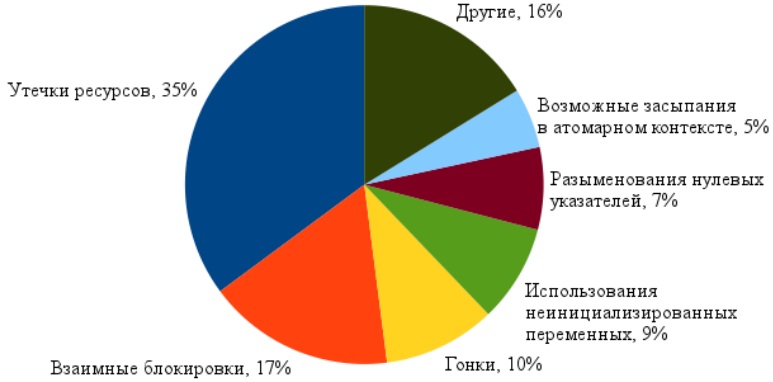


Рис. 10. Распределение возможных последствий выявленных ошибок в модулях ядра ОС Linux.

Диаграмма на рис. 10 демонстрирует распределение возможных последствий выявленных ошибок. Видно, что в большинстве случаев благодаря предложенным исправлениям удалось избежать утечек ресурсов[6]. На втором и третьем местах идут соответственно взаимные блокировки[7] (как следствие отсутствия освобождения примитивов синхронизации в потоке, в котором они были захвачены), гонки[8] (как следствие освобождения примитивов синхронизации в потоке, в котором они не были захвачены). Стоит отметить, что был исправлен ряд ошибок, которые имеют специфичные для модулей ядра ОС Linux проявления. Например, из-за возможных засыпаний в

[6] Определение утечки ресурса. http://cwe.mitre.org/data/definitions/401.html.

[7] Определение взаимной блокировки. http://cwe.mitre.org/data/definitions/833.html.

[8] Определение гонки. http://cwe.mitre.org/data/definitions/362.html.

атомарном контексте может существенно замедлиться работа ядра, а в некоторых случаях даже может зависнуть вся ОС.

Большинство ошибок (около 70%) были обнаружены в коде обработки ошибок, например, после неудачного выделения памяти или проблем при инициализации устройства. Это объясняется тем, что при использовании ОС и при тестировании подобные ситуации происходят достаточно редко, тогда как при статической верификации рассматриваются все возможные пути выполнения.

Практически все ошибки были выявлены с помощью инструмента статической верификации BLAST [30]. Это объясняется тем, что этот инструмент используется в LDV Tools по умолчанию, а также тем, что он был специально оптимизирован для анализа модулей ядра ОС Linux. 4 ошибки были выявлены с помощью инструмента статической верификации CPAchecker [31] (CPAchecker предоставляет дополнительные возможности по анализу функциональных указателей и битовой арифметики по сравнению с BLAST).

В нескольких случаях по результатам обсуждений с разработчиками модулей и ядра ОС Linux исправлений ошибок, найденных с помощью конфигурируемой системы статической верификации LDV Tools, пересматривался дизайн соответствующих подсистем ядра или модулей[9]. Зачастую выявляемые ошибки являлись наведенными вследствие, например, некорректной обработки ошибок[10] или отсутствия инициализации данных[11] (исходно обнаруженные проблемы – утечки памяти)

Другие подходы к обеспечению качества программных систем теоретически смогли бы обнаружить ошибки, выявленные с помощью LDV Tools. Часть этих ошибок могла бы проявиться при использовании ядра ОС Linux либо при проведении тестирования. Однако с одной стороны, для этого может потребоваться достаточно много времени, в том числе для разработки специфичных тестовых сценариев, поскольку большинство ошибок были обнаружены в коде обработки ошибок. А с другой стороны, для таких ошибок, как утечки ресурсов и гонки, бывает достаточно сложно точно идентифицировать их причины.

Большинство выявленных ошибок вероятно можно было бы обнаружить с помощью инструментов, реализующих легковесные подходы статического анализа. Для этого необходимо разработать подходящие для инструментов формальные описания правил корректного использования программного

[9] Обсуждение исправлений ряда ошибок в драйверах сетевых устройств. https://lkml.org/lkml/2012/8/14/128.

[10] Пример существенных исправлений в коде драйвера после обнаружения в нем ошибки. http://linuxtesting.ru/results/report?num=L0130.

[11] Пример исправления инициализации структур драйвера. http://linuxtesting.ru/results/report?num=L0116.

интерфейса ядра ОС Linux, запустить проверку и провести анализ результатов. При этом часть ошибок может быть пропущена (например, несколько десятков ошибок были выявлены при проведении межпроцедурного анализа и при анализе потока управления со сложными зависимостями, что ограниченно поддерживается в легковесных подходах статического анализа), а анализ результатов может быть затруднен из-за большого количества ложных сообщений об ошибках.

4.2. Анализ причин ложных сообщений об ошибках

При проведении экспериментов, результаты которых приведены в данном и трех последующих подразделах, конфигурируемая система статической верификации LDV Tools версии 0.5 запускалась на компьютере с четырехядерным процессором с частотой 3.4 GHz (Intel Core i7-2600), 16 гигабайтами оперативной памяти и ОС Ubuntu 12.04 (ядро ОС Linux версии 3.5, 64-битная архитектура). Для проверки использовался инструмент статической верификации BLAST версии 2.7.2 в конфигурации, используемой по умолчанию. На максимальное количество времени и оперативной памяти, которые мог использовать BLAST на решение одной верификационной задачи, были установлены ограничения 15 минут и 15 гигабайт соответственно.

В табл. 1 приведено распределение ложных сообщений об ошибках по их причинам для трех спецификаций правил. Для составления этого распределения были проанализированы результаты статической верификации всех модулей ядра ОС Linux версии 3.12-rc1 (примерно 4200 модулей[12]). По таблице видно, что модулей, для которых были выданы ложные сообщения об ошибках, не так много относительно общего количества анализируемых модулей (в среднем около 1,5%). Основными причинами ложных сообщений об ошибках являются:

- Нехватка при анализе кода взаимосвязанных модулей, например, когда анализируемый модуль вызывает функции, определенные в других модулях.
- Неточная модель окружения, например, вызовы обработчиков модулей в неправильном порядке.
- Недостаточно точный анализ инструмента статической верификации BLAST, например, вследствие неполноты анализа алиасов.

[12] В LDV Tools для одного модуля может генерироваться несколько моделей окружения. В табл. 1 и далее числа приведены с учетом всех моделей окружения для модулей.

Спецификация правила Причина ложных сообщений об ошибках	Корректное использование мьютексов в одном потоке[13]	Корректное выделение и освобождение блоков-запросов для USB-устройств[14]	Корректная регистрация USB-устройств класса Gadget[15]	Итого
Нехватка при анализе кода взаимосвязанных модулей	4	29	5	38
Неточная модель окружения	24	43	7	74
Неточная спецификация правил	18	6	3	27
Недостаточно точный анализ инструмента статической верификации	17	34	5	56
Итого	63	112	20	195

Табл. 1. Распределение ложных сообщений об ошибках по их причинам для трех спецификаций правил и всех модулей ядра ОС Linux 3.12-rc1.

4.3. Анализ причины пропуска ошибок

Методы статической верификации изначально были нацелены на выявление всех возможных ошибок искомого вида или на доказательство корректности анализируемых программных систем относительно проверяемых правил. Тем не менее, по тем или иным причинам возможен пропуск ошибок.

Для оценки количества пропущенных ошибок конфигурируемая система статической верификации LDV Tools запускалась на 34 модулях ядра ОС Linux разных версий, в которых имелись известные нарушения правил корректного использования программного интерфейса ядра. С помощью инструмента статической верификации BLAST удалось обнаружить 16 из этих ошибок, с помощью CPAchecker[16] – 14. BLAST нашел все те же ошибки,

13 http://forge.ispras.ru/issues/1940.

14 http://forge.ispras.ru/issues/3233.

15 http://forge.ispras.ru/issues/2742.

16 Ревизия SVN 8244, конфигурация, используемая по умолчанию.

которые обнаружил CPAchecker. Для обнаружения 2 ошибок, которые нашел BLAST, CPAchecker не хватило отведенного времени.

Основные причины пропуска остальных ошибок у использованных инструментов статической верификации следующие: недостаток в верификационных задачах кода взаимосвязанных модулей (5), недостаточно точная модель окружения (5), недостаточно точная спецификация правил (3), ошибки в инструментах статической верификации (BLAST: 4, CPAchecker: 3), нехватка памяти (BLAST: 1) и нехватка времени (CPAchecker: 2).

4.4. Анализ времени верификации модулей ядра ОС Linux

Эксперименты показали, что на проверку всех драйверов, которые входят в состав ядра ОС Linux 3.12-rc1 и которые могут быть представлены в виде модулей (примерно 3300), по одной спецификации правил в среднем потребовалось около 25 часов процессорного времени. На проверку всех модулей ядра – около 36 часов процессорного времени. Примерно 70% всего времени анализа заняла работа инструмента статической верификации.

Стоит отметить, что в настоящее время конфигурируемая система статической верификации LDV Tools не работает в параллельном режиме (поскольку инструментам статической верификации требуется большое количество оперативной памяти), поэтому количество ядер процессора практически не влияет на общее время верификации.

4.5. Анализ причин неуспешного завершения работы LDV Tools

Из всех модулей ядра ОС Linux 3.12-rc1 со всеми сгенерированными моделями окружения не удалось верифицировать около 800 (примерно 15% от общего количества) по следующим причинам: ошибки в Command Stream Divider (около 12% от 800), ошибки в Driver Environment Generator (около 11%), ошибки в Rule Instrumentor (около 19%), ошибки в BLAST (около 29%), превышение допустимого ограничения по памяти (около 22%) и по времени (около 7%) у BLAST.

4.6. Возможности LDV Tools для сравнения инструментов статической верификации и их конфигураций

Конфигурируемая система статической верификации LDV Tools позволяет сравнивать инструменты статической верификации и их конфигурации на такой большой, сложной и динамически развивающейся программной системе, как модули ядра ОС Linux [35, 36]. Благодаря этому для проведения статической верификации модулей ядра ОС Linux с целью выявления ошибок удается подобрать инструменты и их конфигурации оптимальным образом с точки зрения уменьшения пропущенных ошибок, количества ложных сообщений об ошибках и потребляемых ресурсов. Кроме того, удается выявить препятствия применения инструментов статической верификации на

практике, например, что в инструменте есть некоторые ошибки или выдаваемые им трассы ошибки содержат недостаточно информации для их наглядной визуализации.

Несколько последних лет с помощью конфигурируемой системы статической верификации LDV Tools готовился набор верификационных задач DeviceDrivers64 для ежегодных соревнований, проходящих в рамках мероприятий ETAPS/Competition on Software Verification [12, 13]. При подготовке набора разработчики LDV Tools отобрали те верификационные задачи, при решении которых были выявлены нарушения правил корректного использования программного интерфейса ядра или инструментам статической верификации потребовалось большое количество ресурсов. С 2013 года DeviceDrivers64 стал самым большим набором по количеству задач (более половины от количества всех задач).

В табл. 2 приведены информация о количестве верификационных задач в наборе DeviceDrivers64, победители ежегодных соревнований ETAPS/Competition on Software Verification на наборе DeviceDrivers64 и их итоговые результаты (набранное количество баллов и время работы). Данные результаты показывают, какие инструменты статической верификации необходимо использовать при проведении проверки модулей ядра ОС Linux.

Набор верификационных задач DeviceDrivers64	1-е место	2-е место	3-е место
SV-COMP 2012 41 задача, максимально 66 баллов	BLAST 55 баллов *1400 с*	CPAchecker-Memo 49 баллов *500 с*	SATabs 32 баллов *3200 с*
SV-COMP 2013 1237 задач, максимально 2419 баллов	UFO 2408 баллов *2500 с*	CPAchecker-Explicit 2340 баллов *9700 с*	BLAST 2338 баллов *2400 с*
SV-COMP 2014 1428 задач, максимально 2766 баллов	BLAST 2682 баллов *13000 с*	UFO 2642 баллов *5700 с*	FrankenBit 2639 баллов *3000 с*

Табл. 2. Победители ежегодных соревнований ETAPS/Competition on Software Verification на наборе верификационных задач DeviceDrivers64.

4.7. Выводы

Анализ результатов практического применения конфигурируемой системы статической верификации LDV Tools наглядным образом продемонстрировал, что инструменты статической верификации могут быть успешно использованы для выявления нарушений правил корректного использования

программного интерфейса ядра ОС Linux в модулях. Также он позволил определить наиболее существенные проблемы в LDV Tools:

- Недостаток в верификационных задачах кода взаимосвязанных модулей.
- Недостаточно точная спецификация π-модели окружения.
- Недостаточно точные спецификации правил.
- Ошибки в компонентах и подкомпонентах LDV Tools.

и используемых инструментах статической верификации:

- Недостаточно точный анализ алиасов, функциональных указателей, битовой арифметики и т.д.
- Ошибки в инструментах, например, при разборе файлов с исходным кодом, подаваемых им на вход.
- Чрезмерное потребление ресурсов.

Благодаря решению данных проблем в будущем можно существенно сократить количество пропущенных ошибок и ложных сообщений об ошибках, избежать неуспешного завершения работы системы статической верификации, а также оптимизировать время ее работы.

5. Заключение

Современные методы и инструменты статической верификации уже позволяют доказать выполнимость специфицированных свойств для средних по размеру программных систем за приемлемое время. Благодаря этому с помощью данных инструментов можно выявить все нарушения правил корректного использования программного интерфейса ядра ОС в модулях. Опыт статической верификации большого количества сложных и активно развивающихся модулей ядра ОС Linux показал важность профессионального решения большого количества инженерных задач, без которого не удается использовать преимущества даже самых передовых методов статической верификации.

В предложенном в работе методе статической верификации можно конфигурировать процесс проверки на каждом из его этапов, благодаря чему реализующая метод конфигурируемая система статической верификации LDV Tools продолжает успешно развиваться, а кроме того уже позволила выявить более 150 критичных ошибок в модулях ядра ОС Linux.

В работе было показано, что модули ядра ОС Linux являются уникальным полигоном для испытания инструментов статической верификации. Это позволяет, с одной стороны, подобрать для проведения верификации оптимальные по ряду критериев инструменты и их конфигурации. С другой

стороны, положительные результаты такого комплексного подхода являются новой движущей силой для развития самих методов и инструментов статической верификации.

В ходе практического применения конфигурируемой системы статической верификации LDV Tools были выявлены проблемы, которые определяют основные направления дальнейшего развития:

- Анализ групп взаимосвязанных модулей.
- Дополнение спецификации π-модели окружения.
- Уточнение существующих спецификаций правил.
- Исправление ошибок в компонентах и подкомпонентах LDV Tools.

Кроме того, планируются развивать следующие направления:

- Продолжение анализа изменений в модулях ядра ОС Linux с целью выявления новых критичных правил корректного использования программного интерфейса ядра и разработка новых спецификаций правил.
- Интеграция новых инструментов статической верификации и подбор оптимальных конфигураций инструментов для проверки тех или иных правил.
- Многоаспектная статическая верификация модулей ядра ОС Linux для выявления всех возможных нарушений нескольких проверяемых правил за один запуск инструмента статической верификации.
- Статическая верификация модулей ядра для различных архитектур, таких как ARM, MIPS, S/390 и PowerPC, а также наиболее часто используемых конфигураций ядра.
- Использование и развитие методов регрессионной статической верификации для проверки изменений в коде модулей и ядра ОС Linux.
- Распараллеливание статической верификации модулей ядра ОС Linux.
- Применение предложенного метода статической верификации модулей ядра ОС Linux для проверки компонентов ядра других ОС.

Разработчикам инструментов статической верификации предлагается разрабатывать более точные методы анализа, исправлять ошибки в инструментах, а также оптимизировать реализацию методов с целью уменьшения потребления ресурсов.

Текущие достижения, проблемы, перспективы и планы регулярно обсуждаются на семинарах Linux Driver Verification Workshop и мероприятиях ETAPS/Competition on Software Verification, в которых наиболее активную

роль играют Институт системного программирования РАН, Москва (разработчик конфигурируемой системы статической верификации LDV Tools) и Университет Пассау, Германия (основной разработчик инструмента статической верификации CPAchecker).

Благодарности

Авторы выражают признательность П. Андрианову, В. Гратинскому, В.А. Захарову, М. Макиенко, В. Морданю, О. Стрикову, А. Страху, П. Шведу и И. Щепеткову за активное участие в проектировании и разработке конфигурируемой системы статической верификации LDV Tools.

Литература

[1] Chou A., Yang J., Chelf B., Hallem S., Engler D. An empirical study of operating system errors. Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP), pp. 73-88, 2001. doi: 10.1145/502034.502042

[2] Swift M., Bershad B., Levy H. Improving the reliability of commodity operating systems. Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP), pp. 73-88, 2003. doi: 10.1145/502034.502042

[3] Palix N., Thomas G., Saha S., Calves C., Lawall J., Muller G. Faults in Linux: ten years later. Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 305-318, 2011. doi: 10.1145/1950365.1950401

[4] Мутилин В.С., Новиков Е.М., Хорошилов А.В. Анализ типовых ошибок в драйверах операционной системы Linux. Труды Института системного программирования РАН, т. 22, стр. 349-374, 2012.

[5] Ball T., Bounimova E., Cook B., Levin V., Lichtenberg J., McGarvey C., Ondrusek B., Rajamani S. K., Ustuner A. Thorough static analysis of device drivers. Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys), pp. 73-85, 2006. doi: 10.1145/1218063.1217943

[6] Glass R.L. Facts and fallacies of software engineering. Addison-Wesley Professional, 2002.

[7] Engler D., Chelf B., Chou A., Hallem S. Checking system rules using system-specific, programmer-written compiler extensions. Proceedings of the 4th conference on Symposium on Operating System Design & Implementation (OSDI), vol. 4, pp. 1-16, 2000.

[8] Аветисян А., Белеванцев А., Бородин А., Несов В. Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ. Труды Института системного программирования РАН, т. 21, стр. 23-38, 2011.

[9] Lawall J. L., Brunel J., Palix N., Rydhof H. R., Stuart H., Muller G. WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code. Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 43—52, 2009. doi: 10.1109/DSN.2009.5270354

[10] Мандрыкин М.У., Мутилин В.С., Хорошилов А.В. Введение в метод CEGAR — уточнение абстракции по контрпримерам. Труды Института системного программирования РАН, т. 24, стр. 219-292, 2013.

[11] Engler D., Musuvathi M. Static analysis versus model checking for bug finding. Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), LNCS, vol. 2937, pp. 191-210, 2004. doi: 10.1007/978-3-540-24622-0_17
[12] Beyer D. Competition on Software Verification. Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp. 504-524, 2012. doi: 10.1007/978-3-642-28756-5_38
[13] Beyer D. Second Competition on Software Verification. Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7795, pp. 594-609, 2013. doi: 10.1007/978-3-642-36742-7_43
[14] Мандрыкин М.У., Мутилин В.С., Новиков Е.М., Хорошилов А.В. Обзор инструментов статической верификации Си программ в применении к драйверам устройств операционной системы Linux. Труды Института системного программирования РАН, т. 22, стр. 293-326, 2012.
[15] Corbet J., Kroah-Hartman G., McPherson A. Linux kernel development. How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It. http://go.linuxfoundation.org/who-writes-linux-2012, 2012.
[16] Ball T., Levin V., Rajamani S.K. A decade of software model checking with SLAM. Communications of the ACM, vol. 54, issue 7, pp. 68-76, 2011. doi: 10.1145/1965724.1965743
[17] Ball T., Bounimova E., Kumar R., Levin V. SLAM2: Static driver verification with under 4% false alarms. Proceedings of the 10th International Conference on Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 35-42, 2010.
[18] Ball T., Rajamani S.K. SLIC: A specification language for interface checking of C. Technical Report MSR-TR-2001-21, Microsoft Research, 2001.
[19] Ball T., Bounimova E., Levin V., Kumar R., Lichtenberg J. The Static Driver Verifier Research Platform. Proceedings of the 22nd International Conference on Computer Aided Verification (CAV), LNCS, vol. 6174, pp. 119–122, 2010. doi: 10.1007/978-3-642-14295-6_11
[20] Witkowski T., Blanc N., Kroening D., Weissenbacher G. Model checking concurrent Linux device drivers. Proceedings of the 22nd IEEE/ACM international conference on Automated Software Engineering (ASE), pp. 501-504, 2007. doi: 10.1145/1321631.1321719
[21] Post H., Küchlin W. Integrated static analysis for Linux device driver verification. Proceedings of the 6th International Conference on Integrated Formal Methods (IFM), LNCS, vol. 4591, pp. 518-537, 2007. doi: 10.1007/978-3-540-73210-5_27
[22] Clarke E., Kroening D., Lerda F. A tool for checking ANSI-C programs. Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 2988, pp. 168-176, 2004. doi: 10.1007/978-3-540-24730-2_15
[23] Clarke E., Kroening D., Sharygina N., Yorav K. SATABS: SAT-based predicate abstraction for ANSI-C. Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 3440, pp. 570–574, 2005. doi: 10.1007/978-3-540-31980-1_40
[24] Мутилин В.С. Верификация драйверов операционной системы Linux при помощи предикатных абстракций. Диссертация на соискание ученой степени кандидата физико-математических наук, Институт системного программирования РАН, 2012.

[25] Захаров И.С., Мутилин В.С., Новиков Е.М., Хорошилов А.В. Моделирование окружения драйверов устройств операционной системы Linux. Труды Института системного программирования РАН, т. 25, стр. 85-112, 2013.
[26] Новиков Е.М. Развитие метода контрактных спецификаций для верификации модулей ядра операционной системы Linux. Диссертация на соискание ученой степени кандидата физико-математических наук, Институт системного программирования РАН, 2013.
[27] Necula G. C., McPeak S., Rahul S.P., Weimer W. CIL: Intermediate language and tools for analysis and transformation of C programs. Proceedings of the 11th International Conference on Conference on Compiler Construction, LNCS, vol. 2304, pp. 213-228, 2002. doi: 10.1007/3-540-45937-5_16
[28] Мутилин В.С., Новиков Е.М., Страх А.В., Хорошилов А.В., Швед П.Е. Архитектура Linux Driver Verification. Труды Института системного программирования РАН, т. 20, стр. 163-187, 2011.
[29] Khoroshilov A., Mutilin V., Novikov E., Zakharov I. Modeling environment for static verification of Linux kernel modules. Proceedings of the 11th International Andrei Ershov Memorial Conference (PSI), 2014.
[30] Beyer D., Henzinger T., Jhala R., Majumdar R. The software model checker BLAST: Applications to software engineering. International Journal on Software Tools for Technology Transfer (STTT), vol. 5, pp. 505-525, 2007. doi: 10.1007/s10009-007-0044-z
[31] Beyer D., Keremoglu M.E. CPAchecker: A tool for configurable software verification. In Proceedings of the 23rd International Conference on Computer Aided Verification (CAV), LNCS, vol. 6806, pp. 184–190, 2011. doi: 10.1007/978-3-642-22110-1_16
[32] Albarghouthi A., Li Y., Gurfinkel A., Chechik M. UFO: A framework for abstraction and interpolation-based software verification. Proceedings of the 24th International Conference on Computer Aided Verification, LNCS, vol. 7358, pp. 672-678, 2012. doi: 10.1007/978-3-642-31424-7_48
[33] Новиков Е.М. Упрощение анализа трасс ошибок инструментов статического анализа кода. Сборник научных трудов научно-практической конференции Актуальные Проблемы Программной Инженерии (АППИ), стр. 215-221, 2011.
[34] Список ошибок, выявленных в модулях ядра ОС Linux с помощью конфигурируемой системы статической верификации Linux Driver Verification Tools. http://linuxtesting.org/results/ldv.
[35] Мандрыкин М.У., Мутилин В.С., Новиков Е.М., Хорошилов А.В., Швед П.Е. Использование драйверов устройств операционной системы Linux для сравнения инструментов статической верификации. Программирование, т. 38, н. 5, стр. 54-71, 2012.
[36] Бейер Д., Петренко А.К. Верификация драйверов операционной системы Linux. Труды Института системного программирования РАН, т. 23, стр. 405-412, 2012.

Configurable Toolset for Static Verification of Operating Systems Kernel Modules

I.S. Zakharov, M.U. Mandrykin, V.S. Mutilin, E.M. Novikov, A.K. Petrenko, A.V. Khoroshilov
Institute for System Programming of RAS (ISP RAS), Moscow, Russia
{ilja.zakharov, mandrykin, mutilin, novikov, petrenko, khoroshilov}@ispras.ru

Abstract. An operating system (OS) kernel is a critical software regarding to reliability and efficiency. Quality of a modern OSs kernel is high enough. Another situation is with kernel modules, e.g. device drivers, which due to various reasons have a significantly lower level of quality. One of the most critical and widespread bugs in kernel modules are violations of rules of correct usage of a kernel API. One can identify all such the violations in modules or prove their correctness with help of static verification tools which needs contract specifications describing formally obligations of a kernel and modules with respect to each other. The paper considers existing methods and toolsets for static verification of kernel modules of different OSs. It suggests a new method for static verification of Linux kernel modules that allows to configure checking at each of its stages. The paper shows how this method can be adapted for checking kernel components of other OSs. It describes an architecture of a configurable toolset for static verification of Linux kernel modules, which implements the proposed method, and demonstrates results of its practical application. Directions of further development are considered in conclusion.

Keywords: operating system kernel; kernel module; software quality; static verification; contract specification; environment model; specification of rule of correct usage of API.

References

[1] Chou A., Yang J., Chelf B., Hallem S., Engler D. An empirical study of operating system errors. Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP), pp. 73-88, 2001. doi: 10.1145/502034.502042

[2] Swift M., Bershad B., Levy H. Improving the reliability of commodity operating systems. Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP), pp. 73-88, 2003. doi: 10.1145/502034.502042

[3] Palix N., Thomas G., Saha S., Calves C., Lawall J., Muller G. Faults in Linux: ten years later. Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 305-318, 2011. doi: 10.1145/1950365.1950401

[4] Mutilin V.S., Novikov E.M., Khoroshilov A.V. Analiz tipovykh oshibok v drajverakh operatsionnoj sistemy Linux [Analysis of typical faults in Linux operating system drivers]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 22, pp. 349-374, 2012 (in Russian).

[5] Ball T., Bounimova E., Cook B., Levin V., Lichtenberg J., McGarvey C., Ondrusek B., Rajamani S. K., Ustuner A. Thorough static analysis of device drivers. Proceedings of

the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys), pp. 73-85, 2006. doi: 10.1145/1218063.1217943
[6] Glass R.L. Facts and fallacies of software engineering. Addison-Wesley Professional, 2002.
[7] Engler D., Chelf B., Chou A., Hallem S. Checking system rules using system-specific, programmer-written compiler extensions. Proceedings of the 4th conference on Symposium on Operating System Design & Implementation (OSDI), vol. 4, pp. 1-16, 2000.
[8] Avetisyan A., Belevantsev A., Borodin A., Nesov V. Ispol'zovanie staticheskogo analiza dlya poiska uyazvimostej i kriticheskikh oshibok v iskhodnom kode programm [Using static analysis for finding security vulnerabilities and critical errors in source code]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 21, pp. 23-38, 2011 (in Russian).
[9] Lawall J. L., Brunel J., Palix N., Rydhof H. R., Stuart H., Muller G. WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code. Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 43—52, 2009. doi: 10.1109/DSN.2009.5270354
[10] Mandrykin M.U., Mutilin V.S., Khoroshilov A.V. Vvedenie v metod CEGAR — utochnenie abstraktsii po kontrprimeram [Introduction to CEGAR — Counter-Example Guided Abstraction Refinement]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 24, pp. 219-292, 2013 (in Russian).
[11] Engler D., Musuvathi M. Static analysis versus model checking for bug finding. Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), LNCS, vol. 2937, pp. 191-210, 2004. doi: 10.1007/978-3-540-24622-0_17
[12] Beyer D. Competition on Software Verification. Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp. 504-524, 2012. doi: 10.1007/978-3-642-28756-5_38
[13] Beyer D. Second Competition on Software Verification. Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7795, pp. 594-609, 2013. doi: 10.1007/978-3-642-36742-7_43
[14] Mandrykin M.U., Mutilin V.S., Novikov E.M., Khoroshilov A.V. Obzor instrumentov staticheskoj verifikatsii Si programm v primenenii k drajveram ustrojstv operatsionnoj sistemy Linux [Static verification tools for C programs and Linux device drivers: A survey]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 22, pp. 293-326, 2012 (in Russian).
[15] Corbet J., Kroah-Hartman G., McPherson A. Linux kernel development. How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It. http://go.linuxfoundation.org/who-writes-linux-2012, 2012.
[16] Ball T., Levin V., Rajamani S.K. A decade of software model checking with SLAM. Communications of the ACM, vol. 54, issue 7, pp. 68-76, 2011. doi: 10.1145/1965724.1965743
[17] Ball T., Bounimova E., Kumar R., Levin V. SLAM2: Static driver verification with under 4% false alarms. Proceedings of the 10th International Conference on Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 35-42, 2010.
[18] Ball T., Rajamani S.K. SLIC: A specification language for interface checking of C. Technical Report MSR-TR-2001-21, Microsoft Research, 2001.
[19] Ball T., Bounimova E., Levin V., Kumar R., Lichtenberg J. The Static Driver Verifier Research Platform. Proceedings of the 22nd International Conference on Computer

Aided Verification (CAV), LNCS, vol. 6174, pp. 119–122, 2010. doi: 10.1007/978-3-642-14295-6_11
[20] Witkowski T., Blanc N., Kroening D., Weissenbacher G. Model checking concurrent Linux device drivers. Proceedings of the 22nd IEEE/ACM international conference on Automated Software Engineering (ASE), pp. 501-504, 2007. doi: 10.1145/1321631.1321719
[21] Post H., Küchlin W. Integrated static analysis for Linux device driver verification. Proceedings of the 6th International Conference on Integrated Formal Methods (IFM), LNCS, vol. 4591, pp. 518-537, 2007. doi: 10.1007/978-3-540-73210-5_27
[22] Clarke E., Kroening D., Lerda F. A tool for checking ANSI-C programs. Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 2988, pp. 168-176, 2004. doi: 10.1007/978-3-540-24730-2_15
[23] Clarke E., Kroening D., Sharygina N., Yorav K. SATABS: SAT-based predicate abstraction for ANSI-C. Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 3440, pp. 570–574, 2005. doi: 10.1007/978-3-540-31980-1_40
[24] Mutilin V.S. Verifikatsiya drajverov operatsionnoj sistemy Linux pri pomoshhi predikatnykh abstraktsij [Linux drivers verification with help of predicate abstractions]. Dissertatsiya na soiskanie uchenoj stepeni kandidata fiziko-matematicheskikh nauk, ISP RAN [PhD thesis, ISP RAS], 2012 (in Russian).
[25] Zakharov I.S., Mutilin V.S., Novikov E.M., Khoroshilov A.V. Modelirovanie okruzheniya drajverov ustrojstv operatsionnoj sistemy Linux [Environment modeling of Linux operating system device drivers]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 25, pp. 85-112, 2013 (in Russian).
[26] Novikov E.M. Razvitie metoda kontraktnykh spetsifikatsij dlya verifikatsii modulej yadra operatsionnoj sistemy Linux. [Development of contract specifications method for verification of Linux kernel modules]. Dissertatsiya na soiskanie uchenoj stepeni kandidata fiziko-matematicheskikh nauk, ISP RAN [PhD thesis, ISP RAS], 2013 (in Russian).
[27] Necula G. C., McPeak S., Rahul S.P., Weimer W. CIL: Intermediate language and tools for analysis and transformation of C programs. Proceedings of the 11th International Conference on Conference on Compiler Construction, LNCS, vol. 2304, pp. 213-228, 2002. doi: 10.1007/3-540-45937-5_16
[28] Mutilin V.S., Novikov E.M., Strakh A.V., Khoroshilov A.V., Shved P.E. Arkhitektura Linux Driver Verification [Linux Driver Verification architecture]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 20, pp. 163-187, 2011 (in Russian).
[29] Khoroshilov A., Mutilin V., Novikov E., Zakharov I. Modeling environment for static verification of Linux kernel modules. Proceedings of the 11th International Andrei Ershov Memorial Conference (PSI), 2014.
[30] Beyer D., Henzinger T., Jhala R., Majumdar R. The software model checker BLAST: Applications to software engineering. International Journal on Software Tools for Technology Transfer (STTT), vol. 5, pp. 505-525, 2007. doi: 10.1007/s10009-007-0044-z
[31] Beyer D., Keremoglu M.E. CPAchecker: A tool for configurable software verification. In Proceedings of the 23rd International Conference on Computer Aided Verification (CAV), LNCS, vol. 6806, pp. 184–190, 2011. doi: 10.1007/978-3-642-22110-1_16
[32] Albarghouthi A., Li Y., Gurfinkel A., Chechik M. UFO: A framework for abstraction and interpolation-based software verification. Proceedings of the 24th International

Conference on Computer Aided Verification, LNCS, vol. 7358, pp. 672-678, 2012. doi: 10.1007/978-3-642-31424-7_48
[33] Novikov E.M. Uproshhenie analiza trass oshibok instrumentov staticheskogo analiza koda. [Simplification of static verifier traces analysis]. Aktual'nye Problemy Programmnoj Inzhenerii [Actual Problems of Software Engineering], pp. 215-221, 2011 (in Russian).
[34] List of bugs found in Linux kernel modules with help of configurable toolset for static verification Linux Driver Verification Tools. http://linuxtesting.org/results/ldv.
[35] Mandrykin M.U., Mutilin V.S., Novikov E.M., Khoroshilov A.V., Shved P.E. Using Linux device drivers for static verification tools benchmarking. Programming and Computer Software, vol. 38, n. 5, стр. 245-256, 2012.
[36] Beyer D. Petrenko A. Linux Driver Verification. Proceedings of the 5th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies, LNCS, vol. 7610, pp. 1-6, 2012. doi: 10.1007/s10009-007-0044-z

Обход неизвестного графа коллективом автоматов

Игорь Бурдонов, Александр Косачев
{igor, kos}@ispras.ru

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 графу переходов, свойства которого нужно исследовать. За последние годы размер реально используемых систем и сетей и, следовательно, размер их моделей и, следовательно, размер исследуемых графов 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 (несколькими параллельно работающими компьютерами с достаточной суммарной памятью). Решению этой задачи посвящена данная работа.

Ключевые слова: исследование графа, обход графа, конечный автомат, расширенный автомат, взаимодействующие автоматы, параллельная обработка, распределенные системы, тестирование.

1. Введение

Задача обхода неизвестного ориентированного графа автоматом используется во многих приложениях. В данной статье подразумевается тестирование детерминированных систем: граф — это граф автомата тестируемой системы, автомат на графе — тестирующая система, а проход по дуге — это тестовое воздействие и наблюдение результата [[1]]. В качестве практического примера можно привести работу [[8]], 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.

Нижняя оценка времени обхода для одного или ограниченного числа автоматов равна $\Omega(nm)$, где n — число вершин графа, а m — число дуг [[2]]. Если число автоматов не ограничено, нижняя оценка $\Omega(m)$.

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. Если автомат один, то существуют алгоритмы с оценкой $\Theta(nm)$ [[2]]. Если число автоматов больше одного, но ограничено, то распараллеливание ускоряет обход, но не меняет порядок времени обхода в наихудшем случае [[7]]. Если число автоматов не ограничено и все вычисления автоматов и передачи сообщений между ними выполняются не дольше (по порядку), чем проход дуги графа, то существуют алгоритмы с оценкой $\Theta(m)$. Это утверждение доказывается в данной статье в конце п.5.3.

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: автомат может писать/читать из текущей вершины символы конечного алфавита. Такой подход может применяться, например, для сети интернета, когда вершина — это узел сети, а проход по дуге — передача сообщения между узлами.

Для одного конечного автомата известен алгоритм с оценкой $\Theta(nm+n^2 loglogn)$ [[4]], а при повторном обходе $\Theta(nm+n^2 l(n))$, где $l(n)$ — число логарифмирований, при котором достигается соотношение $1 \le log(log...(n)...) < 2$ [[5]]. Отличие от нижней оценки $\Omega(nm)$ объясняется тем, что автомату бывает нужно «вернуться» в начало только что пройденной дуги.

Если конечных автоматов несколько, каждый из них может читать пометки в вершинах, оставленные другими автоматами, и обмениваться с ними сообщениями. Для двух автоматов оценка равна уже $\Theta(nm)$. Эти автоматы двигаются синхронно, кроме случая прохода по новой дуге. В этом случае один автомат (первый) идёт по дуге, а второй автомат остаётся на месте, ожидая сообщения от первого автомата. В этом сообщении указывается, нужно ли второму автомату оставаться на месте, поджидая первый автомат (для возвращения первого автомата в начало только что пройденной дуги), или, наоборот, двигаться вперёд, догоняя первый автомат (возвращения по дуге не требуется).

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 конечных 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.

Этот подход впервые был применён в нашей работе [[9]], где предложен алгоритм обхода с оценкой $\boldsymbol{O}(m+n^2)$. В настоящей статье мы предлагаем алгоритм с улучшенной оценкой.

2. Формализация автоматов на графе

2.1. Номер выходящей дуги

Автомат на графе, начиная с начальной вершины графа, двигается по дугам, проходя некоторый маршрут. Обход выполнен, если по каждой дуге прошёл хотя бы один автомат. Когда автомат находится в вершине и хочет пройти по выходящей из неё дуге, он эту дугу должен указать. Для этого используется нумерация дуг, выходящих из вершины. Для детерминированных систем разные дуги, выходящие из одной вершины, имеют разные номера. Дуги, выходящие из вершины v нумеруются от 1 до полустепени выхода вершины (числа выходящих дуг): $1..d_{out}(v)$. Автомат указывает номер выходящей дуги. Однако для того, чтобы выходной алфавит автомата оставался конечным, полустепень выхода вершин графа должна быть ограничена сверху некоторой константой r.

Это ограничение легко снимается, если в каждой вершине v добавить $k=[d_{out}(v)/(r\text{-}1)]+1$ ячеек памяти, каждая из которых ассоциируется с r-1 выходящими дугами, кроме последней ячейки, которая может ассоциироваться с меньшим числом дуг. Эти ячейки связываются в цикл, который будем называть v-циклом. Для автомата добавляется *внутреннее* перемещение на следующую по v-циклу ячейку, которое автоматически настраивает граф на следующую порцию выходящих дуг. Тем самым, автомату, находящемуся в некоторой ячейке нужно идентифицировать внешнюю дугу только в пределах тех r-1 дуг, которые ассоциированы с этой ячейкой, или меньшего числа дуг для последней ячейки. При *внешнем* перемещении по дуге $(v`,v)$ автомат попадает в первую ячейку v-цикла (рис.1).

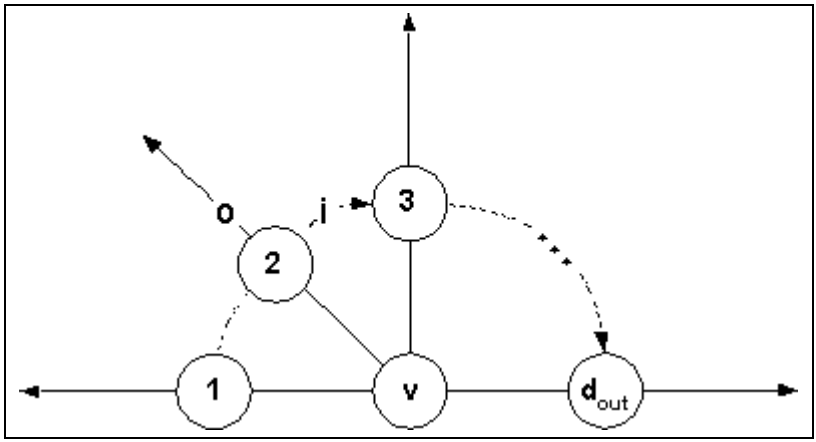


Рис 1. Вершина v и v-цикл ячеек (для r=2)

Это эквивалентно соответствующему преобразованию графа, когда каждая вершина *v*, в которой полустепень выхода больше *r*, заменяется на *v*-цикл вершин. В результате такого преобразования получается граф, в котором полустепень выхода каждой вершины *v* ограничена *r*. Дуги, заканчивавшиеся в вершине *v*, теперь будут заканчиваться в первой вершине *v*-цикла, а остальные вершины *v*-цикла отличаются тем, что в каждую из них входит ровно одна (внутренняя) дуга. В дальнейшем для простоты изложения мы будем рассматривать только такие графы, в которых из каждой вершины выходит не более *r* дуг.

Заметим, что замена вершины *v* на *v*-цикл вершин можно понимать как создание одним автоматом, оказавшимся в вершине *v*, цепочки автоматов — по одному на каждые *r*-1 выходящих дуг или меньшего числа дуг для последнего автомата в цепочке.

2.2. Идентификатор вершины

В процессе обхода графа в суммарной памяти автоматов создаются описания пройденных вершин и выходящих из них дуг. Описание вершины хранится в памяти одного автомата, который ниже будет называться регулятором вершины, и который регулирует движение автоматов, попавших в вершину, указывая им, по каким выходящим дугам им нужно двигаться. Для того, чтобы автомат понял, в какой вершине оказался, вершина снабжается уникальным идентификатором. Попав в вершину, автомат узнаёт её уникальный идентификатор и число выходящих из вершины дуг. По этому идентификатору автомат ищет регулятор вершины, опрашивая другие автоматы. Если регулятор не найден, опрашивающий автомат сам создаёт

описание вершины в своей памяти, становясь регулятором этой вершины. Поиск регулятора вершины не требуется в двух случаях: для терминальной вершины и для вершины только с одной входящей дугой.

Регулятор терминальной вершины не нужно искать, поскольку оказывающиеся в ней автоматы всё равно не могут двигаться дальше. Когда автомат проходит дугу и находит регулятор (или сам становится регулятором) конечной вершины дуги, он может его адрес сообщить регулятору начальной вершины дуги. Это делается для того, чтобы при повторном проходе дуги можно было не проводить поиск регулятора конечной вершины дуги. Поэтому, если известно, что у вершины есть только одна входящая дуга, регулятор этой вершины можно не искать: автомат, первым попавший в эту вершину сразу становится её регулятором. В этих двух случаях вершина может не иметь идентификатора, точнее она снабжается специальным *пустым* идентификатором. Заметим, что дополнительные вершины *v*-цикла, о которых шла речь в п.2.1, имеют по одной входящей дуге и поэтому могут снабжаться пустым идентификатором. Итак, непустой идентификатор обязана иметь каждая нетерминальная вершина, в которую входит более одной дуги. Остальные дуги могут иметь как пустой, так и непустой идентификатор. Все непустые идентификаторы вершин различны.

По техническим причинам мы будем считать, что идентификатор начальной вершины графа всегда не пустой.

Поскольку идентификатор вершины является входным символом автомата, для того, чтобы автомат был конечным, алфавит идентификаторов должен быть конечным. Но тогда для конечности автоматов мы должны рассматривать графы с ограниченным числом вершин с непустыми идентификаторами.

Для того, чтобы обойти это ограничение, предлагается рассматривать расширенный автомат (EFSM), в котором, кроме его состояния, имеется конечное число ячеек памяти, в которых могут храниться идентификаторы вершин и только они: в каждой ячейке по одному идентификатору. Эти ячейки будем называть ячейками идентификаторов (вершин).

Для идентификаторов вершин определены следующие операции и только они:

1. Скопировать идентификатор из указанной ячейки идентификаторов в сообщение, посылаемое другому автомату, как его параметр.
2. Скопировать идентификатор как параметр сообщения, полученного от другого автомата, в указанную ячейку идентификаторов.
3. Сравнить на равенство идентификаторы в двух указанных ячейках идентификаторов.

2.3. Адрес автомата и среда связи автоматов

Будем считать, что среда связи автоматов позволяет передавать сообщение от любого автомата любому указанному автомату за время, ограниченное сверху константой. При посылке сообщения автомат должен указать получателя сообщения как уникальный адрес автомата. Для конечного автомата число таких получателей должно быть конечным и ограниченным: адрес автомата должен пробегать конечный алфавит. Последнее ограничение приводит к тому, что число автоматов ограничено.

Чтобы обойти это ограничение, будем считать, что в расширенном автомате имеется конечное число ячеек памяти для хранения адресов автоматов. Эти ячейки будем называть ячейками адресов (автоматов). Тем самым, граф динамических связей автоматов — это конечный ориентированный граф с ограниченной полустепенью выхода вершин (автоматов).

Для адресов автоматов определены следующие операции и только они:

1. Послать сообщение автомату, адрес которого находится в указанной ячейке адресов.
2. Скопировать адрес из указанной ячейки адресов в посылаемое сообщение как его параметр.
3. Скопировать адрес как параметр полученного сообщения в указанную ячейку адресов.

Будем считать, что выделен один *пустой* адрес, который не может быть адресом никакого созданного автомата.

Мы будем предполагать выполнение следующих правил обмена сообщениями:

1. Для передачи сообщения есть один примитив «послать сообщение», обязательным параметром которого является адрес получателя.
2. Для получения сообщений есть один примитив «принять сообщение от любого отправителя».
3. Сообщения не теряются и не искажаются в среде передачи.
4. При передаче нескольких сообщений от одного и того же отправителя одному и тому же получателю нет обгона сообщений.

2.4. Сообщение

Сообщение, передаваемое от одного автомата другому автомату, является как входным, так и выходным символом автомата. Поэтому для того, чтобы автомат был конечным, сообщение должно быть символом из конечного алфавита. Но тогда сообщение не может содержать идентификаторы вершин и адреса автоматов, если размер графа или число автоматов не ограничены.

Чтобы обойти это ограничение, будем считать, что для расширенного автомата сообщение также «расширено»: оно состоит из *тега*, пробегающего конечный алфавит тегов, и ограниченного конечного числа параметров, которыми могут являться только идентификаторы вершин или адреса автоматов. В общем, это эквивалентно (неограниченной по длине) цепочке ограниченных по длине сообщений.

2.5. Порождение и уничтожение автоматов

Самый первый автомат порождается некоторым *внешним автоматом* и начинает свою работу с ожидания сообщения от этого внешнего автомата.

Для порождения остальных автоматов предлагается использовать сообщение ***создай автомат***, посылаемое внешнему автомату. В ответном сообщении ***автомат создан*** указывается адрес порождённого автомата.

Порождаемый автомат будет начинать работать с начальной вершины графа (если он вообще должен быть связан с графом). Это требование объясняется тем, что мы рассматриваем тестирование системы, которая не допускает произвольное клонирование, а только создание нового экземпляра системы в её начальном состоянии. Исключение составляет создание автомата, которому передаётся граф от другого автомата вместе с текущей вершиной. Передатчик теряет связь с графом.

Уничтожение автомата выполняется внешним автоматом. Для этого внешнему автомату посылается сообщение ***уничтожь автомат***, в котором указывает адрес автомата, который надо уничтожить. Ответ на это сообщение не предусматривается.

2.6. Взаимодействие с графом

Предлагается рассматривать граф, точнее его экземпляр для каждого взаимодействующего с ним автомата, как отдельный автомат графа. Такой автомат графа создаётся внешним автоматом, когда ему посылается сообщение ***создай граф***. В ответном сообщении ***граф создан*** содержится адрес (автомата) графа, идентификатор начальной вершины и число дуг, выходящих из начальной вершины. Созданный автомат графа находится в состоянии, которое соответствует начальной вершине графа.

Далее потребуется только одна операция по взаимодействию с графом, которая реализуется как сообщение, посылаемые автомату графа и ответное сообщение:

1. ***проход по дуге*** графа с параметром: номер выходящей дуги.
2. Ответное сообщение ***ответ на проход*** с параметрами: идентификатор вершины и число выходящих дуг. Имеются в виду идентификатор конечной вершины указанной дуги и число дуг, выходящих из этой вершины.

Для уничтожения графа внешнему автомату посылается сообщение ***уничтожь граф*** с параметром «адрес графа», который надо уничтожить. Ответ на это сообщение не предусматривается.

2.7. Замечание о ячейках автомата и параметрах сообщений

Далее, для простоты изложения, будем считать, что каждая ячейка памяти автомата или параметр сообщения может хранить как идентификатор вершины, так и адрес автомата. Мы будем просто говорить «ячейка» или «параметр».

Кроме того, для удобства изложения будут использоваться дополнительные ячейки и параметры фиксированного размера. Совокупность значений в этих ячейках является, фактически, расширением состояния автомата, а совокупность значений в этих параметрах является, фактически, расширением сообщения как дополнение к его тегу.

Автомат, память которого расширена такими ячейками, эквивалентен коллективу конечных автоматов, которые могут обмениваться между собой сообщениями.

3. Идея алгоритма

3.1. Граф

Рассматривается обход ориентированного графа с выделенной начальной вершиной и пронумерованными выходящими из каждой вершины дугами. Все вершины снабжены идентификаторами, как было описано в п.2.2. *Началом* и *концом* дуги для краткости будем называть, соответственно, начальную и конечную вершины дуги. *Терминальной вершиной* будем называть вершину графа, из которой не выходят дуги. *Терминальной дугой* будем называть дугу, заканчивающуюся в терминальной вершине.

После инициализации в каждый момент времени имеются:

- *Пройденный граф* — подграф графа, определяемый всеми пройденными дугами.
- *Пройденное дерево* — ориентированный от корня, которым является начальная вершина, остов подграфа, получаемого из пройденного графа удалением терминальных дуг. Такое пройденное дерево содержит все нетерминальные пройденные вершины пройденного графа.

В начале работы алгоритма, после инициализации, пройденный граф совпадает с пройденным деревом и состоит из одной начальной вершины графа.

В процессе работы алгоритма пройденный граф и пройденное дерево увеличиваются (добавляются новые дуги и вершины). Пройденное дерево, получающееся в конце работы, будем называть *законченным деревом*. Под

хордой будем понимать нетерминальную дугу, не принадлежащую законченному дереву; пройденная хорда является хордой пройденного дерева. Для данной вершины *входящей* дугой будем называть дугу законченного дерева, заканчивающуюся в этой вершине. У начальной вершины нет входящей дуги, а в любую другую вершину входит ровно одна дуга.

В конце работы алгоритма получается разметка графа: для каждой дуги в её начале указывается её тип: дуга законченного дерева, терминальная дуга или хорда.

3.2. Работа автоматов

Автомат работает в трёх режимах: генератор, регулятор, движок. Обход графа, то есть движение по его дугам, выполняют движки, регуляторы предназначены для управления перемещением движков по дугам, а генератор создаёт новые движки и связанные с ними копии графов.

Каждый создаваемый генератором движок предназначен для того, чтобы, начиная с начальной вершины графа, пройти путь в пройденном дереве, после чего пройти последовательность непройденных дуг, заканчивающуюся хордой или терминальной дугой. С каждой пройденной нетерминальной вершиной связан ровно один регулятор, который направляет движки, приходящие в эту вершину, по тем или иным выходящим из вершины дугам. Для этого движок, оказавшийся в данной вершине, спрашивает у регулятора этой вершины (посылает ему сообщение) ***куда идти***, а регулятор в ответном сообщении ***иди по дуге*** сообщает номер выходящей дуги. В некоторой ситуации (описанной ниже) движок оказывается «лишним» и останавливается, не пройдя ни одной непройденной дуги.

Автомат (копии) графа, связанный с движком «помнит» текущую вершину графа, в которой находится движок. Для прохода по дуге движок посылает автомату графа сообщение ***проход по дуге***, сообщая номер выходящей дуги, и получает в сообщении ***ответ на проход*** идентификатор конца дуги и число выходящих из него дуг.

Вначале извне создаётся генератор. После инициализации генератор одновременно становится регулятором начальной вершины, которая в этот момент времени является единственной пройденной вершиной и единственной вершиной пройденного дерева, а также создаёт первый движок, находящийся в начальной вершине графа.

Для управления обходом графа используются сообщения ***запрос*** и ***конец***. Такое сообщение посылается «по пройденной дуге» в обратном направлении: от автомата, находящегося в конце дуги, регулятору начала дуги. Сообщение ***запрос*** посылается регулятором вершины по входящей дуге и означает, что по этой дуге нужно послать один движок, когда будет такая возможность. Следующий движок по этой дуге можно посылать после того, как будет получен новый ***запрос***. Сообщение ***конец***, посылаемое по пройденной дуге, означает, что по этой дуге больше никогда не нужно посылать движки.

Сообщение ***конец*** посылает по хорде или терминальной дуге движок, который эту дугу проходит. Кроме того, сообщение ***конец*** посылает регулятор вершины по входящей дуге, если в пройденном дереве выше этой вершины нет непройденных дуг (точнее, нет вершин, из которых выходят непройденные дуги). Это происходит тогда, когда по всем дугам, выходящим из вершины, получено сообщение ***конец***. У начальной вершины нет входящей дуги, поэтому регулятор начальной вершины посылает сообщение ***конец*** внешнему автомату, что означает конец работы алгоритма. ***Запросы*** посылаются с некоторым опережением, поэтому может оказаться, что движку, пришедшему в вершину, некуда двигаться. Такой движок становится *ждущим* в вершине (запоминается регулятором вершины). Далее либо по выходящей дуге придёт ***запрос***, и движок будет послан по этой дуге, либо по всем выходящим дугам будут получены сообщения ***конец***, и тогда движок оказывается «лишним» – он должен остановиться.

Каждая дуга имеет в регуляторе её начала три состояния:

– *Активная* = дуга, по которой нужно посылать движки. Это может быть либо непройденная дуга (такие дуги всегда активны), либо дуга пройденного дерева, по которой пришёл ***запрос***, но движок по дуге ещё не послан.

– *Пассивная* = пройденная дуга, по которой пока не нужно посылать движки. Это дуга, по которой был послан движок, но после этого по дуге не приходили сообщения ***запрос*** или ***конец***.

– *Законченная* = пройденная дуга, по ней больше никогда не нужно посылать движки. Это дуга, по которой пришло сообщение ***конец***.

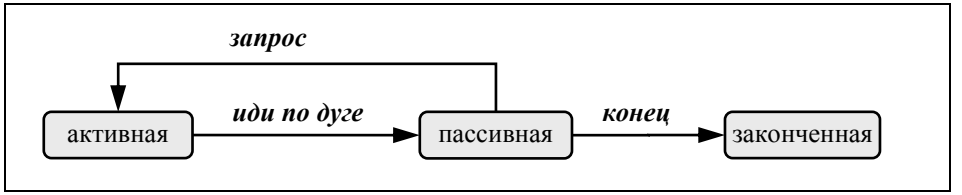


Рис 2. Схема изменения состояния дуги

На 0 изображена схема изменения состояния дуги под влиянием сообщений. В начале все дуги не пройдены, т.е. активные.

Регулятор посылает движки по активным выходящим дугам «по кругу». Для этого у регулятора есть номер текущей выходящей дуги i. Это дуга, по которой последний раз посылался движок, или i=0 в самом начале. Как сказано выше, движок, находясь в вершине пройденного дерева, спрашивает у

регулятора вершины, по какой дуге ему идти, посылая сообщение ***куда идти***. Регулятор корректирует номер *i* текущей дуги: новой текущей дугой становится следующая после дуги *i* активная дуга в цикле выходящих дуг, а если активных дуг больше нет, то *i*:=0. Если после коррекции *i*>0, регулятор посылает движку сообщение ***иди по дуге*** с номером дуги *i*, а сам по входящей дуге посылает ***запрос***. Поскольку у начальной вершины нет входящей дуги, регулятор начальной вершины не посылает сообщения ***запрос***, а генератор продолжает генерацию движков. Если *i*=0, регулятор запоминает адрес движка, движок становится ждущим, ***запрос*** не посылается. Если ждущий движок образуется в начальной вершине, генератор приостанавливает генерацию движков.

После того, как появляется ждущий движок, возможны два случая. 1) Регулятор в дальнейшем получает сообщение ***запрос*** с указанием дуги, по которой регулятор направляет ждущий движок (сообщение ***иди по дуге***), и сам посылает сообщение ***запрос*** регулятору начала входящей дуги. Поскольку у начальной вершины нет входящей дуги, регулятор начальной вершины не посылает сообщения ***запрос***, но генератор возобновляет генерацию движков. 2) Регулятор обнаруживает, что все выходящие дуги стали законченными (вершина терминальная или получено сообщение ***конец*** по последней незаконченной выходящей дуге). Регулятор посылает ждущему движку сообщение ***иди по дуге*** с нулевым номером дуги (это «лишний» движок и он останавливается) и сам посылает сообщение ***конец*** регулятору начала входящей дуги. Поскольку у начальной вершины нет входящей дуги, регулятор начальной вершины посылает сообщение ***конец*** внешнему автомату, и генератор прекращает генерацию движков.

Как создаются регуляторы? Рассмотрим подробнее работу автоматов, связанную с проходом движком непройденной ранее дуги $a—i\rightarrow b$, ведущей из вершины *a*, имеющей в ней номер *i* и ведущей в вершину *b*.

- Если вершина *b* терминальная, для неё не создаётся регулятор. Движок посылает регулятору вершины *a* сообщение ***конец*** с признаком «терминальная вершина» и номером дуги *i*, а сам останавливается.
- Если вершина *b* нетерминальная, то движок должен узнать по идентификатору вершины, пройдена она или ещё нет, то есть имеется ли регулятор этой вершины. Вершина *b* заведомо не пройдена, если её идентификатор пуст (в такую вершину входит только одна дуга). В противном случае производится опрос регуляторов, описанный ниже. Итак:
  - Если вершина *b* уже пройдена, то движок прошёл по хорде. Движок посылает регулятору вершины *a* сообщение ***конец*** с признаком «хорда», номером дуги *i* и адресом регулятора вершины *b*, а сам останавливается.

– Если вершина *b* ещё не пройдена, то движок сам становится регулятором вершины *b* и создаёт новый движок в текущей вершине *b*, передавая ему граф. После того, как этот новый движок будет создан и послан по выходящей дуге, новый регулятор вершины *b* посылает регулятору вершины *a* сообщение ***запрос*** с указанием номера дуги *i* и своим адресом как адресом регулятора вершины *b*.

Опрос регуляторов выполняется тогда, когда движок проходит по непройденной дуге и оказывается в нетерминальной вершине с непустым идентификатором. В этом случае ему нужно найти регулятор той вершины, в которой он оказался, или самому стать регулятором, если поиск не увенчался успехом. Этот поиск выполняется по идентификатору вершины: две вершины не могут иметь один и тот же непустой идентификатор. Для опроса регуляторов используется односторонний список регуляторов вершин с непустыми идентификаторами, регулятор начальной вершины (он же генератор) всегда находится в голове списка (напомним, что идентификатор начальной вершины всегда не пустой). Движок посылает по списку сообщение ***опрос***, в котором указывается (непустой) идентификатор вершины и адрес движка. Если регулятор вершины уже есть, он, получив такое сообщение, посылает ***ответ на опрос***, указывая движку свой адрес. Если регулятора вершины ещё нет, то сообщение ***опрос*** доходит до последнего в списке регулятора, который пересылает его движку, а сам ставит у себя ссылку по списку на этот движок. Получив обратно своё сообщение ***опрос***, движок сам становится регулятором – последним в списке, и создаёт новый движок, передавая ему свой адрес (как адрес регулятора текущей вершины) и адрес графа.

Замечание: Остановка движка означает, что движок уничтожает свою копию графа и самого себя.

4. Подробное описание алгоритма

Цикл работы автомата состоит в следующем:

– ожидание сообщения (любого сообщения от любого автомата),

– выполнение тех или иных действий в зависимости от полученного сообщения и состояния автомата.

Автомат имеет 4 управляющих состояния: 1) начальное состояние, 2) генератор (после инициализации, он же регулятор начальной вершины), 3) регулятор, 4) движок. Полное состояние расширенного автомата определяется его управляющим состоянием и содержанием ячеек памяти.

4.1. Структуры данных

4.1.1. Сообщения

Названия (теги) и параметры сообщений:

1. ***ты генератор***:
   - адрес внешнего автомата,
   - адрес генератора [то есть адрес получателя сообщения];
2. ***создай граф***:
   - адрес отправителя [сообщения];
3. ***граф создан***:
   - адрес [автомата копии] графа,
   - идентификатор начальной вершины,
   - число выходящих [из начальной вершины] дуг;
4. ***создай автомат***:
   - адрес отправителя [сообщения];
5. ***автомат создан***:
   - адрес созданного автомата;
6. ***ты движок***:
   - адрес движка [то есть адрес получателя сообщения],
   - адрес генератора,
   - адрес [автомата копии] графа,
   - идентификатор текущей вершины,
   - адрес регулятора текущей вершины;
7. ***куда идти***:
   - адрес отправителя [сообщения];
8. ***иди по дуге***:
   - номер выходящей дуги,
   - адрес регулятора конца выходящей дуги [может быть пустым];
9. ***проход по дуге***:
   - адрес отправителя [сообщения],
   - номер выходящей дуги;
10. ***ответ на проход***:
    - идентификатор вершины,
    - число выходящих [из этой вершины] дуг;
11. ***запрос***:
    - номер выходящей дуги [номер дуги, по которой в обратном направлении посылается сообщение, в начале дуги],
    - адрес регулятора конца выходящей дуги;
12. ***конец***:
    - номер выходящей дуги [номер дуги, по которой в обратном направлении посылается сообщение, в начале дуги],

- адрес регулятора конца выходящей дуги [может быть пустым],
- тип дуги [*дуга дерева*, *хорда*, *терминальная дуга*];

13. ***опрос***:
    - адрес движка,
    - идентификатор вершины;
14. ***ответ на опрос***:
    - адрес регулятора вершины;
15. ***уничтожь граф***:
    - адрес графа;
16. ***уничтожь автомат***:
    - адрес автомата.

Замечание: В сообщении ***запрос*** параметр «адрес регулятора конца выходящей дуги» лишний, если это повторное сообщение, посылаемое по этой дуге «в обратном направлении». Вместо этого можно было ввести сообщение с новым тегом ***новая вершина*** с такими же двумя параметрами, а в сообщении ***запрос*** оставить только один параметр: «номер выходящей дуги».

В сообщении ***конец***, которое посылает генератор внешнему автомату все параметры лишние. Вместо этого можно было бы ввести сообщение с новым тегом ***конец работы*** без параметров. Также в сообщении ***конец***, которое посылает движок, попавший в терминальную вершину, параметр «адрес регулятора конца выходящей дуги» всегда пустой. Вместо этого можно было бы ввести сообщение с новым тегом ***терминальная вершина***, в котором был только один параметр «номер выходящей дуги». Наконец, в сообщении ***конец***, которое посылается «в обратном направлении» по дуге пройденного дерева, параметр «адрес регулятора конца выходящей дуги» лишний, поскольку он уже известен регулятору начала дуги. Вместо этого можно было бы ввести сообщение с новым тегом ***хорда***, в котором были бы два параметра «номер выходящей дуги» и «адрес регулятора конца выходящей дуги», а сообщение ***конец*** посылать только «в обратном направлении» по дуге пройденного дерева с единственным параметром «номер выходящей дуги». Тем самым, вместо параметра «тип дуги» используются разные теги сообщений: ***терминальная вершина***, ***хорда***, ***конец***.

4.1.2. Ячейки автомата

- *собственный адрес автомата;*
- *адрес внешнего автомата;*
- *адрес* [автомата копии] *графа;*
- *адрес генератора;*
- *адрес ждущего движка;*
- *адрес следующего в списке регуляторов* [вершин с непустым идентификатором]*;*
- *идентификатор текущей вершины;*

- *адрес регулятора текущей вершины;*
- *номер входящей дуги* [пройденного дерева]*;*
- *адрес регулятора начала входящей дуги* [пройденного дерева]*;*
- *число выходящих дуг;*
- *номер текущей выходящей дуги;*
- список — *состояние выходящей дуги* (*i*), где *i*=1..*r;*
- список — *тип выходящей дуги* (*i*), где *i*=1..*r* [*дуга дерева*, *хорда*, *терминальная дуга*]*;*
- список — *адрес регулятора конца выходящей дуги* (*i*), где *i*=1..*r;*
- список — *рабочая ячейка* (*j*) , где *j*=1..3.

4.2. Работа автоматов

Работа начинается с того, что внешний автомат порождает один автомат, находящийся в начальном состоянии, которому посылает сообщение ***ты генератор***. Ниже мы описываем работу автомата при получении того или иного сообщения в зависимости от состояния автомата.

4.2.1. Сообщение ты генератор

Это сообщение может получить только автомат в начальном состоянии, который переходит в состояние «генератор».

Параметры сообщения переписываются в ячейки автомата:
адрес внешнего автомата, *адрес регулятора начала входящей дуги*
:= адрес внешнего автомата,
собственный адрес автомата, *адрес генератора*, *адрес регулятора текущей вершины*
:= адрес генератора.
Инициализируются ячейки:
адрес графа := *пустой*,
адрес ждущего движка := *пустой*.

Генератор создаёт экземпляр [автомата] графа, для чего посылает на *адрес внешнего автомата* сообщение ***создай граф*** с параметром:

адрес отправителя := *собственный адрес автомата*.

Обработка сообщения на этом заканчивается. В дальнейшем генератору может прийти только сообщение ***граф создан***.

4.2.2. Сообщение создай граф

Это сообщение может получить только внешний автомат и только от генератора.

Внешний автомат должен послать ответное сообщение ***граф создан***.

4.2.3. Сообщение граф создан

Это сообщение может получить только генератор в ответ на сообщение ***создай граф***.

Генератор проверяет, создаётся первый экземпляр графа или нет.

Если *адрес графа = пустой*, то создаётся первый экземпляр графа.

Параметры сообщения переписываются в ячейки автомата:

адрес графа := адрес графа,

идентификатор текущей вершины := идентификатор начальной вершины,

число выходящих дуг := число выходящих дуг.

Генератор проверяет, является ли начальная вершина терминальной.

Если *число выходящих дуг* = 0, то начальная вершина терминальная.

В этом случае генератор завершает работу алгоритма.

Для этого он сначала посылает на *адрес внешнего автомата* сообщение ***уничтожь граф*** с параметром: адрес графа := *адрес графа*.

Затем генератор посылает на *адрес внешнего автомата* сообщение ***конец***. Значения параметров в данном случае несущественны.

Обработка сообщения на этом заканчивается. Генератор останавливается.

Если *число выходящих дуг* > 0, то начальная вершина не терминальная.

Генератор становится регулятором начальной вершины, инициализируются ячейки регулятора:

адрес следующего в списке регуляторов := *пустой*,

номер текущей выходящей дуги := 0,

для каждого i=1..*число выходящих дуг* устанавливается

состояние выходящей дуги (i) := *активная*,

адрес регулятора конца выходящей дуги (i) := *пустой*.

Теперь генератор готов выполнять также функции регулятора начальной вершины.

Генератор посылает на *адрес внешнего автомата* сообщение ***создай автомат*** с параметром: адрес отправителя := *собственный адрес автомата*.

Обработка сообщения на этом заканчивается.

Если *адрес графа ≠ пустой*, то первый экземпляр графа был создан ранее.

Параметры сообщения переписываются в ячейки автомата:

адрес графа := адрес графа.

Остальные параметры игнорируются, поскольку они должны совпадать с теми, что были получены при создании первого экземпляра графа.

Генератор посылает на *адрес внешнего автомата* сообщение ***создай автомат*** с параметром: адрес отправителя := *собственный адрес автомата*.

Обработка сообщения на этом заканчивается.

4.2.4. Сообщение создай автомат

Это сообщение может получить только внешний автомат от генератора или движка, ставшего регулятором.

Внешний автомат должен послать ответное сообщение ***автомат создан***.

4.2.5. Сообщение автомат создан

Это сообщение является ответом на сообщение ***создай автомат***. Такой ответ может получить либо генератор, либо регулятор (движок, ставший регулятором). Созданный автомат предназначен для выполнения работы в режиме движка.

Параметры сообщения переписываются в ячейки автомата:

рабочая ячейка (1) := адрес созданного автомата.

Созданному автомату на адрес из *рабочая ячейка* (1) посылается сообщение ***ты движок*** с параметрами:

адрес движка := *рабочая ячейка* (1),
адрес генератора := *адрес генератора*,
адрес графа := *адрес графа*,
идентификатор текущей вершины := *идентификатор текущей вершины*,
адрес регулятора текущей вершины := *адрес регулятора текущей вершины*.
Обработка сообщения на этом заканчивается.

4.2.6. Сообщение ты движок

Это сообщение может получить автомат в начальном состоянии.

Автомат переходит в состояние движок.

Параметры сообщения переписываются в ячейки автомата:

собственный адрес автомата := адрес движка,
адрес генератора := адрес генератора,
адрес графа := адрес графа,
идентификатор текущей вершины := идентификатор текущей вершины,
адрес регулятора текущей вершины := адрес регулятора текущей вершины.

Движок посылает на *адрес регулятора текущей вершины* сообщение ***куда идти*** с параметром: адрес отправителя := *собственный адрес автомата*.

Обработка сообщения на этом заканчивается. В дальнейшем движок может получить только сообщение ***иди по дуге***.

4.2.7. Сообщение куда идти

Это сообщение может получить только регулятор (в том числе генератор как регулятор начальной вершины) и только от движка, оказавшегося в вершине этого регулятора.

Параметры сообщения переписываются в ячейки автомата:

рабочая ячейка (1) := адрес отправителя.

Регулятор корректирует *номер текущей выходящей дуги*. Для этого в цикле выходящих дуг (1..*число выходящих дуг*) после дуги с индексом *номер текущей выходящей дуги* ищется первый такой индекс *i*, что *состояние выходящей дуги* (*i*) = *активная*.

Если такая дуга найдена, то *номер текущей выходящей дуги* := *i*.
Иначе *номер текущей выходящей дуги* := *0*.

Далее регулятор проверяет, есть ли активная дуга.

Если *номер текущей выходящей дуги* > 0, то активная дуга есть.

Регулятор посылает в ответ по адресу из *рабочая ячейка* (1) сообщение ***иди по дуге*** с параметрами:

номер выходящей дуги := *номер текущей выходящей дуги*,

адрес регулятора конца выходящей дуги

:= *адрес регулятора конца выходящей дуги* (*номер текущей выходящей дуги*).

Регулятор корректирует

состояние выходящей дуги (*номер текущей выходящей дуги*) := *пассивная*.

Дальнейшее поведение зависит от управляющего состояния автомата.

Если это не генератор (не регулятор начальной вершины), то он посылает на *адрес регулятора входящей дуги* сообщение ***запрос*** с параметром:

номер выходящей дуги := *номер входящей дуги*.

Обработка сообщения на этом заканчивается.

Если это генератор, то он должен продолжить генерацию движков. Для этого генератор сначала должен создать новый экземпляр графа, поэтому он посылает на *адрес внешнего автомата* сообщение ***создай граф*** с параметром:

адрес отправителя := *собственный адрес автомата*.

Обработка сообщения на этом заканчивается.

Если *номер текущей выходящей дуги* = 0, то активных дуг нет.

Движок, спрашивавший ***куда идти***, становится ждущим. Для этого регулятор запоминает его: *адрес ждущего движка* := *рабочая ячейка* (1).

Заметим, что регулятор начальной вершины, то есть генератор приостанавливает генерацию движков, а регулятор другой вершины не посылает ***запрос***.

Обработка сообщения на этом заканчивается.

4.2.8. Сообщение иди по дуге

Это сообщение движок получает в ответ на сообщение ***куда идти***, которое он посылал регулятору той вершины, в которой сейчас находится.

Движок переписывает

адрес регулятора начала входящей дуги := *адрес регулятора текущей вершины*.

Параметры сообщения переписываются в ячейки автомата:

номер входящей дуги := номер выходящей дуги,

адрес регулятора текущей вершины := адрес регулятора конца выходящей дуги.

Движок проверяет, есть ли ему куда идти.

Если *номер входящей дуги = 0*, то движку идти некуда.

Движок должен уничтожить свою копию графа и сам себя.

Для этого он сначала посылает на *адрес внешнего автомата* сообщение ***уничтожь граф*** с параметром: адрес графа := *адрес графа*.

Затем движок посылает на *адрес внешнего автомата* сообщение ***уничтожь автомат*** с параметром: адрес автомата := *собственный адрес автомата*.

Обработка сообщения на этом заканчивается. Движок останавливается.

Если *номер входящей дуги* $\neq 0$, то движок должен пройти по указанной дуге.

Для этого движок посылает на *адрес графа* сообщение ***проход по дуге*** с параметрами:

адрес отправителя := *собственный адрес автомата*,

номер выходящей дуги := *номер входящей дуги*.

Обработка сообщения на этом заканчивается. В дальнейшем движок может получить только сообщение ***ответ на проход***.

4.2.9. Сообщение проход по дуге

Это сообщение может получить только автомат графа и только от движка.

Автомат графа должен послать в ответ сообщение ***ответ на проход***.

4.2.10. Сообщение ответ на проход

Это сообщение движок получает от автомата графа в ответ на сообщение ***проход по дуге***.

Движок проверяет, прошёл он по новой (непройденной ранее) дуге или по старой дуге.

Если *адрес регулятора текущей вершины ≠ пустой*, то движок прошёл по старой дуге.

В этом случае движок игнорирует параметры сообщения: они ему не нужны, поскольку он знает адрес регулятора конца дуги как *адрес регулятора текущей вершины*.

Движок посылает на *адрес регулятора текущей вершины* сообщение ***куда идти*** с параметром: адрес отправителя := *собственный адрес автомата*.

Обработка сообщения на этом заканчивается. В дальнейшем движок может получить только сообщение ***иди по дуге***.

Если *адрес регулятора текущей вершины = пустой*, то движок прошёл по новой дуге.

Параметры сообщения переписываются в ячейки автомата:

идентификатор текущей вершины := идентификатор вершины,

число выходящих дуг := число выходящих дуг.

Движок проверяет, является ли текущая вершина терминальной.

Если *число выходящих дуг = 0*, то движок попал в терминальную вершину.

Движок посылает на *адрес регулятора начала входящей дуги* сообщение ***конец*** с параметрами:

номер выходящей дуги := *номер входящей дуги*,

адрес регулятора конца выходящей дуги := *пустой*,

тип дуги := *терминальная дуга*.

Движок должен уничтожить свою копию графа и сам себя.

Для этого он сначала посылает на *адрес внешнего автомата* сообщение ***уничтожь граф*** с параметром: адрес графа := *адрес графа*.

Затем движок посылает на *адрес внешнего автомата* сообщение ***уничтожь автомат*** с параметром: адрес автомата := *собственный адрес автомата*.

Обработка сообщения на этом заканчивается. Движок останавливается.

Если *число выходящих дуг > 0*, то движок попал в нетерминальную вершину.

Движок проверяет, не пуст ли *идентификатор текущей вершины*.

Если *идентификатор текущей вершины = пустой*, то в данную вершину входит только одна дуга – та, по которой прошёл движок. Поэтому, поскольку движок прошёл по новой дуге, эта вершина тоже новая.

Движок становится регулятором этой новой вершины, но не вставляется в список регуляторов, поскольку идентификатор вершины пустой.

Для этого меняется управляющее состояние автомата с «движок» на «регулятор» и инициализируются ячейки регулятора:

номер текущей выходящей дуги := 0,

для каждого *i*=1..*число выходящих дуг* устанавливается

состояние выходящей дуги (*i*) := *активная*,

адрес регулятора конца выходящей дуги (*i*) := *пустой*.

Движок, ставший регулятором, создаёт новый движок. Для этого он посылает на *адрес внешнего автомата* сообщение ***создай автомат*** с параметром: адрес отправителя := *собственный адрес автомата*.

Обработка сообщения на этом заканчивается. В дальнейшем движку, ставшему регулятором, может придти сообщение ***автомат создан***, а также сообщения ***опрос***, предназначенные ему как регулятору.

Если *идентификатор текущей вершины ≠ пустой*, движок выполняет опрос регуляторов для того, чтобы узнать, есть ли регулятор вершины, в которую он попал, или ещё нет.

Для этого движок посылает на *адрес генератора* (как первого регулятора в списке регуляторов) сообщение ***опрос*** с параметрами:

адрес движка := *собственный адрес автомата*,

идентификатор вершины := *идентификатор текущей вершины*.

Обработка сообщения на этом заканчивается. В дальнейшем движок может получить либо сообщение ***ответ на опрос***, либо сообщение ***опрос***.

4.2.11. Сообщение запрос

Это сообщение регулятор (в том числе генератор как регулятор начальной вершины) получает «в обратном направлении» по выходящей пассивной дуге от регулятора конца этой дуги.

Параметры сообщения переписываются в ячейки автомата:

рабочая ячейка (1) := номер выходящей дуги,

рабочая ячейка (*2*) := адрес регулятора конца выходящей дуги.

Регулятор проверяет, имеется ли у него ждущий движок.

Если *адрес ждущего движка = пустой*, то ждущего движка нет.

Дуга становится активной и меняется адрес регулятора конца дуги:

состояние выходящей дуги (*рабочая ячейка* (1)) := *активная*,
адрес регулятора конца выходящей дуги (*рабочая ячейка* (1)) := *рабочая ячейка* (*2*).
Обработка сообщения на этом заканчивается.

Если *адрес ждущего движка ≠ пустой*, то есть ждущий движок.
Ждущему движку на *адрес ждущего движка* посылается сообщение ***иди по дуге*** с параметрами:
номер выходящей дуги := *рабочая ячейка* (1),
адрес регулятора конца выходящей дуги := *рабочая ячейка* (*2*).
Дуга остаётся пассивной, а ждущего движка теперь нет:
адрес ждущего движка := *пустой*.
Дальнейшее поведение зависит от управляющего состояния автомата.
Если это не генератор (не регулятор начальной вершины), то он посылает на *адрес регулятора начала входящей дуги* сообщение ***запрос*** с параметрами:
номер выходящей дуги := *номер входящей дуги*,
адрес регулятора конца выходящей дуги := *собственный адрес автомата*.
Обработка сообщения на этом заканчивается.
Если это генератор, то он возобновляет генерацию движков. Для этого генератор сначала должен создать новый экземпляр графа, поэтому он посылает на *адрес внешнего автомата* сообщение ***создай граф*** с параметром:
адрес отправителя := *собственный адрес автомата*.
Обработка сообщения на этом заканчивается.

4.2.12. Сообщение конец

Это сообщение регулятор (в том числе генератор как регулятор начальной вершины) получает «в обратном направлении» по выходящей пассивной дуге от регулятора конца этой дуги или движка (в случае хорды или терминальной вершины).

Параметры сообщения переписываются в ячейки автомата:

рабочая ячейка (1) := номер выходящей дуги,
рабочая ячейка (*2*) := адрес регулятора конца выходящей дуги,
рабочая ячейка (*3*) := тип дуги.

Дуга становится законченной, корректируется адрес регулятора её конца и тип дуги:

состояние выходящей дуги (*рабочая ячейка* (1)) := *законченная*,
адрес регулятора конца выходящей дуги (*рабочая ячейка* (1)) := *рабочая ячейка* (*2*),
тип выходящей дуги (*рабочая ячейка* (1)) := *рабочая ячейка* (*3*).

Регулятор проверяет, есть ли ждущий движок.

Если *адрес ждущего движка = пустой*, то ждущего движка нет. В этом случае обработка сообщения заканчивается.

Если *адрес ждущего движка ≠ пустой*, то есть ждущий движок.

Регулятор проверяет, все ли выходящие дуги теперь законченные.

Замечание: Для оптимизации можно было бы использовать счетчик незаконченных выходящих дуг, который устанавливался бы равным числу выходящих дуг при инициализации ячеек в регуляторе и уменьшался бы на 1 при получении сообщения ***конец***.

Если для каждого *i*=1..*число выходящих дуг* *состояние выходящей дуги* (*i*) = *законченная*, то все выходящие дуги законченные.

Ждущему движку на *адрес ждущего движка* посылается сообщение ***иди по дуге*** с параметром: номер выходящей дуги := 0. Остальные параметры несущественны.

По такому сообщению движок самоуничтожится.

Дальнейшее поведение зависит от управляющего состояния автомата.

Если это не генератор (не регулятор начальной вершины), то он посылает на *адрес регулятора начала входящей дуги* сообщение ***конец*** с параметрами:

номер выходящей дуги := *номер входящей дуги*,

адрес регулятора конца выходящей дуги := *собственный адрес автомата*,

тип дуги := *дуга дерева*.

Обработка сообщения на этом заканчивается.

Если это генератор, то он заканчивается работу алгоритма.

Для этого генератор посылает на *адрес внешнего автомата* сообщение ***конец***, параметры которого несущественны.

Обработка сообщения на этом заканчивается.

Если для некоторого *i*=1..*число выходящих дуг* *состояние выходящей дуги* (*i*) ≠ *законченная*, то есть незаконченная выходящая дуга. Обработка сообщения на этом заканчивается.

4.2.13. Сообщение опрос

Это сообщение используется для поиска регулятора вершины по идентификатору вершины. Оно инициируется движком, прошедшим по новой (ранее непройденной) дуге и попавшим в нетерминальную вершину с непустым идентификатором. Движок направляет это сообщение генератору как регулятору начальной вершины и первому регулятору в списке регуляторов. Далее сообщение передаётся по списку регуляторов либо искомому регулятору вершины, если такой есть, либо, проходя весь список регуляторов, возвращается движку.

Обработка сообщения зависит от управляющего состояния автомата.

Если это регулятор (в том числе генератор как регулятор начальной вершины), то он проверяет идентификатор вершины.

Параметры сообщения переписываются в ячейки автомата:
рабочая ячейка (1) := адрес движка,
рабочая ячейка (*2*) := идентификатор вершины.
Регулятор проверяет, не он ли является искомым регулятором, то есть регулятором вершины с указанным идентификатором.

Если *идентификатор текущей вершины = рабочая ячейка* (*2*), то регулятор является искомым регулятором.

Регулятор посылает движку на адрес из *рабочая ячейка* (1) сообщение ***ответ на опрос*** с параметром: адрес регулятора вершины := *собственный адрес автомата*.
На этом обработка сообщения заканчивается.

Если *идентификатор текущей вершины ≠ рабочая ячейка* (1), то регулятор не является искомым регулятором.

Регулятор проверяет, не является ли он последним в списке регуляторов.

Если адрес следующего в списке регуляторов = 0, то регулятор последний в списке регуляторов. В этом случае регулятор делает движок, инициировавший запрос, последним в списке регуляторов:
адрес следующего в списке регуляторов := *рабочая ячейка* (1).
После этого регулятор пересылает на *адрес следующего в списке регуляторов* сообщение ***опрос*** с теми же параметрами:
адрес движка := *рабочая ячейка* (1),
идентификатор вершины := *рабочая ячейка* (*2*).
На этом обработка сообщения заканчивается.

Если это движок, то искомый регулятор не обнаружен и движок должен сам стать регулятором вершины.

Для этого меняется управляющее состояние автомата с «движок» на «регулятор» и инициализируются ячейки регулятора:
адрес следующего в списке регуляторов := *пустой*,
номер текущей выходящей дуги := 0,
для каждого *i*=1..*число выходящих дуг* устанавливается
состояние выходящей дуги (*i*) := *активная*,
адрес регулятора конца выходящей дуги (*i*) := *пустой*.
Движок, ставший регулятором, создаёт новый движок. Для этого он посылает на *адрес внешнего автомата* сообщение ***создай автомат*** с параметром:
адрес отправителя := *собственный адрес автомата*.
Обработка сообщения на этом заканчивается. В дальнейшем движку, ставшему регулятором, может придти сообщение ***автомат создан***, а также сообщения ***опрос***, предназначенные ему как регулятору, находящемуся в списке регуляторов.

4.2.14.Сообщение ответ на опрос

Это сообщение используется для поиска регулятора вершины по идентификатору вершины в случае, когда искомый регулятор найден. Найденный регулятор посылает сообщение ***ответ на опрос*** движку, инициировавшему опрос регуляторов. В этом случае движок прошёл по хорде и должен самоуничтожиться.

Адрес найденного регулятора как параметр сообщения переписывается в ячейку автомата:

рабочая ячейка (1) := адрес регулятора вершины.

Движок посылает на *адрес регулятора начала входящей дуги* сообщение ***конец*** с параметрами:

номер выходящей дуги := *номер входящей дуги*,
адрес регулятора конца выходящей дуги := *рабочая ячейка* (1),
тип дуги := *хорда*.

Движок должен уничтожить свою копию графа и сам себя.

Для этого он сначала посылает на *адрес внешнего автомата* сообщение ***уничтожь граф*** с параметром: адрес графа := *адрес графа*.

Затем движок посылает на *адрес внешнего автомата* сообщение ***уничтожь автомат*** с параметром: адрес автомата := *собственный адрес автомата*.

Обработка сообщения на этом заканчивается. Движок останавливается.

4.2.15. Сообщение уничтожь граф

Это сообщение может получить только внешний автомат.

Сообщение посылает генератор в случае, когда начальная вершина графа оказалась терминальной, или движок перед тем, как самоуничтожиться.

Внешний автомат освобождает память, занятую автоматом графа.

Ответное сообщение не предусмотрено.

4.2.16. Сообщение уничтожь автомат

Это сообщение может получить только внешний автомат.

Сообщение посылает движок для самоуничтожения.

Внешний автомат осовождает память, занятую движком.

Ответное сообщение не предусмотрено.

4.3. Описание графа в конце работы алгоритма

В конце работы алгоритма все движки и автоматы графов уничтожены. Сохраняются только регуляторы – по одному на каждую нетерминальную вершину графа.

Пройденный граф сопадает со всем графом (выполнен обход), а пройденное дерево содержит все нетерминальные вершины графа – мы назвали его законченным деревом.

Внешний автомат имеет адрес регулятора начальной вершины – это тот автомат, которому он в начале работы посылал сообщение ***ты генератор***. В регуляторе начальной вершины графа в ячейке *адрес регулятора начала входящей дуги* хранится адрес внешнего автомата.

В начале каждой дуги хранится ссылка на её конец, индексируемая номером этой дуги, а в конце каждой дуги законченного дерева хранится её номер в её начале и ссылка на её начало. Для этого используются регуляторы и их ячейки.

В регуляторе каждой нетерминальной вершины, кроме начальной вершины графа, имеется адрес регулятора начала входящей дуги законченного дерева в ячейке *адрес регулятора начала входящей дуги* и номер этой дуги в её начале, хранящийся в ячейке *номер входящей дуги*.

Также в регуляторе каждой нетерминальной вершины имеется число выходящих из этой вершины дуг в ячейке *число выходящих дуг* и описание всех выходящих из этой вершины дуг. Для каждого i=1.. *число выходящих дуг* в ячейке *тип выходящей дуги* (i) хранится тип i-ой выходящей дуги (дуга дерева, хорда или терминальная дуга), а в ячейке *адрес регулятора конца выходящей дуги* (i) хранится адрес регулятора конца i-ой выходящей дуги. Этот адрес пустой, если i-ая дуга терминальная.

5. Оценка сложности алгоритма

Для того, чтобы оценить сложность алгоритма, мы введём формальную модель обхода графа коллективом автоматов, докажем теорему о модели, установим связь модели с алгоритмом и оценим сложность алгоритма на основе теоремы о модели.

5.1. Модель обхода графа коллективом автоматов

5.1.1. Определение модели

Пусть задано дерево с начальной вершиной (корнем) и множеством вершин V и определены две функции $x{:}N\times V\to N$, $y{:}N\times V\to N$, где N – множество натуральных чисел. Будем обозначать: $x_{i,j}=x(i,j)$, $y_{i,j}=y(i,j)$. В вершинах дерева могут находиться точки, которые могут двигаться по дугам дерева. Пусть в момент времени t точка появилась в вершине j, причём в вершине j она является i-ой по счёту точкой (до неё в вершине j побывало i-1 точек). Эта точка задерживается в вершине j на время $y_{i,j}$. После этого точка может двигаться по дуге $j\to j`$, если в вершине $j`$ нет точки. Если таких дуг нет, точка i ожидает появления таких дуг. Если таких дуг несколько, выбирается одна из них недетерминированным образом. Если выбрана дуга $j\to j`$ в момент

времени $t`$, то точка $i$ появляется в вершине $j`$ в момент времени $t`$, и исчезает из вершины $j$ в момент времени $t`+x_{i,j}$. Если j корень, то $x_{i,j}=0$. В момент исчезновения точки из корня в корне появляется следующая точка. В начальный момент времени $t=0$ в корне появляется первая точка, в остальных вершинах точек нет.

Формализуем выбор дуг при движении точек с помощью функции $next:N\times V\times W\to V$, где $W\subseteq 2^V$. Если точка появляется в вершине j в момент времени t и является i-ой по счёту точкой в вершине j, а в ближайший после $t+y_{i,j}$ момент времени, когда множество «свободных» (в которых нет точек) концов выходящих из j дуг не пусто, это множество равно w, то $next(i,j,w)\in w$ и означает конец выбираемой выходящей из j дуги.

5.1.2. Обозначения и ограничения

- $m`$ – число дуг дерева. Число вершин дерева, очевидно, равно $m`+1$.
- Вершины дерева делятся на *простые* и *сложные*, что задаётся функцией $d:V\to\{0,1\}$: $d(j)=1$ тогда и только тогда, когда вершина j сложная. Будем обозначать $d_j=d(j)$.
- $n`$ – число сложных внутренних (отличных от листьев и корня) вершин дерева плюс 1. Тем самым, $n`>1$.
- Будем считать, что задержки ограничены сверху:
  - для $i>1$ $x_{i,j}\le\tau$,
  - для сложной j-ой вершины $y_{1,j}\le n`\tau$,
  - для $i>1$ или простой j-ой вершины $y_{i,j}\le\tau$,

  где τ – натуральное число, означающее длительность такта времени.
- Корень – простая вершина.
- $D`$ – максимальное число сложных нелистовых вершин на пути от корня до листа дерева.

5.2. Теорема о заполнении модели точками

Будем говорить, что модель заполнена точками, если в каждой вершине дерева находится точка. Обозначим через $t(m`,n`,D`)$ – максимальное время заполнения дерева точками для данных $m`$, $n`$ и $D`$.

Теорема 1: $t(m`,n`,D`)\le 4m`\tau+2(n`-1)D`\tau$.

Доказательство: Рассмотрим путь P в дереве от корня до листа и движение точек по этому пути. Перенумеруем вершины дерева (с соответствующим изменением функций x, y и $next$) так, чтобы вершины пути P от корня до листа получили номера $1,2,\ldots,k$. Будем говорить, что точка «уходит с пути» в вершине $j<k$, если при движении точек по дереву эта точка достигает вершины j на пути, из которой она уходит по дуге, не принадлежащей пути, т.е. уходит не в вершину $j+1$. Точка, попадающая в листовую вершину k, очевидно, в ней останавливается. Число точек, уходящих с пути, очевидно, равно $m`+1-k$. Будем говорить, что путь P заполняется точками в момент времени t_P, если в этот момент времени во всех вершинах пути P имеются точки, а в любой

предыдущий момент времени это не так. Оценим время заполнения пути P точками.

Перенумеруем все точки в порядке их генерации: 1,2,…. Очевидно, число сгенерированных точек в момент времени t_P не превосходит m`+1. Обозначим:

- $a(i,j)$ – максимальный номер точки, оказавшейся в вершине j, меньший i. Будем считать, что если все точки с номером меньше i не доходят до вершины j, то $a(i,j)=0$. Очевидно, что $a(i,j)\le i-1$.
- Также будем считать, что $t^+_{0,j}=0$.
- $f(j)$ – номер точки, которая первой попадает в вершину j. Очевидно, что $f(j)=i$ тогда и только тогда, когда $a(i,j)=0$ и $a(i+1,j)>0$, что эквивалентно $a(i,j)=0$ и $a(i+1,j)=i$.
- $t_{i,j}$ – время появления точки i в вершине j.
- $t^+_{i,j}$ – время исчезновения точки i из вершины j.
- d_j – равно 1, если вершина j сложная, и равно 0, если вершина j простая.
- $D_j = d_1+\ldots+d_j$ – число сложных вершин среди вершин $1..j$.
- $D_P = d_{k-1}$.
- $D_0 = 0$.
- $g(i,j) = r$, понимаемый как «локальный» номер точки i в вершине j, т.е. длина последовательности $i_1<i_2<\ldots<i_r$ такой, что $i_r=i$, для каждого $v=i_1,i_2,\ldots,i_r$ $a(v+1,j)=v$ и для каждого $w<i$ и отличного от $i_1,i_2,\ldots,i_r$ $a(w+1,j)<w$.
- $X_{i,j} = x_{g(i,j),j}$.
- $Y_{i,j} = y_{g(i,j),j}$.

Рассмотрим соотношения между временами, следующие из определения модели. Каждое такое соотношение выражает время прихода точки в вершину или время ухода точки из вершины через аналогичные времена, но, быть может, для других точек и вершин. Каждое такое соотношение, в зависимости от дополнительных условий, имеет вид $t = t$`$+\Delta$ или $t \le t$`$+\Delta$, где t и t` имеют вид $t_{i,j}$ или $t^+_{i,j}$, а $\Delta$ – ограничение сверху на приращение времени. Мы будем говорить, что каждое такое соотношение определяет *шаг* перехода от времени $t$ к времени $t$`. С каждым временем $t_{i,j}$ или $t^+_{i,j}$ свяжем величину $2i+j+\pi$, где $\pi=0$ для $t_{i,j}$ и $\pi=1$ для $t^+_{i,j}$. Мы покажем приращение этой величины на каждом шаге, т.е. при переходе от t к t`, и ограничение сверху на приращение времени $\Delta\ge t-t$`.

0. Рассматриваем $t_{i,j}$ для $i=1$ и $j=1$.
   Время $t_{1,1}$ – это время прихода точки 1 в вершину 1, то есть момент появления первой точки в начале пути P, то есть в корне дерева. По определению модели это время равно 0.
   Соотношение 0: $t_{1,1}=0$.
   $2i+j+\pi=2\cdot1+1+0=3$.
1. Рассматриваем $t_{i,j}$ для $i>1$ и $j=1$.
   Время $t_{i,1}$ – это время прихода точки i в вершину 1, то есть момент появления точки i в начале пути P, то есть в корне дерева. По определению

модели любая точка, кроме первой, появляется в корне дерева в тот момент времени, когда предыдущая точка из этого корня уходит.

Соотношение 1: $t_{i,1}=t^{+}_{i-1,1}$.

Шаг $t_{i,1}\rightarrow t^{+}_{i-1,1}$.

Приращение $2i+j+\pi$: $[2(i-1)+1+1] - [2i+1+0] = -1$.

Приращение времени: = 0.

2. Рассматриваем $t_{i,j}$ для $j>1$ при условии, что точка i бывает в вершине j; это условие эквивалентно условию $a(i+1,j)=i$.

Если $j>1$, точка i появляется в вершине j в тот момент времени, когда выполнены два условия: 1) точка i прождала в предыдущей вершине j положенное время задержки $Y_{i,j-1}$, 2) из вершины j ушла предыдущая точка $a(i,j)$:

$t_{i,j}=max\{t_{i,j-1}+Y_{i,j-1},t^{+}_{a(i,j),j}\}$.

Заметим, что если предыдущей точки нет, то есть точка i появляется первой в вершине j, то требуется выполнение только одного условия 1. Но в этом случае $a(i,j)=0$, поэтому по определению $t^{+}_{a(i,j),j}=0$ и, следовательно, $t_{i,j}=max\{t_{i,j-1}+Y_{i,j-1},0\}=t_{i,j-1}+Y_{i,j-1}$.

2.1. Рассматриваем случай, когда точка i не первая в вершине j-1, что эквивалентно условию $g(i,j-1)>1$.

Тогда по ограничению $Y_{i,j-1}\le\tau$.

2.1.1. Рассматриваем случай $t_{i,j-1}+Y_{i,j-1} \ge t^{+}_{a(i,j),j}$.

Соотношение 2.1.1: $t_{i,j} \le t_{i,j-1}+\tau$.

Шаг $t_{i,j}\rightarrow t_{i,j-1}$.

Приращение $2i+j+\pi$: $[2i+(j-1)+0] - [2i+j+0] = -1$

Приращение времени: $\le \tau$.

2.1.2. Рассматриваем случай $t_{i,j-1}+Y_{i,j-1} < t^{+}_{a(i,j),j}$.

Соотношение 2.1.2: $t_{i,j} = t^{+}_{a(i,j),j}$.

Шаг $t_{i,j}\rightarrow t^{+}_{a(i,j),j}$.

Приращение $2i+j+\pi$: $[2a(i,j)+j+1] - [2i+j+0] \ge [2(i-1)+j+1] - [2i+j+0] = -1$.

Приращение времени: = 0.

2.2. Рассматриваем случай, когда точка i первая в вершине j-1, что эквивалентно условию $g(i,j-1)=1$.

В этом случае по ограничению $Y_{i,j-1}\le\tau+(n`-1)d_{j-1}\tau$.

Также в этом случае до точки i никакая точка не была в вершине j: $a(i,j)=0$.

Поэтому $max\{t_{i,j-1}+Y_{i,j-1},t^{+}_{a(i,j),j}\} = max\{t_{i,j-1}+Y_{i,j-1}, 0\} = t_{i,j-1}+Y_{i,j-1}$.

Соотношение 2.2: $t_{i,j} \le t_{i,j-1}+\tau+(n`-1)d_{j-1}\tau$.

Шаг $t_{i,j}\rightarrow t_{i,j-1}$.

Приращение $2i+j+\pi$: $[2i+(j-1)+0] - [2i+j+0] = -1$

Приращение времени: $\le \tau+(n`-1)d_{j-1}\tau$.

3. Рассматриваем $t^{+}_{i,j}$ для $j>1$ при условии, что точка i бывает в вершине j; это условие эквивалентно условию $a(i+1,j)=i$.

3.1. Рассматриваем случай, когда точка i бывает в вершине j+1, что эквивалентно $a(i+1,j+1)=i$.

Точка i покидает вершину j через интервал времени $X_{i,j+1}$ после её появления в следующей вершине j+1.

$t^{+}_{i,j}=t_{i,j+1}+X_{i,j+1}=max\{t_{i,j}+Y_{i,j},t^{+}_{a(i,j+1),j+1}\}+X_{i,j+1}$.

3.1.1. Рассматриваем случай, когда точка i не первая в вершине j, что эквивалентно условию $g(i,j)>1$.

Тогда по ограничению $Y_{i,j}\leq\tau$. Также по ограничению $X_{i,j+1}\leq\tau$.

3.1.1.1. Рассматриваем случай $t_{i,j}+Y_{i,j} \geq t^{+}_{a(i,j+1),j+1}$.

Соотношение 3.1.1.1: $t^{+}_{i,j} \leq t_{i,j}+\tau+\tau$.

Шаг $t^{+}_{i,j}\rightarrow t_{i,j}$.

Приращение $2i+j+\pi$: $[2i+j+0] – [2i+j+1] = -1$

Приращение времени: $\leq 2\tau$.

3.1.1.2. Рассматриваем случай $t_{i,j}+Y_{i,j} < t^{+}_{a(i,j+1),j+1}$.

Соотношение 3.1.1.2: $t^{+}_{i,j} = t^{+}_{a(i,j+1),j+1}+\tau$.

Шаг $t^{+}_{i,j}\rightarrow t^{+}_{a(i,j+1),j+1}$.

Приращение $2i+j+\pi$: $[2a(i,j+1)+j+1] – [2i+j+1] \geq [2(i-1)+j+1] – [2i+j+1] = -2$.

Приращение времени: $= \tau$.

3.1.2. Рассматриваем случай, когда точка i первая в вершине j, что эквивалентно условию $g(i,j)=1$.

В этом случае по ограничению $Y_{i,j}\leq\tau+(n`-1)d_j\tau$. Также по ограничению $X_{i,j+1}\leq\tau$.

Также в этом случае до точки i никакая точка не была в вершине j+1: $a(i,j+1)=0$.

Поэтому $max\{t_{i,j}+Y_{i,j},t^{+}_{a(i,j+1),j+1}\}+X_{i,j+1} = max\{t_{i,j}+Y_{i,j},0\}+X_{i,j+1} = t_{i,j}+Y_{i,j}+X_{i,j+1}$.

Соотношение 3.1.2: $t^{+}_{i,j} \leq t_{i,j}+\tau+(n`-1)d_j\tau+\tau$.

Шаг $t^{+}_{i,j}\rightarrow t_{i,j}$.

Приращение $2i+j+\pi$: $[2i+j+0] – [2i+j+1] = -1$

Приращение времени: $\leq 2\tau+(n`-1)d_j\tau$.

3.2. Рассматриваем случай, когда точка i доходит до вершины j, что эквивалентно $a(i+1,j)=i$, в которой уходит с пути, что эквивалентно $a(i+1,j+1)<i$.

3.2.1. Рассмотрим случай, когда $t_{i,j}+Y_{i,j}\geq t^{+}_{a(i,j+1),j+1}$.

Это случай, когда точка i доходит до вершины j в момент времени $t_{i,j}$ и после задержки $Y_{i,j}$, то есть в момент времени $t_{i,j}+Y_{i,j}$, следующая вершина j+1 на пути свободна. В этот момент времени точка i может двигаться из вершины j, поэтому она должна покинуть вершину j. В рассматриваемом случае точка уходит с пути.

Имеем $t^{+}_{i,j}=t_{i,j}+Y_{i,j}$.

3.2.1.1. Рассмотрим случай, когда точка i не первая в вершине j, что эквивалентно $g(i,j)>1$.

Тогда по ограничению $Y_{i,j} \le \tau$.

Соотношение 3.2.1.1: $t^+_{i,j} \le t_{i,j}+\tau$.

Шаг $t^+_{i,j} \to t_{i,j}$.

Приращение $2i+j+\pi$: $[2i+j+0] - [2i+j+1] = -1$

Приращение времени: $\le \tau$.

3.2.1.2. Рассмотрим случай, когда точка i первая в вершине j, что эквивалентно $g(i,j)=1$.

Тогда по ограничению $Y_{i,j} \le \tau+(n`-1)d_j\tau$.

Соотношение 3.2.1.2: $t^+_{i,j} \le t_{i,j}+\tau+(n`-1)d_j\tau$.

Шаг $t^+_{i,j} \to t_{i,j}$.

Приращение $2i+j+\pi$: $[2i+j+0] - [2i+j+1] = -1$

Приращение времени: $\le \tau+(n`-1)d_j\tau$.

3.2.2. Рассмотрим случай, когда $t_{i,j}+Y_{i,j} < t^+_{a(i,j+1),j+1}$.

Это случай, когда точка i доходит до вершины j в момент времени $t_{i,j}$ и после задержки $Y_{i,j}$, то есть в момент времени $t_{i,j}+Y_{i,j}$, следующая вершина $j+1$ на пути занята. Тогда точка i обязана покинуть вершину j после того, как следующая вершина $j+1$ освободится, но уйти с пути она может и раньше.

Соотношение 3.2.2: $t^+_{i,j} \le t^+_{a(i,j+1),j+1}$.

Шаг $t^+_{i,j} \to t^+_{a(i,j+1),j+1}$.

Приращение $2i+j+\pi$: $[2a(i,j+1)+j+1+1] - [2i+j+1] \le [2(i-1)+j+1+1] - [2i+j+1] = -1$.

Приращение времени: ≤ 0.

Прежде всего, заметим, что для каждых i, j при условии, что точка i генерируется и достигает вершины j, для времени $t_{i,j}$ можно сделать шаг $t_{i,j} \to t`$, определяемый каким-то соотношением, а в случае $j<k$ для времени $t^+_{i,j}$ также можно сделать шаг $t^+_{i,j} \to t`$, определяемый каким-то соотношением. На каждом шаге величина $2i+j+\pi$ уменьшается не менее, чем на 1, а минимальное значение 3 эта величина принимает для $t_{1,1}=0$. Поэтому для $t_{m`+1,1}$ число шагов не превосходит $[2(m`+1)+1+0]-3 = 2m`$.

Приращение времени на шаге ограниченно сверху: А) числом 2τ или В) числом $2\tau+(n`-1)d_j\tau$. Для $d_j=0$ имеем $2\tau+(n`-1)d_j\tau = 2\tau$, а для $d_j=1$ имеем $2\tau+(n`-1)d_j\tau = (n`+1)\tau$. Поскольку $n` \ge 1$, $(n`+1)\tau \ge 2\tau$, то есть в варианте В ограничение сверху не меньше, чем в варианте А.

Теперь посмотрим, в каких соотношениях используется d_j в приращении времени. Это соотношения 2.2, 3.1.2 и 3.2.1.2. Соотношение 2.2 определяет шаг $t_{i,j+1} \to t_{i,j}$ при условии, что точка i бывает в вершине $j+1$ и является первой точкой в вершине j. Соотношения 3.1.2 и 3.2.1.2 определяют шаг $t^+_{i,j} \to t_{i,j}$ при условии, что точка i первой приходит в вершину j. Поскольку в каждой вершине только одна точка является первой, d_j используется не более двух

раз: на шаге $t_{i,j+1} \rightarrow t_{i,j}$ и на шаге $t^{+}_{i,j} \rightarrow t_{i,j}$ при условии в обоих случаях, что точка i первая в вершине j. Следовательно, вариант В для приращения времени возможен не более, чем на двух шагах для данного j. Поэтому общее число шагов с вариантом В не превосходит $2D_P$. Поскольку ограничение сверху для варианта В не меньше, чем для варианта А, общее приращение времени на всех шагах для $t_{m`+1,1}$ не превосходит $(2m`-2D_P)2\tau + 2D_P(n`+1)\tau = 4m`\tau + 2(n`-1)D_P\tau$.

Путь P заполняется точками в момент времени t_P, причём в этот момент времени число сгенерированных точек не превосходит $m`+1$. Это заполнение происходит тогда, когда последняя сгенерированная точка появляется в вершине 1, что происходит не позже, чем появление точки $m`+1$ в вершине 1. Иными словами, $t_P \le t_{m`+1,1}$.

Заполнение пути P не обязательно означает заполнение всего дерева, поскольку после заполнения пути точки могут уходить с пути, если дерево не заполнено. Однако существует такой путь в дереве, момент заполнение которого совпадает с моментом заполнения всего дерева.

В каждой вершине дерева отметим выходящую дугу, по которой последний раз двигалась точка до заполнения дерева. Путь от корня до листа по отмеченным дугам будем называть *последним путём* и обозначать L.

Лемма 1: Последний путь заполняется одновременно с заполнением всего дерева.

Доказательство:

Рассмотрим конечный отрезок последнего пути, полностью заполненный точками.

Покажем, что эти точки не уходят с пути.

Действительно, допустим противное: из какой-то вершины отрезка точка уходит с пути. Рассмотрим последнюю такую вершину j. Эта вершина j не может быть листовой, так как из листовых вершин точки не уходят. Уход точки с пути из вершины пути означает, что точка передвинулась из этой вершины по дуге дерева, не лежащей на этом пути. Но тогда получится, что последняя дуга, по которой точка вышла из вершины j не лежит на последнем пути. Мы пришли к противоречию, следовательно, утверждение верно.

Следовательно, в тот момент времени, когда последний путь оказывается заполненным, ни одна точка из его вершины не уходит с пути. А тогда всё дерево заполнено, так как в противном случае была бы незаполненная вершина, не лежащая на последнем пути, и для её заполнения самая верхняя точка, общая для пути в эту вершину и последнего пути, должна была бы уйти с последнего пути. Поэтому в тот момент времени, когда последний путь оказывается заполненным, всё дерево также оказывается заполненным.

Лемма 1 доказана.

Таким образом, имеем: $t(m`,n`,D`) = t_L \le t_{m`,1} \le 4m`\tau+2(n`-1)D_L\tau \le 4m`\tau+2(n`-1)D`\tau$.
Теорема 1 доказана.

5.3. Прагматика: связь модели с алгоритмом. Сложность алгоритма

- Дерево модели получается из исходного графа следующим образом. Берём законченное дерево (пройденное дерево в конце работы алгоритма) и для каждой хорды добавляем новую вершину, которую делаем новым концом хорды. После этого добавляем терминальные дуги графа.
- m – число дуг в исходном графе. Очевидно, $m`=m$.
- Простая вершина – это терминальная вершина, вершина с пустым идентификатором или начальная вершина, т.е. корень дерева модели. Корень считается простой вершиной, поскольку переход движка по дуге, ведущей в начальную вершину, в модели превращается в переход точки в листовую вершину, а движок, сгенерированный в начальной вершине, не производит опрос регуляторов.
  Сложная вершина – это любая другая вершина, т.е. нетерминальная вершина с непустым идентификатором, отличная от корня. В такой вершине происходит опрос регуляторов.
- n – число нетерминальных вершин с непустым идентификатором. Очевидно $n`=n$, если начальная вершина нетерминальная и имеет непустой идентификатор, и $n`=n+1$, если начальная вершина терминальная или имеет пустой идентификатор. Поэтому всегда $n`\le n+1$.
- D – максимальное число нетерминальных вершин с непустым идентификатором на пути в графе от начальной вершины, не считая самой начальной вершины. Очевидно, что $D`=D$.
- Точка в модели соответствует движку. Отличие только в том, что движок, прошедший по терминальной дуге (нулевое число выходящих дуг как параметр сообщения ***ответ на проход***) или хорде (сообщение ***ответ на опрос***), а также ждущий движок, получивший сообщение ***иди по дуге*** с нулевым номером дуги, останавливается и уничтожается, а в модели точка остаётся на месте.
- Появление точки в корне дерева модели соответствует началу генерации движка, а появление точки в другой вершине – началу перемещения движка по дуге, ведущей в эту вершину (сообщение ***иди по дуге***).
- Исчезновение точки из корня дерева модели соответствует началу перемещения движка из начальной вершины по выходящей дуге, что совпадает с началом генерации следующего движка.
  Исчезновение точки из некорневой вершины дерева модели соответствует получению регулятором начала входящей в эту вершину дуги сообщения ***запрос*** по этой дуге, что происходит после того, как по этой дуге пошёл движок (сообщение ***иди по дуге***).

- $x_{1,1}$=0 (корень дерева модели имеет номер j=1) означает, что генерация первого движка начинается в начальный момент времени 0.
- $x_{i,1}$=0 (корень дерева модели имеет номер j=1) для i>1 означает, что генерация движка начинается в тот момент времени, когда предыдущий движок пошёл по дуге из начальной вершины: регулятор начальной вершины не посылает генератору сообщение ***запрос***; вместо этого генератор, совпадающий с регулятором начальной вершины, сразу начинает генерировать следующий движок.
- $x_{i,j}$, если j>1, т.е. вершина j не корень дерева модели, означает время передачи из регулятора вершины j в регулятор предшествующей по дереву вершины сообщения ***запрос***.
- $y_{1,1}$ означает время генерации первого движка, т.е. время передачи следующей цепочки сообщений:
  - ***ты генератор***, ***создай граф***, ***граф создан***, ***создай автомат***, ***автомат создан***, ***куда идти***.
- $y_{i,1}$, где i>1, означает время генерации i-го движка, т.е. время передачи следующей цепочки сообщений:
  - ***создай граф***, ***граф создан***, ***создай автомат***, ***автомат создан***, ***куда идти***.
- $y_{1,j}$ для j>1 означает время движения 1-го движка по дуге плюс опрос регуляторов, если конец этой дуги – сложная вершина, т.е. время передачи одной из следующих цепочек сообщений:
  - терминальная вершина j: ***иди по дуге***, ***проход по дуге***, ***ответ на проход***;
  - идентификатор нетерминальной вершины j пустой: ***иди по дуге***, ***проход по дуге***, ***ответ на проход***, ***создай автомат***, ***автомат создан***, ***куда идти***;
  - идентификатор нетерминальной вершины j не пустой: ***иди по дуге***, ***проход по дуге***, ***ответ на проход***, ***опрос***…***опрос***, ***создай автомат***, ***автомат создан***, ***куда идти***.
- $y_{i,j}$ для j>1 и i>1 означает время прохода i-го (не первого) движка в вершину j, т.е. время передачи следующей цепочки сообщений (без опроса): ***иди по дуге***, ***проход по дуге***, ***ответ на проход***, ***куда идти***.
- Задержка τ означает максимальное время передачи любой указанной выше цепочки сообщений, кроме той, где участвуют опросы.
- Задержка n`τ – это ограничение сверху на передачу цепочки сообщений с опросом списка регуляторов. В список попадает регулятор нетерминальной вершины с непустым идентификатором, а также регулятор начальной вершины. Число таких регуляторов равно числу сложных нелистовых вершин дерева модели плюс 1, т.е $n$`. В модели не учитываются задержки при опросе регуляторов в листовых вершинах дерева, т.е. задержки при опросе после прохода движка по хорде, когда он вместо сообщения ***опрос*** получает сообщение ***ответ на опрос***. Если же

движок проходит в новую нетерминальную вершину с непустым идентификатором, то он получит сообщение ***опрос***, после чего становится регулятором этой вершины и вставляется в список регуляторов. А это означает, что в этом случае при опросе число регуляторов в списке меньше, по крайней мере на 1, т.е. не превосходит $n`$-1. Цепочка с опросом регуляторов всегда содержит подпоследовательность сообщений ***иди по дуге***, ***проход по дуге***, ***ответ на проход***, ***опрос*** (***опрос***, который посылает движок), ***создай автомат***, ***автомат создан***, ***куда идти***, время передачи которых тоже ограничим временем τ. Остальная часть цепочки – это последовательность сообщений ***опрос***, посылаемых регуляторами в списке, длина которого не превосходит $n`$-1. Поэтому задержка на передачу сообщений всей цепочки не превосходит $\tau+(n`-1)\tau=n`\tau$.

На основе рассмотренной выше связи модели с алгоритмом можно заключить, что в момент заполнения модели точками в каждой вершине исходного графа либо 1) находится движок, который выполняет опрос регуляторов, либо 2) находится ждущий движок, либо 3) движок находился раньше, а потом был уничтожен. В случае 1 движок прошёл по хорде и находится в её конце, который соответствует листовой вершине дерева модели. В случае 3 регулятор вершины уже послал сообщение ***конец*** по входящей дуге, если это не начальная вершина, или алгоритм закончил свою работу посылкой сообщения ***конец*** внешнему автомату. После момента заполнения модели точками на исходном граф выполняются следующие действия: А) движки, выполняющие опрос регуляторов, завершают эти опросы и посылают сообщения ***конец***; В) сообщения ***конец*** распространяются по цепочке регуляторов на пути дерева к регулятору начальной вершины. Действия А требуют время, не превосходящее время опроса регуляторов после прохода по хорде, когда в списке регуляторов могут оказаться регуляторы всех нетерминальных вершин с непустым идентификатором и регулятор начальной вершины, т.е. не превосходящее $(n+1)\tau$. Распространение сообщений ***конец*** происходит по пути в законченном дереве от листьев до корня. Длина такого пути не превосходит числа дуг графа m, а время передачи одного сообщения можно считать не превосходящим τ. Поэтому действия В требуют время, не превосходящее $m\tau$.

Таким образом, учитывая теорему 1, алгоритм заканчивает работу за время, не более $[4m`\tau+2(n`-1)D`\tau]+(n+1)\tau+m\tau \le 4m\tau+2nD\tau+(n+1)\tau+m\tau$. Поскольку, очевидно, $n \le m+1$. Поэтому $4m\tau+2nD\tau+(n+1)\tau+m\tau \le 4m\tau+2nD\tau+(m+1+1)\tau+m\tau = 6m\tau+2nD\tau+2\tau = \boldsymbol{O}(m+nD)$, где

- m – число дуг в исходном графе,
- n – число нетерминальных вершин с непустым идентификатором,
- D – максимальное число нетерминальных вершин с непустым идентификатором на пути в графе от начальной вершины, не считая самой начальной вершины.

Во введении упоминалось утверждение о том, что если число автоматов не ограничено и все вычисления автоматов и передачи сообщений между ними выполняются не дольше (по порядку), чем проход дуги графа, то существуют алгоритмы, работающие $\Theta(m)$ времени. В этом случае мы можем считать, что все вершины простые, т.е. нет опроса регуляторов, длительность которого зависит от n. Поэтому в этом случае из доказанной сложности алгоритма следует оценка $6m\tau+2\tau = O(m)$. Нижняя оценка времени работы любого алгоритма обхода равна $\Omega(m)$ при условии, что за один такт можно сгенерировать не более одного автомата и каждый автомат за один такт проходит не более одной дуги. Поэтому получается оценка $\Theta(m)$.

Для того, чтобы оценить время работы алгоритма снизу, нужно предположить, что задержки $x_{i,j}$ и $y_{i,j}$ ограничены снизу. Для любого i и $j>1$ можно считать $x_{i,j}\geq 1$. Для $i>1$ и любого j можно считать, что $y_{i,j}\geq 1$. Если вершина j простая, то можно считать, что $y_{1,j}\geq 1$. А если вершина j сложная, то $y_{1,j}$ ограничено снизу 1 плюс число сложных вершин, которых достигли точки к тому моменту времени, когда в вершину j приходит первая точка. Пусть эти условия выполнены и, кроме того, $m+D\geq 2n+1$. Рассмотрим граф, изображённый на 0. Мы оставляем без доказательства утверждение о том, что на этом графе алгоритм работает $\Omega(m+nD)$.

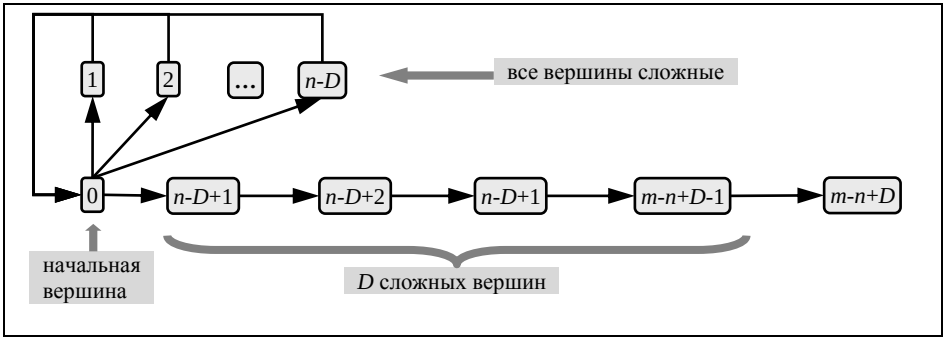


Рис. 3. Пример графа, где алгоритм работает время $\Omega(m+nD)$

6. Возможные оптимизации

6.1. Хорда

В рассмотренном выше алгоритме движок, прошедший по хорде (и узнавший об этом в результате опроса регуляторов), прекращает свою работу и уничтожается. Вместо этого можно было бы оптимизировать алгоритм, разрешив такому движку продолжать работу по обходу графа.

Движок узнаёт о том, что он прошёл по хорде, когда получает от регулятора конца хорды сообщение ***ответ на опрос***, в котором сообщается адрес этого регулятора. После этого движок мог бы послать этому регулятору сообщение ***куда идти***, и, получив ответное сообщение ***иди по дуге***, идти по указанной выходящей дуге, если в этом ответном сообщении указан ненулевой номер выходящей дуги. Нулевой номер выходящей дуги в сообщении ***иди по дуге*** регулятор указывает в том случае, когда идти некуда: все выходящие из вершины дуги пассивные и уже есть ждущий в этой вершине движок.

При такой модификации возникает следующая проблема. Когда движок попадает в вершину и не становится ждущим движком, регулятор вершины посылает по этой дуге в обратном направлении сообщение ***запрос***. До модификации движок, прошедший хорду, не посылает регулятору вершины сообщение ***куда идти***, поэтому регулятор не посылает сообщение ***запрос***. После модификации регулятор будет стараться направить по выходящим дугам все движки, приходящие в вершину, включая те, что приходят по хорде. Это может привести к появлению «лишних» сообщений ***запрос***, которые направляются в обратном направлении не по пассивной, а по активной дуге пройденного дерева. Такие «лишние» ***запросы*** могут привести к генерации «лишних» движков, то есть движков, которые уничтожаются, не пройдя ни по одной непройденной ранее дуге. Заметим, что в исходном алгоритме в каждой нетерминальной вершине уничтожается ровно один такой «лишний» движок, а именно ждущий движок, если все дуги, выходяшие из вершины, уже законченные.

Возможны два решения этой проблемы.

1. «Лишние» сообщения ***запрос*** допускаются. Это может привести к дополнительному появлению «лишних» движков. Но число таких движков, очевидно, не превосходит числа нетерминальных вершин (как раньше) плюс число хорд, что не влияет на порядок верхней оценки сложности алгоритма.
2. «Лишние» сообщения ***запрос*** не допускаются. Регулятор узнаёт, как движок пришёл в вершину: по входящей дуге пройденного дерева (для начальной вершины – сгенерирован генератором) или по хорде, и в последнем случае не посылает сообщение ***запрос***. Для этого достаточно в параметрах сообщения ***куда идти*** указать признак того, что движок прошёл по хорде. Или ввести для этого случая новое сообщение ***куда идти после хорды*** с теми же параметрами, что и в сообщении ***куда идти***. Получив это новое сообщение, регулятор не посылает сообщение ***запрос***. В этом случае дополнительных «лишних» движков не появляется.

6.2. Петля

Когда движок проходит по непройденной дуге $a—i\rightarrow b$, он должен опросить регуляторы, чтобы узнать, новая вершина b или уже пройденная. Однако, если $a=b$, то есть дуга $a—i\rightarrow b$ является петлёй, то вершина b заведомо пройденная, а её регулятор – это регулятор вершины a. В этом случае опрос регуляторов

можно не проводить. Для того, чтобы узнать, является ли дуга $a—i\rightarrow b$ петлёй, достаточно сравнить идентификаторы вершин a и b. Идентификатор вершины b движок узнаёт, пройдя по дуге $a—i\rightarrow b$, то есть получив от графа сообщение ***ответ на проход***. Как движок узнает идентификатор вершины a?

При создании движка ему посылается ***ты движок***, в котором указывается как параметр идентификатор текущей вершины: начальной вершины, если движок создаётся генератором, или другой вершины, если движок создаётся регулятором (движком, ставшим регулятором новой вершины). Этот идентификатор вершины движок запоминает как *идентификатор текущей вершины*. После прохода по дуге движок получает сообщение ***ответ на проход***, в котором указывается идентификатор вершины, в которой движок оказался. Если это новая дуга, движок запоминает этот идентификатор как *идентификатор текущей вершины*. Однако, если это старая дуга, то в исходном алгоритме движок не запоминает идентификатор текущей вершины, поскольку ему известен адрес регулятора этой вершины.

Для оптимизации «петля» нужно следующее:

1. После прохода новой дуги движок сначала сравнивает идентификаторы в ячейке *идентификатор текущей вершины* и в параметрах сообщения ***ответ на проход***. Если они не совпали, движок запоминает новый идентификатор текущей вершины, дальнейшее поведение, как раньше. Если же идентификаторы совпали, то есть пройдена петля, движок выполняет те действия, которые он обычно выполняет после прохода по хорде и завершении опроса регуляторов.
2. После прохода старой дуги движок переписывает идентификатор вершины из параметров сообщения ***ответ на проход*** в ячейку *идентификатор текущей вершины*.

6.3. Кластеризация автоматов по машинам

Напомним, что под машиной мы понимаем процессор с памятью. Автомат работает в одной машине, но в одной машине может быть несколько автоматов. Передача сообщения между автоматами в одной и в разных машинах занимает существенно разное время, поэтому имеет смысл оптимизировать распределение автоматов по машинам.

Сравнивая нижнюю оценку $\boldsymbol{\Omega}(m)$ и верхнюю оценку $\boldsymbol{O}(m+nD)$, видим, что оптимизировать можно только опрос регуляторов, требующий $\boldsymbol{O}(n)$ тактов на каждый опрос и суммарно дающий $\boldsymbol{O}(nD)$ тактов.

Поскольку объём памяти машины $\boldsymbol{O}(1)$, в ней помещается $\boldsymbol{O}(1)$ автоматов. Если список регуляторов кластеризован, т.е. все регуляторы из одной машины расположены подряд, то та часть времени опроса, которая приходится на регуляторы одной машины равна $\boldsymbol{O}(1)$. Тем самым, время одного опроса равно $\boldsymbol{O}(p)$, где p — число машин. Это, конечно, не влияет на порядок оценки, поскольку $\boldsymbol{O}(n)=\boldsymbol{O}(p)$. Это равенство означает наличие такого коэфициента k,

что $n=kp+\boldsymbol{o}(p)$, а на практике значение k очень важно, так как означает ускорение обхода в k раз.

Для кластеризации списка регуляторов нужно выделять память под регуляторы отдельно от памяти для движков и связанных с ними графов. Но в описанном выше алгоритме регулятор – это бывший движок. Поэтому приходится кластеризовать суммарно регуляторы и движки. Память под регуляторы и движки выделяется сначала в первой машине, а когда в ней нет места, то во второй машине, и так далее. В каждый момент времени имеется текущая машина, в которой эти автоматы создаются. Важно отметить, что освобождение памяти в машине из-под уничтожаемого движка не должно приводить к её использованию для новых движков и регуляторов, если это не текущая машина.

Но что делать, когда машины закончились, то есть текущая машина – последняя в списке машин и память в ней закончилась? Можно закончить работу, считая что граф слишком большой. Но можно продолжить обход с понижением скорости до тех пор, пока будет хватать памяти. Для этого будем использовать для создания новых движков и регуляторов свободную память уже не только в текущей машине. Опять сначала в первой машине, потом во второй машине и так далее по циклу. Скорость уменьшится из-за фрагментации по машинам списка регуляторов. Но мы всё же сможем обходить графы больших размеров.

6.4. Работа с ограниченной памятью

Что делать, если в процессе работы алгоритма памяти не хватает, то есть внешний автомат не может выполнить запрос на создание автомата или графа?

Если не может быть удовлетворён запрос на создание автомата (движка), поступивший от движка, ставшего регулятором, то можно вместо ответного сообщения ***автомат создан*** посылать сообщение ***нет памяти для нового автомата***. Получив такое сообщение движок, ставший регулятором, уничтожает связанный с ним граф, что приводит к появлению свободной памяти.

Если не может быть удовлетворён запрос на создание автомата (движка) или графа, поступивший от генератора, то такой запрос во внешнем автомате становится ждущим. Ждущий запрос может быть удовлетворён позже, когда появится свободная память из-под уничтожаемых графов.

Исключение составляет случай, когда уничтожается последний граф, но в этом случае никакая работа продолжена быть не может из-за нехватки памяти. Внешний автомат может выделить этот случай, «помня» число существующих графов и при уничтожении последнего графа сообщать о невозможности продолжать работу.

7. Заключение

Дальнейшие исследования обхода графов коллективом автоматов возможны по нескольким направлениям.

Первое направление: недетерминированные графы. Граф недетерминирован, если выходящая из вершины дуга идентифицируется неоднозначно: одному номеру выходящей дуги соответствует несколько дуг с общей начальной вершиной и разными конечными вершинами. Обход такого графа может пониматься двояко. 1) Полный обход графа, когда требуется проход по каждой дуге графа. 2) Частичный обход графа, когда нужно пройти хотя бы по одной дуге с данной начальной вершиной и данным номером выходящей дуги. Частичный обход графа применяется, когда есть уверенность, что интересующие нас свойства графа могут быть исследованы при таком обходе.

Для того, чтобы гарантировать возможность полного обхода недетерминированного графа, нужны какие-то дополнительные предположения. Одним из таких предположений может служить предположение об ограниченном недетерминизме, которое сводится к следующему: проходя t раз из данной вершины по выходящей дуге с данным номером мы пройдём все выходящие из этой вершины дуги с этим номером, где t заранее известная константа. Существуют алгоритмы такого обхода одним автоматом, оставляющем «пометки» в вершинах графа [[6]]. Можно поставить задачу разработки нового алгоритма такого обхода коллективом взаимодействующих автоматов (без «пометок» в вершинах).

Для того, чтобы гарантировать возможность частичного обхода недетерминированного графа, также нужны дополнительные предположения. Одним из таких предположений является, так называемая Δ-достижимость вершин графа из начальной вершины [[3]]. Говоря неформально, это означает, что существует алгоритм движения из начальной вершины в заданную вершину, который гарантированно обеспечивает достижение заданной вершины при любом недетерминированном выборе выходящей дуги в рамках множества дуг с тем же началом и тем же номером дуги.

Второе направление: использование внешней памяти. Предложенный в данной статье алгоритм обходит граф только в том случае, когда регуляторы всех нетерминальных вершин помещаются в суммарной памяти машин. Если это не так, становится актуальным вопрос об использовании дополнительной внешней памяти. Поскольку код программы регулятора, и вообще, всех автоматов: регуляторов, движков и генератора, один и тот же для всех этих автоматов, разные автоматы отличаются только состоянием и содержимым ячеек памяти. Поэтому речь идёт о размещении на внешней памяти этого набора данных. Адрес автомата, тем самым, может указывать на такой набор данных как в оперативной памяти той или иной машины, так и быть адресом по внешней памяти. Передача сообщения автомату, располагающемуся на

внешней памяти, потребует подкачки информации с внешней памяти, для чего, возможно, потребуется откачка части автоматов на внешнюю память.

Обход графа коллективом автоматов с использованием внешней памяти лежит на стыке двух направлений: 1) методы эффективной работы с внешней памятью, в частности, стратегий страничной и сегментной подкачки, 2) алгоритмы обхода графов коллективом автоматов. Сама постановка такой задачи является новой. Здесь потребуются новые алгоритмы (или модификации имеющихся алгоритмов) работы с внешней памятью, учитывающие специфику обхода графов, и алгоритмы обхода графов, учитывающие специфику работы с внешней памятью.

Третье направление: графы специального вида. Большой интерес представляет разработка новых алгоритмов исследования графов, принадлежащих тому или иному подклассу графов. Такие подклассы имеет смысл рассматривать при сочетании двух условий: эти подклассы имеют практическое значение (графы переходов исследуемых систем и сетей относятся к такому подклассу) и на этих подклассах алгоритмы могут работать быстрее.

Литература:

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

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

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

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

[7] 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.

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

[9] 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.

Graph learning by a set of automata

Igor Burdonov, Alexander Kosachev
ISP RAS, Moscow,Russia
{igor, kos}@ispras.ru

Abstract. Graph learning by automata is a basic task in many applications. Among these applications are verification and testing of software and hardware systems, network exploration including Internet and GRID basing on formal models. The system or network model, in the final analysis, is a transition graph with properties to be examined. In the recent years the size of the systems and networks in everyday use and, consequently, the size of their models and graphs to be learned grows constantly. Problems arise when graph exploration by a single automaton (computer) either requires a lot of time, or a lot of computer memory, or both. Therefore, there is a task of parallel and distributed graph exploration. This task is formalized as graph learning by a set of automata (a set of computers with sufficient total memory working in parallel). This paper covers the solution of this task. First of all the behavior of the automaton on the graph is formalized. Then the algorithm of the graph learning is described in detail. The estimation of the algorithm is $\boldsymbol{O}(m+nD)$. Some ways of possible optimization is discussed.

Keywords: graph learning, traversal of an unknown graph, finite automaton (state machine), extended finite state machine, communicating automata, parallel processing, distributed systems, testing.

References:

[1] 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.
[2] 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.
[3] 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.
[4] 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.
[5] 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.
[6] Bourdonov I.B., Kossatchev A.S. Complete Open State Testing of Limitedly Nondeterministic Systems. Programming and Computer Software, Vol. 35, No. 6, 2009,pp.301-313
[7] 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)
[8] A. Demakov, A. Kamkin, A. Sortov. "High-Performance Testing: Parallelizing Functional Tests for Computer Systems Using Distributed Graph Exploration". Open Cirrus Summit 2011, Moscow.
[9] 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)

Описание аппаратных конфигураций гостевых систем в эмуляторе QEMU в виде отдельных текстовых файлов

Горемыкин О.В.
ИСП РАН, Москва
goremykin@ispras.ru

Аннотация. QEMU – свободное программное обеспечение с открытым исходным кодом, предназначенное для эмуляции различных платформ. В данной статье описывается разработанный способ описания аппаратных конфигураций гостевых систем в эмуляторе QEMU с помощью отдельных текстовых файлов, использующих удобный для чтения и написания текстовый формат JSON. Внешние конфигурационные файлы позволят избежать необходимости перекомпилировать QEMU при любых изменениях аппаратных конфигураций гостевых систем, а также избавят от необходимости разбираться во всех тонкостях внутренней реализации эмулятора.

Ключевые слова: QEMU; эмуляция; аппаратная конфигурация

1. Введение

Эмулятор в вычислительной технике – это комплекс программных или аппаратных средств, а также их сочетание, предназначенный для выполнения функций одной вычислительной системы (гостевой) на другой (целевой), таким образом, чтобы эмулируемое поведение как можно ближе соответствовало поведению оригинальной системы [1].

В настоящее время существует множество систем эмулирования, одной из них является QEMU [2]. Данный эмулятор имеет два режима работы: режим полносистемной эмуляции и режим эмуляции приложения. В первом режиме эмулируется все аппаратное обеспечение, включая периферийные устройства, и в полученной виртуальной машине загружается своя операционная система. Второй режим позволяет запускать отдельные пользовательские приложения, написанные для одной архитектуры, на другой [3].

Все аппаратные конфигурации гостевых систем в QEMU представлены в виде файлов, написанных на языке Си. Данные файлы являются громоздкими и сложными для восприятия и редактирования. Добавление нового устройства в реализованную систему представляет собой множество вызовов различных

функций с большим числом параметров. Также, после добавления устройства или любых других изменений в гостевой системе, необходимо заново собирать QEMU. Поэтому, с целью упрощения добавления новых гостевых систем или внесения в них изменений, разрабатывается средство описания аппаратных конфигураций гостевых систем с использованием текстовых файлов.

Подобный способ уже реализован в эмуляторе Simics[4], который имеет свой внутренний язык для моделирования устройств – Device Modeling Language (DML). Для описания конфигураций гостевых систем в Simics используются отдельные скриптовые файлы, которые полностью описывают аппаратные и программные особенности эмулируемых систем. Но данный эмулятор является платаным программным обеспечением и его код отсутствует в свободном доступе.

2. Реализация гостевых систем в QEMU

В QEMU гостевые системы представлены в виде структур, одним из полей которых является функция инициализации. В функции инициализации происходит создание центрального процессора, памяти, таймеров, контроллеров прерываний и различных периферийных устройств. Параметры данной функции соответствуют аргументам запуска QEMU, которые указывает пользователь.

Весь процесс инициализации гостевой системы можно условно поделить на три части:

1) инициализация памяти;
2) инициализация процессора;
3) инициализация периферийных устройств.

2.1. Модель памяти гостевой системы

Память в QEMU моделируется с помощью ациклического графа объектов MemoryRegion, листья которого являются RAM и MMIO(memory-mapped I/O).

В QEMU у гостевых систем существует четыре основных типа памяти [3]:

- RAM: диапазон RAM памяти целевой системы, которая доступная гостевой системе;
- MMIO: адреса гостевой системы, которым соответствуют указатели на функции чтения и записи, обеспечивающие пересылку данных между CPU и устройствами;
- Alias: диапазон памяти гостевой системы, позволяющий через свои адреса получать доступ к ячейкам памяти другого диапазона памяти;
- Контейнер: единая область памяти фиксированного размера, содержащая подобласти с разными смещениями относительно своего начала.

В QEMU возможна ситуация, когда алиасы или контейнеры перекрывают адресное пространство друг друга. Для разрешения коллизий используются приоритеты. Приоритеты локальны и сравниваются между подобластями одного уровня внутри контейнера. Перекрытие областей памяти служит исключительно для удобства моделирования памяти гостевой системы.

Во время инициализации гостевой памяти происходит создание объектов, каждый из которых характеризует одну из приведенных выше типов памяти.

2.2. Модель процессора и периферийных устройств гостевой системы

Для моделирования различных аппаратных компонентов гостевых систем в QEMU используется объектная модель (QOM – QEMU Object Model). QOM обеспечивает создание и регистрацию произвольных типов, а также создание экземпляров этих типов. Каждый QOM объект состоит из двух структур: Class и Object. Class содержит перечень указателей на функции и общие для всех классов поля. Object содержит поля, значения которых индивидуальны для каждой копии объекта [3].

Каждый объект может иметь свойства. Свойства описываются структурой «ObjectProperty». Данная структура содержит информацию об имени свойства и его типе, указатели на функции доступа к свойству, указатель на внутренние данные свойства, а также деструктор. Все свойства объекта образуют единый список.

На основе QOM в QEMU создаются процессор, различные устройства, а также шины. При создании новых устройств создаются и инициализируются новые объекты QOM.

У каждого устройства в QEMU имеется указатель на шину, к которой оно подключено, а также список шин, которыми оно управляет. Аналогично с шинами: каждая шина имеет указатель на одно родительское устройство и список указателей на дочерние устройства. Таким способом в QEMU задается дерево, где на каждом уровне находятся либо устройства, либо шины. Корнем данного дерева является системная шина.

3. Реализация гостевых систем с использование конфигурационного файла

3.1. Язык описания аппаратных конфигураций

Для описания аппаратных конфигураций в виде отдельных текстовых файлов был выбран текстовый формат JSON (JavaScript Object Notation)[5].

Текст в формате JSON представляет собой одну из двух структур:

- коллекция пар ключ/значение. В разных языках, эта концепция реализована как объект, запись, структура, словарь, хэш, именованный список или ассоциативный массив;

- упорядоченный список значений. В большинстве языков это реализовано как массив, вектор, список или последовательность.

Коллекция пар ключ/значение в JSON называется объектом, а упорядоченный список значений называется массивом. Значение может быть строкой в двойных кавычках, числом, true, false, null, объектом или массивом. Объекты и массивы могут быть вложенными друг в друга.

Данный формат был выбран по следующим причинам:

- большие расходы на разработку собственного текстового формата, а также разработка, внедрение и сопровождение инструмента, позволяющего переводить данные из текстового документа во внутреннее представление QEMU;
- текстовый формат JSON используется «QEMU Monitor Protocol»; он необходим для взаимодействия QEMU с другими программами, поэтому в QEMU уже встроен транслятор формата JSON, который осуществляет однозначное преобразование поступивших данных во внутренне представление QEMU(QOM объекты);
- данный текстовый формат является простым и в тоже время способен полностью покрыть все возможные аппаратные конфигурации гостевой системы.

Далее рассмотрим способ инициализации памяти, процессора, периферийных устройств и шин с помощью конфигурационного файла.

3.2. Описание аппаратных конфигураций гостевых систем с использованием формата JSON.

Для инициализации гостевой системы используется текстовый файл аппаратных конфигураций. Аппаратные конфигурации в текстовом файле представлены в виде объекта, ключи которого характеризуют различные аппаратные свойства гостевой системы. Для указания файла был добавлен новый параметр запуска QEMU – config_file «file_name».

3.2.1 Описание памяти гостевой системы

Конфигурации памяти гостевой системы представлены в виде массива. Элементами массива являются объекты. Каждый объект массива описывает определенную область памяти. Описания полей объекта приведены в табл. №1.

Наименование	Тип JSON	Свойства
name	строка	название области памяти, используется исключительно для отладки
id	строка	уникальный идентификатор области памяти; используется для доступа к указанной области

parent	строка	строкой является *id* области памяти, в которую необходимо добавить подобласть; если не указывать данный параметр, то по умолчанию считается, что *parent* – выделенная для гостевой системы память
size	строка	размер создаваемой области памяти; размер задается в десятеричной или шестнадцатеричной системе счисления в виде строки, например, «0xff9a»; возможно в качестве параметра указать «*ram_size*», что будет соответствовать размеру выделенной для гостевой системы памяти
offset	строка	смещение создаваемой подобласти относительно *parent*; смещение задается в таком же виде, как и поле *size*; если данное поле не указано, то будет создан контейнер
alias_to	строка	созданная область памяти будет являться алиасом к области памяти с *id*, указанным в строке
readonly	булев тип	по умолчанию данное поле имеет значение false; если данное поле имеет значение true, то созданная область будет доступна только для чтения
overlap_priority	число	по умолчанию созданные области памяти имеют приоритет 0, но для разрешения коллизий при перекрытии областей памяти необходимо использовать приоритеты

Табл. 1. Атрибуты объекта JSON, описывающего область памяти.

Все поля, за исключением size, не обязательны, так как при создании области памяти достаточно знать только её размер. Однако, объект с единственным полем size будет бесполезен, так как будет задавать пустой контейнер, который из-за отсутствия id нельзя будет использовать.

Данная структура объектов, описывающих области памяти, благодаря наличию уникальных идентификаторов, позволяет полностью моделировать древовидную структуру памяти гостевой системы аналогично тому, как описано в разделе 2.1.

Ниже, в качестве примера, приведен объект, отвечающий за инициализацию памяти в гостевой системе «kzm» архитектуры ARM с помощью конфигурационного файла.

«memory»: [

```
  {«name»:    «kzm.ram»,    «size»:    «ram_size»,    «offset»:
«0x80000000», «id»: «ram»},
  {«name»:   «ram.alias»,   «size»:   «ram_size»,   «offset»:
«0x88000000», «alias_to»: «ram»},
  {«name»:    «kzm.sram»,    «size»:    «0x4000»,    «offset»:
«0x1FFFC000»}
].
```

В данном примере создаются три разных области памяти, одна из которых является алиасом. Так как поле «parent» явно не указано, то все смещения «offset» создаваемых областей памяти берутся относительно начала выделенной для гостевой системы памяти.

3.2.2. Описание устройств гостевой системы

Устройства (за исключением процессора) в конфигурационном файле представлены в виде массива объектов. Из-за наличия зависимостей между устройствами их необходимо инициализировать в определенном порядке, например, перед инициализацией контроллера прерываний необходимо создать процессор. Каждое устройство характеризуется атрибутами, которые задаются с помощью полей объекта. В табл. №2 приведены основные атрибуты устройств.

Наименование	Тип JSON	Свойства
type	строка	данное поле однозначно определяет тип устройства во время создания QOM объекта
id	строка	уникальный идентификатор устройства
addr	строка или массив строк	MMIO: адреса гостевой системы, которым соответствуют указатели на функции чтения и записи, обеспечивающие пересылку данных между CPU и устройствами
irq	объект	объект состоит из строки *device_id* и массива *num*; *device_id* – уникальный идентификатор устройства, к которому адресовано прерывание, *num* – массив номеров прерываний
gpio_out	объект	объект состоит из строки *device_id* и массива *num*; *device_id* – уникальный идентификатор устройства, от которого необходимо обрабатывать прерывания, *num* – массив номеров прерываний

prop_set	объект	объект состоит из строк *name* и *value*, где *name* – название свойства, а *value* – его значение; объект *prop_set* необходим для инициализации имеющихся в устройстве дополнительных атрибутов; все добавляемые свойства должны быть явно закреплены за устройствами с определенными наименованиями
bus	объект	объект состоит из строк *device_id* и *property*; данным объектом является дочерняя шина со свойством *property* у устройства с *id* равным *device_id;* если объект *bus* явно не указан, то устройство будет подключено к системной шине

Табл. 2. Атрибуты объекта JSON, описывающего периферийные устройства.

Процессор в конфигурационном файле представлен в виде отдельного объекта. У него есть два атрибута:

1) «model_default»: наименование модели процессора, которая будет использоваться при инициализации, если другая модель не будет указана во время запуска QEMU;
2) «id»: уникальный идентификатор, который необходим для последующего использования процессора при инициализации устройств.

Рассмотрим на примере платы Versatile инициализацию процессора и контроллера прерываний PL190 (в данном примере пропущена инициализация памяти и устройств, отличных от PL190):

```
{
    «cpu»:  { «model_default»:  «arm926»,  «id»: «cpu»},
    «device»:
    [
          { «type»: «pl190»,   «addr»: «0x10140000»,   «id»:
«vectored interrupt controller»,
       «irq»: { «device_id»:  «cpu»,  «num»:  [0, 1] }}
    ]
}
```

Из примера видно, что для создания контроллера прерываний необходимо указать его название, адрес MMIO и номера прерываний, которые будут поступать на процессор. Прерывания с номерами 0 и 1 являются

прерываниями типа IRQ(Interruppt Request) и FIQ (Fast Interrapt Request) соответственно. После инициализации контроллера прерываний и процессора появляется возможность создания и других устройств.

4. Заключение

Разработанный способ описания аппаратных конфигураций позволяет инициализировать память гостевой системы, процессор и устройства вне исходного кода QEMU. Благодаря этому, появляется возможность без перекомпиляции эмулятора вносить изменения в аппаратные конфигурации гостевых систем, а также описывать новые системы.

Данный способ описания был протестирован на гостевых системах архитектуры ARM. Так как инициализация гостевых систем происходит один раз при запуске QEMU, то использование конфигурационного файла никак не сказывается на скорости работы эмулятора.

Описанный подход универсальный, поэтому позволяет легко включать в свою реализацию новые архитектуры гостевых систем.

Список литературы

[1] James Smith, Ravi Nair. Virtual Machines: Versatile Platforms for Systems and Processes. Morgan Kaufmann Publishers is an imprint by Elsevier, 500 Sansome Street, Suite 400, San Francisco, CA 94111, 2005.

[2] Fabrice Bellard, QEMU, a fast and portable dynamic translator, In Proc. of the USENIX Annual Technical Conference, pages 41–46, April 2005.

[3] QEMU – Open Source Processor Emulator. http://wiki.qemu.org . Дата обращения: 19.03.2014.

[4] Wind River Simics. http://www.windriver.com/simics. Date of visit: 19.03.2014.

[5] The JSON Data Interchange Format, Standart ECMA-404, Geneva, 2013.

Description of hardware configurations of guest systems in QEMU emulator as separate text files

O.V.Goremykin
ISP RAS, Moscow, Russia
goremykin@ispras.ru

Abstract. QEMU is an open-source full system emulator based on the dynamic binary translation approach. QEMU emulates a complete hardware environment including CPUs, peripheral devices (VGA cards, network interfaces, IDE controllers, sound and USB components, and other). Guest system hardware configuration initialized at QEMU startup. During initialization phase QEMU creates virtual CPUs, peripheral devices and memory hierarchy. Hardware configuration is a part of QEMU source code. Being written in C language hardware configuration can be difficult to edit. Also, QEMU must be recompiled after any change in hardware configuration like addition of a new device. A method of the guest system hardware configuration description as an external file is discussed in this paper. JSON format was selected for configuration files. QEMU has parser for this format readily available because JSON is used by QEMU Monitor Protocol. Each file describes hardware configuration of one board. It consists of sequence of JSON objects. Each object describes properties one peripheral device, bus, CPU or memory segment. Each object has a unique ID associated with it and can reference other objects by their IDs. An arbitrary board configuration can be described with this format. The proposed method was implemented in QEMU and tested on ARM guest systems. The method is generic and can be applied to any board with any CPU architecture.

Keywords: QEMU; emulation; hardware configuration.

References

[1] James Smith, Ravi Nair. Virtual Machines: Versatile Platforms for Systems and Processes. Morgan Kaufmann Publishers is an imprint by Elsevier, 500 Sansome Street, Suite 400, San Francisco, CA 94111, 2005.
[2] Fabrice Bellard, QEMU, a fast and portable dynamic translator, In Proc. of the USENIX Annual Technical Conference, pages 41–46, April 2005.
[3] QEMU – Open Source Processor Emulator. http://wiki.qemu.org. Date of visit: 19.03.2014.
[4] Wind River Simics. http://www.windriver.com/simics. Date of visit: 19.03.2014.
[5] The JSON Data Interchange Format, Standart ECMA-404, Geneva, 2013.

Межпроцедурный анализ помеченных данных на базе инфраструктуры LLVM.

Кошелев В.К., Избышев А.О, Дудина И.А
ИСП РАН, Москва
{vedun, izbyshev, eupharina}@ispras.ru

Аннотация. В данной работе рассматривается задача анализа помеченных данных. Для её решения предлагается статический межпроцедурный котекстно-потоковый объектно-чувствительный алгоритм, производится оценка характеристик данного алгоритма, обсуждаются особенности реализации на базе компиляторной инфраструктуры LLVM и приводятся результаты практического тестирования.

Ключевые слова: статический анализ; межпроцедурный анализ; класс IFDS; анализ потоков данных; анализ помеченных данных

1. Введение

Разработка программного обеспечения (ПО) с повышенными требованиями к безопасности сопряжена с необходимостью дополнительного контроля качества. В частности, важной задачей является включение в процесс разработки ПО поиска уязвимостей, связанных с нарушением политик безопасности. Данный поиск может осуществляться с помощью статического анализа программного кода. Статический анализ производится без реального выполнения программы и затрагивает весь программный код, анализируя редко выполняемые фрагменты кода, тестирование которых затруднено сложностью воспроизведения.

Для поиска таких уязвимостей как, «утечка критических данных», «использование константных паролей» или же «отправка нешифрованных критических данных», необходимо использовать методы статического анализа, способные эффективно отслеживать потоки данных в программе. В данном случае задача отслеживания потоков данных может быть сведена к задаче анализа помеченных данных.

В данной работе предлагается межпроцедурный контекстно-зависимый алгоритм анализа помеченных данных, позволяющий реализовать поиск данных уязвимостей. Алгоритм поиска реализован на базе компиляторной инфраструктуры LLVM. В качестве представления программ, над которым осуществляется анализ, используется LLVM-биткод. Использование LLVM-биткода позволяет унифицировать представление программ на различных

языках программирования и использовать обширный набор анализов, уже реализованных в инфраструктуре LLVM.

Данная работа разбита на восемь частей. Во второй части производится обзор существующих подходов к анализу помеченных данных. В третьей части рассматривается решение задач из класса IFDS (класс IFDS представляет собой класс межпроцедурных задач анализа потоков данных с конечным множеством фактов и дистрибутивными передаточными функциями). Четвёртая часть посвящена сведению задачи анализа помеченных данных к задаче из класса IFDS. В пятой части рассматривается проблема анализа псевдонимов в контексте анализа помеченных данных. В шестой части приводится рад оптимизаций, позволяющих существенно уменьшить потребление памяти алгоритма. В седьмой части обсуждаются результаты тестирования реализации алгоритма анализа. Восьмая часть представляет собой заключение данной работы.

2. Существующие подходы к анализу помеченных данных.

Для начала сформулируем задачу межпроцедурного анализа помеченных данных. Пусть в программе заданы два множества функций: истоки и стоки. Задачей анализа помеченных данных является поиск всех таких параметров вызовов функций из множества стоков, что они зависят по данным от какого-либо параметра вызова функции из множества истоков. При этом считается, что все переходы по ветвлениям всегда являются допустимыми. Результатом такого анализа помеченных данных является набор путей в программе, вдоль которых помеченные данные распространяются от функции из множества истоков до функции из множества стоков.

На листинге 1 приведён пример программы на языке C, содержащий утечку помеченных данных. Данная программа обрабатывает события, полученные в ходе вызова функции `get_event`. Функция `get_event` возвращает либо событие `RECEIVE`, `PROCESS` или `SEND`, либо 0, если необходимо завершить работу. Предположим, что было получено событие `RECEIVE`, тогда в строке 11 функция source поместит помеченные данные в переменную `c`, после чего управление передаётся на строку 9. Далее, при получении события `PROCESS`, указатель на помеченные данные будет скопирован из переменной c в переменную d (строка 12). Наконец, при получении события `SEND`, помеченные данные, доступные из переменной d, передадутся в функцию sink. Таким образом, путь, на котором произойдёт утечка, будет выглядеть следующим образом: 7 – 9 – 11 – 9 – 12 – 4 – 12 –9 – 13.

```
1 char get_event(void); void source(void **c); void
sink(void **c);
2
3 void copy(void **a, void *b) {
```

```
4       *a = b;
5  }
6  int main(void) {
7       char *c, *d, e;
8       do {
9             e = get_event();
10            switch (e){
11                  case RECEIVE: source(&c); break;
12                  case PROCESS: copy(&d, c); break;
13                  case SEND: sink(&d); break;
14            }
15      } while (e);
16      return 0;
17 }
```

Листинг 1. Пример утечки помеченных данных.

Разрабатываемый в ИСП РАН инструмент статического анализа Svace осуществляет анализ помеченных данных для поиска уязвимостей типа «использование непроверенных входных данных»[1]. В настоящий момент Svace использует инфраструктуру анализа общего назначения для поддержки широкого класса проверок. Данная инфраструктура отвечает за построение модели памяти, основанной на графе объектов, и множества допустимых значений для данных объектов. Анализ функций программы осуществляется в обратном топологическом порядке относительно графа вызовов. Данный подход позволяет одновременно использовать целый набор проверок, что положительно сказывается на суммарном времени анализа, однако в случае, если функции из множеств истоков и стоков находятся достаточно далеко друг от друга, точность инструмента Svace оказывается недостаточной.

Недостаточная точность связана с компромиссом между точностью и масштабируемостью, к которому вынуждены были прибегнуть разработчики Svace. Данный компромисс заключается в жёстком ограничении размера графа объектов: если в ходе анализа функции размер графа объектов превышает допустимый максимум, система автоматически удаляет одну из вершин из графа в соответствии с заранее заданным критерием. Так как максимально допустимое число вершин в графе, как правило, намного меньше количества объектов в программе, то для достаточно длинной цепочки зависимостей можно почти гарантированно утверждать, что один из объектов, участвующих в данной цепочке, будет удалён при анализе. При этом изменение стратегии удаления вершин не меняет суть дела, так как анализ идёт в обратном топологическом порядке и в момент удаления ещё неизвестно, какие объекты будут помечены, а какие нет.

В университете Райса был предложен подход[2] к анализу помеченных данных на основе адаптации алгоритма Sparse Conditional Constant

Propagation(SCCP)[3]. Данный подход предполагает использование в SCCP вместо решётки для констант решётки из двух элементов «помеченные данные» и «непомеченные данные». Для моделирования памяти предлагается использовать Array SSA Form, представляя всю память как один массив. Однако в оригинальной статье утверждается[4], что Array SSA From для SCCP работает лишь с константными индексами массива и поддержка символьных значений является предметом дальнейших исследований. С другой стороны, тезис об использовании Array SSA Form никак более не поясняется, таким образом невозможно причины выбора данного представления.

Команда разработчиков проекта Flowdroid из EC SPRIDE предложила свести задачу анализа помеченных данных к задаче из класса IFDS[5]. Их главной целью было создание средства анализа Java-приложений для платформы Android, позволяющего находить утечки пользовательских данных, поэтому точность анализа была основным приоритетом. Предложенный ими подход является потоково-,объектно- и контекстно-чувствительным, однако объём потребляемой памяти оставляет желать лучшего. В ходе тестирования средства анализа Flowdroid было установлено, что на достаточно больших реальных приложениях (около десяти тысяч рёбер в графе вызовов) анализ не успевает завершиться до того, как израсходует всю доступную память (При тестировании для анализа выделялось 24 гигабайта оперативной памяти). Более того, другие пользователи проекта FlowDroid также подтверждают нехватку памяти для завершения анализа [6]. Так как целью данной работы является, прежде всего, анализ достаточно больших приложений, то подход проекта FlowDroid должен быть доработан в сторону уменьшения потребления памяти.

3. Класс задач IFDS

Как уже отмечалось ранее, класс задач IFDS[7] представляет собой класс межпроцедурных задач анализа потоков данных с конечным множеством фактов и дистрибутивными передаточными функциями (В данной работе нас интересуют только задачи с операцией объединения в качестве meet-оператора). При этом дистрибутивность передаточных функций означает, что для любого факта d, подмножества фактов анализа потоков данных D' и передаточной функции $F_{<v,u>}$ верно, что $F_{<v,u>}(d \cup D') = F_{<v,u>}(d) \cup F_{<v,u>}(D')$. Из данного определения следует, что для пустого множества и любого факта d всегда верно, что $F_v(\emptyset) \subseteq F_v(d)$. Поэтому введём дополнительный факт λ и новую передаточную функцию F'_v, такую, что:

$$F'_{<v,u>}(d) = F_{<v,u>}(d) \setminus F_{<v,u>}(\emptyset);\ F'_{<v,u>}(\lambda) = F_{<v,u>}(\emptyset) \cup \lambda;\ F'_{<v,u>}(\emptyset) = \emptyset.$$

Далее будем считать, что над передаточными функциями уже было проделано данное преобразование и выделен так называемый «пустой» факт λ.

Введём понятие точного решения для задачи из класса IFDS. Для этого рассмотрим все корректные пути в программе (под корректностью понимается, что при очередном возврате из функции управление должно быть передано именно в точку возврата предыдущего вызова данной функции в данном пути). Будем говорить, что для вершины u межпроцедурного графа потока управления верен факт анализа потока данных d, если найдётся такой путь $P = (e, v_1, v_2, \dots v_m, u)$, где e – точка входа в программу, что $d \subseteq F_{<v_m,u>}(F_{<v_{m-1},v_m>}(\dots F_{<e,v_1>}(\lambda)))$, т.е. d принадлежит к композиции передаточных функций. Тогда под точным решением будем понимать множество пар $< v, D_v >$, где v – вершина из межпроцедурного графа потока управления, а D_v – множество всех верных для неё фактов анализа потоков данных.

Для нахождения точного решения[7] задачи из класса IFDS было предложено сведение задачи межпроцедурного анализа потока данных к задаче достижимости на графе. Для этого межпроцедурный граф потока управления G преобразуется в расширенный граф G'. Вершины расширенного графа представляют собой декартово произведение $V_{G'} = V_G \times D$, где V_G – множество вершин в исходном графе, а D – множество фактов анализа потоков данных. Наличие ребра между вершинами в графе G' определяется следующим образом: между вершинами $< v, d > u\ < u, d' >$ есть ребро тогда и только тогда, когда ребро $< v, u > \in G\ u\ d' \in F_{<v,u>}(d)$. Такое определение графа G' с учётом дистрибутивности передаточных функций гарантирует тот факт, что существование корректного пути из начальной вершины $< e, \lambda >$ в вершину $< v, d >$ равносильно тому, что факт d верен в вершине v. Стоит отметить, что контекстная чувствительность анализа достигается как раз за счёт требования существования корректного пути.

Для поиска точного решения был предложен алгоритм, основанный на динамическом программировании, позволяющий найти его за время $O(|D|^3 * |E|), где\ |D|$ – мощность множества фактов анализа потока данных, а $|E|$ – мощность множества ребёр в графе G. Оригинальный алгоритм требовал построения графа G' в явном виде, однако граф G' может содержать до $O(|D| * |E|)$ вершин и $O(|D|^2 * |E|)$ рёбер, что делает невозможным его построение для реальных программ при хоть сколько-нибудь большой мощности множества D. К счастью, граф G' зачастую оказывается

несвязанным, и из начальной вершины $< e, \lambda >$ доступна лишь незначительная его часть. Поэтому впоследствии были предложены улучшения оригинального алгоритма, не требующие более построения G' в явном виде. В данной работе используется вариация такого алгоритма: пожертвовав полной поддержкой рекурсивных функций, была достигнута возможность полного анализа пары «функции и её контекста».

```
1 Global FunctionValues = Map(),Result = Map()
2 Function DFSBFS(F, C)
3   V = F.getEntry(), Q = Result.getQueue(<F, C>)
4   while (!Q.isEmpty())
5     V, D = Q.pop()
6     if (V.isCall())
7       DestF = V.getDest()
8       DestD = caller2callee(V, D)
9       if (not FunctionValues.contains(<DestF, DestD>))
10         DFSBFS(DestF, DestD)
11      for D' in callee2caller(V, FunctionValues
[<DestF, DestD>])
12        Q.addIfNotVisited(<V.getReturnSide(), D'>)
13     for D' in call2return(V, D)
14        Q.addIfNotVisited(<V.getReturnSide(), D'>)
15    else if (V.isReturn()) FunctionValues[<F,
C>].insert(D)
16    else for V' in V.succs()
17      for D' in normal(V, V', D)
18.       Q.addIfNotVisited(<V', D'>)
```

Листинг 2. Псевдокод алгоритма решения задачи IFDS.

Рекурсивная функция DFSBFS (листинг 2) строит для каждой функции и контекста её вызова множество посещённых в ней вершин графа G'(глобальная переменная Result). Разберём подробнее алгоритм работы данной функции. В начале выполнения функции (строчка 3) происходит получение точки входа в функцию – вершины V – и получение очереди Q, поддерживающие две операции: добавить пару «вершина и факт анализа потока данных» в очередь, если они до этого ещё не были добавлены, и получить очередную пару из очереди. Далее, пока очередь не пуста, происходит обход в ширину вершин графа G'(4-18). Отдельно рассматривается случай, когда вершина графа G представляет собой вызов функции(6-14). Основная идея алгоритма заключается в замене повторного анализа функций с данным контекстом на использование уже ранее

посчитанного множества фактов анализа потоков данных. Поэтому, если вызываемая функция с полученным контекстом ещё не анализировалась, то необходимо вызвать её анализ при помощи рекурсивного вызова функции DFSBFS(9-10), после этого результат анализа будет гарантированно содержаться в переменной `FunctionValues`. Далее полученные значения преобразуются с помощью передаточной функции Callee2Caller, и новые вершины добавляются к обходу в ширину, если это необходимо (11-12). Несложно показать, что данный алгоритм, так же, как и исходный, будучи запущенным от точки входа программы и факта λ, будет иметь вычислительную сложность $O(|D|^3 * |E|)$, однако при этом будут проанализированы лишь вершины, доступные из начальной вершины.

Заметим, что в реальной реализации данного алгоритма для каждой посещённой вершины сохраняется её предок с целью обеспечить возможность восстановления пути после нахождения обнаружения утечки. Поэтому после окончания анализа данные о посещённых вершинах не могут быть освобождены и хранятся всё время работы алгоритма, составляя основную часть расходов оперативной памяти.

4. Сведение задачи анализа помеченных данных к задаче из класса IFDS

Данный раздел посвящён заданию множества фактов анализа потоков данных и передаточных функций, позволяющих решать задачу анализа помеченных данных. Так же, как и разработчики проекта Flowdroid, в данном алгоритме используется модель памяти, основанная на путях доступа (access path).

Путь доступа представляет собой имя переменной в программе, к которой последовательно применялись операции смешения на константу (в байтах) и разыменования. Соответственно, использование путей доступа в качестве фактов анализа потоков данных позволяет сделать анализ объектно-чувствительным. Например, если в программе имеется переменная int **p, то на её основе могут быть получены такие пути доступа, как *(p + 4) или же *(*(p + 8)). Формально путь доступа задаётся именем переменной N и последовательностью смещений и разыменований. Так как два подряд идущих смещения гарантированно можно объединить в одно, то будем считать, что для задания смещений и разыменований достаточно одной последовательности смещений, если считать, что между двумя смещениями обязательно происходит разыменование. Таким образом, путь доступа формализуется как имя переменной и последовательность чисел.

Для построения передаточных функций необходимо ввести операции над путями доступа. Всего в данной работе будут рассмотрены четыре операции по преобразованию путей доступа и один предикат, задающий частичный порядок.

Для пути доступа WholeAP путь SubAP является подпутём при выполнении следующих условий. Во-первых, пути доступа WholeAP и SubAP имеют одно и то же имя переменной. Во-вторых, длина последовательности смещений у SubAP меньше либо равна длине последовательности у WholeAP. В-третьих, все смещения, кроме последнего, у SubAP совпадают со смещениями у WholeAP, а последнее либо совпадает, либо равно 0. Например, для доступа *(*(p + 4)+2) + 0 путь p + 0 является подпутём доступа, а путь *(p + 0) + 0 – нет.

Операция Callee2Caller(CallerAP, CalleeAP) используется для объединения пути доступа CalleeAP с путём доступа CallerAP. Семантика данного объединения задаётся следующим выражением, где символ “;” отделяет имя переменной от последовательности смещений.

$$CallerAP = (A; a_0, \dots a_n);$$

$$CalleeAP = (B; b_0, \dots b_m);$$

$$Callee2Caller = (A; a_0, \dots, a_n + b_0, b_1, \dots, b_m)$$

Операция Caller2Callee(CallerAP, Fact, B) используется для передачи пути доступа из вызывающей функции в вызываемую функцию. Символ B означает имя аргумента для нового пути доступа в вызываемой функции. Предполагается, что путь CallerAP является подпутём для пути доступа Fact.

$$CallerAP = (A; a_0, \dots a_n);\ Fact = (A;\ a_0, \dots a_{n-1}, c_0, \dots c_m)$$

$$Caller2Callee = (B;\ c_0 - a_n, c_1 \dots c_m)$$

Операция ChangeSubPath(OldAP, NewAP, AP) используется для моделирования копирования из одного пути доступа в другой путь доступа. В данном случае OldAP является подпутём для AP.

$$OldAP = (A; a_0, \dots a_n);\ NewAP = (B; b_0, \dots b_m);$$

$$AP = (A; a_0, \dots a_{n-1}, c_0, \dots c_m)$$

$$ChangeSubPath = (B;\ b_0 \dots b_m - a_n + c_0, c_1 \dots c_m)$$

Операция Deref(AP) используется для разыменования пути доступа AP.

$$AP = (A; a_0, \dots a_n);$$

$$Deref = (A; a_0, \dots a_n, 0);$$

Для поддержки такого случая смещения переменной на не константное значение, введём специальное смещение «*» со следующей семантикой: «*» равно любому числу и в том числе «*», операции сложения и вычитания «*» с использованием в качестве операнда всегда дают результат «*». Множество значений смещений может быть дополнено значением «*» с сохранением семантики введённых операцией.

В LLVM-биткоде результат выполнения каждой инструкции, если он имеется, сохраняется в псевдорегистр, представленный в SSA-форме. Для введения передаточных функций рассмотрим аналоги наиболее важных инструкций из LLVM-биткода.

- *store(B, A)* – сохранить по адресу A значение B
- *val = load(A)* – загрузить значение по адресу A в псевдорегистр val
- *val = getelementptr(A, offset)* – прибавить к адресу A offset байт и поместить в псевдорегистр val (на самом деле в llvm инструкция getelementptr работает не байтовыми смещениями, а со смещениями, исчисляющихся в членах типа, на который указывает A, но для упрощения повествования, будем считать, что getlementptr оперирует непосредственно с байтовыми смещениями).
- *val = call(foo, [arg1, arg2, ...])* – вызов функции foo.
- *ret [A]* – возврат из функции, если необходимо возвращается значение A
- *val = phi(val1, label1, val2, label2, ...)* – Фи функция для значений val1, val2 и.т.д, соответствующих базовым блокам label1, label2 и т.д.
- *val = binop(A ,B)* – бинарная операция над значениями, результат сохраняется в псевдорегистр *val*.
- *val = unop(A)* – унарная операция, результат сохраняется в псевдорегистр *val*.
- *val = alloca(type)* – операция выделения sizeof(type) памяти на стеке, в псевдорегистр val помещается указатель на выделенную память.

Так как данный статический анализ игнорирует условия на переходах, заменим стандартные инструкции перехода на jmp и jmpcond.

- *jmp label* – безусловный переход на метку label.
- *jmpcond label* – переход на метку label либо же переход на следующую инструкцию.

Сопоставим каждому псевдорегистру и аргументу функции путь доступа следующим образом. Аргументу функции с именем name соответствует путь доступа $name + 0$, псевдорегистрам инструкций val = phi(…), val = call(…) и val = alloca(size) соответствует путь доступа val + 0. Псевдорегистру инструкции val = load(A) соответствует путь доступа *(AP(A)), где AP(A) – путь доступа, соответствующий значению A. Аналогично, для инструкции val = getelementptr(A, size), псевдорегистру val будет соответствовать путь доступа AP(A) + size.

На листинге 3 приведён пример биткода, соответствующего примеру программы, содержащей утечку данных (листинг 1). В комментариях после инструкций приведены пути доступа, соответствующие их псевдорегистрам.

```
1  declaration i8 get_event(void)
2
3  definition void copy(i8 **a, i8 *b) // a + 0 и b + 0
```

```
4    store b, a
5    ret
6
7  definition int main(void)
8   i8** c = alloca(i8*)// c + 0
9   i8** d = alloca(i8*)// d + 0
10  loop_start:
11    i8 e = get_event()// e + 0
12    jmpcond receive
13    jmpcond process
14    jmpcond send
15    jmp end
16    receive:
17      call source(c)
18      jmp end
19    process:
20      c_value = load(c) // *(c + 0)
21      call copy(d, c_value)
22      jmp end
23    send:
24      call sink(d)
25      jmp end
26    end:
27    jmpcond loop_start
28  ret 0
```

Листинг 3. Примера утечки помеченных данных .

Реализуем передаточные функции Normal, Caller2Callee, Callee2Callee и Call2Return для алгоритма DFSBFS, используя вышеописанные операции преобразования. Метод `getArg(i)` для функции возвращает путь доступа, соответствующий i-тому формальному параметру. Функция `ReportLeak(V, D)` сообщает о найденной утечке.

```
  normal(V, V', D)
1  Result = Set()
2  if (V is store(B, A))
3    if (not (deref(A) is subpath for D))
4      Result.insert(D)
5    if (B is subpath for D)
6      Result.insert(ChangeSubPath(B, deref(A), D))
7  else
8    if (V is val = phi(a1, … an))
9      for i in (1, n) if (ai is subpath for D)
10       Result.insert(ChangeSubPath(ai,val,D))
```

```
11    if (V is val = binop(A, B) and (D is subpath for A or B))
12      Result.insert(val)
13   if (V is val = unop(A) and (D is subpath for A))
14      Result.insert(val)
15   Result.insert(D)
16   return Result

  caller2callee(V, D)
1   Result = Set()
2   if V is val = call Func(a1, … , an)
3     for i in (1, n)
4       if (ai is subpath for D)
5         Result.insert(Caller2Callee(ai, D,
Func.getArg(i))
6   return Result

  callee2caller(V, D)
1   Result = Set()
2   if V is ret from val = call Func(a1, … , an)
3     for i in (1, n)
4       if (Func.getArg(i) is subpath for D)
5         Result.insert(Callee2Caller(ai, D)
6   return Result

  call2return(V, D)
1   Result = Set()
2   Flag = True
3   if V is call Func(a1, … , an)
4       for i in (1, n)
5         If (ai is subpath for D)
6           Flag = false
7     if (Flag) Result.insert(D)
8     if (Func == Source)
9       Result.insert(deref(a1))
10    if (Func == Sink and a1 is subpath for D)
11      ReportLeak(V, D)
12  return Result
```

Листинг 4. Реализация передаточных функций.

Имея передаточные функции, применим алгоритм поиска утечек помеченных данных к примеру из листинга 3. Анализ начинается со строки 8 и пустого факта λ, далее с ним посещаются все инструкции. При посещении инструкции вызова функции source (строка 17) порождается факт *(c + 0) и посещает все инструкции с 11 по 28, в том числе и вызов функции copy. При обработке

вызова функции copy с фактом *(c + 0), передаточная функция устанавливает, что *(c + 0) является подпутём для параметра c_value, которому соответствует путь доступа *(c + 0), и производит преобразование Caller2Callee, получая из факта *(c + 0) факт b + 0. После этого происходит анализ функции copy с контекстом b + 0. Он начинается с посещения фактом b + 0 инструкции копирования (строка 4), где в ходе вызова передаточной функции устанавливается, что записываемое значение является подпутём для факта b + 0, поэтому, с учётом разыменования, выполняется преобразование ChangeSubPath и порождается факт *(a + 0). После этого факт *(a + 0) достигает инструкции возврата ret и записывается в результат вызова функции copy с контекстом b + 0. После завершения анализа функции copy с контекстом b + 0, анализ функции main с контекстом λ продолжается с обратного преобразования результатов вызова. В результате преобразования Callee2Caller факт *(a + 0) функции copy преобразуется в факт *(d + 0) функции main. Далее данный факт также посещает все инструкции с 11 по 28 и инициирует анализ функции copy с контекстом *(a + 0). После достижения фактом *(d + 0) инструкции на строке 24 передаточная функция обнаружит, что данный вызов является вызовом функции из множества стоков, и путь доступа параметра вызова функции d + 0 является подпутём для факта *(d + 0), следовательно, найдена утечка помеченных данных.

5. Анализ псевдонимов

Ключевой задачей для анализа помеченных данных является анализ псевдонимов. В данной работе используется подход, схожий с подходом проекта Flowdroid. Данный подход решает задачу анализа псевдонимов как задачу из класса IFDS. Главное преимущество использующегося подхода заключается в поиске псевдонимов не для всех переменных, а лишь для переменных, содержащих помеченные данные.

Стоит заметить, что данный алгоритм ищет не псевдонимы в обычном смысле, а такие пути доступа (назовём их неактивными путями доступа), которые при выполнении определённой инструкции (назовём её инструкцией активации) будут содержать помеченные данные. Иными словами, они являются псевдонимом для какого-то другого пути доступа, при выполнении записи помеченных данных в который данные в них также окажутся помеченными.

Поиск данных путей доступа осуществляется лениво: при очередной операции записи в память или же возврата из функции, в ходе которых образовались новые помеченные данные, для каждого нового помеченного пути доступа происходит поиск связанных с ним неактивных путей доступа. Для поиска неактивных путей доступа используется комбинация прямой и обратной IFDS-задач. Оба анализа используют множество фактов, основанное на паре «путь доступа и инструкция активации». Для начального факта инструкция активации совпадает с инструкцией, породившей новый факт, а путь доступа с

новым помеченным путём доступа. Передаточные функции для обратного анализа задаются следующим образом. Функции caller2callee и callee2caller не порождают никаких фактов, т.к. на данный момент используется внутрипроцедурная версия обратного анализа. Функция call2return передаёт полученный факт через точку вызов, реализация функции normal приведена на листинге 5. Функция `InjectForward(V, D)` добавляет пару `<V, D>` в прямой анализ псевдонимов, в тот же контекст, из которого была вызвана функция normal.

```
    normal(V, V', D)
1  Result = Set()
2  if (V is store(B, A))
3    if (deref(A)is subpath for D)
4      injectForward(V, D)
5      Result.insert(ChangeSubPath(deref(A), B, D))
6    else
7      Result.insert(D)
8  else if (V is val = phi(a1, …, an))
9    if (val is subpath for D)
10    for i in (1, n)
11      Result.insert(ChangeSubPath(val, ai, D))
12    injectForward(V, D)
13  else
14     Result.insert(D)
15 else if (V is val = alloca(…) or val = call(…) or V
is entry)
16   injectForward(V, D)
17 else
18   Result.insert(D)
19 return Result
```

Листинг 5. Обратная передаточная функция.

Обратим внимание на то, что как только факты анализа достигают точки своего определения (строки 4, 12 и 16), они передаются обратно прямому анализу. В данном случае прямой анализ использует те же передаточные функции, что и обычный анализ помеченных данных, с той лишь разницей, что инструкции unop и binop более не порождают помеченные данные. Более того, если прямой анализ обнаруживает запись помеченных данных в новый путь доступа, он запускает обратный анализ, беря инструкцию активации из факта. Если же факт достигает своей инструкции активации, то он дальше не передаётся, а вместо этого возвращается в анализ помеченных данных уже активированным.

```
1  definition copy(i8 **a, i8 *b)
2  a.addr = alloca (i8***)
```

```
3  b.addr = alloca (i8**)
4  store a, a.addr
5  store b, b.addr
6  a_value = load a.addr
7  b_value = load b.addr
8  store b_value, a_value
9  ret
```

Листинг 6. Альтернативная реализация функции copy.

Для демонстрации работы анализа псевдонимов, рассмотрим альтернативную реализацию функции copy из листинга 3. Данная реализация (листинг 6) генерируется компиляторной инфраструктурой LLVM в случае, когда не используется оптимизация отображения переменных из памяти на регистры. Так же, как и в примере из листинга 3, будем считать, что функция copy анализируется с контекстом b + 0. Несложно заметить, что оригинальный алгоритм не способен найти передачу помеченных данных в путь доступа относительно аргумента a, так как она происходит неявно. Для записи фактов будем использовать следующую нотацию: «путь доступа» для активных фактов и «путь доступа, число» для неактивных фактов с инструкцией активации на строке «число».

Рассмотрим работу алгоритма с учётом анализа псевдонимов. В начале работы алгоритм проталкивает факт «b + 0» до инструкции на строке 5, где происходит его запись в путь доступа «*(b.addr + 0)» далее запускается анализ псевдонимов, который, ничего нового не обнаружив, завершается. Далее путь факт «*(b.addr + 0)» доходит до инструкции на строке 8, порождая новый факт «*(*(a.addr + 0) + 0)», для данного факта также запускается анализ псевдонимов с фактом «*(*(a.addr + 0) + 0), 8». Идя назад, данный факт встречает инструкцию на строке 4, что приводит к тому, что в прямой анализ псевдонимов добавляется факт «*(*(a.addr + 0) + 0), 8», а обратный продолжается с фактом «*(a + 0), 8». После достижения точки входа в функцию факт «*(a + 0), 8» также добавляется в прямой анализ. В свою очередь, прямой анализ проталкивает факт «*(a + 0), 8» до его инструкции активации на строке 8, активирует его и возвращает в анализ помеченных данных. После этого анализ помеченных данных посещает инструкцию на строке 9 с фактом «*(a + 0)» и добавляет его к результатам анализа функции.

К сожалению, если считать, что факты анализа потока данных, различающиеся только инструкциями активации, разные, то количество фактов анализа может увеличиться в $O(E)$ раз, что недопустимо с точки зрения масштабируемости. Для решения данной проблемы факты, различающиеся лишь инструкцией активации, считаются одинаковыми. Эта эвристика позволяет существенно уменьшить число фактов, получаемых при анализе псевдонимов, однако в случае, если один и тот же путь доступа помечается в нескольких различных инструкциях, то инструкция для

активации неактивных фактов будет выбрана произвольно. Из-за этого может возникнуть ситуация, когда неактивный факт попал в функцию из множества стоков, пройдя породившую его инструкцию, но не пройдя выбранную. На данный момент неясно, насколько такая ситуация распространена в реальных приложениях, однако без данной эвристики анализ приложений даже из ста тысяч инструкций нецелесообразен.

6. Оптимизация времени работы алгоритма

Работа анализа псевдонимов устроена таким образом, что большинство неактивных фактов при своей генерации сначала поднимаются обратным анализом до точки входа в функцию, после чего спускаются прямым анализом, посещая каждую инструкцию. Поэтому, если функция из нескольких тысяч инструкций анализируется с десятками различных контекстов и при этом генерируется несколько сотен неактивных фактов, то значительное количество памяти тратится на посещение инструкций, неспособных породить новых фактов. Для решения данной проблемы вместо передачи факта к смежным вершинам можно передавать его сразу в точку ближайшего использования относительно исходной вершины.

Если в инструкцию могут быть переданы только факты, имена переменных путей доступа которых используются в данной инструкции, то в рамках одного контекста общее число посещений данной инструкции ограничено произведением числа возможных путей доступа для данного имени на количество аргументов у инструкции. Так как число путей доступа может быть произвольно большим даже для ограниченной длины самого пути, то необходимо отдельно ограничить число поддерживаемых путей некоторой константой A. Осталось оценить число различных контекстов, в которых может быть проанализирована функция. Пусть количество аргументов у функции не превосходит K, тогда всего общее количество контекстов без учёта глобальных переменных будет равно $\mathrm{O}(\mathrm{K} * A)$. Так как всего в программе $\mathrm{O}(\mathrm{E})$ инструкций, то суммарное количество посещённых вершин, и соответственно объём занятой памяти, будут равны $\mathrm{O}(\mathrm{K} * \mathrm{A}^2 * \mathrm{E})$. При этом затраты времени окажутся равными $\mathrm{O}(\mathrm{TF} * \mathrm{K} * \mathrm{A}^2 * \mathrm{E})$, где $\mathrm{O}(\mathrm{TF})$ – временная сложность вычисления передаточной функции.

Для того чтобы иметь возможность переходить на ближайшее использование, предлагается для каждого имени, использующегося в инструкции, предподсчитать ближайшие использования. Так как в каждой функции имеется $\mathrm{O}(\mathrm{E_F})$ инструкций, а ближайшие использования могут быть найдены, например, поиском в ширину за время $\mathrm{O}(\mathrm{E_F})$, то для всех инструкций они могут быть найдены за время $\mathrm{O}(\mathrm{E_F^2})$. Т.е. суммарное время работы алгоритма для всех функций составит:

$$O(\sum_{F} E_F^2)$$

Соответственно, суммарная сложность анализа помеченных данных равна:

$$O\left(TF * K * E * A^2 + \sum_{F} E_F^2\right)$$

Однако данная оценка неверна для прямой фазы алгоритма анализа псевдонимов, т.к. инструкции, способные выступать в роли инструкций активации, должны быть посещены неактивными фактами. Данный факт приводит к тому, что инструкции store и call могут быть посещены $O(E_F * A)$ раз. Учитывая, что в реальных приложениях доля таких инструкций достигает 25%, суммарная оценка времени работы алгоритма составляет:

$$O\left(\sum_{F} TF * K * A^2 * E_F^2\right)$$

7. Результаты

Предложенный алгоритм поиска помеченных данных реализован в качестве прохода для компиляторной инфраструктуры LLVM. Кроме того, в LLVM были также реализованы вспомогательные проходы, а именно проход, отвечающий за построение межпроцедурного графа потока управления, проход, отвечающий за анализ виртуальных вызовов в C++ и проход, реализующий инфраструктуру для разработки дополнительных проверок.

Межпроцедурный граф потока управления отличается от внутреннего представления LLVM, а именно, у него выделены точки входа в функцию, точки возврата после вызова функции и построены все межпроцедурные рёбра. Для построения межпроцедурного графа потока управления был реализован анализирующий проход, преобразующий внутреннее представление LLVM в межпроцедурный граф потока управления. Для построения межпроцедурных рёбер для виртуальных вызов в C++ данный проход использует результаты анализатора виртуальных вызовов, разработанного в ИСП РАН.

Дополнительные проверки, основанные на данном анализе, предполагается реализовать с помощью модификации передаточных функций. Разработчику новой проверки предоставляется возможность проанализировать построенный межпроцедурный граф потока управления и пометить вершины, для которых должна быть запущена дополнительная реализация передаточных функций. Также он имеет возможность указать, должна ли эта реализация быть запущена до, после или же вместо оригинальной реализации. В качестве тестовой проверки была реализована проверка того, что константные значения не передаются в указанные функции.

Отчёты о найденных трассах могут быть либо показаны последовательным списком инструкций, либо же представлены в виде размеченного межпроцедурного графа потока управления, демонстрирующего последовательность распространения помеченных данных.

Прежде чем перейти к результатам тестирования, необходимо заметить, что алгоритм использует целый набор эмпирических констант, от значения которых напрямую зависят результаты анализа. В данном тестировании максимальная длина пути доступа была ограничена четырьмя, а максимальное количество путей доступа для одного имени – тридцатью.

Тестирование проводилось как на синтетических тестах, разработанных специально для тестирования данной реализации, так и на наборе реальных приложений, а именно stl vector, md5 и openvpn. Необходимо заметить, что дополнительную сложность при тестировании представляет задача сборки тестируемого проекта в LLVM-биткод, т.к. такая сборка почти никогда не поддерживается разработчиками.

Целью тестирования на stl vector была проверка способности анализа отслеживать помеченные данные, даже если они были добавлены в контейнер. Так как stl vector является шаблонным классом, то его исходный код автоматически оказывается доступным в виде LLVM-биткода при анализе. Тестирование показало, что реализованный алгоритм может успешно отслеживать добавление и извлечение помеченных данных из контейнера stl vector.

Целью тестирования на алгоритме шифрования md5 была проверка возможности отслеживать зависимость зашифрованной строки от пароля, который изначально подвергался шифрованию. При тестировании было установлено, что выходной буфер алгоритма зависит от изначальной строки.

Целью тестирования на приложении openvpn (свободное клиент-серверное приложения для установления vpn-соединений) было установление того, что данные, посылаемые в зашифрованный канал, зависят от данных, полученных из сети. Данная зависимость была успешно установлена.

Название	Количество LLVM-инструкций	Время работы	Потребление памяти	Количество шагов анализа помеченных данных	Количество шагов анализа псевдонимов
stl vector	порядка 100	0.03 сек.	Меньше 10МБ	7383	18181

md5	порядка 1000	0.1 сек	Меньше 10МБ	25540	32806
openvpn	порядка 100 000	58 сек	Около 2ГБ	13840217	26471821

Табл. 1. Результаты тестирования.

Тестирование проводилось на компьютере с процессором Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz и 32 гигабайтами оперативной памяти. В табл 1. содержатся численные результаты тестирования. Однако, как было замечено [8], в случае анализа помеченных данных, эффективность подходов и время работы может кардинально различаться в зависимости от классов приложений и использующихся стоков и истоков, поэтому данные, приведённые в табл. 1, носят относительный характер.

8. Заключение

В данной работе рассмотрена задача анализа помеченных данных и предложен алгоритм её эффективного решения. Данный алгоритм реализован на базе компиляторной инфраструктуры LLVM. Проведено тестирование данной реализации на наборе реальных программ. Тестирование показало, что предложенный алгоритм способен эффективно находить утечки данных в программах среднего размера (несколько сотен тысяч инструкций). Учитывая, что основной целью данной работы является достижение линейной зависимости числа шагов от количества инструкций в анализируемой программе на достаточно широком наборе приложений, необходимо создать широкую базу тестовых приложений с размеченными стоками и истоками. Данный набор приложений позволит уточнить эмпирические значения, использующиеся в алгоритме, и адаптировать их к анализу больших приложений (несколько миллионов инструкций).

Список литературы

[1] В.П. Иванников, А.А. Белеванцев, А.Е. Бородин, В.Н. Игнатьев, Д.М. Журихин, А.И. Аветисян, М.И. Леонов. Статический анализатор Svace для поиска дефектов в исходном коде программ. Труды ИСП РАН, том 25, 2013, с. 231-249.

[2] Vivek Sarkar. Security Analysis of LLVM Bitcode Files for Mobile. High confidence software and system conference, 2014.

[3] Mark N. Wegman, F. Kenneth Zadeck. Constant propagation with conditional branches. ACM Trans. Program. Lang. Syst. 13, 2 (April 1991), pp.181-210.

[4] Vivek Sarkar, Kathleen Knobe. Enabling Sparse Constant Propagation of Array Elements via Array SSA Form, 5th International Symposium, SAS'98, 1998, pp 33-56.

[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps, PLDI'14. 2014

[6] Li Li, Alexandre Bartel, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, Patrick McDaniel. I know what leaked in your pocket: uncovering privacy leaks on Android Apps with Static Taint Analysis. http://arxiv.org/abs/1404.7431. 29 Apr 2014.

[7] Thomas Reps, Mooly Sagiv, Susan Horwitz. Interprocedural Dataflow Analysis via Graph Reachability. University of Copenhagen. 1994

[8] Yannis Smaragdakis, George Kastrinis, George Balatsouras. Introspective Analysis: Context-Sensitivity, Across the Board. , PLDI'14. 2014

Interprocedural taint analysis for LLVM-bitcode

V.K. Koshelev, A.O. Izbyshev, I.A. Dudina
ISP RAS, Moscow, Russia
{vedun, izbyshev, eupharina}@ispras.ru

Abstract. Today the development cycle of many application classes requires a security analysis stage. Taint analysis is widely used to check programs for different security vulnerabilities. This paper describes static interprocedural flow, context, and object-sensitive taint analysis approach for C/C++ applications. Our taint analysis algorithm is based on the Flowdroid project's approach, but in contrast to Flowdroid, which aims to analyze Java bytecode, our approach handles LLVM bitcode and pointer arithmetic. Primary drawback of the Flowdroid approach is a memory usage issue which arises during analysis of medium size applications (around 10 000 edges in the call graph). To achieve scalability of the approach, we suggest a set of heuristics which helps to significantly decrease memory usage of the algorithm. The testing of real-world applications shows that such heuristics make precise taint analysis suitable for the medium size programs. Using our approach, we implemented general taint analysis framework as an LLVM pass. Additional security checks (e.g. Use of Hard-coded Password, Information Exposure, etc.) can be implemented on top of this framework. We have also developed auxiliary passes which resolve targets of virtual calls and build interprocedural control flow graph according to the results.

Keywords: static analysis, taint analysis, IFDS, dataflow, alias analysis.

References

[1] V.P. Ivannikov, A.A. Belevantsev, A.E. Borodin, V.N. Ignat'ev, D.M. Zhurihin, A.I. Avetisjan, M.I. Leonov. Staticheskij analizator Svace dlja poiska defektov v ishodnom kode program [Static analyzer Svace for finding of defects in program source code]. Trudy ISP RAN [Proceedings of the ISP RAS], tom 25, 2013, s. 231-249.

[2] Vivek Sarkar. Security Analysis of LLVM Bitcode Files for Mobile. High confidence software and system conference, 2014.

[3] Mark N. Wegman, F. Kenneth Zadeck. Constant propagation with conditional branches. ACM Trans. Program. Lang. Syst. 13, 2 (April 1991), pp.181-210.

[4] Vivek Sarkar, Kathleen Knobe. Enabling Sparse Constant Propagation of Array Elements via Array SSA Form, 5th International Symposium, SAS'98, 1998, pp 33-56.

[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps, PLDI'14. 2014

[6] Li Li, Alexandre Bartel, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, Patrick McDaniel. I know what leaked in your pocket: uncovering privacy leaks on Android Apps with Static Taint Analysis. http://arxiv.org/abs/1404.7431. 29 Apr 2014.

[7] Thomas Reps, Mooly Sagiv, Susan Horwitz. Interprocedural Dataflow Analysis via Graph Reachability. University of Copenhagen. 1994

[8] Yannis Smaragdakis, George Kastrinis, George Balatsouras. Introspective Analysis: Context-Sensitivity, Across the Board. , PLDI'14. 2014

Внесение неисправностей в программу с использованием детерминированного воспроизведения

П. М. Довгалюк, Ю. В. Маркин
ИСП РАН, Москва
{Pavel.Dovgaluk, ustas}@ispras.ru

Аннотация: В данной работе представлен подход к внесению неисправностей в программу, использующий механизм детерминированного воспроизведения в симуляторе. Внесение неисправностей используется для проверки качества покрытия кода тестами, способности системы реагировать на некорректные данные или механизма обработки исключений. При внесении неисправностей предлагается использовать детерминированное воспроизведение работы программ, что позволит упростить инициализацию системы, ускорить ее тестирование с помощью фаззинга, а также изучить альтернативные пути выполнения программы при ее отладке.

Ключевые слова: детерминированное воспроизведение; виртуальная машина; отладка; внесение неисправностей; фаззинг; симулятор; гейзенбаг; QEMU

1. Введение

Внесение неисправностей (fault injection) – это искажение кода или данных программы с целью проверки качества покрытия кода тестами, способности системы реагировать на некорректные данные или механизма обработки исключений [1].

Внесение неисправностей имеет ряд особенностей, которые затрудняют его применение в определенных случаях. Например, при выполнении фаззинга сетевого протокола может возникнуть потребность в обеспечении одинаковой реакции сервера на каждый инициализирующий пакет, посылаемый тестируемой программой. В некоторых случаях программа сама может быть недетерминированной и из-за этого вести себя по-разному даже на одинаковых наборах входных данных.

В статье предлагается метод, позволяющий решить эти проблемы с помощью детерминированного воспроизведения. Детерминированное воспроизведение программы заключается в записи сценария ее выполнения и последующем его (возможно, многократном) воспроизведении. Детерминированное воспроизведение чаще всего используется для отладки и динамического

анализа программ. Например, при анализе ошибок в многопоточных программах возможность наблюдать один и тот же сбой при каждом запуске программы очень важна. Присутствие множества побочных факторов, влияющих на ход выполнения такой программы, приводит к тому, что повторный запуск на тех же входных данных может задействовать другую траекторию выполнения. Существует специальный термин для обозначения ошибок, для которых «вручную» отследить все условия срабатывания не представляется возможным – гейзенбаг. Без специальных средств инструментальной поддержки процесса отладки, например, детерминированного воспроизведения, гарантировать, что в сложной многопоточной программе при тех же входных данных гейзенбаг проявится в том же месте (и что он вообще проявится) практически невозможно.

Предлагается дополнить метод детерминированного воспроизведения возможностью отклоняться от записанного сценария работы посредством замены части входных данных. Пользователь сможет значительно экономить свое время при анализе и отладке программ с использованием детерминированного воспроизведения. Например, отклонение от записанного хода выполнения программы может понадобиться при изучении покрытия кода тестами, обратной отладке или тестировании методом внесения неисправностей.

2. Подобные работы

Внедрение неисправностей может выполняться на этапе компиляции (мутационное тестирование) или на этапе выполнения программы. Например, XEMU [2] реализует мутационное тестирование в симуляторе при выполнении отдельных пользовательских приложений.

Другой подход к внесению неисправностей – это модификация программы или данных во время выполнения. Существует множество систем, использующих модификацию во время выполнения. Как правило, эти системы выполняют эмуляцию аппаратных неисправностей в микросхемах памяти или процессоре, а также в шинах данных [3], [4], [5], [6]. Другой вид неисправностей – это коммуникационные неисправности – потеря, задержка, искажение или дублирование сообщений [3], [7].

Фаззинг – это генерация заведомо некорректных входных данных для программы с целью вызова ее сбоя [8]. Наиболее популярный способ фаззинга на сегодняшний день – это обращение с системой как с черным ящиком. В этом случае при генерации входных данных не используется никаких знаний о тестируемой системе, кроме, возможно, формата принимаемых ею данных. В работе [9] рассматриваются подходы к фаззингу драйверов USB-устройств. Один из них заключается в помещении между USB-устройством и его драйвером (выполняющимся в виртуальной среде) «прослойки», позволяющей модифицировать или подменять передаваемые пакеты. Другой – в реализации аппаратного устройства, имитирующего требуемый протокол. Главное

достоинство программного решения, использующего виртуальную среду – это полный контроль над выполнением программы, а также возможность сохранять и восстанавливать состояние виртуальной машины. Проблема такого подхода заключается в невозможности эффективного изменения определенного пакета после того, как множество пакетов уже было передано, ведь поведение системы может меняться от запуска к запуску.

Другой подход к фаззингу – это использование символьного выполнения программы [10]. Проводится тестирование «белого ящика» – чтобы найти входные данные, приводящие к сбою программы, производится ее символьное выполнение с последующим решением уравнений, позволяющих восстановить данные, приводящие к повышению степени покрытия кода. Так происходит поиск новых путей исполнения программы. Если в результате на одном из путей программа достигает состояния, которое определяется как сбой, то в ней есть ошибка, проявляющаяся на найденных входных данных. Этот подход доказал свою эффективность на практике, описанной в работе [10], а также широким применением в компании Microsoft при тестировании программ, работающих с файлами сложного формата [11]. В то же время, существующие реализации обладают рядом ограничений. Большинство из них (например, [11], [12]) проводит анализ лишь отдельных программ, откуда следует невозможность анализа драйверов и операционных систем. Существует и реализация полносистемного анализа, описанная в [13]. В ней все необходимые данные постоянно хранятся в памяти, что делает невозможным анализ сложных программных компонентов, либо всего программного стека конкретного протокола. Кроме того, все описанные реализации обладают тем же недостатком, что и подходы к фаззингу черного ящика – невозможность тестирования всех фаз работы драйвера, работающего по определенному протоколу.

Использование детерминированного воспроизведения для решения задач, подобных внесению неисправностей, уже становилось предметом внимания исследователей и разработчиков ПО. В отладчике EPDB [14], реализующем метод обратной отладки, существует возможность изменения входных данных, чтобы отклониться от записанного сценария. Основное ограничение этого отладчика в том, что он существует лишь для программ на языке Python. К тому же, в настоящее время проект не развивается.

Другим примером, где имеется подобная возможность, является система динамического анализа Crosscut [15]. В ней детерминированное воспроизведение используется для сбора трасс программ, выполняющихся внутри виртуальной машины. Чтобы получать данные из гостевой операционной системы и запущенных в ней процессов, Crosscut в определенные моменты времени выполняет фрагменты кода, отличающиеся от тех, что выполнялись при записи сценария работы. Например, для трассировки программы на языке Perl в интерпретатор встраивается код, активируемый на определенных шагах воспроизведения. При этом перед

выполнением кода трассировки необходимо сохранять состояние процессора и оперативной памяти, чтобы затем использовать их для продолжения воспроизведения работы виртуальной машины.

Гораздо более близкий аналог предлагаемого метода реализуется в симуляторе Simics [16], [17]. Этот симулятор позволяет выполнять обратную отладку сложных (в том числе и состоящих из нескольких компьютеров) систем. Однако симулятор является закрытым, и никаких деталей о принципах и методах, лежащих в его основе, не публиковалось.

Отличие разработанного метода от отладчиков уровня приложений (таких как EPDB) заключается в том, что он работает на уровне всей системы и поэтому может применяться для изменения поведения на более низком уровне (например, при работе в ядре операционной системы). В реализациях подобных Crosscut возможно выполнение ограниченного фрагмента кода в нужной точке воспроизведения, но невозможно создание ответвлений с продолжением работы программы от произвольной точки некоторого сценария ее работы. Кроме того, предлагаемый метод позволяет выполнять полноценный фаззинг протоколов с тестированием полного стека реализации протокола, а не отдельных приложений в рамках операционной системы. Симулятор Simics ориентирован лишь на обратную отладку и не содержит в себе механизмов для фаззинга выполняющихся в нем программ.

Описанные выше программные системы, реализующие внесение неисправностей (и, в частности, фаззинг) обладают следующими недостатками. Многократные запуски одного сценария с частично изменяющимися входными данными могут иметь фазы работы, которые не отличаются от запуска к запуску. Это приводит к выполнению лишней работы при повторении этих фаз. Кроме того, программа может по-разному себя повести при разных запусках независимо от подаваемых входных данных, что изменит ход выполнения, и при тестировании могут проявиться совсем разные дефекты.

Вместо предлагаемого детерминированного воспроизведения для перехода к определенной фазе выполнения можно использовать предварительно сохраненные снимки системы. Но снимки занимают значительное место на диске (ведь нужно как минимум делать копию оперативной памяти системы), а на их создание затрачивается больше времени, чем на запись журнала недетерминированных событий. Поэтому сохранение снимков может повлиять на работу системы, замедляя ход ее выполнения, что сделает анализ приложений реального времени невозможным.

Для проверки результатов внесения неисправностей можно использовать сравнение трасс выполнения программ. Предлагаемый метод позволяет уменьшить число и объем используемых трасс. В качестве эталона можно использовать трассу оригинального сценария выполнения, а дополнительные трассы получать, начиная воспроизведение с моментов внесения изменений. Кроме того, использование детерминированного воспроизведения

гарантирует, что отличия в трассах выполнения будут обусловлены только внесенными неисправностями, а не какими-либо другими внешними факторами.

3. Описание метода

Детерминированное воспроизведение работы системы означает, что во время записи сценария работы с ней создается журнал недетерминированных событий.

Но полностью детерминированное (без возможности внести изменения) воспроизведение не всегда удобно. Например, при тестировании или отладке программ бывает необходимо изменить часть данных, поступающих в систему. В этом случае пользователю придется записать новый сценарий работы с программой. При этом в ряде случаев он может столкнуться с затруднениями:

1. Сценарий достаточно длительный, а изменения нужно внести ближе к его концу.
2. На работу программы влияют источники недетерминизма (многопоточность, неинициализированная память, внешняя среда), из-за чего сложно воспроизвести ту же самую ошибку, которая была записана в оригинальном сценарии.
3. Подготовка к запуску сценария требует сложной настройки окружения.

Поэтому предлагается не записывать новый сценарий, а создавать ответвление от текущего посредством изменения некоторых данных, поступающих из журнала в определенный момент воспроизведения. На рис. 1 изображен пример. Во время воспроизведения сценария программы в точке B была произведена замена записанных входных данных на другие. В результате, выполнение пошло по другому пути, и программа перешла в состояние C' вместо C, которое было бы достигнуто при нормальном воспроизведении.

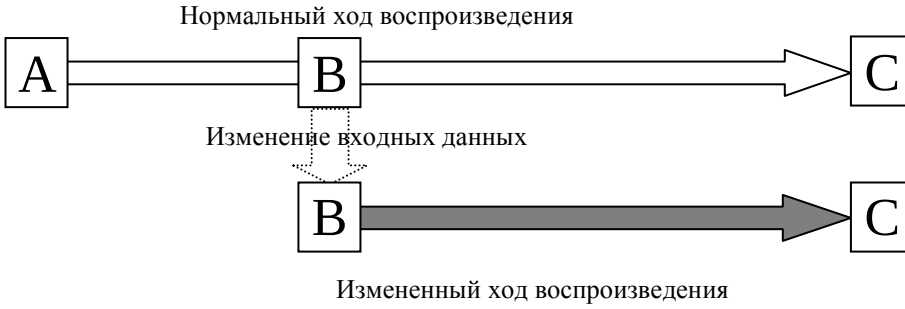


Рис. 1. Изменение состояния программы во время ее воспроизведения.

В рамках работы было реализовано несколько способов перехода к выполнению программы из режима ее воспроизведения. В зависимости от того, с какой целью восстанавливается выполнение программы, варианты могут быть следующие:

1. Изменение переменных, регистров или ячеек памяти с помощью отладчика.
2. Изменение получаемого сетевого пакета, записанного в журнал недетерминированных событий.
3. Включение с помощью специальной команды в произвольный момент воспроизведения журнала. Этот способ может быть использован для того, чтобы изменить входные данные, подаваемые в программу через ее пользовательский интерфейс.

В режиме воспроизведения работы программы весь ввод данных в систему симулируется – соответствующие события загружаются из журнала и передаются виртуальным устройствам ввода (мышь, клавиатура, сетевая карта, микрофон). После переключения на режим выполнения, виртуальные мышь, клавиатура и микрофон снова становятся доступны для передачи команд от пользователя.

Во время воспроизведения работы системы весь ввод в нее поступает из заранее записанного журнала событий. В тот момент, когда воспроизведение прерывается и начинается нормальное выполнение, необходимо прекратить чтение событий из журнала и начать взаимодействие с пользователем.

Основная проблема при переключении на источники входных данных заключается в том, что источники реального времени также считаются внешними устройствами и их показания записываются в журнал событий. Поэтому при переходе в режим выполнения программы из режима воспроизведения, показания счетчиков времени должны соответствовать новому состоянию системы.

Реализация детерминированного воспроизведения, используемая в данной работе, основана на симуляторе QEMU [18] и подробно описана в работе [19]. Для реализаций платформ i386 и ARM, которые поддерживают детерминированное воспроизведение, симулируются следующие аппаратные счетчики времени:

- TSC (Time Stamp Counter) – счетчик тактов с последнего сброса процессора. Возвращается как результат работы инструкции RDTSC процессора i386.
- i8254 – микросхема программируемого таймера, используемая начиная с самых первых моделей IBM PC.
- MC146818 – микросхема часов реального времени, впервые появившаяся в IBM PC/AT.

- HPET (High Precision Event Timer) – таймер событий высокой точности, призванный заменить i8254 и MC146818 в задачах замера небольших промежутков времени.
- ACPI PM timer (Advanced Configuration and Power Interface Power Management timer) – таймер в составе интерфейса управления конфигурацией и питанием.
- PL031 – часы реального времени ARM PrimeCell.
- DS1338 – микросхема часов реального времени с интерфейсом I2C.

Программа не должна «заметить» переключение симулятора из режима воспроизведения в режим выполнения по необычной работе одного из таймеров – в противном случае работа программы может по тем или иным причинам измениться по сравнению с ожиданиями пользователя. Это изменение может быть случайно или намеренно заложено программистом. Например, в [20] описывается ряд антиотладочных приемов, основанных на использовании инструкции RDTSC.

В QEMU все виртуальные счетчики времени реализованы через использование нескольких источников реального времени хостовой машины. Они же используются для измерения периодов, не относящихся к часам, например, промежутков времени между обновлениями экрана:

- Функция QueryPerformanceCounter (Windows) или clock_gettime (Linux)
- Функция gettimeofday
- Функция gmtime_r
- Функция time
- Счетчик тактов хостового процессора

Все эти источники являются внешними для виртуальной машины, и поэтому обращения к ним записываются в журнал событий. Во время воспроизведения нужные показания источников реального времени извлекаются из журнала. Таким образом, в случае переключения из режима воспроизведения в режим выполнения программы значения таймеров моментально изменятся на текущие для хостовой машины значения, поскольку записанные показания таймеров отличаются от текущих.

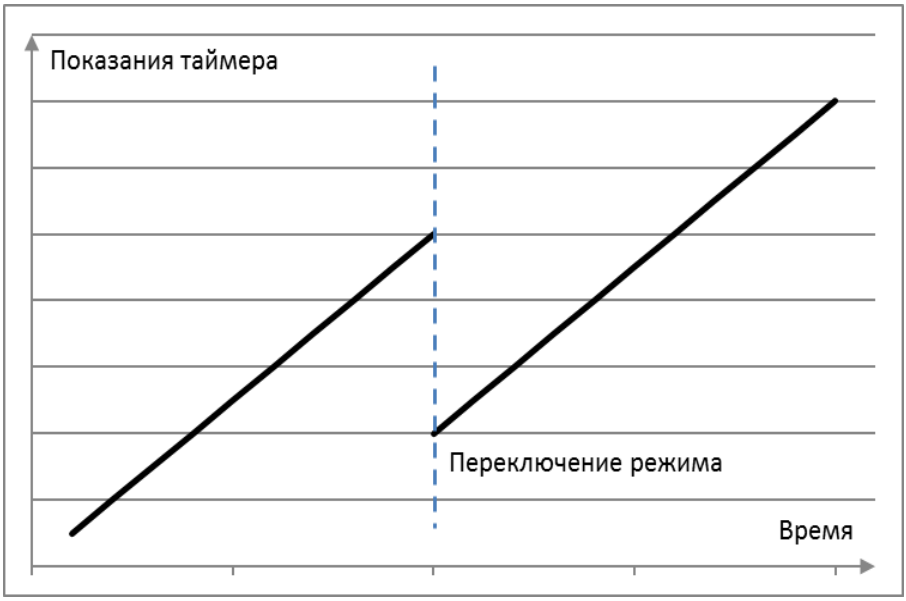


Рис. 2. Изменение показаний таймера при переключении из режима воспроизведения в режим выполнения.

Чтобы такого «скачка времени» не происходило, необходимо изменить способ вычисления значений таймеров в режиме выполнения. При передаче показаний таймера хостовой машины виртуальным устройствам нужно вычитать из них смещение – разницу между показаниями таймера хоста и показаниями таймера, считанными из журнала событий в тот момент, когда происходит переключение режима.

С подключением реальных устройств ввода к клавиатуре, мыши или микрофону таких проблем не возникает, так как они получают извне только требуемые изменения в своем состоянии. Поэтому скачков в передаваемых в гостевую систему значениях не происходит.

Совершенно иначе обстоит дело с сетевым адаптером. Здесь получаемые системой пакеты формируются на удаленной машине. А ее состояние невозможно восстановить при переходе программы в режим выполнения. Поэтому при подмене сетевого пакета дальнейшее взаимодействие программы с внешними сервисами уже не имитируется. Тем не менее, можно проверить непосредственную реакцию программы на заменяемый пакет с учетом всего предыдущего взаимодействия (т.е. осуществлять фаззинг сетевого протокола). Но ту часть взаимодействия с сервером, которая должна следовать за подмененным пакетом, проверить этим способом не удастся.

4. Применение метода

В качестве примера продемонстрируем возможности предлагаемого метода для фаззинга сетевого протокола. Для этого было разработано приложение, подключающееся к запущенному симулятору и выполняющее несколько итераций фаззинга заданного сетевого пакета. Каждая итерация состоит из трех этапов:

- переход на начальный шаг исследуемого сценария;
- изменение содержимого входящего сетевого пакета;
- продолжение выполнения программы до момента наступления некоторого события.

Отличительная особенность фаззера – восстановление контекста обработки сетевых пакетов на каждой новой итерации (для stateful-протоколов). Это происходит благодаря детерминированному воспроизведению при переходе на начальный шаг исследования – в каждой итерации будет получено одно и то же состояние системы после выполнения этого перехода.

4.1. Входные параметры

Чтобы начать фаззинг сетевого приложения, необходимо задать следующие параметры:

- Список динамических библиотек, производящих разбор (построение) сетевых пакетов. Для того чтобы модифицировать сетевые пакеты, необходимо осуществлять разбор этих пакетов. Структура полезной нагрузки сетевого пакета определяется приложением, в работе которого данный сетевой пакет используется. Для каждого нового формата создается динамическая библиотека, в которую помещается код разборщика (построителя) сетевых пакетов в соответствии с заданным форматом.
- Идентификатор сетевого пакета, содержимое которого в рамках данного исследования будет изменяться. Для этого пакета необходимо указать динамическую библиотеку, при помощи которой будет производиться разбор (построение), а также файл с набором параметров, необходимых для разбора (например, ключи шифрования)
- Список целевых адресов. Если значение счетчика инструкций процессора совпадет хотя бы с одним значением из этого списка, задача фаззинга будет считаться выполненной. По целевому адресу не обязательно должен располагаться код анализируемого сетевого приложения – там, к примеру, может находиться код динамической библиотеки, которую анализируемое приложение использует в своей работе.

- Начальный и конечный шаги исследования. Шаги журнала событий, которые соответствуют получению выбранных для модификации сетевых пакетов, должны лежать в диапазоне начального и конечного шагов исследования. Попадание управления на какой-либо адрес из списка целевых адресов должно произойти в диапазоне начального и конечного шагов исследования. Таким образом, в качестве начального шага исследования всегда подходит шаг журнала событий, соответствующий получению сетевого пакета, содержимое которого будет изменено. В качестве конечного шага исследования можно выбирать достаточно большие значения. В то же время, чем меньше разность конечного и начального шагов, тем меньше времени потребуется для проведения одной итерации исследования.
- Модифицирующий файл. Необходимо указать файл, содержимое которого будет использовано для модификации сетевого пакета.

4.2. Критерий остановки

Выполнение программы, использующей измененный сетевой пакет, прекращается, если наступает одно из событий:

- достижение конечного шага исследования;
- аварийное завершение работы эмулятора.

Каждая итерация характеризуется отдельным файлом, в который записываются инструкции, исполняемые симулятором во время этой итерации. В случае наступления одного из перечисленных выше событий, сохранение исполняемого кода прекращается и начинается его анализ. Список целевых адресов сопоставляется с адресами инструкций файла данной итерации. Если найдено хотя бы одно совпадение, анализатор принимает решение о прекращении исследования. В противном случае начинается новая итерация. В случае аварийного завершения работы симулятора происходит его перезапуск.

4.3. Тестирование

Анализ корректности работы фаззера проводится на контрольном примере. Используется предварительно записанный сценарий выполнения программы «FuzzTest.exe», а также файл «Original.pcap», в котором сохранен трафик виртуального сетевого интерфейса симулятора. Программа «FuzzTest.exe» отправляет сетевой пакет, содержащий запрос ip-адресов серверов с доменным именем «lenta.ru». После получения ответа программа сравнивает значение последнего из пришедших ip-адресов с ip-адресом «0.0.0.0». В случае совпадения происходит ошибка – разыменование нулевого указателя.

Перед запуском фаззера необходимо найти адрес инструкции, выполнение которой приводит к ошибке. Это можно сделать с помощью дизассемблера IDA Pro. Интересующая нас инструкция находится по адресу «0x40135E». Для

работы с пакетами протокола DNS используется модуль «Nslookup.dll». В окне добавления сетевого пакета вводим идентификатор пакета (349). Никаких дополнительных параметров в данном случае не требуется.

После нажатия на кнопку «Start Fuzzing» появится окно, отображающее промежуточное представление выбранного сетевого пакета (рис. 3). Необходимо выбрать *модифицирующий* файл. Каждый модифицирующий файл представляет собой последовательность блоков. Блок включает в себя идентификатор вершины промежуточного представления, а также набор новых значений для модификаций:

```
[id_1]
id_1_new_value_1
id_1_new_value_2
...
id_1_new_value_n_1

[id_2]
id_2_new_value_1
id_2_new_value_2
...
id_2_new_value_n_2
...
[id_m]
id_m_new_value_1
id_m_new_value_2
...
id_m_new_value_n_m
```

В качестве модифицирующего использовался файл следующего содержания:

```
[24]
01020304
19203940
00000000
```

На рис. 3 тег с идентификатором «24» соответствует ip-адресу, который будет изменяться. В данном случае будет выполнено три итерации, то есть ip-адрес будет изменен три раза, так как именно столько различных IP-адресов записано в модифицирующем файле. По окончании работы фаззера в консоль выводится сообщение «Fuzzing completed». В папке «FuzzerLog» появилась папка с результатами фаззинга, содержащая четыре файла: «asm_log_1.txt», «asm_log_2.txt», «asm_log_3.txt», «fuzz_log.txt». В первых трех файлах находится ассемблерный код, исполняемый виртуальной машиной. В последнем файле для каждой проведенной итерации сохранен список соответствующих изменений сетевого пакета. В поле «Result» последней итерации записано значение «success». Это значит, что на последней итерации симулятор выполнил интересующую нас инструкцию по адресу «0x40135E».

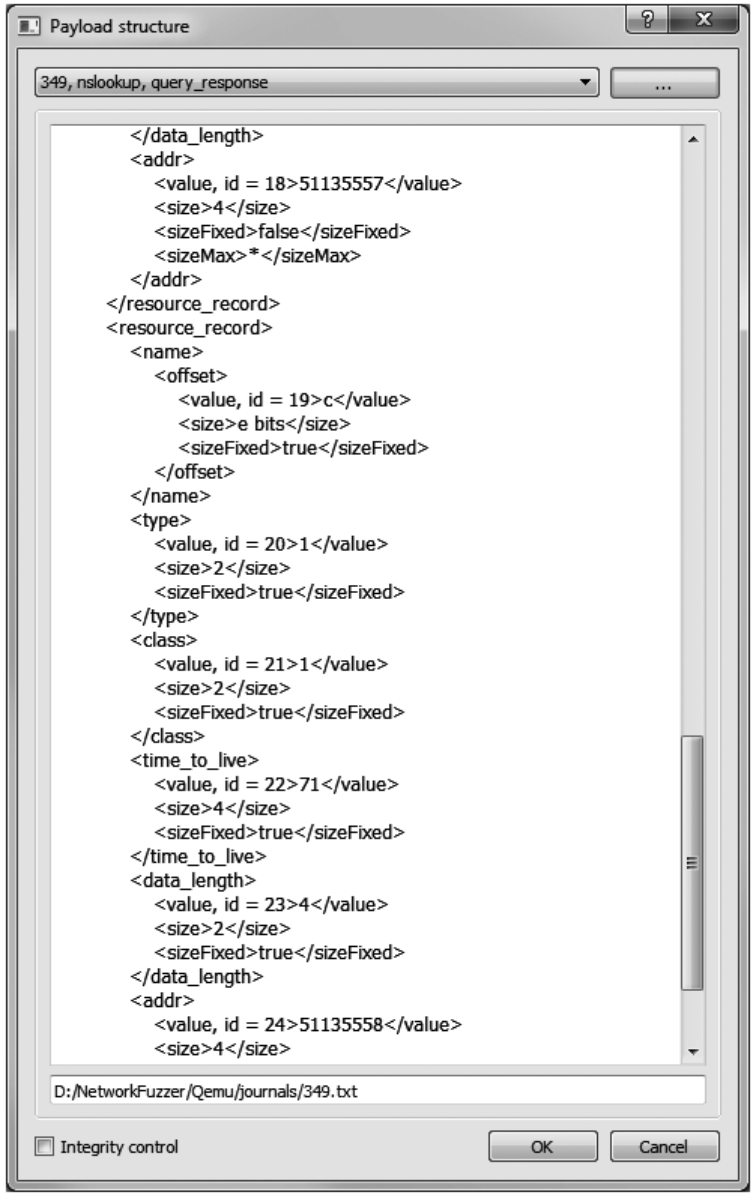


Рис. 3. Окно промежуточного представления.

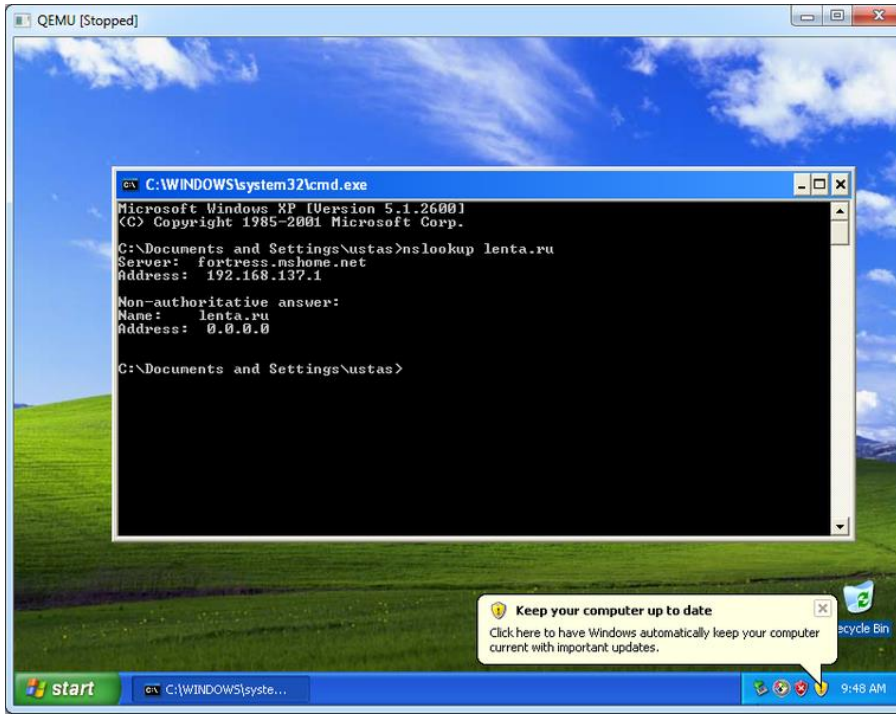


Рис. 4. Результат последней итерации фаззинга.

5. Заключение

В рамках работы был разработан метод, позволяющий переходить от воспроизведения программ к их выполнению. Благодаря этому методу расширяются возможности по внесению неисправностей, а также отладке приложений. Например, при фаззинге сетевых протоколов каждый сгенерированный сетевой пакет не требует нового сеанса взаимодействия с удаленным компьютером в отличие от обычных методов тестирования. Таким образом, разработанный метод способен сократить время, затрачиваемое на тестирование как клиентских, так и серверных приложений.

Предполагается дальнейшее развитие метода по ряду направлений. Во-первых, при отладке может оказаться полезной возможность построения дерева путей выполнения программы, по которому можно передвигаться вверх и вниз (назад и вперед во времени), переключаясь между ветвями и изучая состояние программы в зависимости от входных данных и взаимодействия с пользователем.

Второе направление заключается в разработке средств символьного выполнения. Само по себе символьное выполнение работает медленно, и это может повлиять на realtime-приложения. Использование детерминированного воспроизведения может позволить анализировать такие приложения. Похожий подход реализован в системе динамического анализа SAGE [11], которая работает на уровне отдельных приложений.

Еще одно направление исследований – это дополнение разработанного метода для проведения более полного фаззинга сетевых протоколов. Текущая реализация не позволяет продолжить сетевое взаимодействие после изменения определенного пакета. Решением этой проблемы могут стать синхронные запись и воспроизведение работы сразу нескольких виртуальных машин, например как в симуляторе Simics [16].

Аналогично фаззингу сетевых протоколов, можно использовать детерминированное воспроизведение и при фаззинге протоколов взаимодействия USB-устройств с виртуальной машиной. USB-устройство является внешним по отношению к виртуальной машине, поэтому при фаззинге протокола возникают те же сложности, что и при работе с удаленной системой по сети. Например, в работе [9] описаны подходы к фаззингу протоколов работы с USB-устройствами. Авторам удалось обеспечить устойчивое воспроизведение найденных сбоев лишь при подмене инициализирующего пакета, а в остальных случаях недетерминированность обмена с устройством не давала возможности воспроизводить сбои.

Список литературы

[1] Bieman J. M., Dreilinger D., Lin L. Using Fault Injection to Test Software Recovery Code // Final report, Colorado advanced software institute, 1995, 48 pages.

[2] Becker M., Baldin D., Kuznik C., Joy M. M., Xie T., Mueller W. XEMU: an efficient QEMU based binary mutation testing framework for embedded software. // Proceedings of the Tenth ACM International Conference on Embedded Software, ACM New York, NY, USA 2012, pp. 33-42.

[3] Han S., Shin K. G., Rosenberg H. A. DOCTOR: An Integrated Software Fault Injection Environment for Distributed Real-time Systems. // In Proc. 2nd Annual IEEE Int. Computer Performance and Dependability Symp. (IPDS'95). Erlangen, Germany, 1995. P. 204-213.

[4] Kanawati J., Abraham J. FERRARI: A Tool for the Validation of System Dependability Properties. // In Proc. 22nd IEEE Int. Symp. on Fault Tolerant Computing (FTCS-22). Boston, Massachusetts, USA, 1992. P. 336-344.

[5] Segall Z., Vrsalovic D., Siewiorek D., Yaskin D., Kownacki J., Barton R., Dancey A., Robinson T. FIAT – Fault Injection Based Automated Testing Environment. // In Proc. 18th IEEE Int. Symp. on Fault Tolerant Computing (FTCS-18). Tokio, Japan, 1988. P. 102-107.

[6] Li Y., Xu P., Wan H. A Fault Injection System Based on QEMU Simulator and Designed for BIT Software Testing // Applied Mechanics and Materials, vol. 347-350, 2013. pp. 580-587.

[7] Dawson S., Jahanian F., Mitton T. ORCHESTRA: a probing and fault injection environment for testing protocol implementations. In proceeding of: Computer Performance and Dependability Symposium, 1996, page 56.
[8] B.P. Miller, L. Fredriksen, and B. So, An Empirical Study of the Reliability of UNIX Utilities, Communications of the ACM 33, 12 (December 1990), pages 32-44
[9] Jodeit M., Johns M. USB Device Drivers: A Stepping Stone into Your Kernel. // Proceedings of the 2010 European Conference on Computer Network Defense, IEEE Computer Society, Washington, DC, USA, 2010. pp. 46-52
[10] Cha S. K., Avgerinos T., Rebert A., Brumley D. Unleashing mayhem on binary code. // SP '12 Proceedings of the 2012 IEEE Symposium on Security and Privacy. IEEE Computer Society Washington, DC, USA 2012, pp. 380-394.
[11] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. Queue 10, 1, Pages 20 (January 2012), 8 pages.
[12] The KLEE Symbolic Virtual Machine. http://klee.github.io/klee, 21.04.2014
[13] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: a platform for in-vivo multi-path analysis of software systems. SIGARCH Comput. Archit. News 39, 1 (March 2011), 265-278
[14] EPDB – a reversible debugger for Python. https://code.google.com/p/epdb, 21.04.2014
[15] Chow J., Lucchetti D., Garfinkel T., Lefebvre G., Gardner R., Mason J., Small S., Chen P. M. Multi-stage replay with Crosscut // VEE '10 Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments. – New York, NY, USA : ACM, 2010. – pp. 13-24.
[16] Jakob Engblom . Back to Reverse Execution. http://blogs.windriver.com/tools/2013/06/back-to-reverse-execution.html, 21.04.2014
[17] Engblom, J. A review of reverse debugging. System, Software, SoC and Silicon Debug Conference (S4D), 2012, pages 1-6.
[18] Bellard F. QEMU, a fast and portable dynamic translator. // In USENIX 2005 Annual Technical Conf. pages 41–46, Apr. 2005.
[19] Довгалюк П. Детерминированное воспроизведение процесса выполнения программ в виртуальной машине / Труды Института системного программирования РАН. Т. 21 / под ред. В.П. Иванникова. М.: ИСП РАН. 2011. С. 123-132.
[20] Bania P. Playing with RDTSC. http://www.piotrbania.com/all/articles/playing_with_rdtsc.txt, 21.04.2014

Using Deterministic Replay for Software Fault Injection

P. M. Dovgalyuk, Y. V. Markin
ISP RAS, Moscow, Russia
{Pavel.Dovgaluk, ustas}@ispras.ru

Abstract: This paper presents method of improving software fault injection by using deterministic replay. Fault injection and fuzzing are the methods of testing used for checking code coverage quality, improving error handling, and robustness testing. Fuzzing can hardly be applied for stateful communication protocols because of program state could change when restarting an application. The main idea of our method is to inject faults while replaying program deterministically. Deterministic replay requires program execution recording for latter replaying. Recorded log includes user input, incoming network packets, USB input, and hardware timers. During replay we read these events from the log and put them back into the simulator instead of reading inputs or receiving packets from the network. After injecting the fault in replay mode the program execution is different. It means that we should stop the replaying and start normal program execution from that program state. During the execution we simulate all hardware timers to make this mode switching imperceptible to the program. With the help of deterministic replay we can accelerate system initialization, eliminate non-deterministic data sources effect, and simplify environment setup, because the whole program execution before injecting fault is recorded. On the basis of the method the network fuzzer was built. The fuzzer modifies selected network packet saved during session recording and sends it back into the simulator. This phase is repeated from the same program state until the bug in the program was found.

Keywords: deterministic replay; virtual machine; debugging; fault injection; fuzzing; simulator; heisenbug; QEMU

References

[1] Bieman J. M., Dreilinger D., Lin L. Using Fault Injection to Test Software Recovery Code. Final report, Colorado advanced software institute, 1995, 48 pages.
[2] Becker M., Baldin D., Kuznik C., Joy M. M., Xie T., Mueller W. XEMU: an efficient QEMU based binary mutation testing framework for embedded software. Proceedings of the Tenth ACM International Conference on Embedded Software, ACM New York, NY, USA 2012, pp. 33-42. doi: 10.1145/2380356.2380368
[3] Han S., Shin K. G., Rosenberg H. A. DOCTOR: An Integrated Software Fault Injection Environment for Distributed Real-time Systems. In Proc. 2nd Annual IEEE Int. Computer Performance and Dependability Symp. (IPDS'95). Erlangen, Germany, 1995. P. 204-213. doi: 10.1109/IPDS.1995.395831
[4] Kanawati J., Abraham J. FERRARI: A Tool for the Validation of System Dependability Properties. In Proc. 22nd IEEE Int. Symp. on Fault Tolerant Computing (FTCS-22). Boston, Massachusetts, USA, 1992. P. 336-344. doi: 10.1109/FTCS.1992.243567
[5] Segall Z., Vrsalovic D., Siewiorek D., Yaskin D., Kownacki J., Barton R., Dancey A., Robinson T. FIAT – Fault Injection Based Automated Testing Environment. In Proc. 18th IEEE Int. Symp. on Fault Tolerant Computing (FTCS-18). Tokio, Japan, 1988. P. 102-107. doi: 10.1109/FTCS.1988.5306

[6] Li Y., Xu P., Wan H. A Fault Injection System Based on QEMU Simulator and Designed for BIT Software Testing. Applied Mechanics and Materials, vol. 347-350, 2013. pp. 580-587.
[7] Dawson S., Jahanian F., Mitton T. ORCHESTRA: a probing and fault injection environment for testing protocol implementations. In proceeding of: Computer Performance and Dependability Symposium, 1996, page 56. doi: 10.1109/IPDS.1996.540200
[8] B.P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. Communications of the ACM 33, 12 (December 1990), pages 32-44. doi: 10.1145/96267.96279
[9] Jodeit M., Johns M. USB Device Drivers: A Stepping Stone into Your Kernel. Proceedings of the 2010 European Conference on Computer Network Defense, IEEE Computer Society, Washington, DC, USA, 2010. pp. 46-52. doi: 10.1109/EC2ND.2010.16
[10] Cha S. K., Avgerinos T., Rebert A., Brumley D. Unleashing mayhem on binary code. SP '12 Proceedings of the 2012 IEEE Symposium on Security and Privacy. IEEE Computer Society Washington, DC, USA 2012, pp. 380-394. doi: 10.1109/SP.2012.31
[11] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. Queue 10, 1, Pages 20 (January 2012), 8 pages. doi: 10.1145/2090147.2094081
[12] The KLEE Symbolic Virtual Machine. http://klee.github.io/klee
[13] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: a platform for in-vivo multi-path analysis of software systems. SIGARCH Comput. Archit. News 39, 1 (March 2011), 265-278. doi: 10.1145/1961295.1950396
[14] EPDB – a reversible debugger for Python. https://code.google.com/p/epdb
[15] Chow J., Lucchetti D., Garfinkel T., Lefebvre G., Gardner R., Mason J., Small S., Chen P. M. Multi-stage replay with Crosscut. VEE '10 Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments. – New York, NY, USA : ACM, 2010. – pp. 13-24. doi: 10.1145/1735997.1736002
[16] Jakob Engblom . Back to Reverse Execution. http://blogs.windriver.com/tools/2013/06/back-to-reverse-execution.html
[17] Engblom, J. A review of reverse debugging. System, Software, SoC and Silicon Debug Conference (S4D), 2012, pages 1-6.
[18] Bellard F. QEMU, a fast and portable dynamic translator. In USENIX 2005 Annual Technical Conf. pages 41–46, Apr. 2005.
[19] Dovgalyuk P. Determinirovannoe vosproizvedenie protsessa vypolneniya programm v virtual'noj mashine [Deterministic replay of software in virtual machine]. Trudy Instituta sistemnogo programmirovaniya RAN [The Proceedings of ISP RAS], T. 21, pod red. V.P. Ivannikova. M.: ISP RAN. 2011. S. 123-132. (in Russian)
[20] Bania P. Playing with RDTSC. http://www.piotrbania.com/all/articles/playing_with_rdtsc.txt

Применение информационных технологий (генетические алгоритмы, нейронные сети, параллельные вычисления) в анализе безопасности АЭС.

Ю.Б. Воробьев, П. Кудинов**, М. Ельцов**, К. Кёоп**, К.Н. Чыонг Ван*, yura@npp.mpei.ac.ru, pkudinov@kth.se, marti@safety.sci.kth.se, kaspar@safety.sci.kth.se, nhattvk@gmail.com, * НИУ «МЭИ», Красноказарменная 14, Москва, Россия, ** Королевский технологический институт, Валхалавеген 79, Стокгольм, Швеция*

Аннотация. В статье рассматриваются три направления использования информационных технологий в анализе безопасности атомных электростанций (АЭС). Это: динамический вероятностный анализ безопасности АЭС, возможности использования кодов расчетной гидродинамики (CFD) для моделирования сложных процессов ядерных энергетических установок и проблема идентификации аварии на АЭС. Используемые технологии повышают эффективность как обработки информации – генетические алгоритмы (ГА), нейронные сети (НС), так и расчетов – параллельные вычисления. Применение новых технологий основывается на их гармоничном сочетании с существующими классическими методами анализа безопасности АЭС.

Ключевые слова: информационные технологии, анализ безопасности, мета-модель

1. Введение

События последних лет показывают, что, несмотря на всю глубину проработки и сложность используемых методов в анализе безопасности АЭС, явления «невероятные» с точки зрения используемых стандартных методов продолжают случаться (достаточно вспомнить аварию на АЭС Фукусима), обнаруживая уязвимости в сложной системе АЭС. Использование традиционных методов основанных на априорных предположениях о сложных феноменах и зависимостях может приводить к ложному консерватизму. Решение данной проблемы может основываться на применении новых методов, дающих новое качество обработки имеющейся информации. При этом существенным фактором является их гармоничное использование

совместно с традиционными подходами анализа безопасности, что обеспечивает преемственность и апробированность результатов.

В статье рассматриваются три направления применения информационных технологий (ИТ), по которым авторы наработали существенный положительный опыт.

2. Метод динамического вероятностного анализа безопасности АЭС с использованием информационных технологий

2.1. Проблемы стандартного ВАБ

Вероятностный анализ безопасности (ВАБ) является важным инструментом для рассмотрения потенциально опасных инженерных систем в ядерной энергетике, авиакосмическая индустрии и т.д. ВАБ базируется на множестве предположений по поводу возможных аварийных сценариев, предполагаемых консервативными, «декомпозиции» сложных проблем на множество предопределенных последовательностей и т.д. Также существуют проблемы самого ВАБ:

- учет взаимодействия между детерминистическими и вероятностными величинами
- Булева алгебра ограничивает возможности учета порядка возникновения событий
- использование Марковских моделей

Все это может приводить к чрезмерному или ложному консерватизму, к недооценке или пропуску потенциально опасных аварийных сценариев.

С конца 80-х детерминистические динамические модели, называемые кодами наилучшей оценки, получили признание как инструменты анализа безопасности АЭС. Однако они используются до сих пор достаточно оторвано от ВАБ, что затрудняет их применение в анализе риска и идентификации опасных состояний АЭС.

Динамический ВАБ (ДВАБ) является подходом, нацеленный на улучшение качества анализа через связь детерминистической и вероятностной ветвей анализа безопасности и является дополнением к стандартному ВАБ. ДВАБ методы могут быть сгруппированы в 4-е класса:

- Марковские модели [1];
- графические модели [2];
- непосредственное моделирование системы на основе метода Монте Карло (МК) [3];
- дискретные динамические деревья событий (ДДДС) [4].

Наиболее гибкими для применения в анализе безопасности АЭС являются МК и ДДДС методы. Основная проблема в применении ДДДС методов

заключается в комбинаторном взрыве при уточнении дискретности. В МК методах, комбинаторный взрыв избегается, но они требуют огромного количества вариантов расчетов детерминистического кода для уверенной идентификации маловероятных (10^{-6} - 10^{-8}), но потенциально опасных событий.

В статье рассматривается подход ДВАБ метод на основе применения ИТ - генетического алгоритма (ГА-ДВАБ) позволяющий проводить исследование пространства неопределенностей и сценариев АЭС наиболее эффективным образом с вычислительной точки зрения.

2.2. Подход ГА-ДВАБ

Нарушение нормальной работы АЭС, ее уязвимость заключается в сценариях, содержащих начальные события, отказы компонент, действия оператора, функционирование систем контроля и безопасности, приводящих к отказу барьеров безопасности (например, оболочки тепловыделяющих элементов (ТВЭЛ) и т.п). Уязвимости могут быть идентифицированы в процессе исследования пространства неопределенностей АЭС (неопределенности, носящие чисто случайный характер и неточности моделирования). Критические для безопасности параметры (например, максимальная температура оболочки ТВЭЛов и т.п.) могут быть использованы как функция, управляющая процессом исследования уязвимости АЭС. АЭС является сложной, нелинейной системой и, следовательно, упомянутая функция будет нелинейная с локальными экстремумами.

Существуют две типичные задачи идентификации ДВАБ анализа:

1. наихудшего сценария с наихудшими последствиями;
2. области отказа или подобластей в пространстве возможных сценариев АЭС, где превышается барьер, связанный с некоторыми пределами безопасности.

В данной работе усилия концентрируются на идентификации потенциальных уязвимостей АЭС. Для этого применяется адаптивная стратегия на основе алгоритмов семейства поиска глобального оптимума, позволяющая проводить более детальное исследование параметрического пространства в потенциально опасных областях (областях отказа). В методе используется генетический алгоритм (ГА)[5]. ГА является эвристическим методом, использующий технику и терминологию, заимствованную из биологии, для нахождения глобального оптимума функции $\boldsymbol{Y=F(U)}$. $\boldsymbol{U}$ – вектор параметров и представляет неопределенности, связанные с моделированием и функционированием АЭС (например, времена отказа или срабатывания оборудования, задержки активирования систем безопасности, коэффициенты теплообмена и т.п.). ГА не имеет ограничений на тип функции приспособленности $\boldsymbol{F(U)}$ и на независимые параметры $\boldsymbol{U}$ (непрерывные и дискретные могут быть использованы), что делает простым его использование совместно с детерминистическими системными кодами. В ГА относительно

легко реализуются параллельные расчеты, что серьезно снижает расчетное время для детального ДВАБ анализа.

Шаги использования ГА-ДВАБ следующие. Вначале определяется пространство поиска на основе вектора $\boldsymbol{U}$. Функция $\boldsymbol{F}$ базируется на критических значениях системных параметров. Далее пространства событий АЭС и его параметры неопределенностей $\boldsymbol{U}$ отображаются на вектор параметров ГА - $\boldsymbol{X}$ (хромосомы по терминологии ГА). Третий шаг концентрируется на исследовании пространства неопределенностей для нахождения условий, при которых нарушаются требования безопасности. ГА исследует пространство неопределенностей для нахождения множества значений $\boldsymbol{X}$, при которых имеется определенная степень нарушения барьеров безопасности, описанных функцией $\boldsymbol{Y=F(U)}$. Степень нарушения $\boldsymbol{Y_{TAR}}$, может быть определена, например, на основании требований контролирующей организации. Области в пространстве значений входных параметров $\boldsymbol{U}$, в которых $\boldsymbol{Y > Y_{TAR}}$, называются областями отказа. Для идентификации наиболее худшего сценария исходная $\boldsymbol{Y}$ может быть использована непосредственно как функция приспособленности, но в некоторых случаях $\boldsymbol{Y}$ должна быть преобразована. После нахождения наихудшей аварии или идентификации областей отказа могут быть оценены их вероятностные характеристики.

ГА-ДВАБ поход реализован на базе NPO (Nuclear Plant Optimizer) кода [6] разработанного для автоматизации анализа безопасности АЭС с использованием параллельных расчетов. Для ГА-ДВАБ адаптированы следующие основные системные коды: Relap5/Parcs, MELCOR 1.8x, 2.x [7]. Важным аспектом, повышающим эффективность ДВАБ расчетов, является возможность их реализации в системе параллельных вычислений. Это может быть реализовано двояко. Первый вариант базируется на распараллеливании расчетов самих детерминистических кодов, что обычно реализуется разработчиками кодов. В настоящее время этот процесс находиться на начальной стадии и большинство системных кодов выполняются в последовательном режиме. Поэтому первый вариант не используется в ГА-ДВАБ. Во втором случае распараллеливанию подлежат непосредственно сами варианты расчетов, распределяемые на узлы кластера, что оказывается очень эффективным т.к. время коммуникации между узлами пренебрежимо мало по сравнению с расчетным временем каждого варианта. В NPO используется система распараллеливания PVM в стандартной концепции slave – master. Использование достаточно старой PVM обусловлено двумя причинами. В первую очередь историческими – в момент создания NPO у авторов отсутствовала возможность использования суперкомпьютеров. Во вторых, в силу особенностей расчетов переход на более современную систему MPI не дает какого либо преимущества по расчетам, но дает возможность использовать суперкомпьютеры, что является мотивацией для дальнейших работ.

2.3. Пример использования ГА-ДВАБ подхода

Задача поиска наиболее опасной ситуации рассматривается для реакторной установки (РУ) ВВЭР-1000/В320. Для детерминистического анализа использовался код Relap5 mod 3.3. Возможные аварии основывались на вариации: размера и места течи из первого контура РУ, комбинаций отказов систем безопасности и временных запаздываний их активации, параметров моделей кода Relap5, действий оператора АЭС. Размерность параметрического пространства равна 58. Вероятностные характеристики параметров выбирались по результатам анализа неопределенностей (АН) [8]. Функция приспосабливаемости ГА базировалась на максимальном значении температуры оболочки ТВЭЛов ($\mathbf{T_{об}}$) в активной зоне реактора.

В ГА-ДВАБ задаче было рассчитано 943 варианта, в 7-и из которых $\mathbf{T_{об}}$ превышает 1300К°. Для расчетов использовался кластер из 72 CPU. Расчетное время составило 54,1 часов. Наихудшие аварийные сценарии представлены на рис. 1. При увеличении количества расчетных вариантов в ГА-ДВАБ задаче до 1700 удалось найти более опасные варианты аварий с $\mathbf{T_{об}}$ близкой к 1700К°.

Анализ найденных аварийных ситуаций определил, что причиной повышения $\mathbf{T_{об}}$ является сложное взаимодействие между различными компонентами системы и действиями оператора АЭС по активации системы безопасности (JDH to PRZ на рис 1) [9]. Таким образом, решение ГА-ДВАБ задачи позволило найти неочевидный и парадоксальный аварийный случай, когда правильные действия оператора приводят сложную систему АЭС к опасным последствиям, что не определяется стандартными средствами ВАБ.

На основе проведенных расчетов были выделены следующие проблемы:

- Большая размерность параметрического пространства приводит к так называемому «проклятию размерности» [10] при проведении вероятностных расчетов
- Для сложных задач возможен немонотонный характер изменения функции приспособленности ГА на пространстве неопределенностей
- Сложный тип границ области отказа, возможность несвязанных областей отказа
- Сильное изменение чувствительности функции приспособленности для различных варьируемых параметров
- Существенная корреляция между входными параметрами модели

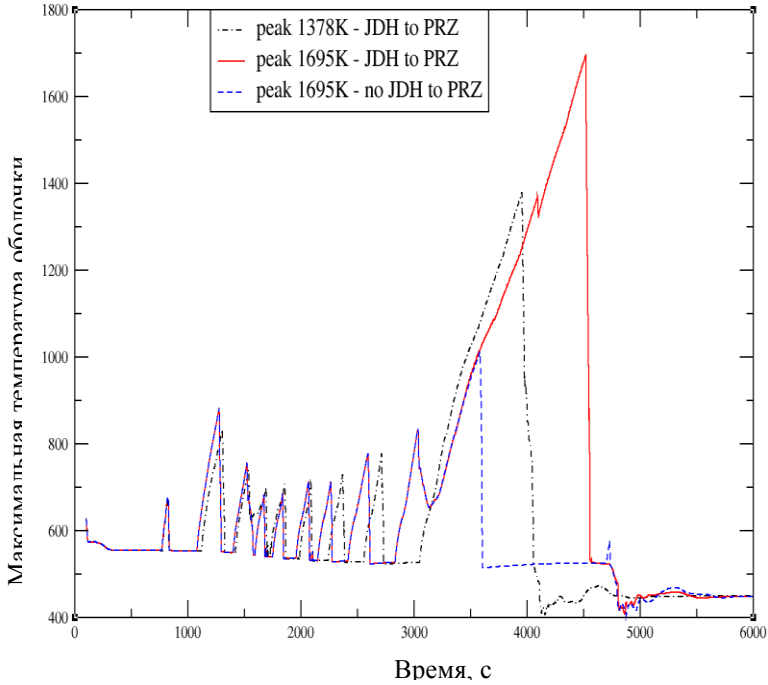


Рис 1 Изменение $T_{об}$ *в зависимости от времени аварии для различных вариантов максимальной температуры: 1378К и 1695К для активации системы безопасности в компенсатор давления (JDH to PRZ); для аварии с 1695К, но без активации системы безопасности (no JDH to PRZ).*

- Необходимость оценок областей малой вероятности – 10-5÷10-6

Для их решения необходимо:

- модификация стандартного алгоритма ГА
- использование нескольких методов вероятностных оценок для последующей кросс-верификации результатов.

2.4. Повышение эффективности ГА процедур для ГА-ДВАБ

Данное направление базируется на следующем. Во первых, необходима модификация стандартных параметров ГА. Например, необходимо увеличивать вероятность мутации до диапазона 0,4 – 0,5 в противовес часто

используемым значениям 0,01. Это дает лучшее исследование параметрического пространства и уменьшает корреляцию между точками. Далее, были разработаны алгоритмы, модифицирующие стандартный ГА, для увеличения эффективности выделения областей отказа, идентификации их границ. Например, вводиться ограничение на функцию приспособленности на основе параметра $\boldsymbol{C}$, который определяется в процентах от $\boldsymbol{Y_{TAR}}$. Тогда функция приспособленности $\boldsymbol{F_i}$ члена популяции $\boldsymbol{i}$ модифицируется (для верхней границы):

$$F_i' = \begin{cases} F_i \ ; & F_i < Y_{TAR} + Y_{TAR} \cdot C \\ Y_{TAR} + Y_{TAR} \cdot C; & F_i \geq Y_{TAR} + Y_{TAR} \cdot C \end{cases} , \qquad (1)$$

что делает все точки внутри области отказа одинаково привлекательными для процесса поиска.

Как пример можно рассмотреть оптимизацию механизма мутации для генов непрерывного (вещественного) типа. В применяемом ГА мутация определяет новое значения гена $\boldsymbol{x_{in}}$ по выражению: $\boldsymbol{x_{in}}=\boldsymbol{x_{io}}+ \boldsymbol{g_m g_s \Delta x_i}$. Здесь $\boldsymbol{x_{io}}$ старое значение, $\boldsymbol{g_m}$ случайное число, $\boldsymbol{\Delta x_i}$ – максимально возможный разброс значения для гена, $\boldsymbol{g_s}$ – параметр, определяющий разброс значений при мутации. Для решения стандартных оптимизационных задач $\boldsymbol{g_s}$=0,08 обеспечивает наибольшую эффективность ГА. При изучении влияния параметра $\boldsymbol{g_s}$ задача ГА-ДВАБ была создана на основе функции, имеющей множество локальных минимумов:

$$Y = F(x_1,\ldots,x_N) = 10N + \sum_{n=1}^{N}[x_n^2 - 10\cos(2\pi x_n)] \qquad (2)$$

Рассматривалось влияние параметра $\boldsymbol{g_s}$ на результат вероятностных оценок с $\boldsymbol{N}$=50. На рис 2 представлены результаты для параметров: **CoV** - коэффициента вариации, $\boldsymbol{V_r}$ - степень покрытия пространства неопределенностей в процессе поиска ГА, количества расчетов (2), найденный минимум (2). Рис 2 (а), показывает, что параметр **CoV** уменьшается, а число расчетов (2) увеличивается в зависимости от $\boldsymbol{g_s}$. На рис 2 (б) лучшее покрытие параметрического пространства $\boldsymbol{V_r}$ достигается с увеличением $\boldsymbol{g_s}$, в то время как нахождение глобального минимума **Ошибка! Источник ссылки не найден.** становиться хуже. Баланс между разными параметрами достигается для $\boldsymbol{g_s}$ в диапазоне 0,25 – 0,3 , использование которого в вышерассмотренном варианте ВВЭР-1000 подтвердило лучшее исследование области отказа по результирующих вероятностным оценкам.

2.5. Методы оценки результирующей вероятности для ГА-ДВАБ

В задачах определения областей отказа необходимо оценивать вероятность попадания в них. Для ГА-ДВАБ был разработан метод вероятностных оценок на основе определения статистических характеристик полученных ГА

выборок, который эффективно работает для относительно малых размерностей параметрического пространства. Для больших размерностей, желательно иметь несколько вариантов оценки вероятности для их кросс-верификации. В настоящее время разработано два метода и исследуются другие.

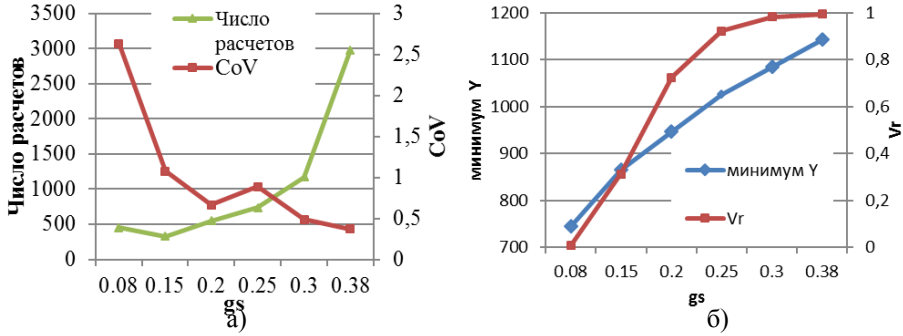


Рис 2 Зависимости: ***CoV*** *и число вызовов (2) (а);* $\mathbf{V_r}$ *и найденный минимум* ***Y*** *(б) от* $\mathbf{g_s}$ *.*

2.5.1 Метод вероятностных оценок на основе метода существенной выборки

В методе используется тот факт, что ГА принадлежит к классу стохастических алгоритмов, т.к. имеются вероятностные процедуры в операциях скрещивания и мутации. Поэтому по варьируемым параметрам $\boldsymbol{U}$ и результирующим значениям $\boldsymbol{Y=F(U)}$ имеется статистические данные. На их основе необходимо оценить вероятность $\boldsymbol{P(Y>Y_{TAR})}$ или $\boldsymbol{P(Y<Y_{TAR})}$. В данной работе используется подход схожий с методом существенной выборки метода МК. Исходная функция плотности распределения $\boldsymbol{U\rightarrow f(U)}$ заменяется смещенной функцией $\boldsymbol{h(U)}$ и результирующая вероятность можно оценить как:

$$P'(Y > Y_{TAR}) = \frac{1}{N_{GA}} \sum_{j=1}^{N_{GA}} I(Y > Y_{TAR}) \frac{f(U_j)}{h(U_j)}; \;\; I(Y > Y_{TAR}) = \begin{cases} 1, Y > Y_{TAR} \\ 0, Y \leq Y_{TAR} \end{cases} \qquad (2)$$

Оценка $\boldsymbol{h(U)\rightarrow h'(U)}$ определяется на основе статистических данных ГА. В настоящее время реализованы и развиваются два подхода. В первом для $\boldsymbol{h'(U)}$ используется приближение на основе ступенчатой функции построенной по типу гистограммы. Во втором применяется метод оценок плотности распределения на основе ядерных функций (KDE) [10].

2.5.2 *Метод вероятностных оценок на основе нейронных сетей*

В методе нейронные сети (НС) используются для аппроксимации поведения сложной, нелинейной функции. Трехслойная НС прямого распространения применяется в текущей работе. Применение НС для мета-моделирования включает следующие шаги. Сначала детерминистический код используется для генерации обучающего множества $\boldsymbol{A_t} = \{\boldsymbol{U_t}, \boldsymbol{Y_t}\}$ на основе $\boldsymbol{Y=F(U)}$. Затем параметры НС настраиваются в процессе обучения НС для аппроксимации отображения $\boldsymbol{F}$. Обычно метод МК используется для создания $\boldsymbol{A_t}$. В данном подходе ГА-ДВАБ используется для генерации $\boldsymbol{A_t}$. Для обучения НС может быть применен стандартный алгоритм обратного распространения ошибки на основе минимизации среднеквадратической ошибки - $\boldsymbol{E}$. Важно заметить, что НС обучается на основе ГА-ДВАБ данных, смещенных в сторону области отказа. Поэтому НС будет всегда отображать поведение $\boldsymbol{U}$, $\boldsymbol{Y}$ в области отказа лучше, чем в других. Вычислительные ресурсы, требуемые для НС, являются малыми. Поэтому НС мы сможем использовать как мета-модель вместе с МК для определения вероятностных характеристик областей отказа.

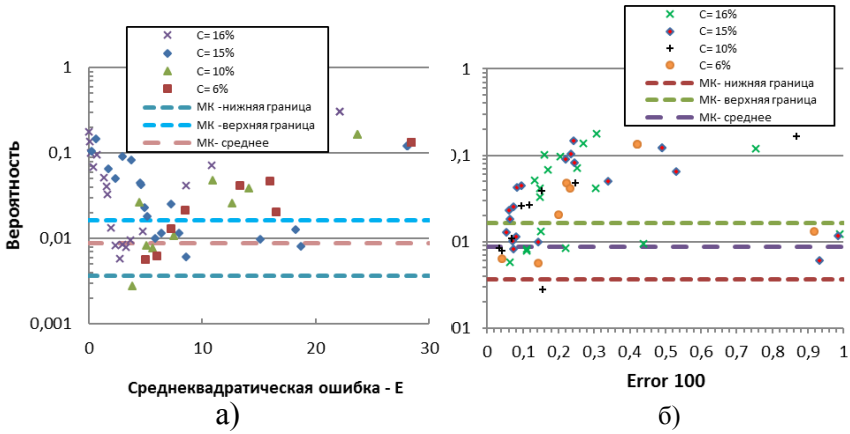


Рис 3 Вероятностные оценки на основе НС как функция: среднеквадратической ошибки (а); Error100 (б) для различных значений параметра С из (1). Доверительный интервал для МК метода показан для ВВЭР-1000 варианта на основе 1773 расчетов.

Вероятность попадания в область отказа оцененная с помощью НС как функция среднеквадратической ошибки процесса обучения для различных

значений параметра алгоритма выделения областей отказа представлена на рис.3(а). Для каждого множества обучения для различных данных идентификации области отказа имеется оптимальная стадия обучения НС, которая соответствует $\boldsymbol{E}$ в диапазоне 3 – 8. Перетренировка НС (малые $\boldsymbol{E}$) приводит к ухудшению вероятностных оценок по сравнению с референтными значениями метода МК.

Для решения проблемы был выработан подход контроля процесса обучения на основе вероятностных характеристик НС мета-модели. Используется тот факт, что данные сгенерированные для начальной популяции ГА являются чисто вероятностными и могут быть использованы для референтных вероятностных оценок (MC_{GA}). Был введен параметр:

$$\text{Error100} = \sum_{i=1}^{9}\left(\frac{100\%-i\cdot 10\%}{100\%} - P(Y_N > Y_{MC_{GA}i})\right)^2 \qquad (3)$$

$Y_{MC_{GA}i}$ определяется так, что $P\left(Y_{MC_{GA}} > Y_{MC_{GA}i}\right) = i \cdot 10\%,\ i = 1,2,..,9$. Если Error100 мало, то это означает, что вероятностные оценки на основе обученной НС близки к референтным данным метода МК. На рис 3(б) показано, что если Error100 уменьшается, то имеется лучшее согласование с данными полученными по МК. Таким образом, Error100 может быть использован как дополнительный критерий для контроля процесса обучения НС.

3. Использование мета-моделей при применении CFD кодов в анализе безопасности АЭС

3.1. Проблемы использования CFD кодов в анализе безопасности АЭС

В настоящее время анализ безопасности АЭС базируется на применении одномерных системных кодов типа Relap5, являющимися адекватными при моделировании интегральных характеристик АЭС для анализа переходных и аварийных процессов. В тоже время при необходимости учета трехмерных характеристик оборудования, получения соответствующих расчетных данных успешно используются CFD коды, как правило, на более локальном уровне, чем системные коды. С точки зрения точности желательным является использование CFD кодов для всей АЭС, однако в настоящее время для этого существует ряд ограничений: вычислительных ресурсов, способности CFD кодов полномасштабно моделировать динамику многофазных процессов. Кроме того, это часто и не нужно т.к. многие теплогидравлические процессы на АЭС существенно одномерны. Оптимальным является совместное моделирование на основе одномерного системного кода и CFD кода - (CFD-1D система). При конкретной реализации в модели выделяется область, где

значимы трехмерные эффекты – CFD домен и те места, где одномерное приближение является адекватным – домен системного кода. Связь между доменами осуществляется на основе методов декомпозиции или наложения доменов.

При декомпозиции доменов расчетная область делится между CFD и системным кодом, а связь осуществляется на границах. В мире имеется положительный опыт по реализации данной связи – например [11]. Основная проблема заключается в нестабильности расчетов связанной с трудностью достижения сходимости по параметрам на интерфейсах между доменами.

Другое направление, основанное на наложении расчетных доменов [12], применяет схему, когда домен одномерного системного кода охватывает всю моделируемую систему, а домен CFD кода ту часть, где важны трехмерные эффекты. Системный код предоставляет граничные условия для CFD кода, а он в свою очередь замыкающие величины для системного. Это могут быть, например, перепад давления, коэффициент теплообмена и т.п. В этом случае расчетная схема оказывается численно значительно более стабильной по сравнению с вариантом декомпозиции.

3.2. Использование мета-моделирования

По обоим направлениям имеется положительный опыт реализации, однако если необходимо использовать систему CFD-1D не для одного, но множества (от сотни до тысяч) расчетов, например, в рамках АН, ДВАБ расчетов и т.п. данная схема оказывается существенно более медленная по сравнению с расчетами на основе одномерного системного кода. Одно из решений проблемы базируется на использовании мета-моделей. Алгоритм заключается в следующем:

- проведение $\boldsymbol{N_{\text{пер}}}$ первоначальных расчетов на основе CFD или CFD-1D системы – данные настройки $\boldsymbol{Y_{\text{пер}}} = \mathbf{F}(\boldsymbol{U_{\text{пер}}}) \rightarrow \{\boldsymbol{Y_{\text{пер}}}, \boldsymbol{U_{\text{пер}}}\}$
- подбор параметров метамодели на основе $\{\boldsymbol{Y_{\text{пер}}}, \boldsymbol{U_{\text{пер}}}\}$
- замена CFD-1D системы на совместный расчет метамодели с одномерным системным кодом (метамодель-1D система) и выполнение расчетов для исследования пространства состояний моделируемой системы (АЭС)

Здесь имеются следующие проблемы. Необходимо определиться с выбором $\boldsymbol{N_{\text{пер}}}$. Для этого можно использовать формулу Уилкса (см. ниже (3)), дающую необходимое число статистических испытаний для получения доверительного интервала для $\boldsymbol{Y}$ с определенной уверенностью. Также для этого можно использовать априорные знания о типе функции $\boldsymbol{F}$.

Другой вопрос касается выбора типа мета-модели. Возможны следующие варианты:

1. Использование поверхности отклика на основе регрессионной модели. Часто применяется подход на основе поверхностей 2-го и 3-го порядка правомерных в случае $\boldsymbol{F}$ простого типа.
2. Модели на основе использования априорных физических знаний.
3. Нейронные сети. Данное направление также можно рассматривать как подход черного ящика, т.к. какие либо предположения о $\boldsymbol{F}$ не используются. С помощью НС возможно отобразить нелинейность $\boldsymbol{F}$ любого типа, но существует трудность физической интерпретации НС.

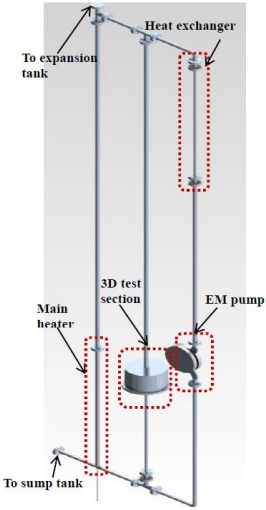
При использовании направления (2) часто используется подход по построению суррогатной модели основанный на огрублении исходной модели, на основе имеющихся априорных физических знаний проводится переход от трехмерной модели к двух или одно мерной, используются асимптотические приближения и т.д.: $\boldsymbol{Y}=\boldsymbol{F}(\boldsymbol{U}) \rightarrow \boldsymbol{Y}=\boldsymbol{F}_{sim}(\boldsymbol{U},\boldsymbol{A})$. Параметры $\boldsymbol{A}$ находятся на основе подгонки под имеющиеся экспериментальные или расчетные данные (см. например [13]). Как правило, $\boldsymbol{A}$ не является функцией от $\boldsymbol{U}$: $\boldsymbol{A}\neq\boldsymbol{f}(\boldsymbol{U})$. В альтернативном подходе к направлению (2) можно совместить гибкость мета-моделирования на основе НС и имеющиеся априорные знания. В этом случае $\boldsymbol{A}=\boldsymbol{f}(\boldsymbol{U})$ и $\boldsymbol{f}$ ищется на основе НС. Достоинство подхода заключается в возможности более гибкой и точной аппроксимации. Также в $\boldsymbol{F}_{sim}$ можно использовать более общие априорные соображения. Трудность же лежит в более сложном процессе обучения НС. Работы по данному направлению находятся в процессе развития.

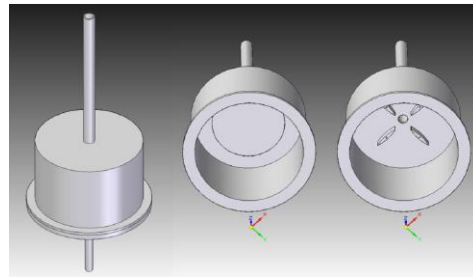
В текущей работе представлены результаты разработки мета-моделирования по направлению (3) на основе CFD-1D модели для установки TALL-3D Королевского технологического института (Стокгольм) [12] призванной моделировать процессы в ядерных реакторах с жидко-металлическим теплоносителем. Она представляет собой петлю с тремя нитками и одной 3D секцией бассейнового типа – рис. 4. В контуре теплогидравлические процессы адекватно описываются одномерным приближением. В 3D секции из-за формирования температурной стратификации необходимо использовать 3D моделирование. Температура на выходе 3D секции влияет на расход в контуре и его распределения по ниткам, что важно для моделирования процесса естественной циркуляции и его срыва.

3.3. Разработка метамодели для установки TALL-3D

Для моделирования TALL-3D была выбрана схема с наложением расчетных доменов [12] . Одномерная часть модели выполнена на коде Relap5 – LBE (ENEA). CFD код Star-CCM+ использовался для 3D секции, где применялась 2D полихедральная сетка, содержащая 91011 объемов. Связь между кодами осуществлялась на основе Java интерфейса кода Star-CCM+. Проведенные расчеты на основе созданной CFD-1D системы показали, что имеется значимое отличие от моделирования только одномерным кодом Relap5 в

области переходных режимах, где точность моделирования температурной стратификации оказывает существенное влияние.





а) б)

Рис. 4 а) Схема TALL-3D экспериментальной установки; б) Увеличенный вид 3D секции.

Задача текущей работы заключалась в исследовании возможности применения мета-моделирования по направлению (3) для CFD-1D системы. Для этого были выбраны следующие 4 режима работы установки TALL-3D, являющиеся наиболее представительными:

- **Режим 1** – переход от вынужденной к естественной циркуляции; обогрев 3D секции - постоянно
- **Режим 2** - переход от вынужденной к естественной циркуляции; обогрев 3D секции с 0 сек
- **Режим 3** – естественная циркуляция; обогрев 3D секции с 0 сек
- **Режим 4** – переход от естественной циркуляции к вынужденной; обогрев 3D секции с 0 сек

Между моделью 3D секции и Relap5 передавались следующие моделируемые величины:

- Расход на входе 3D секции - независимая переменная
- температуры входа и выхода 3D секции - (зависимая/независимая переменная в соответствии со знаком расхода)

НС использовалась той же архитектуры, что и ранее. Обучение осуществлялось на основе минимизации среднеквадратичной ошибки.

Учитывалась возможность влияния предистории Δt независимых переменных на основе формирования входных нейронов для некой временной точки t_i на основе данных из интервала $[t_i , t_i - \Delta t]$, что является физически обоснованным динамикой процессов температурной стратификации в 3D секции. Особенностью данной системы является изменение знака расхода в рассматриваемых режимах. Это меняет зависимую переменную выходная/входная температура, что в определенном смысле конфликтует с попыткой учесть предисторию и существенно затрудняет обучение НС. Решение было найдено на основе выделения в динамическом процессе переходного окна по параметру Δt и временной точки смены знака расхода. Т.к. динамика процессов разная в рамках переходного окна, когда имеется внутреннее возмущение в 3D секции, и другими областями, то логично выделить две независимые НС для их моделирования. Это решило проблему с настройкой НС. В результате НС имеет следующие входные узлы: предистория Δt изменения независимых параметров, мощность, подводимая к 3D секции, интегральное значение расхода с последней смены знака, идентификация – положительный/отрицательный расход. Использовались следующие данные: 8508 точек для обучения, 1193 для проверки, переходное окно – 222 точки. Узлы НС: 59 входных – 41 скрытых – 1 выходной. Точность моделирования НС приведена в табл.1, которая для текущих целей признана достаточной, хотя может быть еще далее улучшена.

Тип данных	**Среднеквадратичное отклонение, σ**	**Максимальное отклонение, °C**
Обучение	0,547977	7,22
Переходное окно	0,200846	1,02
Проверки	0,57545	10,49

Табл. 1 Точность моделирования настроенной НС

На основе настроенной НС моделировалась работа системы метамодель-1D. Результаты моделирования **Режима 2** с помощью НС, а также на основе кода Relap5, CFD-1D системы и результаты настойки НС представлены на рис. 5. Видно, что имеется значимое различие между одномерным кодом Relap5 и CFD-1D системой. Также заметно, что мета-модель на основе НС обеспечивает хорошую воспроизводимость CFD-1D системы. Схожие результаты были получены по остальным режимам. В результате проведенных работ был создан программный скелет NeuroMeta системы мета-моделирования для вышерассмотренного направления 3 и полученный опыт показывает, что требуемое качество отображения достижимо, но нужна адаптация структуры НС под текущую задачу.

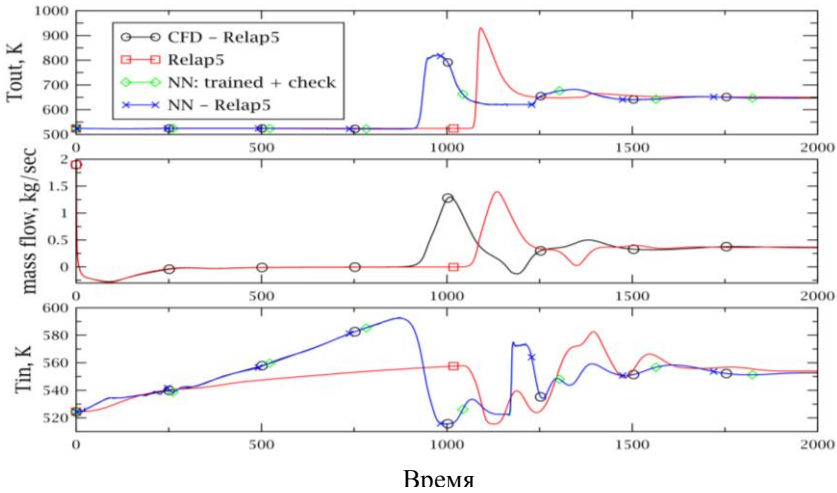


*Рис. 5 Моделирование **Режима 2** (входная температура – Tin, расход – mass flow, выходная температура – Tout) для установки TALL-3D на основе кода Relap5, CFD-1D системы (CFD-Relap5), НС: обучение и проверка (NN: trained+checked), НС: моделирования связи с Relap5 (NN-Relap5)*

4. Проблема идентификации аварии на АЭС

4.1. Представление проблемы

При работе АЭС происходит взаимодействие компонентов и физических процессов, что определяет сложное поведение, как отдельных элементов, так и всей системы при нормальной эксплуатации и, особенно, в аварийных режимах. Поток информации, поступающий к оператору, характеризуется многомерностью, взаимовлиянием между компонентами, наложением стохастических погрешностей. Это затрудняет правильную идентификацию состояний системы, прогнозирование их развития и планирование эффективных противоаварийных мероприятий, что особенно актуально в том случае, когда необходимо быстро принять решение. В дополнение необходимо учесть погрешность (неопределенность) моделирования современными расчетными кодами типа Relap5, что затрудняет прогнозирование характеристик реальных аварийных процессов и, следовательно, корректную идентификацию стадий аварий. Существующие системы поддержки оператора на АЭС зачастую базируются на определении предаварийной ситуации и факте возникновения аварии на основе идентификации отклонения контролируемых параметров от номинальных значений. Однако для принятия действенных противоаварийных мероприятий необходима система, позволяющая также определять тип аварийной ситуации

и осуществлять поддержку оператора или кризисного центра непосредственно во время протекания аварии.

4.2. Постановка задачи

Постановка задачи может быть следующая. При работе АЭС контролируется вектор параметров $\boldsymbol{Y}$. Также имеется множество возможных аварийных ситуаций на АЭС - $A = \{A_i\}_{i=1}^{n}$ (может формироваться на основе ВАБ/ДВАБ). Задача заключается в установлении соответствия между реализацией множества $\boldsymbol{A} \rightarrow \boldsymbol{Y_A}$ и конкретным элементом множества $\boldsymbol{A} \rightarrow \boldsymbol{A_i} \rightarrow \boldsymbol{Y_{Ai}}$, что можно определить из анализа динамики АЭС во время аварии с помощью системных кодов типа Relap5 на основе расчета значений вектора наблюдений $\boldsymbol{Y}$: $\boldsymbol{A_i} \rightarrow \boldsymbol{Y_{Aic}}$. Задача системы идентификации состоит в автоматизации процесса соотнесения во время аварии величин $\boldsymbol{Y_{Ai}}$ и $\boldsymbol{Y_{Aic}}$ (реальных и смоделированных с помощью системного кода) и определении соответствующего элемента $A_i \in A$. Расчетные коды моделируют процессы с погрешностью $\Delta\boldsymbol{Y_{Ai}}=\boldsymbol{Y_{Aic}} - \boldsymbol{Y_{Ai}}$, которая носит, в общем случае, случайный характер. Кроме того существуют параметры модели аварии имеющие чисто стохастическую природу. Например, текущая мощность реактора, температура охлаждающей воды САОЗ, отказы систем и т.д., дающие дополнительную стохастическую составляющую в $\Delta\boldsymbol{Y_{Ai}}$. Анализ ИТ и учет выше приведенных условий определяет, что для соотнесения величин $\boldsymbol{Y_{Aic}}$ и $\boldsymbol{Y_{Ai}}$ как во время наступления аварии, так и в процессе ее протекания, подходят НС , а $\Delta\boldsymbol{Y_{Ai}}$, предлагается учитывать в рамках метода АН.

4.3. Идентификация аварий на АЭС на основе применения нейронных сетей

В АН, неопределенности необходимо учитывать через соответствующие вероятностные распределения и использовать при генерации набора данных по каждой $\boldsymbol{A_i}$. Существуют разные методы для анализа неопределенностей, но для сложных системных кодов и большого количества параметров методы на основе метода МК являются оптимальными. В данном подходе используются формула Уилкса для определения необходимого минимального и достаточного количества расчетов [8]:

$$1-\frac{\alpha}{100}^{K} - K(1-\frac{\alpha}{100}^{K})\frac{\alpha}{100}^{K-1} \geq \frac{\beta}{100}; \quad P\{P(m \leq Y \leq M) \geq \alpha\} \geq \beta \qquad (3)$$

Здесь: $\boldsymbol{P}$- вероятность, $\boldsymbol{Y}$- выходные данные, $\boldsymbol{m}$, $\boldsymbol{M}$ - минимальное и максимальное значение, $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$- вероятности, $\boldsymbol{K}$ – минимальное число расчетов. Тогда для получения 95% ($\boldsymbol{\alpha}$) доверительного интервала изменения $\boldsymbol{Y}$ с 95% уверенностью ($\boldsymbol{\beta}$), необходимо выполнить $\boldsymbol{K}$ =93 расчёта.

Для повышения эффективности расчетов АН в подходе используется метод латинских гиперкубов (ЛГ), позволяющий улучшить эффективность

стандартного метода МК, а также параллельные вычисления, реализованные в программной системе NPO. Метод МК оказывается недостаточно эффективным, когда имеется большая размерность пространства и имеются длинные «хвосты» функций плотности распределения. ЛГ является одним из подвидов метода МК и дает больше гарантии, что точки могут оказаться в маловероятных областях.

Важной компонентой предлагаемого подхода является использование ИТ на основе НС. При этом значимым вопросам являются правильный выбор архитектуры НС и процедуры ее обучения. Организация процедуры обучения НС базируется на обеспечении максимального качества идентификации. Для этого исходные данные для ее тренировки (настройки) случайным образом разделяются на два подмножества – обучающее и контрольное в пропорции 75% и 25%, что часто используется в НС. Для обучения применяется алгоритм обратного распространения ошибки с адаптивной коррекцией. Формирование обучающего и контрольного множеств для НС является важным элементом архитектуры предлагаемого подхода. Так как на сегодняшний день отсутствуют фактические данные изменения параметров АЭС по полному спектру возможных аварий $\boldsymbol{A}$, то в данном методе обучающее множество образуется с помощью моделирования на основе системных кодов.

Отладка и проверка работоспособности предлагаемого метода проводилась на основе моделей АЭС для системного кода Relap5 для РУ ВВЭР-440/В213 и ВВЭР-1000/В320. Множество $\boldsymbol{A}$ было сформировано на основе аварий типа течи из первого контура с возможной вариацией места - холодная или горячая нитка ГЦТ. Размер течей варьировался с шагом 10 мм от 20 до 90 мм условного диаметра. На это накладывались возможные комбинации отказов системы безопасности различного типа. Всего в множество $\boldsymbol{A}$ было включено 62 типов аварий.

Для АН необходимо выделить соответствующие параметры модели, для чего использовались результаты работ [8], где количество параметров неопределенностей достигает 171. Однако имеет смысл учитывать только оказывавшие наибольшее влияние на характеристики безопасности АЭС – т.е. использовать результаты анализа чувствительности. В результате было отобрано 9 наиболее существенных факторов.

Другой важной задачей является правильный выбор множества параметров для мониторинга состояния АЭС. При рассмотрении модели конкретных РУ были выбраны 24 параметра: давление на входе и выходе активной зоны, расход по петлям и т.д.

Значимым моментом является определение оптимальных параметров НС - количества слоев и нейронов в каждом слое, т.к. существует известная проблема переобученности НС при слишком большом количестве нейронов, и плохое прогнозирование в противном случае. Исследования зависимости качества обучения НС (средней вероятности ошибки на контрольной выборке) от числа нейронов в скрытом слое показали: существует оптимум в

количестве нейронов, после которого дальнейшее их увеличение не приводит к улучшению качества сети; наличие локальных оптимумов в зависимости.

Для работ с НС была разработана программа neuroV для выполнения операций по автоматизированному нахождению параметров НС, ее обучению и распознаванию аварийной ситуации. В ней реализовано два алгоритма поиска оптимальной структуры НС. Первый использует последовательный перебор всех возможных комбинаций нейронов по заданному количеству скрытых слоев. Однако он будет неэффективен при наличии нескольких скрытых слоев и широком диапазоне изменения количества нейронов, а также при наличии локальных оптимумов. Для решения проблемы был разработан метод поиска архитектуры НС на основе ГА. Пример расчетов по нахождению оптимальной структуры НС с использованием ГА с помощью программы neuroV приведен на рис.6. Использовалась НС с двумя скрытыми слоями, в которых нейроны варьировались соответственно в диапазонах: 20 ÷ 100; 1÷100. Возможное количество всех комбинаций составляет 8000 вариантов. На рис.7 представлены аппроксимирующие поверхности для нормированной ошибки обучения в зависимости от варьирования числа нейронов по слоям для разных вариантов: в ГА2 использовался постоянный параметр скорости обучения, в ГА3 адаптивный. Видно, что зависимость имеет немонотонный характер и использование стандартных методов оптимизации будет неэффективно. Однако ГА позволяет находить глобальный оптимум за существенно меньшее число шагов – 30 ÷ 50. Для повышения вычислительной эффективности процесса поиска ГА в программе neuroV имеется возможность его распараллеливания, реализованная на основе PVM.

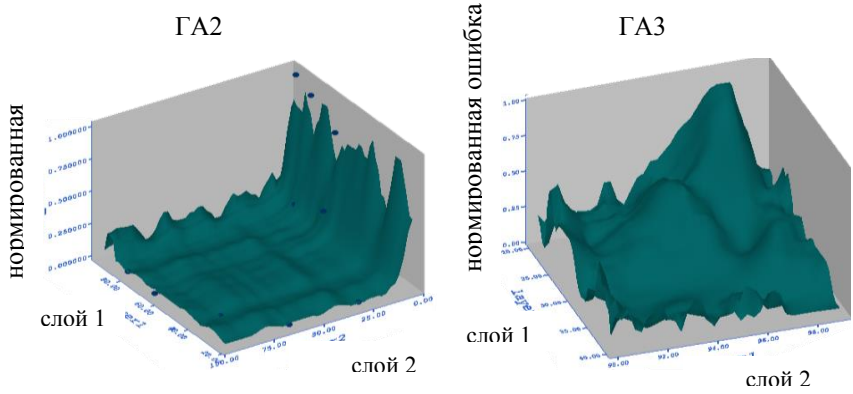


Рис.6 Аппроксимирующая поверхность зависимости ошибки обучения от числа нейронов в слоях для вариантов обучения ГА2 и ГА3.

После определения оптимальной структуры нейронной сети необходимо осуществить настройку системы. Для этого был разработан итерационный алгоритм на основе итераций «обучение – проверка», функционирование

которого в neuroV обеспечивает успешное обучение НС распознаванию типа аварии. Также были показаны хорошие обобщающие свойства настроенной системы идентификации в рамках исследования возможности качественно правильного распознавания аварий, на которые система не была первоначально настроена.

В дальнейших работах планируется изучение возможности использования системы для идентификации стадий развития аварий и формирования обучающего множества с использованием результатов ДВАБ, рассмотренного ранее. Эффективность настройки НС также может быть серьезно увеличена на базе использования технологии CUDA.

5. Заключение

В статье были рассмотрены три направления использования ИТ в ядерной энергетике. Было показано, что применяемые новые технологии совместно с существующими позволяют получить качественно новый уровень анализа, существенно повысить его вычислительную эффективность. Хотя представленные методы используются в ядерной энергетике, в тоже время данный опыт не теряет своей общности при его применении к другим сложным системам, например в аэрокосмической индустрии, химической промышленности и т.д..

Список литературы

[1] Belhadj M, Hassan M, Aldemir T, On the need for dynamic methodologies in risk and reliability studies, Reliability Engineering and System Safety, 1992, V38**,** 219-236

[2] Coppit D, Sullivan K J, Dugan J B. Formal semantics of models for computational engineering- a case study on dynamic fault trees, Software Reliability Engineering,11th International Symposium, San Jose, CA. USA, 8 –11 Oct. 2000, 270-282

[3] Marseguerra M, Zio E, Devooght J, Labeau P E. A concept paper on dynamic reliability via Monte Carlo simulation, Mathematics and Computers in Simulation, 1998,V47 **,** 371-382

[4] Hakobyan A P. Severe Accident Analysis using Dynamic Accident Progression Event Trees, PhD Thesis, Ohio State University, 2006, 225

[5] Mitchell Melanie , An Introduction to Genetic Algorithms, A Bradford Book The MIT Press, Cambridge, Massachusetts, London, England, Fifth printing, 1999, 158

[6] Воробьев Ю.Б., Кузнецов В.Д., Использование современных интегральных кодов для управления безопасностью АЭС, Вестник МЭИ, 2001, №5, 31-37

[7] Байбаков В.Д., Воробьев Ю.Б., Кузнецов В.Д., Коды для расчетов ядерных реакторов, Издательство МЭИ, М., 2003, 162

[8] Мансури Масуд, Анализ неопределенностей параметров при моделировании динамических процессов в контурах АЭС с ВВЭР, дис. к.т.н., МЭИ, 2005, 166

[9] Vorobyev Y, Kudinov P. Development and Application of a Genetic Algorithm Based Dynamic PRA Methodology to Plant Vulnerability Search, proceedings ANS PSA 2011 International Topical Meeting on Probabilistic Safety Assessment and Analysis, Wilmington, NC, on CD-ROM, American Nuclear Society, LaGrange Park, IL, on CD-ROM, March 13-17, 2011, 15

[10] David W. Scott , Stephan R. Sain, Multi-dimensional Density Estimation, Handbook of Statistics, 2005, V24, 229 - 261

[11] D.L. Aumiller, E.T. Tomlinson, R. C. Bauer, A Coupled RELAP5-3D/CFD Methodology with a Proof-of-Principle Calculation, Nuclear Engineering and Design, 2001,V205, 83-90
[12] M. Jeltsov, K. Kööp, W. Villanueva, P. Kudinov, Development of multi-scale simulation methodology for analysis of heavy liquid metal thermal hydraulics with coupled STH and CFD codes, Proceedings of The 9th International Topical Meeting on Nuclear Thermal-Hydraulics, Operation and Safety (NUTHOS-9) N9P0298 Kaohsiung, Taiwan, September 9-13, on CD-ROM, 2012, 18
[13] S. E. Yakush, N. T. Lubchenko and P. Kudinov, SURROGATE MODELS FOR DEBRIS BED DRYOUT, The 15th International Topical Meeting on Nuclear Reactor Thermal - Hydraulics, NURETH-15, Pisa, Italy, May 12-17, on CD-ROM, 2013, 16

Application of information technologies (genetic algorithms, neural networks, parallel calculations) in safety analysis of Nuclear Power Plants

Yu.B. Vorobyev, P. Kudinov**, M. Jeltsov**, K. Kööp**, T.V.K. Nhat*,*
yura@npp.mpei.ac.ru, pkudinov@kth.se, marti@safety.sci.kth.se,
*kaspar@safety.sci.kth.se, nhattvk@gmail.com, *SRU «MPEI»,*
*Krasnokazarmennaya 14, Moscow, Russia, **Royal Institute of Technology,*
Valhallavägen 79, Stockholm, Sweden

Annotation. This paper investigates important issues in three types of safety assessment methodologies commonly applied for Nuclear Power Plants (NPP). These methodologies are i) dynamic probabilistic safety assessment (DPSA) where application of genetic algorithm (GA) is shown to improve the efficiency of the analysis, ii) deterministic safety assessment (DSA) with meta model representation of the system using pre-performed computational fluid dynamics (CFD) code and iii) vulnerability search (e.g. identification of accident scenarios in an NPP) with application of neural network (NN). The use of advanced computational tools and methods such as genetic algorithms, neural networks and parallel computations improve the efficiency of safety analysis. To achieve the best effect, these advanced technologies are to be integrated with existing classical methods of safety analysis of the NPP.

Key words: information technologies, safety analysis, meta-modelling

References

[1] Belhadj M, Hassan M, Aldemir T, On the need for dynamic methodologies in risk and reliability studies, Reliability Engineering and System Safety, 1992, V38**,** 219-236
[2] Coppit D, Sullivan K J, Dugan J B. Formal semantics of models for computational engineering- a case study on dynamic fault trees, Software Reliability Engineering,11th International Symposium, San Jose, CA. USA, 8 –11 Oct. 2000, 270-282
[3] Marseguerra M, Zio E, Devooght J, Labeau P E. A concept paper on dynamic reliability via Monte Carlo simulation, Mathematics and Computers in Simulation, 1998,V47 **,** 371-382
[4] Hakobyan A P. Severe Accident Analysis using Dynamic Accident Progression Event Trees, PhD Thesis, Ohio State University, 2006, 225
[5] Mitchell Melanie , An Introduction to Genetic Algorithms, A Bradford Book The MIT Press, Cambridge, Massachusetts, London, England, Fifth printing, 1999, 158
[6] Vorob'ev YU.B., Kuznetsov V.D. , Ispol'zovanie sovremennykh integral'nykh kodov dlya upravleniya bezopasnost'yu AEHS [The use of modern best estimate codes for controlling of the NPP safety], Vestnik MEHI [Herald of MPEI], 2001, №5, pp 31-37 (in Russian).
[7] Bajbakov V.D., Vorob'ev YU.B., Kuznetsov V.D. , Kody dlya raschetov yadernykh reaktorov [Calculation codes for nuclear reactors], , Izdatel'stvo MEHI [MPEI Publ.], M., 2003, 162 p (in Russian).
[8] Mansuri Masud, Analiz neopredelennostej parametrov pri modelirovanii dinamicheskikh protsessov v konturakh AEHS s VVEHR [Uncertainty analysis of parameters by modeling of dynamic processes for NPP with VVER], dis. k.t.n. [Phd thesis], MEHI [MPEI], 2005, 166 p (in Russian).

[9] Vorobyev Y, Kudinov P. Development and Application of a Genetic Algorithm Based Dynamic PRA Methodology to Plant Vulnerability Search, proceedings ANS PSA 2011 International Topical Meeting on Probabilistic Safety Assessment and Analysis, Wilmington, NC, on CD-ROM, American Nuclear Society, LaGrange Park, IL, on CD-ROM, March 13-17, 2011, 15
[10] David W. Scott , Stephan R. Sain, Multi-dimensional Density Estimation, Handbook of Statistics, 2005, V24, 229 - 261
[11] D.L. Aumiller, E.T. Tomlinson, R. C. Bauer, A Coupled RELAP5-3D/CFD Methodology with a Proof-of-Principle Calculation, Nuclear Engineering and Design, 2001,V205, 83-90
[12] M. Jeltsov, K. Kööp, W. Villanueva, P. Kudinov, Development of multi-scale simulation methodology for analysis of heavy liquid metal thermal hydraulics with coupled STH and CFD codes, Proceedings of The 9th International Topical Meeting on Nuclear Thermal-Hydraulics, Operation and Safety (NUTHOS-9) N9P0298 Kaohsiung, Taiwan, September 9-13, on CD-ROM, 2012, 18
[13] S. E. Yakush, N. T. Lubchenko and P. Kudinov, SURROGATE MODELS FOR DEBRIS BED DRYOUT, The 15th International Topical Meeting on Nuclear Reactor Thermal - Hydraulics, NURETH-15, Pisa, Italy, May 12-17, on CD-ROM, 2013, 16

Обзор методов упрощения полигональных моделей на графическом процессоре

Гонахчян В.И.
ИСП РАН, Москва
pusheax@ispras.ru

Аннотация. Упрощение полигональных моделей является одной из распространенных методик, позволяющих увеличить скорость растеризации масштабных сцен, состоящих из большого количества сложных объектов. Традиционные алгоритмы, как правило, основанные на последовательном исключении ребер и граней, имеют высокую вычислительную сложность, что является препятствием для реализации ряда графических приложений на CPU. С развитием технологий программирования графического процессора открываются новые возможности для эффективной параллельной реализации данных алгоритмов. В работе обсуждаются некоторые известные алгоритмы упрощения полигональных моделей, использующие возможности распараллеливания независимых операций исключения ребер и спекулятивных оценок визуального качества редуцируемого полигонального представления. Сравниваются основные характеристики описанных алгоритмов и параллельных программ, а также даются рекомендации по их практическому использованию.

1. Введение

Трехмерная компьютерная графика — научная область, имеющая многочисленные приложения в визуализации научных и инженерных расчетов, САПР, анимации, играх. Как правило, необходимость реалистичной визуализации, желательно, в режиме реального времени приводит к довольно жестким требованиям к эффективности растеризации динамических пространственно–трехмерных сцен. В случае сложных сцен, состоящих из тысяч и миллионов элементов с индивидуальным геометрическим представлением, удовлетворить этим требованиям удается далеко не всегда даже с учетом перманентного технологического прогресса в развитии графических процессоров.

Одним из перспективных подходов к эффективной растеризации является использование альтернативных, как правило, упрощенных полигональных представлений для элементов сцен (LOD — аббревиатура от Level Of Details).

При растеризации сцены учитывается фактор близости элементов к камере обзора (более точно, угловое расстояние или пиксельное разрешение элементов), и оригинальная геометрическая модель каждого индивидуального элемента подменяется упрощенным представлением, которое обеспечивает необходимое визуальное качество итогового изображения. В частности, элементы с разрешением меньше одного экранного пиксела, вообще исключаются из обработки, а элементы, находящиеся на значительном удалении от камеры, заменяются полигональными моделями с меньшим количеством граней.

На рис. 1 тестовая геометрическая модель представлена четырьмя альтернативными полиэдрами с общим количеством треугольных граней 69666, 2500, 250 и 75 соответственно. При уменьшении количества граней визуальное качество модели заметно ухудшается. Однако по мере удаления от камеры это требование становится все менее критичным, и можно воспользоваться упрощенным представлением, которое бы существенно ускорило процесс растеризации.

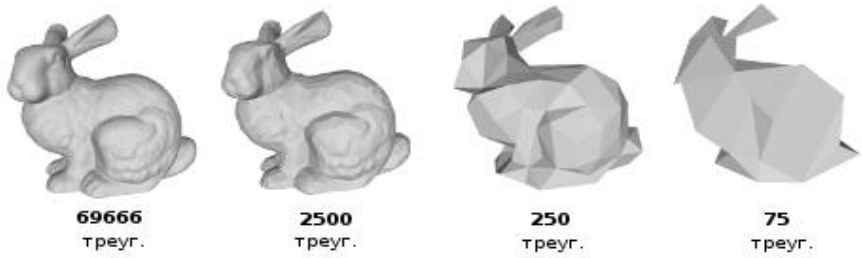


Рисунок 1. Уровни детализации полигональной модели Stanford bunny

На рис. 2 показана смена упрощенных представлений при удалении от камеры. Первый полиэдр содержит 20000 граней. Количество граней уменьшается в два раза при переходе к следующему представлению. Видно, что визуальное качество при этом остается примерно одинаковым.

Рассмотрим области применения методов упрощения полигональных сеток. В технологиях компьютерного зрения, в частности, технологиях лазерного сканирования используются программы для обнаружения, отслеживания, классификации объектов. Результатом их применения часто являются полигональные сетки с миллионами вершин. Примечательно, что необходимость их упрощения часто диктуется не только требованиями ускорить процесс визуализации, но и желанием увеличить точность представления исходных данных и снизить требования к вычислительным ресурсам, необходимым для работы с ними. При этом важно сохранить индивидуальные особенности объектов, связанные с их пропорциями,

наличием острых углов и дырок, для последующего отображения и распознавания.

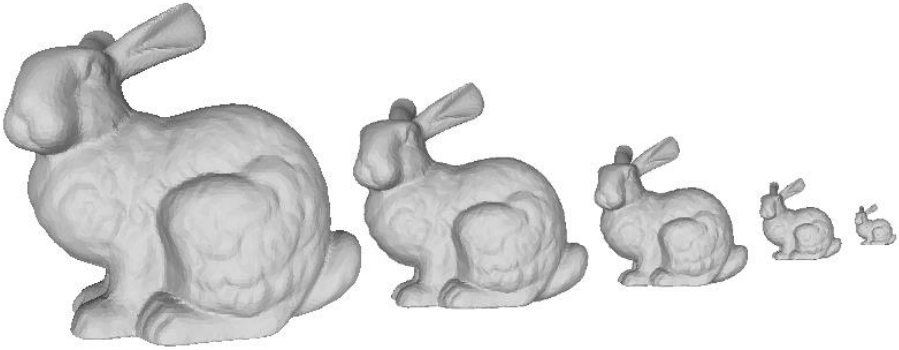


Рисунок 2. Смена уровней детализации полигональной модели при разных удалениях от экрана

Построение упрощенных геометрических моделей, обеспечивающих необходимое визуальное качество с точки зрения психофизики восприятия графических образов человеком, представляет собой нетривиальную задачу, порой требующую привлечения многих прикладных специалистов. Однако разработан мощный арсенал автоматических методов инкрементального упрощения исходной полигональной модели. В настоящей работе обсуждаются подобные возможности, при этом главное внимание уделяется методам, обладающим внутренним параллелизмом и допускающим эффективную реализацию на графических процессорах.

Традиционные автоматические методы упрощения моделей, как правило, основанные на последовательном исключении ребер и граней, имеют высокую вычислительную сложность, что является препятствием для реализации ряда графических приложений на CPU [5]. Вместе с тем, данные методы допускают распараллеливание независимых операций исключения ребер и спекулятивных оценок визуального качества редуцируемого полигонального представления. В работе [22] описана одна из первых программных реализаций параллельного алгоритма для составления многоуровневых сеток [18]. С развитием технологий программирования графического процессора открываются новые возможности для эффективной реализации данных алгоритмов в графических приложениях реального времени. Существующие инструменты программирования GPU, в частности, шейдерные языки CG, GLSL, HLSL и интегрированные GPGPU системы CUDA, OpenCL, предоставляют развитые средства реализации низкоуровневых операций с высокой степенью параллелизма.

Растеризация на GPU предполагает конвейеризацию вычислений с явным выделением следующих этапов:

- Этап обработки вершин, на котором к каждой вершине трехмерной модели применяются трансформации для позиционирования в системе координат сцены;
- Этап обработки фрагментов, на котором определяется цвет каждого пиксела с учетом источников света, материалов;
- Этап формирования итогового изображения с учетом видимости объектов сцены.

Более подробно графический конвейер обсуждается в [9].

Выделяют вершинные, геометрические и фрагментные шейдерные программы. Они соответствуют этапам графического конвейера. Организация шейдерных вычислений естественным образом ложится на архитектуру графического процессора. Вершины и пикселы обрабатываются параллельно по принципу SIMD (Single Instruction Multiple Data). Фрагментные процессоры имеют доступ к текстурам. Кадровый буфер может быть считан как текстура при последующих проходах отрисовки. Обычно шейдеры используются по прямому назначению, однако функциональная полнота операций графического процессора позволяет выполнять любые вычисления. Высокая степень параллельности сделала шейдеры хорошим инструментом для реализации различных операций с матрицами и массивами.

В отличие от шейдерного программирования в рамках концепции GPGPU можно использовать как графические, так и центральные процессоры (CPU). Существует две основных платформы для GPGPU программ: CUDA и OpenCL. CUDA — закрытая интегрированная система, разрабатываемая компанией Nvidia. OpenCL — открытый стандарт, разрабатываемый консорциумом Khronos Group.

В данной статье используются термины, применяемые в модели вычислений OpenCL. Устройство (device) состоит из вычислительных модулей (compute unit) с обрабатывающими элементами (processing element). Роль устройства может выполнять графический процессор или центральный процессор. Вычисления описываются на языке OpenCL C (подмножество ISO C '99) в ядрах (kernel) — специальных функциях. Ядра выполняются рабочими элементами (work item) в рабочих группах (work group). Рабочий элемент — экземпляр ядра программы с уникальным идентификатором. В рабочей группе есть локальная память (local memory) с возможностью быстрого доступа. Вычисления соответствуют модели SIMD (Single Instruction Multiple Data) — одна и та же программа выполняется несколькими вычислительными модулями с разными входными данными. В парадигме GPGPU обращение к глобальной памяти устройства и обмен данными между устройствами стоят относительно дорого.

В настоящей работе обсуждаются некоторые известные параллельные алгоритмы упрощения полигональных моделей, сравниваются их основные

характеристики, а также даются рекомендации по их практическому использованию.

2. Классификация методов упрощения

Алгоритмы упрощения сеток различаются по типам поверхностей, к которым они применяются. Одни работают только на многообразиях (manifold), другие также допускают полигональные сетки, не являющиеся многообразиями (non–manifold). Рассмотрим проблемы, которые возникают при упрощении топологии полигональной модели. Полигональная сетка, каждое ребро которой входит в два треугольника, является двумерным многообразием. Двумерное многообразие с границей допускает ребра, которые входят только в один треугольник. Немногообразные полигональные сетки имеют особенности, которые затрудняют их упрощение. В частности, существуют ребра, которые входят в более чем два треугольника, и вершины, которые являются единственными связующими точками для двух множеств треугольников. Такие участки полигональной сетки требуют отдельной обработки алгоритмом. Большинство алгоритмов пропускает данную часть модели. В случае, когда визуальное качество критично (например, для метода конечных элементов), алгоритмы могут завершиться с ошибкой при нахождении подобных участков. Чтобы избежать этого, требуется предварительно убедиться, что исходная модель является многообразием, и при необходимости преобразовать ее в требуемое представление. Однако есть алгоритмы, которые справляются и с немногообразными полигональными сетками [8].

Алгоритмы также различаются по способу удаления геометрии: прореживание (decimation), выборка (sampling).

Прореживание — итеративный метод удаления геометрии полигональной модели. За одну итерацию объединяются вершины, удаляется вершина, ребро или треугольник с последующей триангуляцией для заполнения дырки при необходимости. При достижении целевого числа полигонов алгоритм завершается.

Рассмотрим основные операции по удалению геометрии при прореживании. На рис. 3а изображена операция объединения вершин, в результате которой три вершины замещаются одной. При этом удаляются все ребра и треугольники с их участием, затем выполняется триангуляция. На рис. 3б изображена операция удаления вершины со всеми зависимыми ребрами и треугольниками. На рис. 3в представлена операция удаления ребра, в результате которой два треугольника становятся вырожденными. При схлопывании ребра (edge collapse) удаляется ребро и вырожденные треугольники. Схлопывание пары (pair collapse) работает таким же образом, но допускает отсутствие ребра между вершинами. На рис. 3г показана операция удаления треугольника с последующей триангуляцией.

Выборка позволяет за несколько шагов существенно сократить количество граней модели. При выборке исходную модель пересекают с трехмерной сеткой заданного размера и упрощают геометрию в каждой ячейке. Обычно используют равномерную сетку. Недостаток этого подхода заключается в том, что плотность вершин в моделях зачастую неравномерная. В результате качество может ухудшиться.

Помимо основных категорий в обзоре также рассматриваются следующие характеристики методов:

- Используемая операция упрощения,
- Точность представления исходной сетки,
- Сравнение производительности и визуального качества с реализацией на CPU.

3. Методы упрощения

Метод итеративного прореживания полигональной сетки [12] предназначен для упрощения результата алгоритма "marching cubes". За каждый проход принимается решение об удалении каждой вершины. Если вершину можно удалить без изменения топологии с допустимой точностью, то алгоритм удаляет ее и все связанные с ней треугольники. Это оставляет дырку в модели, которая затем заполняется треугольниками. Упрощение продолжается до тех пор, пока не закончатся подходящие вершины. Алгоритм, приведенный в работе [16], расширяет данный метод путем прореживания с постепенным уменьшением точности представления.

При кластеризации вершин [13] происходит наложение равномерной сетки на модель для упрощения топологии. В каждой ячейке оставляют одну вершину, которая входит в треугольники с наибольшей суммарной площадью. Этот метод имеет линейную сложность (O(n) для n вершин), но дает относительно низкое визуальное качество из-за изменения топологии и отсутствия возможности указать точность упрощения. Алгоритм, приведенный в работе [15], расширяет данный метод, используя более эффективное пространственное деление, которое называется кластеризацией с плавающей ячейкой (floating–cell clustering). В нем вершины сортируются по приоритету, и на месте вершины с наивысшим приоритетом создается ячейка, в которой производится упрощение. Далее выбирается следующая вершина, и процесс повторяется. Отсутствие сетки в этом алгоритме позволяет упрощать модели вне зависимости от положения и ориентации. Алгоритм, приведенный в работе [4], в отличие от [15], работает с моделью во внешней памяти, используя оперативную память только для хранения результата упрощения. Для оценки точности представления используется квадратичный многочлен [1].

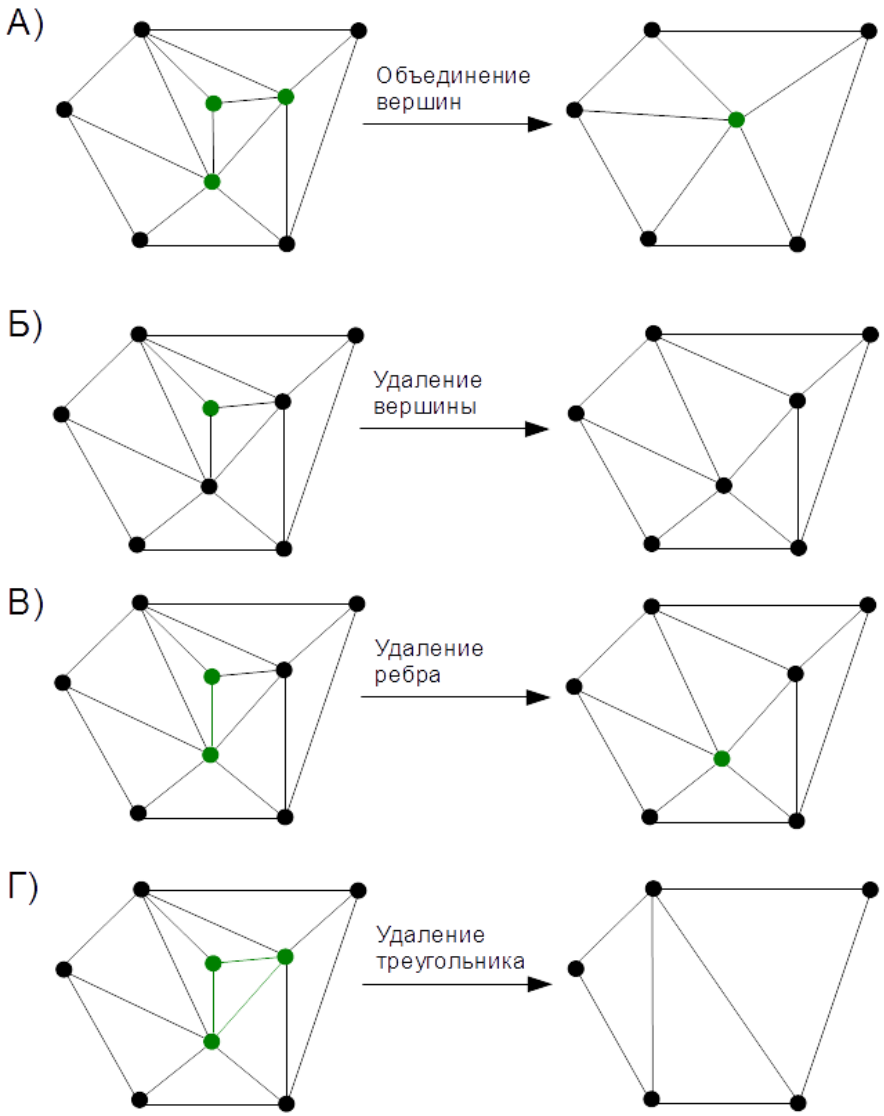


Рисунок 3. Базовые операции по изменению геометрии при прореживании

Метод воксельного упрощения [14] позволяет упрощать топологию постепенно с использованием техники обработки сигналов. Сначала на полигональную модель накладывается трехмерная сетка из вокселей. Каждому вокселю присваивается значение 1, когда он находится внутри модели, либо 0 — вне модели. Затем применяется фильтр нижних частот, который удаляет

характерные особенности на высоких частотах. Далее применяется алгоритм "marching cubes", который восстанавливает полигональное представление модели. Главным недостатком алгоритма является ухудшение визуального качества за счет удаления ряда характерных особенностей.

В работе [17] используются конверты упрощения (envelopes) для гарантирования точности упрощения. Создаются две граничные поверхности, которые смещены относительно исходной на константы δ и -δ. В случае самопересечения этих поверхностей алгоритм понижает значение константы. Затем производится упрощение и делается проверка того, что упрощенная поверхность содержится в пределах граничных поверхностей. Данный метод сохраняет топологию и работает только на ориентируемых многообразиях.

В работе [18] описан способ прореживания многообразия для создания многоуровневых сеток (progressive mesh). Многоуровневая сетка состоит из начального представления и набора операций по разделению вершин (vertex split). Операция разделения вершины является обратной по отношению к операции схлопывания ребра. Применение всех операций по разделению вершин восстанавливает исходную модель. При этом многоуровневые сетки рассчитаны на визуализацию в реальном времени частично восстановленной модели. В работе [21] представлена общая техника на основе метода [18], которая жертвует скоростью для того, чтобы принимать на вход произвольные полигональные сетки.

В работе [19] описан алгоритм, использующий квадратичный многочлен для определения точности представления в вершинах. Квадратичный многочлен задается матрицей размерности 4x4, которая позволяет найти сумму квадратов расстояний от вершины до плоскостей соседних треугольников. Сначала выполняются операции по удалению вершин, которые дают наивысшую точность упрощения. Алгоритм итеративно объединяет пары вершин, между которыми может и не быть ребра. Для определения точности упрощения в новой вершине выполняют сложение матриц, соответствующих квадратичным многочленам удаленных вершин. Это позволяет упрощать топологию при гарантированной точности представления исходной сетки. Невозможно подобрать одну точность упрощения, которая дает высокое визуальное качество для различных полигональных сеток. Алгоритм, приведенный в [20], является развитием данного метода для упрощения полигональных моделей с приписанными материалами.

4. Методы упрощения на графическом процессоре

Метод, предложенный в работе [2], совмещает выборку и прореживание полигональных моделей в реальном времени с помощью шейдерных программ для DirectX 10 [3], не сохраняет топологию, работает на многообразиях. За основу взят алгоритм из работы [4]. Предложен способ кластеризации вершин на графическом процессоре. Граничная рамка объекта равномерно разбивается на ячейки. В каждой ячейке оставляют одну вершину,

которая дает наивысшую точность упрощения согласно квадратичному многочлену для определения суммы квадратов расстояний до соседних граней [1]. Алгоритм работает в три прохода графического процессора, используя сохраненные буферы на последующих проходах. Высокая степень параллельности достигается путем применения независимых операций в каждой ячейке кластера. Использование операции схлопывания пары позволяет получить качественно лучшие результаты по сравнению с традиционной кластеризацией. Реализованный алгоритм дает прирост производительности от 15 до 22 раз по сравнению с версией на CPU. Этот метод можно использовать для упрощения автоматически сгенерированной геометрии или при загрузке моделей для повышения производительности. Упрощенные модели можно использовать для обнаружения столкновений.

Метод, предложенный в работе [6], выполняет прореживание полигональных моделей с помощью OpenCL, сохраняет топологию, работает на многообразиях. Исходная модель разбивается на сегменты по 700 треугольников, чтобы не выйти за пределы 32КБ локальной памяти, отведенной для вычислительного модуля. Далее выполняется параллельное упрощение на обрабатывающих элементах. Результат передается в основную память для встраивания в исходную модель. При упрощении используется операция схлопывания ребра. Для оценки стоимости схлопывания ребра используется объемная точность упрощения [7]. Следующее ребро выбирается без сортировки параллельно с помощью обрабатывающих элементов. При достижении заданной точности упрощение прекращается. Утверждается, что алгоритм дает прирост производительности, сравнимый с кластеризацией вершин (от 15 до 22 раз согласно [2]), но обладает низким визуальным качеством из-за поддержания границ сегментов в исходном виде для выполнения встраивания.

Метод, предложенный в работе [8], выполняет прореживание полигональных моделей с помощью шейдерных программ с использованием OpenGL, сохраняет топологию, работает на многообразиях и немногообразных полигональных сетках. Традиционные методы оценки точности упрощения основаны на форме объектов. В статье обсуждаются пользовательские критерии для задания приоритета удаления ребер. Если конечный элемент достигает предельного размера, то его упрощение прекращается вне зависимости от того, сколько в нем осталось граней. В статье описан гибридный подход, при котором одна часть операций выполняется центральным процессором, а другая — графическим процессором. Алгоритм выделяет независимые множества и сортирует их по значению дискретной кривизны (discrete curvature). Вершины независимого множества не должны быть непосредственными соседями (one–ring neighbourhood) других вершин этого же множества. Независимые множества сохраняются в текстурах графического процессора, который применяет критерий для схлопывания ребер и выполняет перестроение сетки. В качестве критерия для схлопывания ребра используется значение односторонней метрики Хаусдорфа. Для этого

каждой вершине назначается область предельной точности (envelope) [17]. После операции схлопывания ребра область становится активной. При пересечении каждой активной области хотя бы с одним треугольником упрощенной модели упрощение считается корректным. Далее эта область назначается упрощенному треугольнику. Применение критерия на графическом процессоре ограничивается независимым множеством вершин. Критерий по объему применяется на CPU. Если удаление ребра меняет топологию, то его пропускают. Немногообразные полигональные сетки обрабатываются центральным процессором. Утверждается, что алгоритм в 8 раз быстрее версии на CPU при упрощении моделей с большим количеством граней. Этот подход используется в методе конечных элементов, где важно работать с упрощенным представлением и требуется удаление характерных особенностей объекта.

Метод, предложенный в работе [10], выполняет прореживание полигональных моделей с помощью OpenCL, сохраняет топологию, работает на многообразиях. Применяется операция схлопывания ребра для постепенного упрощения модели. Для задания точности упрощения в вершине используют квадратичный многочлен [1]. Весь алгоритм, за исключением стадии инициализации данных, выполняется на графическом процессоре. На вход передается модель и желаемое количество вершин на выходе. Алгоритм поддерживает в оперативной памяти массивы с дополнительной информацией о вершинах. Для параллелизации работы определяются независимые множества модели. Независимое множество состоит из вершин, расстояние между которыми в графе вершин модели не меньше трех. На каждой итерации алгоритм находит независимые множества у всей модели, сортирует их параллельно с использованием сортировки "bitonic sorting", упрощает на вычислительных модулях GPU. Утверждается, что прирост производительности составляет 1.5–2.5 раза по сравнению с версией на CPU при сравнимом визуальном качестве.

Метод, предложенный в работе [11], выполняет прореживание полигональных моделей с помощью CUDA, сохраняет топологию, работает на многообразиях. В статье описан гибридный подход к упрощению моделей, который использует центральный и графический процессоры в заданном процентном соотношении. Задачи постепенно назначаются на свободные потоки CPU и GPU в цикле. Стоит отметить, что при этом не учитываются особенности работы графической системы, как это сделано в работе [6], идет расчет на оптимизацию вычислений в CUDA. При выборе следующего треугольника для параллельного упрощения используется механизм блокировки вершин на основе метода "test–and–set". При успешной блокировке треугольник помечается, и производится операция схлопывания ребра. Таким образом, гарантируется равномерность упрощения. Таблицы CPU–GPU при сравнении соотношений 1:0 и 0:1 показывают незначительный прирост производительности в 15% на больших моделях.

Все рассмотренные методы выполняют прореживание геометрии. В [2] используется гибридный подход — выборка с прореживанием. Во всех методах, кроме [2], в качестве операции упрощения используется схлопывание ребра. В [2] используется операция схлопывания пары. В результате все методы, за исключением [2], сохраняют топологию поверхности. В [8] в качестве исходной модели допускаются немногообразные полигональные сетки. Остальные характеристики приведены в табл. 1.

Статья	Реализация	Метод оценки точности упрощения	Производительность по сравнению с CPU
Decoro–Tatarchuk [2]	Шейдеры	Квадратичный многочлен	15–22 раза
Vad'ura [6]	OpenCL	Объемная мера	—
Hjelmervik–Leon [8]	OpenGL	Расстояние Хаусдорфа	8 раз
Papageorgiou–Platis [10]	OpenCL	Квадратичный многочлен	1.5–2.5 раза
Shontz–Nistor [11]	CUDA	—	15%

Таблица 1. Основные характеристики методов упрощения моделей на GPU

5. Заключение

Таким образом, рассмотрены некоторые известные алгоритмы упрощения полигональных моделей, использующие возможности распараллеливания независимых операций исключения ребер и спекулятивных оценок визуального качества редуцируемого полигонального представления. Проведено сравнение основных характеристик описанных алгоритмов и реализующих их параллельных программ. В частности, разобраны случаи, в которых технологии программирования графических процессоров имеют наибольший успех.

Шейдерные методы дают большой прирост производительности [2], но также имеют свои недостатки. Они предназначены для упрощения в реальном времени, что сильно ограничивает их область применения. Графический процессор в первую очередь рассчитан на растеризацию сцен, а не предобработку трехмерных данных, что вызывает сложности в написании и поддержке кода подобных методов. Также при использовании кластеризации

вершин, которая хорошо распараллеливается с помощью шейдеров, визуальное качество получается хуже, чем при прореживании.

Методы с использованием интегрированных сред OpenCL, CUDA проще в реализации. Допускается загрузка результатов вычислений в основную память. Однако при подходах, которые не учитывают особенности вычислительной модели графического процессора, прирост производительности меньше, чем у шейдерных методов [11], [10]. В [6] показано, что требуется учет низкоуровневых деталей (количества локальной памяти и потоковых процессоров), чтобы задействовать одновременно максимальное количество потоковых процессоров. При этом визуальное качество получается хуже последовательной реализации из-за наличия границ между сегментами с повышенной плотностью вершин.

Отметим, что в качестве основной операции по упрощению чаще всего выбирают схлопывание ребра. Это делается не только из соображений простоты, но и потому что при этой операции не требуется последующая триангуляция, как при удалении треугольников или вершин. Это хорошо подходит для графической системы, т. к. она плохо приспособлена для добавления дополнительных треугольников при заполнении дырок.

Не все алгоритмы допускают эффективную параллельную реализацию. Для обсуждаемых задач, как правило, удается выделить независимые операции или подзадачи, которые можно выполнить одновременно. Однако узким местом остается необходимость организации эффективного обмена данными, для чего часто развертываются дополнительные структуры данных в основной памяти [8], [10]. Сбалансированная загрузка центрального и графического процессоров также является проблемой с учетом специфики решаемых подзадач.

Список литературы

[1] M. Garland, P. Heckbert, "Surface simplification using quadric error metrics," in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pp. 209–216, 1997.

[2] C. DeCoro, N. Tatarchuk, "Real–time Mesh Simplification Using GPU," in *Proceedings of the symposium on interactive 3D graphics and games*, 2007.

[3] D. Blythe, "The Direct3D 10 System," *ACM Transactions on Graphics*, vol. 25, no. 3, pp. 724–734, 2006.

[4] P. Lindstrom, "Out–of–core simplification of large polygonal models," in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, 259–262, 2000.

[5] J. Rossignac, P. Borrel, "Multi–resolution 3D approximations for rendering complex scenes.," *Modeling in Computer Graphics: Methods and Applications*, pp. 455–465, 1993.

[6] J. Vad'ura, "Parallel mesh decimation with GPU," 2011.

[7] A. Gueziec, "Surface simplification inside a tolerance volume," *Second Annual International Symposium on Medical Robotics and Computer Aided Surgery*, pp. 123–139, 1995.

[8] J. Hjelmervik, J. Leon, "GPU–accelerated shape simplification for mechanical based applications," *Shape modeling international*, pp. 91–102, IEEE Computer Society, 2007.
[9] D. Shreiner, M. Woo, J. Neider, T. Davis, "OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2," 2005.
[10] A. Papageorgiou, N. Platis, "Triangular mesh simplification on the GPU," *NASAGEM Geometry Processing Workshop*, Computer Graphics International, 2013.
[11] S. Shontz, D. Nistor, "CPU–GPU algorithms for triangular surface mesh simplification," in *Proceedings of the 21st international meshing roundtable*, pp. 475–492, Springer, Berlin, 2013.
[12] W. Schroeder, J. Zarge, and W. Lorensen, "Decimation of Triangle Meshes," *ACM Siggraph Computer Graphics*, vol. 26, no. 2, pp. 65–70, 1992.
[13] J. Rossignac and P. Borrel, "Multi–Resolution 3D Approximations for Rendering Complex Scenes," *Geometric Modeling in Computer Graphics*, pp. 455–465, 1993.
[14] T. He, L. Hong, A. Kaufman, A. Varshney and S. Wang, "Voxel–Based Object Simplification," in *Proceedings of the 6th conference on Visualization '95*, pp. 296–303, 1995.
[15] K. Low and T. Tan, "Model Simplification Using Vertex–Clustering," in *Proceedings of the 1997 symposium on Interactive 3D graphics*, pp. 75–82, 1997.
[16] W. Schroeder, "A Topology Modifying Progressive Decimation Algorithm," in *Proceedings of the 8th conference on Visualization '97*, pp. 205–212, 1997.
[17] J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. Brooks, and W. Wright, "Simplification Envelopes," in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pp. 119–128, 1996.
[18] H. Hoppe, "Progressive Meshes," in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pp. 99–108, 1996.
[19] M. Garland and P. Heckbert, "Surface Simplification Using Quadric Error Metrics," in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pp. 209–216, 1997.
[20] M. Garland and P. Heckbert, "Simplifying Surfaces with Color and Texture using Quadric Error Metrics," in *Proceedings of the conference on Visualization '98*, pp. 263–269, 1998.
[21] J. Popovic and H. Hoppe, "Progressive Simplicial Complexes," in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pp. 217–224, 1997.
[22] F. Dehne, C. Langis, G. Roth, "Mesh simplification in parallel," in *Proceedings of the 4th International Conference on Algorithms and Architectures for Parallel Processing*, pp. 281–290, 2000.

Survey of polygonal surface simplification algorithms on GPU

V.I. Gonakhchyan
ISP RAS, Moscow, Russia
pusheax@ispras.ru

Annotation. Rendering time depends on number of faces in polygonal mesh. Displaying large number of objects or objects with a lot of faces can result in performance degradation. One of the ways to overcome this problem is to use several levels of detail for polygonal mesh. Levels of detail with low quality are used when object is far away from camera and difference in number of faces is not apparent to viewer. Coarse meshes can be generated manually which is difficult and time consuming. For many applications (computer graphics, computer vision, finite element analysis) it's preferable to use automated methods for polygonal surface simplification to increase performance. Traditional methods for surface simplification are designed for CPU. GPU methods provide improvements in performance and have comparable visual quality. This survey covers algorithms on GPU. They are based on either shader programming or general purpose computing frameworks like OpenCL and CUDA. Shader methods are very restrictive. They are designed to render images in real–time. General purpose computing frameworks provide flexible APIs but require attention to hardware implementation details to avoid performance bottlenecks. Main features of algorithms are provided for comparison as the result. The most used operation for geometry simplification is edge collapse. It's difficult to avoid expensive data exchange between GPU and CPU. It's important to take into account computational model of GPU to increase number of stream processors working in parallel.

Key words: mesh simplification, level–of–detail, GPU programming

References

[1] Garland M., Heckbert P. "Surface simplification using quadric error metrics" in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pp. 209–216, 1997.
[2] DeCoro C., Tatarchuk N. "Real–time Mesh Simplification Using GPU" in *Proceedings of the symposium on interactive 3D graphics and games*, 2007.
[3] Blythe D. "The Direct3D 10 System" in *ACM Transactions on Graphics*, vol. 25, no. 3, pp. 724–734, 2006.
[4] Lindstrom P. "Out–of–core simplification of large polygonal models" in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, 259–262, 2000.
[5] Rossignac J., Borrel P. "Multi–resolution 3D approximations for rendering complex scenes" *Modeling in Computer Graphics: Methods and Applications*, pp. 455–465, 1993.
[6] Vad'ura J. "Parallel mesh decimation with GPU" 2011.

[7] Gueziec A. "Surface simplification inside a tolerance volume" *Second Annual International Symposium on Medical Robotics and Computer Aided Surgery*, pp. 123–139, 1995.
[8] Hjelmervik J., Leon J. "GPU–accelerated shape simplification for mechanical based applications" *Shape modeling international*, pp. 91–102, IEEE Computer Society, 2007.
[9] Shreiner D., Woo M., Neider J., Davis T. "OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2", 2005.
[10] Papageorgiou A., Platis N. "Triangular mesh simplification on the GPU" *NASAGEM Geometry Processing Workshop*, Computer Graphics International, 2013.
[11] Shontz S., Nistor D. "CPU–GPU algorithms for triangular surface mesh simplification" in *Proceedings of the 21st international meshing roundtable*, pp. 475–492, Springer, Berlin, 2013.
[12] Schroeder W., Zarge J. and Lorensen W. "Decimation of Triangle Meshes" *ACM Siggraph Computer Graphics*, vol. 26, no. 2, pp. 65–70, 1992.
[13] Rossignac J. and Borrel P. "Multi–Resolution 3D Approximations for Rendering Complex Scenes" *Geometric Modeling in Computer Graphics*, pp. 455–465, 1993.
[14] He T., Hong L., Kaufman A., Varshney A. and Wang S. "Voxel–Based Object Simplification" in *Proceedings of the 6th conference on Visualization '95*, pp. 296–303, 1995.
[15] Low K. and Tan T. "Model Simplification Using Vertex–Clustering" in *Proceedings of the 1997 symposium on Interactive 3D graphics*, pp. 75–82, 1997.
[16] Schroeder W. "A Topology Modifying Progressive Decimation Algorithm" in *Proceedings of the 8th conference on Visualization '97*, pp. 205–212, 1997.
[17] Cohen J., Varshney A., Manocha D., Turk G., Weber H., Agarwal P., Brooks F. and W. Wright "Simplification Envelopes" in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pp. 119–128, 1996.
[18] Hoppe H. "Progressive Meshes" in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pp. 209–216, 1997.
[19] Garland M. and Heckbert P. "Surface Simplification Using Quadric Error Metrics" in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pp. 209–216, 1997.
[20] Garland M. and Heckbert P. "Simplifying Surfaces with Color and Texture using Quadric Error Metrics" in *Proceedings of the conference on Visualization '98*, pp. 263–269, 1998.
[21] Popovic J. and Hoppe H. "Progressive Simplicial Complexes" in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pp. 217–224, 1997.
[22] Dehne F., Langis C. and Roth G. "Mesh simplification in parallel" in *Proceedings of the 4th International Conference on Algorithms and Architectures for Parallel Processing*, pp. 281–290, 2000.

Перспективные схемы пространственно-временной индексации для визуального моделирования масштабных индустриальных проектов

Золотов. В. А., Семенов В. А.
ИСП РАН, Москва
{vladislav.zolotov, sem}@ispras.ru

Аннотация. Технологии и системы визуального пространственно-временного моделирования проектов получили распространение в последние годы. Однако невозможность применения к масштабным индустриальным проектам и программам является одним из ключевых факторов, сдерживающим их широкое применение. В работе обсуждаются проблемы эффективности и масштабируемости систем данного класса, а также пути решения на основе индексации проектных данных. Для предложенных схем пространственно-временной индексации и выделенных типов запросов описаны алгоритмы исполнения, а также получены оценки их вычислительной сложности. Полученные оценки подтверждают эффективность и перспективность предложенных схем индексации, а выработанные рекомендации помогают в их практической реализации.

Ключевые слова: 4D моделирование, пространственная индексация, бинарные деревья поиска, октальные деревья, вычислительная сложность

1. Введение

С начала 2000 годов технологии визуального пространственно-временного (4D) моделирования проектов получили значительное распространение и с успехом применяются в различных индустриальных областях [1, 2]. Традиционные методы календарно-сетевого планирования предполагают широкое использование визуальных инструментов для представления проектов в виде сетевых структур, диаграмм Ганта, графиков сбалансированности ресурсов и освоения финансовых средств. Однако они оказались крайне ограниченными для анализа индустриальных программ с учетом пространственного и временного фактора. В отличие от них, технологии 4D моделирования позволяют визуально реконструировать ход выполнения проектных работ, выявить потенциальные проблемы еще на стадии проектирования и планирования проекта и, тем самым, уменьшить технологические риски и финансовые издержки в ходе его реализации [3, 4]. В

частности, визуальное моделирование позволяет определить пространственные коллизии между элементами воздвигаемых конструкций, обнаружить отсутствие опорных элементов, а также выявить конфликтные ситуации, связанные с одновременным проведением нескольких работ в одной и той же пространственной зоне. В более широком смысле 4D моделирование подразумевает не только визуализацию существующего проекта, но и его многофакторный анализ и планирование, благодаря которым достигаются следующие важные цели:

- Улучшенная координация участников проекта за счет единой картины проектной площадки и прозрачной визуальной интерпретации хода проектных работ;
- Высокая достоверность календарно-сетевых графиков, подготовленных и верифицированных на отсутствие пространственно-временных конфликтов;
- Скорректированная логистика проекта с учетом пространственного планирования рабочих зон и оптимального размещения и перемещения ресурсов.

Преимущества 4D технологий оказались настолько очевидными и востребованными для широких индустриальных сообществ [5], что программные компании, специализирующиеся, главным образом, в области создания систем автоматизированного проектирования и систем управления проектами, расширили свои линейки продуктов соответствующими решениями. В первую очередь, к ним следует отнести Autodesk NavisWorks, Bentley Schedule Simulator, Intergraph Schedule Review, AstaPowerproject BIM, VICO, Synchro. Однако, несмотря на популярность перечисленных систем, все они имеют серьезный недостаток, связанный с невозможностью эффективного применения к масштабным индустриальным программам. Их анализ предполагает обработку значительных объемов проектной информации, консолидирующая как структуру планов, детальные календарно-сетевые графики проектных работ, так и точные трехмерные геометрические модели многочисленных элементов возводимых объектов.

Подход, реализуемый в традиционных системах планирования и управления проектами посредством WBS (Work Breakdown Structure) структур, учитывает естественную иерархическую организацию проектных планов и позволяет составлять частичные календарно-сетевые графики. Однако подход не обобщается на случай пространственно-временных данных, с которыми оперируют системы 4D моделирования в ходе трехмерной анимации хода работ, при поиске пространственных конфликтов, анализе загруженности рабочих зон и т.п.

В то же время, хорошо зарекомендовали себя структуры пространственной индексации, которые позволяют существенно снизить вычислительные расходы на исполнение запросов, связанных с анализом больших

многомерных данных [6, 7]. В частности, при анализе и растеризации больших статических трехмерных сцен часто применяют октальные деревья, k-d деревья, BSP-деревья, BRep-индексы, R-деревья. Непосредственное применение данных структур к динамическим сценам крайне неэффективно, поскольку они не учитывают характер пространственно-временной когерентности сцен и требуют значительных накладных расходов для перманентного обновления пространственных индексов.

В данной работе предлагается и исследуется несколько перспективных схем пространственно-временной индексации для приложений визуального моделирования масштабных индустриальных проектов. Они реализуют принцип декомпозиции и обеспечивают эффективный доступ к актуальным проектным данным, релевантным заданному интервалу времени и заданной области пространства. Предложенные схемы индексации анализируются в контексте типовых запросов, возникающих в приложениях визуального моделирования проектов.

Следует отметить, что ранее предпринимались попытки использования декомпозиции для индексации временных и пространственных данных. В частности, в ряде работ решалась задача эффективной растеризации трехмерных данных при генерации видео с использованием дерева Финкельштейна [8], временного дерева пространственной декомпозиции [9] и дерева Шена [10]. В наших работах близкие структуры с успехом применялись для определения столкновений в динамических сценах [11, 12]. Безусловно, подобные структуры представляют несомненный интерес и в контексте обсуждаемых задач визуального моделирования [12, 13, 14].

Раздел 2 представляет собой короткое введение в системы визуального моделирования и планирования проектов. В нем также приводятся характеристики типовых сцен и запросов, возникающих в данных системах. В разделе 3 подробно описывается три альтернативные схемы пространственно-временной индексации сцен, а также указываются их принципиальные отличия от упомянутых выше деревьев. Раздел 4 посвящен выводу оценок вычислительной сложности разрешения типовых запросов, которые систематизируются и обсуждаются в разделе 5. В заключении подводятся итоги проведенного исследования.

2. Постановка задачи

Современные системы визуального моделирования и планирования проектов довольно разнообразны по своим возможностям, однако имеют общие функции, связанные с визуализацией и анализом динамических пространственно-трехмерных сцен. Как правило, сцены, возникающие в данных системах, имеют следующие особенности:

- *Высокая сложность*. Предполагается, что сцены могут состоять из сотен тысяч и миллионов объектов, имеющих собственное

геометрическое представление и индивидуальное динамическое поведение. Допускается, что составные объекты имеют сложную иерархическую организацию;

- *Смешанное геометрическое представление.* Объекты сцены представляются геометрическими каноническими примитивами, неявно заданными параметрическими кривыми и поверхностями, сплайнами, выпуклыми и невыпуклыми многогранниками, сложными формами в граничном и конструктивном твердотельном представлении (BREP, CSG);
- *Псевдо-динамика.* События в сцене строго детерминированы и связаны с появлением, перемещением и удалением объектов в дискретные моменты времени, обычно определяемые началом или окончанием соответствующих проектных работ. Предположение о превалирующем дискретном характере событий существенно, поскольку позволяет исключить из рассмотрения случаи непрерывного перемещения объектов. Данные случаи неизбежно моделируются дискретным образом, но могут приводить к необходимости обработки огромного числа событий, значительно превышающим число проектных работ.

Эффективность визуализации и анализа подобных сцен во многом определяет возможности систем визуального моделирования для планирования масштабных индустриальных проектов и программ. На сегодняшний день эта проблема не решена. Универсальные реляционные или объектно-ориентированные СУБД позволяют организовать хранение больших объемов проектных данных во внешней памяти и обеспечить доступ к ним в ходе решения соответствующих вычислительных задач. Однако даже если все проектные данные размещены в оперативной памяти, возникает проблема эффективного исполнения запросов, которые обычно связаны с выборкой и анализом данных, релевантных заданному временному интервалу и пространственной области. Проведенные вычислительные эксперименты показывают, что без организации пространственно-временной индексации разрешить подобные запросы в разумное время невозможно. Для анализа альтернативных схем индексации мы выделили четыре типовых операций (или запросов):

- развертывание структур пространственно-временной индексации (Q1);
- реконструкция сцены на заданный момент времени (Q2);
- выборка объектов, лежащих в области видимости (Q3);
- анимация выполнения проекта с изменением положения камеры (Q4).

Предполагается, что развертывание структур индексации (Q1) выполняется до начала визуализации и анализа сцены. Развернутые структуры могут многократно использоваться при разрешении серий пространственно-

временных запросов. В ряде случаев исполнение запросов требует обновления соответствующих индексов.

Проиллюстрируем выделенные запросы, следуя принципам функционирования системы Synchro и используя экранный снимок системы, приведенный на рис.1. Как видно, графический интерфейс пользователя совмещает в себе диаграмму Ганта, типичную для систем управления проектами и используемую для визуального представления календарно-сетевых графиков, и окна просмотра трехмерных сцен, характерные для систем компьютерной графики и систем автоматизации проектирования. Временной репер, отображаемый на диаграмме Ганта красной вертикальной линией, соответствует текущему проектному времени. Диаграмма Ганта и окна просмотра сцен синхронизованы между собой в том смысле, что трехмерная модель возводимого сооружения в окнах просмотра соответствует текущему проектному времени. Перемещая временной репер по календарной оси, можно наблюдать за ходом проектных работ и связанными с ними изменениями в трехмерной модели сооружения. Результаты анимации могут быть сохранены как серии изображений или сгенерированный видео файл и использоваться в дальнейшем в качестве иллюстраций к проектной документации. Камеры просмотра в каждом из окон установлены таким образом, чтобы иметь возможность наблюдать за наиболее важными зонами проекта. Иногда в ходе анимации проектных работ требуется плавно перемещать фокус камеры таким образом, чтобы новые активные зоны проектной площадки попадали в поле зрения наблюдателя. Обсуждаемая система представляет для этого необходимые инструменты, позволяющие согласовать перемещения камеры с проектным временем.

Описанные сценарии функционирования системы предполагают исполнение перечисленных выше запросов. Любое перемещение временного репера приводит к необходимости обновления окон просмотра, для чего требуется реконструировать представление сцены на заданный момент времени (Q2). Поскольку состояние сцены меняется в моменты времени, ассоциируемые с началом и окончанием соответствующих работ, процесс реконструкции состоит в инкрементальных обновлениях, порождаемых проектными событиями.

Заметим, что для обновления графического контента окон, вообще говоря, нет необходимости реконструировать и растеризовать всю сцену. Запрос Q3 предполагает выборку только тех объектов, которые попадают в призму видимости камеры. Техника удаления невидимых объектов (frustum culling) широко применяется в приложениях компьютерной графики для оптимизации процессов растеризации. Данный запрос исполняется всякий раз, когда пользователь осуществляет навигацию по сцене, меняя положение камеры, даже если проектное время зафиксировано.

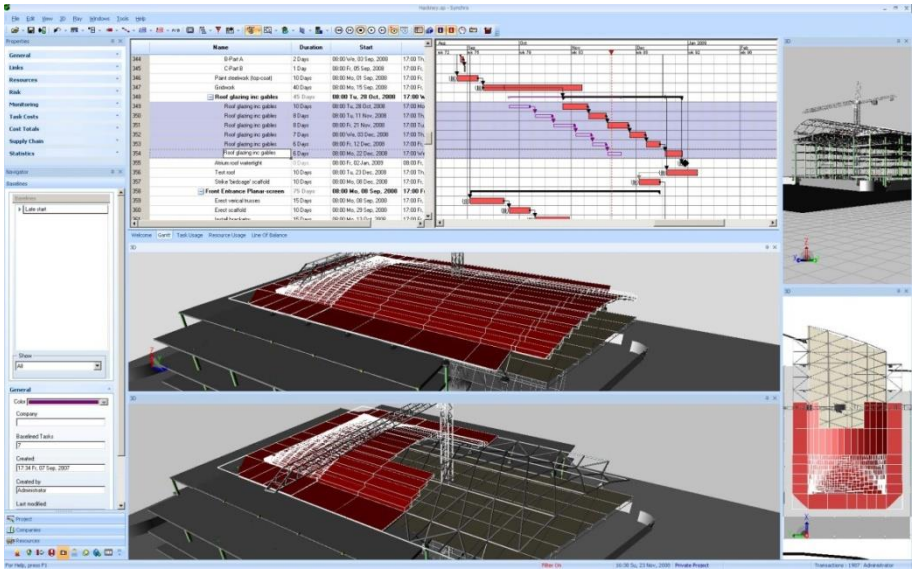


Рисунок 1. Графический интерфейс системы визуального моделирования Synchro

Запрос Q4 соответствует описанному выше сценарию анимации проектных работ при перемещении положения камеры. Данный сценарий предполагает одновременное и согласованное изменение представления сцены и положения камеры вида.

Таким образом, запрос Q2 подразумевает визуализацию проектных данных при фиксированном положении камеры, но в разные моменты проектного времени, запрос Q3 — визуализацию данных с разных ракурсов, но в фиксированный момент времени, а запрос Q4 — визуализацию данных с разных ракурсов в разные моменты времени. Запрос Q1 соответствует предварительной подготовке структур индексации, необходимых для последующего разрешения запросов других типов.

3. Схемы пространственно-временной индексации

Обсудим возможные подходы к организации пространственно-временных индексов для проектных данных. Прежде всего, обсудим базовые структуры данных, которые могли бы использоваться для этих целей.

Для временной индексации событий целесообразным представляется применение сбалансированных бинарных деревьев поиска, которые позволяют эффективно работать с множествами элементов данных даже в тех случаях, когда их ключевые значения изменяются, например, в ходе коррекции временных параметров в календарно-сетевых графиках.

Структуры, подобные AA-дереву, AVL-дереву или красно-черному дереву, позволяют выполнить поиск, вставку и удаление элементов за логарифмическое время. При этом обход всех элементов выполняется за линейное время.

В качестве пространственного индекса для статической трехмерной сцены можно было бы использовать регулярное октальное дерево, формируемое в результате рекурсивного разбиения пространства сцены на восемь одинаковых октантов. Процедура разбиения обычно применяется до тех пор, пока ячейка больше заданного размера, а количество объектов сцены, локализуемых в непустой ячейке, превышает заданный порог. Методы пространственной декомпозиции и, в частности, октальные деревья позволяют значительно сократить трудоемкость выполнения типовых запросов, связанных с выборкой объектов по заданной области, поиском ближайшего соседа или определением столкновений.

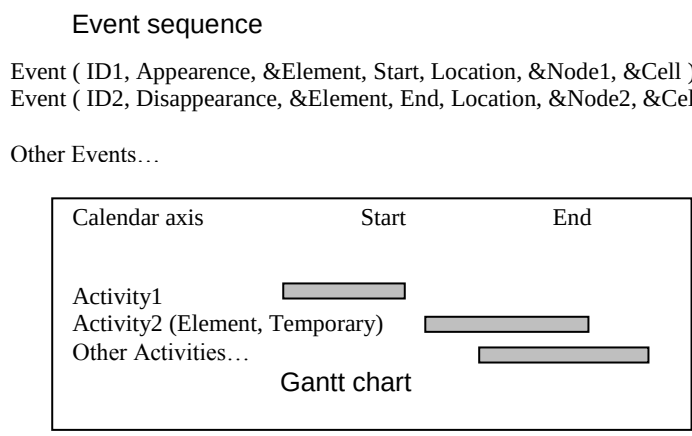


Рисунок 2. Интерпретация событий в приложениях визуального моделирования

Пространственное октальное дерево и временное дерево событий реализуют очевидный принцип: анализу необходимо подвергать лишь те объекты сцены, которые находятся в рассматриваемой области пространства, а обрабатывать лишь те события, которые происходят в интересующем интервале времени. Данный принцип достаточно естественен для обсуждаемых приложений визуального моделирования и возникающих в них сцен. События в них строго детерминированы и обычно связаны с началом или окончанием проектных работ. События визуально интерпретируются в соответствии с назначенным поведенческим шаблоном, как операции появления или удаления соответствующих объектов сцены в заданные моменты времени в заданном месте. Операция перемещения объекта всегда может быть смоделирована

парой операций: удалением объекта с предшествующей позиции и появлением в новой позиции.

Рассмотрим пример календарно-сетевого графика, представленного на рисунке 2, и способ визуальной интерпретации событий в нем. Работа “Activity 2” начинается и заканчивается в точках временной оси “Start” и “End” соответственно. С данной работой ассоциирован трехмерный объект “Element”, которому приписан шаблон “Temporary”. Данный шаблон подразумевает появление объекта в сцене в момент начала соответствующей работы и его удаление в момент завершения работы. Объект занимает положение “Location” на всем протяжении выполнения работы.

Дадим формальное определение пространственного события. Событие — это семерка {ID, TYPE, ELM, TIME, LOC, NODE, CELL}, где ID — идентификатор события, TYPE — тип события (появление, исчезновение или перемещение), ELM — ссылка на трехмерный объект сцены, TIME — время события, LOC — место события (более точно, положение объекта, которое обычно задается в виде матрицы трансформации). Будем также считать, что при наличии развернутого пространственно-временного индекса, структура события может быть дополнена ссылками на элементы временного и пространственного деревьев (NODE и CELL соответственно). До сих пор мы предполагали, что события индексируются с помощью временного дерева, а объекты сцены — с помощью пространственного. Однако пространственное представление динамической сцены непосредственно связано с последовательностью происходящих в ней событий, поэтому обсуждаемые структуры индексации не могут рассматриваться как независимые и необходима организация единого пространственно-временного индекса. Обсудим три альтернативные схемы организации подобного индекса с использованием описанных выше деревьев.

3.1. Схема A. Временно-пространственная декомпозиция

Первая из обсуждаемых схем представляет собой естественное обобщение октальных деревьев на динамический случай. Временное дерево выступает в роли первичного индекса, который развертывается единожды для всех событий, происходящих в детерминированной псевдодинамической сцене. С помощью однонаправленных ассоциаций узлам временного дерева ставятся в соответствие соответствующие события. Такая организация позволяет определять события, происходящие в сцене на заданном временном интервале, достаточно эффективно.

Пространственное октальное дерево строится на любой заданный момент времени и затем обновляется с учетом текущего проектного времени и прошедших за соответствующий период событий. Вообще говоря, пространственное дерево может претерпевать существенные изменения в ходе моделирования масштабного проекта с большим числом объектов и связанных с ними событий. Однако, как правило, обновления носят локальный характер

и могут быть организованы путем инкрементальной коррекции лишь тех ячеек пространственного дерева, в которых обрабатываемое событие происходит.

Октанты пространственного дерева содержат в себе списки трехмерных объектов сцены, локализуемых в них на текущий момент времени. Это позволяет эффективно разрешать ряд запросов. Например, поиск объектов в заданной пространственной области может быть осуществлен путем последовательного анализа ячеек, лежащих в заданной области или пересекающих ее, и сбором всех располагающихся в них объектов. Анализ организуется как обход октального дерева в глубину с отсечением тех октантов и их дочерних поддеревьев, которые заведомо не попадают в интересующую область поиска.

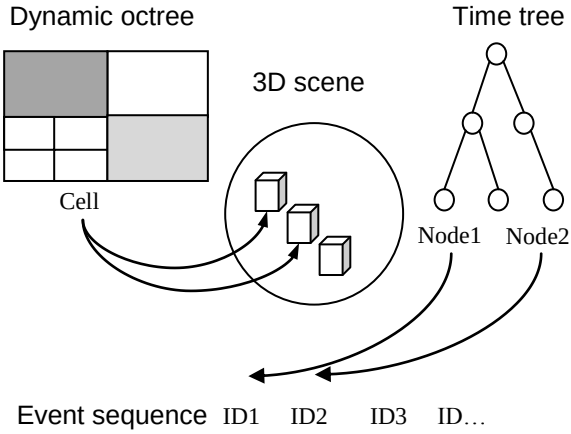


Рисунок 3. Схема временно-пространственной декомпозиции

На рисунке 3 представлена описанная схема индексации проектных данных, а также показаны связи между пространственным октальным деревом, временным деревом, объектами сцены и событиями, происходящими в ней.

В данной работе предполагается, что габариты объектов сопоставимы или меньше размеров листовых октантов и поэтому все объекты сцены локализуются в них. В более общем случае, когда трехмерный объект имеет большие габариты или пересекается гранями октантов верхних уровней, он всегда может быть ассоциирован с одним из родительских октантов или сразу с несколькими листовыми октантами. Для краткости изложения и упрощения проводимого в дальнейшем анализа эффективности схем индексации эти особенные случаи не рассматриваются.

3.2. Схема B. Событийная индексация

Данную схему индексации можно рассматривать в качестве развития предыдущего варианта, поскольку применяются схожие структуры данных и

реализуются близкие принципы индексации. Однако она имеет ряд принципиальных отличий, которые проиллюстрированы на рис. 4.

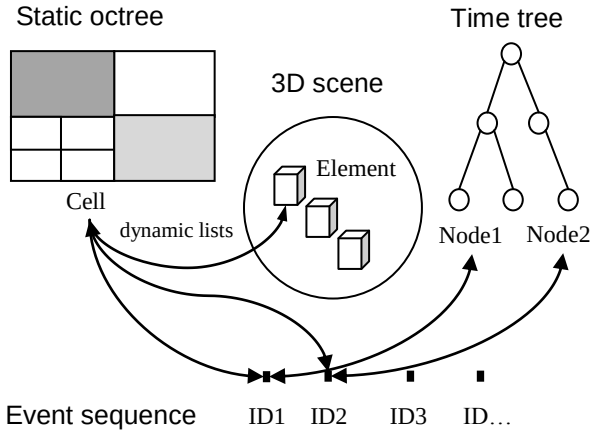


Рисунок 4. Схема событийной индексации

Октанты пространственного дерева хранят списки всех событий, происходящих в них в ходе всего периода моделирования сцены. Такие списки насчитываются предварительно и затем используются для разрешения пространственных запросов. В отличие от предыдущей схемы, в которой октанты хранят списки объектов, локализуемых в них на текущий момент времени и корректируемых по мере наступления новых событий, обсуждаемые структуры индексации полностью статичны и не нуждаются в перманентных обновлениях в процессе моделирования.

Тем не менее, для быстрого разрешения ряда пространственных запросов в сцене, реконструируемой на заданный момент времени, иногда желательно поддерживать в октантах и динамические списки локализуемых в них объектов. Данный вариант индексации в определенном смысле дополняет предыдущую схему, однако возвращает нас к необходимости обновлений.

В подобном варианте схема обладает большей гибкостью, поскольку допускает несогласованные обновления динамических списков объектов. Состояния списков в разных октантах могут соответствовать разным моментам времени и, следовательно, не отражать реальное представление динамической сцены на текущий момент времени. Например, в ходе навигации обычно визуализируются относительно небольшие фрагменты сцены и, вообще говоря, не возникает необходимости в полной ее реконструкции. Вместо этого, реконструкции подлежат лишь те фрагменты сцены, которые принадлежат видимым октантам. В тем случаях, когда в область видимости попадает вся сцена, то обновляются все списки объектов.

Для поддержки данной схемы в каждом октанте необходимо дополнительно хранить также проектное время, которому соответствуют насчитанные списки объектов.

3.3. Схема C. Пространственно-временная декомпозиция

Наконец, обсудим схему пространственно-временной декомпозиции. В каждом октанте пространственного дерева развертывается и хранится временное дерево событий, происходящих в нем в ходе всего периода моделирования сцены. Пространственное дерево в данной схеме используется в качестве первичного индекса, а частичные временные деревья — в качестве вторичного индекса (смотри рис. 5). Подобная организация хорошо подходит для разрешения сложных запросов, адресуемых как к области пространства, так и временному интервалу. Вся структура индексации развертывается один раз перед началом моделирования и может использоваться без каких-либо обновлений. Однако, как и в предыдущем случае, мы предпочитаем дополнить ее динамическими списками объектов для более эффективного разрешения пространственных запросов. Пожалуй, единственным недостатком данной схемы является проблематичность разрешения временных запросов, например, связанных с определением событий, происходящих в заданном интервале времени. Для этого пришлось бы организовать полный обход пространственного дерева и выполнить поиск событий в каждом листовом октанте, используя частичные временные деревья.

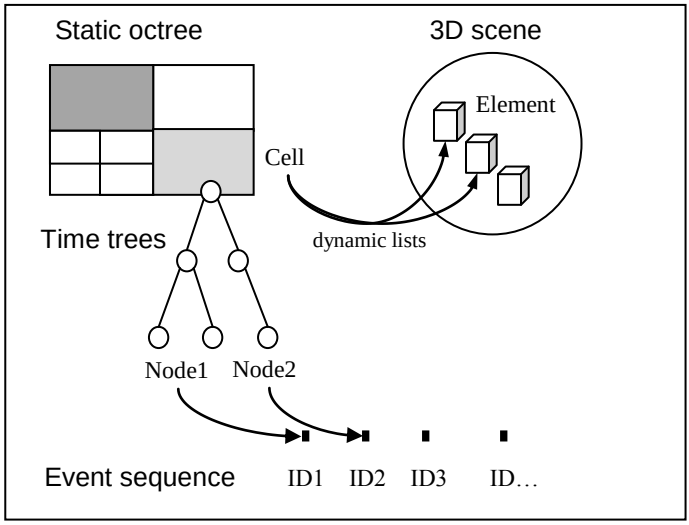


Рисунок 5. Схема пространственно-временной декомпозиции

По построению, описанная структура индексации очень похожа на дерево Шена [10], используемое в приложениях растеризации сцен. Однако

ключевым отличием и важным преимуществом описанного варианта является прямое ассоциирование узлов временных деревьев с событиями, что позволяет избежать избыточное хранение объектов сцены, свойственное структуре Шена.

4. Анализ эффективности

Для оценки эффективности предложенных схем пространственно-временной индексации сделаем ряд предположений относительно характеристик сцен, подлежащих визуальному моделированию и анализу, а именно:

— сцена представляет собой куб со стороной x;
— все объекты являются кубами одинакового размера Δx;
— события в сцене происходят последовательно, причем каждое событие инициирует появление одного нового объекта в положении, зафиксированном на протяжении всего периода моделирования сцены;
— перед началом моделирования динамическая сцена пуста, а в конце моделирования содержит N объектов, равномерно распределенных по объему сцены.

Линейным фактором сцены назовем величину $\delta = \frac{\Delta x}{x}$, выражающую относительные линейные габариты объекта, а объемным фактором — величину $\nu = \left(\frac{\Delta x}{x}\right)^3$, выражающую относительный объем индивидуального объекта. Введем также параметры α, β, γ, характеризующие класс рассматриваемых сцен и способы навигации в них. Параметр $\alpha > 0$ определяет пространственную наполненность сцены на момент окончания моделирования $\alpha = \nu N$. Объем видимой области относительно объемы всей сцены определим как $\beta \in [0,1]$. Отношение длины анализируемого временного интервала ко всему периоду моделирования сцены обозначим как $\gamma \in [0,1]$.

Получим оценки сложности исполнения типовых запросов в указанных выше предположениях относительно характеристик сцен. Оценки вычислительной сложности будут получены в среднем. Итак, пусть C_{Time} — стоимость локализации события во временном дереве, C_{Box} — стоимость локализации объекта в октальном дереве, C_{Cone} —стоимость проверки факта пересечения конуса и куба. Для простоты предположим, что расходы на организацию структур данных в памяти и навигации по ним пренебрежимо малы. Также не

будем учитывать расходы на формирование результатов запросов и растеризацию моделируемых сцен.

4.1. Развертывание пространственно-временного индекса (Q1)

Прежде всего, оценим вычислительную стоимость развертывания и поддержки структур данных для различных схем пространственно-временной индексации. Анализ проведем отдельно для пространственного октального дерева и временного дерева. Поскольку в начальном состоянии сцена пуста, развертывание пространственного дерева в схеме временно-пространственной декомпозиции не требует вычислительных затрат.

Согласно сделанным выше предположениям объекты сцены локализуются в листьях октального дерева на глубине $L = \log(1/\delta)$, если $\alpha > 1$ и $L(N) = \frac{1}{3}\log(N)$ в противном случае. Оценим среднее количество листовых октантов, которым может быть приписан один и тот же объект. Вероятность одновременного пересечения объекта тремя секущими плоскостями на уровне $l \in [1, L]$ равна δ_l^3, где $\delta_l = 2^l \delta$ — эффективный относительный размер объекта. В этом случае объект приписывается сразу восьми дочерним октантам уровня $l+1$. Вероятность пересечения объекта любыми двумя секущими плоскостями равна $3\delta_l^2(1-\delta_l)$. В этом случае объект приписывается четырем октантам следующего уровня. Вероятность пересечения объекта одной из секущих плоскостей равна $3\delta_l(1-\delta_l)^2$. В этом случае объект локализуется в двух октантах. И, наконец, вероятность того, что ни одна из секущих плоскостей не пересекает объект равна $(1-\delta_l)^3$. Тогда в среднем каждый объект ассоциируется с R_l октантами уровня l, число которых выражается следующим образом:

$$R_l = 8\delta_l^3 + 12\delta_l^2(1-\delta_l) + 6\delta_l(1-\delta_l^2) + (1-\delta_l)^3$$

Тогда стоимость локализации объекта выражается как $C_{Box}R(L)$, где $R(L) = \sum_{l=1}^{L} R_l$, а стоимость развертывания всего октального дерева определяется как $C_{Box}R(L)N$, поскольку сцена содержит ровно N объектов. Можно показать, что функция $R(L)$ может быть представлена как

$$R(L) = \frac{1}{98}(5\delta^3 2^{3L+4} - 35\delta^3 2^{3L+4} L - 80\delta^3 - 49\delta^2 2^{2L+3} + 147\delta^2 2^{2L+3} L$$
$$- 392\delta^2 - 147\delta 2^{L+2} + 147\delta 2^{L+2} L + 588\delta + 49L^2 + 49L)$$

и для нее имеет место оценка $R(L) < 4(L+1)L$.

Во всех рассматриваемых схемах временные деревья — статические структуры, формируемые один раз. Для схем B и C стоимость развертывания временного дерева равна $C_{Time} \log(N!)$, где число объектов N совпадает с числом событий в сцене. В случае схемы пространственно-временной декомпозиции частичные временные деревья формируются в каждом 8^L октантов пространственного дерева. Поскольку в каждом октанте происходит в среднем $\frac{N}{8^L}$ событий, общая стоимость развертывания всех временных деревьев равна $C_{Time} 8^L \log((\frac{N}{8^L})!)$.

4.2. Реконструкция сцены на заданный момент времени (Q2)

Запрос реконструкции сцены на заданный момент времени выполняется в предположении, что состояние сцены известно для предыдущего момента времени и требуется обновить сцену на текущий момент времени. Для этого требуется установить все события, прошедшие за данный временной интервал, и соответствующим образом скорректировать представление сцены.

Для схемы событийной индексации запрос можно исполнить двумя способами. Первый состоит в том, чтобы вначале определить все прошедшие события, используя временное дерево. Это может быть выполнено за $C_{Time} \gamma N$, где фактор γ определяет интенсивность событий на временном интервале таким образом, что значение $\gamma = 0$ соответствует случаю отсутствия событий, а значение $\gamma = 1$ соответствует случаю реконструкции событий всего периода моделирования. При таком способе требуется дополнительный анализ принадлежности объектов области видимости.

Второй способ подразумевает использование пространственного дерева в качестве первичной структуры поиска. С этой целью осуществляется навигация по видимым октантам и в каждом из них проводится поиск событий, лежащих в интересующем временном интервале. Затраты могут быть оценены как $C_{Time} \beta R(L) N$, где множитель $R(L)$ отражает тот факт, что объект может быть ассоциирован сразу с несколькими октантами уровня L.

Таким образом, при оптимальной стратегии трудоемкость реконструкции сцены на основе схемы событийной индексации составляет

$$C2_B = \min((C_{Time} + C_{Cone})\gamma N, C_{Time}\beta R(L)N)$$

При использовании схемы пространственно-временной декомпозиции обход октального дерева должен быть дополнен быстрым поиском событий с применением развернутых частичных временных деревьев. Принимая во внимание равномерное распределение объектов по сцене, число событий на заданном временном интервале, попадающих в область видимости, определяется величиной $\beta\gamma R(L)N$, а затраты на реконструкцию сцены могут быть оценены как

$$C2_C = C_{Time}\beta\gamma R(L)N$$

Наконец, оценим затраты на выполнение запроса при помощи схемы временно-пространственной декомпозиции. Поиск событий на интересующем временном интервале занимает $C_{Time}\gamma N$. Чтобы обновить динамические списки объектов в октантах пространственного дерева требуется локализовать и вставить γN объектов. Если глубина октального дерева на текущий момент времени составляла $1 \le L' \le L$, то затраты на обновление пространственного дерева составляют $C_{Box}R(L')N$. Следовательно, трудоемкость реконструкции сцены при помощи схемы A можно оценить следующим образом:

$$C2_A = C_{Time}\gamma N + C_{Box}R(L')\gamma N$$

4.3. Поиск объектов в области видимости (Q3)

Данный запрос может быть разрешен одинаково для всех рассматриваемых схем индексации. Для этого необходимо выполнить обход видимых октантов пространственного дерева и установить видимые объекты и связанные с ними события. Для схем B и C это может быть выполнено за $C_{Cone}\beta\sum_{l=1}^{L} 8^l = C_{Cone}\beta\frac{8(8^L-1)}{7}$, однако обновление динамических списков объектов в октантах может требовать значительных дополнительных расходов, особенно в тех случаях, когда значительная часть сцены попадает в область видимости. Оценки затрат на обновление динамических списков получены в предыдущем разделе и составляют

$$C2_B = \min((C_{Time} + C_{Cone})\gamma N, C_{Time}\beta R(L)N)$$

$$C2_C = C_{Time}\beta\gamma R(L)N$$

Следовательно, трудоемкость разрешения запроса поиска видимых объектов можно оценить как

$$C3_B = \min((C_{Time} + C_{Cone})\gamma N, C_{Time}\beta R(L)N) + 8C_{Cone}\frac{\beta(8^L - 1)}{7}$$

$$C3_C = C_{Time}\beta\gamma R(L)N + 8C_{Cone}\frac{\beta(8^L - 1)}{7}$$

В случае схемы временно-пространственной декомпозиции оценка трудозатрат должна учитывать глубину дерева на текущий момент времени $1 \le L' \le L$. Поскольку данная схема предусматривает поддержку визуального представления на текущий момент времени, затраты оцениваются как

$$C3_A = 8C_{Cone}\frac{\beta(8^{L'} - 1)}{7}$$

4.4. Анимационный запрос (Q4)

Анимационный запрос предполагает одновременное изменение положения камеры и проектного времени. Поэтому его можно рассматривать как композицию запросов Q2 и Q3. В случае схемы A трудоемкость выполнения запроса оценивается как сумма затрат на разрешение вспомогательных запросов:

$$C4_A = C2_A + C3_A$$

Для схем B и C трудоемкость разрешения анимационного запроса совпадает с затратами на выборку объектов в области видимости. Действительно, динамические списки объектов в видимых октантах обновляются уже в ходе разрешения запроса на выборку видимых объектов. Таким образом, имеет место

$$C4_B = C3_B$$

$$C4_C = C3_C$$

5. Сравнительный анализ

Сравним три представленные схемы индексации с учетом требований, предъявляемых к производительности разрешения типовых запросов. Сравнительный анализ вычислительных затрат позволяет сформулировать рекомендации по эффективному применению схем индексации в приложениях визуального моделирования, работающих с масштабными индустриальными проектами. Оценки сложности, полученные для предложенных схем, отражены в таблице 1.

Прежде всего, необходимо отметить, что стоимость развертывания индексов значительно варьируется для различных схем. Асимптотические оценки

сложности построения временного дерева совпадают для всех схем и составляют $O(N \log N)$. Оценки сложности развертывания октального дерева в схемах событийной индексации и пространственно-временной декомпозиции составляют $O(N \log^2 N)$. В схеме временно-пространственной декомпозиции октальное дерево не содержит элементов перед началом моделирования и, следовательно, его построение не требует каких-либо затрат.

Реконструкция сцены на заданный момент времени может быть выполнена за $O(N \log^2 N)$. Однако, полученные асимптотические оценки различаются постоянным множителем. Наименее трудоемкой для разрешения этого типа запросов оказывается схема C.

QQ1	$C1_A = C_{Time} \log(N!)$ $C1_B = C_{Time} \log(N!) + C_{Box} R(L) N$ $C1_C = C_{Time} 8^L \log((\frac{N}{8^L})!) + C_{Box} R(L) N$
Q2	$C2_A = C_{Time} \gamma N + C_{Box} R(L') \gamma N$ $C2_B = \min((C_{Time} + C_{Cone}) \gamma N, C_{Time} \beta R(L) N)$ $C2_C = C_{Time} \beta \gamma R(L) N$
Q3	$C3_A = C_{Cone} \frac{8\beta(8^{L'} - 1)}{7}$ $C3_B = C_{Cone} \frac{8\beta(8^{L} - 1)}{7} + \min((C_{Time} + C_{Cone}) \gamma N, C_{Time} \beta R(L) N)$ $C3_C = C_{Cone} \frac{8\beta(8^{L} - 1)}{7} + C_{Time} \beta \gamma R(L) N$
Q4	$C4_A = C2_A + C3_A$ $C4_B = C3_B$ $C4_C = C3_C$

Таблица 1. Вычислительная трудоемкость разрешения запросов с применением различных схем пространственно-временной индексации

Схема A оказывается наиболее подходящей для поиска объектов в области видимости, поскольку октальное дерево всегда актуализировано на текущее проектное время и соответствует визуальному представлению сцены, подлежащему анализу. Схема C выглядит более предпочтительно, чем схема B, поскольку коррекция динамических списков объектов выполняется несколько эффективней за счет того, что события уже приписаны соответствующим октантам.

Схемы пространственно-временной декомпозиции и событийной индексации наиболее хорошо подходят для разрешения анимационных запросов, связанных с одновременным изменением камеры и проектного времени. По сложности разрешения анимационный запрос эквивалентен поиску объектов в заданной области, поскольку анализу и обработке подлежат те же самые видимые октанты. При использовании схемы временно-пространственной декомпозиции с необходимостью обновляется все октальное дерево. Суммируя вышесказанное, каждая из представленных схем обладает собственными достоинствами и недостатками.

В качестве рекомендаций для выбора схемы индексации, релевантной особенностям конкретного приложения, можно предложить следующие:

— в приложениях, реализующих развитые средства навигации и методы пространственно-временной верификации проектов, целесообразно применение схемы A;
— для приложений, ориентированных на событийное моделирование и предполагающих быструю пространственно-временную реконструкцию и анализ цепочек событий, предпочтительно использовать схему B;
— приложения с превалирующей анимационной функциональностью наиболее эффективно реализуются с применением схемы C.

Следует отметить, что обсуждаемые приложения консолидируют в себе функции визуального моделирования и календарно-сетевого планирования проектов и предполагают эффективное исполнение всех упомянутых выше типов запросов. Схема временно-пространственной декомпозиции представляется наиболее перспективной в этом случае, поскольку обеспечивает сбалансированные затраты на их разрешение.

6. Заключение

Таким образом, в работе обсуждены проблемы эффективности и масштабируемости, возникающие при построении систем визуального моделирования масштабных индустриальных проектов. Возможный подход к их решению состоит в применении пространственно-временной индексации проектных данных. В работе предложены и исследованы три альтернативные схемы, учитывающие особенности пространственно-временной когерентности сцен и обеспечивающие быстрое разрешение типовых запросов, связанных с развертыванием и обновлением индексов, реконструкцией сцен, поиском

видимых объектов и анимацией динамических сцен. Для предложенных схем и выделенных типов запросов описаны алгоритмы исполнения запросов и получены оценки их вычислительной сложности. Полученные оценки подтверждают эффективность и перспективность предложенных схем индексации, а выработанные рекомендации помогают в их практической реализации.

Список литературы

[1] BIM, GSA, "GSA Building Information Modeling Guide Series 04 - 4D Phasing" US General Services Administration, Public Building Service, Technical Report 2009.

[2] Heesom, D., Mahdjoubi, L. Trends of 4D CAD applications for construction planning.// Construction Management and Economics, 22, 2004. — с. 171-182.

[3] Sriprasert, E., Dawood, N. Requirements identification for 4D constraint-based construction planning and control system.// Conference Proceedings - Distributing Knowledge in Building — University of Teeside, Middlesbrough, 2002.

[4] Visualization for decision making in construction planning. Visualization and intelligent design in egineering and architecture. Retik, A.; J.J. Connor et al. — New York, Elsevier Science, 1993. — с. 587-599.

[5] Seliga, C. "Revel Selects Synchro for $2 Billion Atlantic City Casino Project." *Prime Newswire June*, 2007.

[6] Jimenez, P., Thomas, F., Torras, C. 3D collision detection: a survey.// Computers and Graphics, 25, 2001. — с. 269-285.

[7] Klosowski, J.T., Held, M., Mitchel, J. S. B., Sowizral, H., Zikan, K. Efficient collision detection using bounding volume hierarchies of k-DOPs.// Visualization and Computer Graphics, IEEE Transactions on Volume 4, Issue I, 1998. — с. 21-36.

[8] Finkelstein, A., Jacobs, C. E., Salesin, D. H. Multiresolution video.// Proceedings of ACM SIGGRAPH, 1996. — с. 281-290.

[9] Zhiyan, D.Y.J., Chiang, H.W.S. Out-of-core volume rendering for time-varying fields using a space-partitioning (SPT) tree.// Visualizatin Symposium, Pacific Vis '09, 2009. — с. 73-80.

[10] Shen, H.W., Chiang, L.J., Ma, K.L. A fast volume rendering algorithm for time-varying field using a time-space partitioning (TSP) tree.// In Proceedings of IEEE Visualization, 1999. — с. 371-377.

[11] Semenov, V.A., Kazakov, K.A., Zolotov, V.A., Jones, H., Jones, S. Combined strategy for efficient collision detection in 4D planning applications.// Computing in Civil and Building Engineering — Nottingham, UK, June 2010. — с. 31-39.

[12] Золотов, В.А., Семенов, В.А. Исследование и развитие метода декомпозиции для анализа больших пространственных данных.// Труды Института Системного Программирования., 25, 2013. — с. 121-166.

[13] Shagam, J., Pfeiffer, J. "Dynamic Irregular Octrees" NMSU-CS-2003-004, Technical Report 2003.

[14] Sudarsky, O., Gotsman, C. Dynamic Scene Occlusion Culling.// IEEE Transactions on Visualization and Computer Graphics, 5, 1, 1999. — с. 13-29.

[15] Whang, J.W., Song, J.W., Chang, J.Y., Kim, J.Y., Cho, W.S., Park, C.M., Song, I.Y. Octree-R: An adaptive octree for efficient ray tracing.// IEEE Transactions on Visualization and Computer Graphics, 1, 4, 1995. — с. 343-349.

Effective spatio-temporal indexing methods for visual modeling of large industrial projects

V.A. Zolotov, V.A. Semenov
ISP RAS, Moscow, Russia)
{vladislav.zolotov, sem}@ispras.ru

Abstract. As opposed to traditional project planning and scheduling methods, 4D modeling technologies allow performing more comprehensive analysis taking into account both spatial and temporal factors. Such analysis enables to identify critical issues in early design and planning stages and to save significantly on the total project implementation costs. Unsurprisingly that there is a large interest to these emerging technologies and both free and commercial systems have been recently developed and applied in different industrial domains. However, being applied to large industrial projects and programs, most systems face significant problems relating to performance degradation and limited interactive capabilities what ultimately neglects the key benefits of visual 4D modeling technologies. In this paper a promising approach to rising up the performance and scalability of 4D modeling applications by means of indexing project data has been discussed. Three alternative indexing schemes have been proposed and described in conformity to the features of spatio-temporal coherence of arising dynamic scenes. These schemes have been investigated against the efficiency criteria to resolve typical requests connected with deployment and updates of indexes, reconstruction of scenes, analysis of visible objects, and animation of dynamic scenes. Conducted computational complexity analysis and obtained asymptotic estimates confirm the efficiency of the proposed indexing schemes and cause their introduction in industrial practice. Recommendations on their practical use have been also provided.

Keywords: 4D modeling, spatial indexing, binary search trees, octrees, computational complexity

References

[1] BIM, GSA, "GSA Building Information Modeling Guide Series 04 - 4D Phasing" *US General Services Administration, Public Building Service*, Technical Report 2009.
[2] Heesom, D., Mahdjoubi, L. Trends of 4D CAD applications for construction planning. *Construction Management and Economics*, 22, 2004, pp. 171-182.
[3] Sriprasert, E., Dawood, N. Requirements identification for 4D constraint-based construction planning and control system. *Conference Proceedings - Distributing Knowledge in Building*, University of Teeside, Middlesbrough, 2002.
[4] *Visualization for decision making in construction planning. Visualization and intelligent design in egineering and architecture.* / Retik, A.; J.J. Connor. New York, Elsevier Science, 1993, pp. 587-599.
[5] Seliga, C. "Revel Selects Synchro for $2 Billion Atlantic City Casino Project." *Prime Newswire June*, 2007.
[6] Jimenez, P., Thomas, F., Torras, C. 3D collision detection: a survey. *Computers and Graphics*, 25, 2001, pp. 269-285.

[7] Klosowski, J.T., Held, M., Mitchel, J. S. B., Sowizral, H., Zikan, K. Efficient collision detection using bounding volume hierarchies of k-DOPs. *Visualization and Computer Graphics, IEEE Transactions on Volume 4, Issue I*, 1998, pp. 21-36.
[8] Finkelstein, A., Jacobs, C.E., Salesin, D.H. Multiresolution video. *Proceedings of ACM SIGGRAPH*, 1996, pp. 281-290.
[9] Zhiyan, D. Y. J., Chiang, H. W. S. Out-of-core volume rendering for time-varying fields using a space-partitioning (SPT) tree. *Visualizatin Symposium, Pacific Vis '09*, 2009, pp. 73-80.
[10] Shen, H.W., Chiang, L.J., Ma, K.L. A fast volume rendering algorithm for time-varying field using a time-space partitioning (TSP) tree. *In Proceedings of IEEE Visualization*, 1999, pp. 371-377.
[11] Semenov, V.A., Kazakov, K.A., Zolotov, V.A., Jones, H., Jones, S. Combined strategy for efficient collision detection in 4D planning applications. *Computing in Civil and Building Engineering.* Nottingham, UK, June 2010, pp. 31-39.
[12] Zolotov, V.A., Semenov, V.A. Issledovanie i razvitie metoda dekompozitsii dlya analiza bol'shikh prostranstvennykh dannykh [On application of spatial decomposition method for large data sets indexing]. *Trudy ISP RAN* [*The Proceedings of ISP RAS*], 2013, vol 25, pp. 121-166 (in Russian).
[13] Shagam, J., Pfeiffer, J. "Dynamic Irregular Octrees" *NMSU-CS-2003-004*, Technical Report 2003.
[14] Sudarsky, O., Gotsman, C. Dynamic Scene Occlusion Culling. *IEEE Transactions on Visualization and Computer Graphics*, 5, 1, 1999, pp. 13-29.
[15] Whang, J.W., Song, J.W., Chang, J.Y., Kim, J.Y., Cho, W.S., Park, C.M., Song, I.Y. Octree-R: An adaptive octree for efficient ray tracing. *IEEE Transactions on Visualization and Computer Graphics*, 1, 4, 1995, pp. 343-349.

Комбинированный метод верификации масштабных моделей данных

В.А. Семенов, С.В. Морозов, Д.В. Ильин
ИСП РАН, Москва, Россия
sem@ispras.ru, serg@ispras.ru, denis.ilyin@ispras.ru

Аннотация. Статья адресована актуальной проблеме верификации масштабных моделей данных, применяемых в различных индустриальных областях и специфицируемых на популярных универсальных объектно-ориентированных языках, таких как EXPRESS, UML/OCL. Основные преимущества языков информационного моделирования (высокая выразительность, декларативность, богатство синтаксических конструкций) отрицательно сказываются в процессе автоматической верификации спецификаций. Существующие методы верификации моделей вследствие высокой вычислительной сложности не могут использоваться для решения данной проблемы. В статье предлагается комбинированный метод верификации объектно-ориентированных моделей данных, основанный на последовательной редукции к нескольким постановкам математических задач: линейного программирования, удовлетворения ограничений (CSP), выполнимости булевых формул (SAT). Детально исследуется ключевая проблема определения необходимого количества экземпляров для генерации корректной коллекции объектов и ее редукция к задаче целочисленного линейного программирования. Работа поддержана РФФИ (грант 13-07-00390).

Ключевые слова: верификация моделей, объектно-ориентированное моделирование, UML/OCL, EXPRESS, логическое программирование в ограничениях, линейное целочисленное программирование, семантическая реконсиляция

1. Введение

В настоящее время значительную роль в процессе разработки сложных программных систем играет моделирование. Применение модельно-ориентированного подхода (MDA) [1] позволяет решить многие проблемы программной инженерии, в частности, проблемы мобильности, интероперабельности, масштабируемости программного обеспечения, и, в конечном итоге, повысить качество и сократить ресурсы, затрачиваемые на его разработку и тестирование. В рамках данного подхода модели программной системы являются базовыми спецификациями для всех последующих этапов разработки. В связи с этим особую важность при разработке на основе моделей (Model Driven Development) приобретает корректность используемых спецификаций. Выявление ошибок

проектирования на первоначальной стадии разработки позволяет сократить ее время, уменьшить риски и, как результат, снизить ее стоимость.

В работе [2] выделяются две различных постановки задачи проверки корректности концептуальных схем: внутренняя и внешняя проверки. Фундаментальным свойством корректности в контексте внутренней проверки является выполнимость (satisfiability) моделей, а именно, жизнеспособность (liveliness) их классов и ассоциаций, исполняемость (executability) операций, безызбыточность (nonredundancy) семантических ограничений. Задача в данной постановке может быть автоматически решена методами формальной верификации. С точки зрения внешней проверки наиболее важным фактором является применимость (usability) моделей к конкретным требованиям пользователя. Задача в данной постановке сводится к валидации полноты покрытия семантики заданной предметной области и не может быть решена только формальными методами.

Программные средства верификации и валидации для автоматической или автоматизированной проверки корректности концептуальных схем существуют [11, 17, 26, 28, 31, 34], но их высокая вычислительная сложность является значительным недостатком. Основные преимущества языков информационного моделирования (высокая выразительность, декларативность, богатство синтаксических конструкций) отрицательно сказываются в процессе автоматической верификации спецификаций. В связи с постоянно возрастающей сложностью используемых в программной инженерии моделей возникает необходимость разработки и применения эффективных методов для их верификации.

Настоящая работа адресована проблеме верификации масштабных моделей данных, применяемых в различных индустриальных областях и специфицируемых на популярных универсальных объектно-ориентированных языках (EXPRESS [3], UML/OCL [4, 5] и т.п.). Примерами таких моделей служат IFC (архитектура и строительство) [6], CIS/2 (строительство с использованием стальных конструкций) [7], STEP (машиностроение) [8], ISO 15926 (нефтегазодобывающая промышленность) [9]. Данная проблема является актуальной, поскольку масштабные индустриальные модели разрабатываются крупными консорциумами участников в течение продолжительного времени (см. пример в табл. 1) и могут содержать трудно обнаруживаемые визуально ошибки, такие как избыточные либо переопределенные ограничения. Подобные ошибки являются одной из причин периодического пересмотра утвержденных информационных стандартов и принятия поправок к ним.

	IFC 1.5.1	IFC 2.0	IFC 2x	IFC 2x2	IFC 2x3	IFC 4
Год принятия	1997	1999	2000	2003	2006	2013
Объектные типы	186	290	370	623	653	766
Типы, определяемые пользователем	95	157	228	312	327	391
Вычисляемые атрибуты	44	45	41	55	96	105
Процедуры и функции	25	27	25	37	38	42
Ограничения в объектных типах	107	168	196	271	340	638
Ограничения в определяемых пользователем типах	12	12	13	16	24	24
Ограничения уникальности	1	3	14	14	17	4
Глобальные ограничения	0	0	3	3	2	2

Табл. 1. Эволюция модели IFC

В разделе 2 проводится аналитический обзор существующих методов верификации моделей. В разделе 3 предлагается комбинированный метод верификации объектно-ориентированных моделей данных, основанный на последовательной редукции к нескольким постановкам математических задач: линейного программирования, удовлетворения ограничений (CSP), выполнимости булевых формул (SAT). В разделе 4 детально исследуется ключевая проблема определения необходимого количества экземпляров для генерации корректной коллекции объектов и ее редукция к задаче целочисленного линейного программирования. В разделе 5 демонстрируется пример применения предложенного комбинированного метода. В заключении приводятся выводы о перспективности применения метода для верификации масштабных индустриальных моделей данных и намечаются направления его дальнейшего исследования и развития.

2. Аналитический обзор существующих методов верификации моделей

Существуют многочисленные решения для верификации моделей, базирующиеся как на UML/OCL, так и на других языковых средствах (диаграммы «сущность-связь» (ER) [10], язык Alloy [11], языки объектно-ориентированных баз данных [12], дескрипционные логики [13]). Лежащие в их основе подходы основываются на редукции исходной задачи верификации модели к той или иной математической постановке и ее дальнейшему решению известными методами. Как правило, большинство существующих средств проверяют выполнимость исходной модели, т.е. возможность создания некоторого набора экземпляров ее классов, не нарушающего определенных в ней ограничений. В ряде из них осуществляются дополнительные проверки, например, локализация избыточных или зависимых ограничений. В данном разделе представлены краткие описания нескольких известных средств верификации, сгруппированные по используемым математическим методам. В последнем подразделе проводится сравнение данных решений.

2.1. Методы, основанные на системах линейных неравенств

Метод, основанный на трансформации концептуальных схем в системы линейных неравенств, был впервые предложен Ленцерини и Нобили [14] для верификации ER-диаграмм, которые включают сущности (классы), бинарные связи (ассоциации) и ограничения кардинальности. Система формируется по набору связей и ограничений кардинальности заданной ER-модели за полиномиальное время относительно размера исходной схемы. В работе было введено определение сильной выполнимости модели, которое гарантирует проверку корректности ограничений кардинальности и подразумевает возможность создания набора экземпляров, включающего как минимум один экземпляр каждой сущности и не нарушающего ни одного из определенных ограничений. Теоретически доказано, что ER-модель является сильно выполнимой тогда и только тогда, когда существует решение соответствующей системы неравенств. Система может быть решена методами линейного программирования. В работе [14] был также представлен алгоритм, позволяющий локализовать подмножество ограничений, нарушающих выполнимость ER-модели. Он основан на преобразовании исходной ER-диаграммы в ориентированный мультиграф и поиске в нем критических циклов.

Мараее и Балабан [15] развили оригинальный метод Ленцерини–Нобили для верификации диаграммы классов UML, включающей не только бинарные ассоциации с ограничениями кардинальности, но и отношения обобщения/специализации с соответствующими ограничениями ”complete”, “incomplete”, “disjoint”, “overlapping”. Алгоритм формирования системы неравенств также имеет полиномиальную сложность, но обладает рядом

ограничений. В частности, он неприменим к UML диаграммам с циклическими иерархиями классов при наличии ограничений обобщения/специализации. Прочие конструкции языка UML (n-арные ассоциации, агрегации, композиции, сложные ограничения кардинальности в виде комбинации интервалов, ограничения OCL) также не обрабатываются.

2.2. Методы, основанные на трансформации в реляционные логики

Наиболее известным средством верификации данного класса является язык Alloy [16]. Он использует логику предикатов первого порядка для описания моделей структурированных данных вместе с семантическими ограничениями и рассматривается как простое приложение, которое позволяет проектировать модели и проверять их корректность с помощью анализатора Alloy. Проектировщик описывает модель на языке Alloy, который имеет текстовую нотацию, затем анализатор переводит конструкции языка в логическую формулу в конъюнктивной нормальной форме и разрешает её с помощью решателя SAT (другими словами, находит такое состояние модели, при котором логическая формула возвращает истинное значение). Синтаксис языка Alloy совместим с UML, но оценка покрытия UML текстовой нотацией Alloy весьма затруднительна. Первая версия Alloy имела ограниченные объектно-ориентированные свойства. В более поздних версиях появилась поддержка наследования, полиморфизма, мульти-арных отношений, кванторов.

UML2Alloy [17] играет важную роль в создании моста между языками UML/OCL и Alloy. UML2Alloy — приложение для верификации диаграмм классов UML с помощью анализатора Alloy в контексте технологии MDD. Кроме диаграмм классов UML поддерживаются также инварианты OCL. Трансформация модели UML/OCL в конструкции языка Alloy осуществляется автоматически, в качестве исходных данных принимается файл в формате XMI, сгенерированный с помощью ArgoUML [18].

Известны многочисленные альтернативные решения для автоматической верификации UML моделей, основанные на трансформации диаграмм UML с OCL ограничениями в логики первого порядка и дальнейшей редукции к задаче SAT. Среди них особо отметим решение, предложенное университетом Бремена [19]. В отличие от UML2Alloy в нем поддерживается больше конструкций языка UML/OCL. Оно также позволяет не только решать задачу верификации UML/OCL модели быстрее, но и дополнительно обнаруживать зависимые инварианты (т.е. инвариант, который логически следует из другого).

Все решения, основанные на редукции к SAT, имеют один общий недостаток: ограниченную поддержку арифметических выражений (поддерживается только целочисленная арифметика без мультипликативных операций). При трансформации в логическую формулу арифметические выражения

преобразуются в логические, оперирующие с булевскими переменными. Исходная семантика при этом теряется. Кроме того, расширение области значений целочисленных переменных может привести к комбинаторному взрыву в размере генерируемой логической формулы.

2.3. Методы, использующие дескрипционные логики

Дескрипционные логики — семейство формализмов для представления знаний. Они позволяют выводить одни знания из других, представленных в базе знаний. В последнее время дескрипционные логики широко применяются в концепции семантической паутины (Semantic Web) для построения онтологий [20]. Верификация UML диаграмм с помощью данного формализма основана на их трансляции в ту или иную дескрипционную логику, а затем использовании одной из известных машин ДЛ-вывода для проверки выполнимости концептов и непротиворечивости утверждений. Любой дескрипционной логике свойственен компромисс между выразительностью и сложностью логического вывода. Выразительные логики, как правило, не могут применяться для вывода высокой сложности. С другой стороны, логики с эффективными процедурами вывода не обладают высокой выразительностью. По этой причине степень покрытия семантики языка UML дескрипционными логиками не является высокой. Ограничения OCL не поддерживаются ни одним из известных решений.

Среди существующих решений для верификации UML диаграмм отметим следующие. Первое разработано в университете Рима [21]. Оно транслирует диаграмму классов UML в дескрипционную логику ALCQI и использует ДЛ-машины FaCT [22] и RACER [23] для проверки выполнимости исходной модели. Из ограничений целостности поддерживаются только ограничения кардинальности атрибутов и ассоциаций, а также ограничения наследования ”disjoint” и ”overlapping”. В работе теоретически доказывается экспоненциальная сложность данной задачи. Второе решение, разработанное в университете Брюсселя [24], основано на использовании дескрипционной логики ALCQRIFO и машины вывода Loom [25]. Оно позволяет анализировать совместно диаграммы классов, последовательностей и состояний и находить такие нестыковки между ними, как отсутствие определений сущностей, атрибутов и операций, провисшие ассоциации, нарушение порядка выполнения и т.п. Конфликты подобного рода, как правило, возникают в процессе редактирования моделей UML, когда изменения, сделанные, например, в статической модели (диаграмме классов), забывают перенести на связанные с ней динамические модели. Используемая дескрипционная логика имеет высокую выразительность, однако в работе не приводятся теоретические и экспериментальные оценки сложности и не исследуется применимость данной логики к анализу масштабных UML моделей.

2.4. Методы, использующие логики высших порядков

Решения, основанные на использовании логик высшего порядка, проводят верификацию посредством доказательства теорем. Автоматическое доказательство не поддерживается, и требуется участие высококвалифицированного пользователя, что является существенным недостатком данных средств. Это объясняется фактом невозможности автоматически различить, где произошла ошибка: в формулировке теоремы или в выбранной стратегии доказательства. Рассмотрим два наиболее известных средства, основанных на данном формализме.

Prototype Verification System (PVS) [26] — среда верификации формальных спецификаций, интегрированная в Emacs. Она включает в себя язык спецификаций, основанный на классической типизированной логике высшего порядка, а также программу доказательства теорем, поддерживающую несколько процедур принятия решений. Существуют методики трансформации диаграмм классов (исключая отношения обобщения/специализации) и состояний UML, а также ограничений OCL в язык PVS [27].

Higher-order logic and object constraint language (HOL-OCL) [28] — приложение для верификации диаграмм классов UML, которая проводится посредством доказательства теорем с помощью универсального пакета Isabelle/HOL [29]. С помощью HOL-OCL возможно найти эквивалентные ограничения, но он не способен определить подмножество ограничений, нарушающих выполнимость исходной модели. HOL-OCL поддерживает верификацию диаграмм классов UML, за исключением стереотипов и ограничений кратности концов ассоциаций. С помощью HOL-OCL возможна также проверка выполнимости инвариантов OCL, при этом поддерживаются только основные типы данных OCL: целочисленный, действительный, строковый и логический. Обобщенные типы OCL, такие как OclVoid, OclModelElementType и OclType, не могут быть явно описаны в его внутреннем представлении.

2.5. Методы удовлетворения ограничений

Средства верификации данного класса редуцируют задачу верификации моделей к задаче удовлетворения ограничений (CSP) [30] и ее решению методами логического программирования. Задача формулируется как конечное множество переменных и множество определенных на них ограничений. Решение осуществляется в два этапа. На первом определяются кардинальные числа для множеств экземпляров каждого класса и ассоциации. На втором находятся допустимые значения атрибутов и ролей данных экземпляров. Области определения значений переменных, как правило, ограничиваются, что является существенным недостатком. Невозможность найти решение при текущих заданных границах может вовсе не означать его отсутствия в случае установки границ другим способом.

Наиболее известным приложением, относящимся к данному классу средств верификации моделей является UMLtoCSP [31, 32]. Данное приложение принимает модель UML в формате ArgoUML XMI и текстовый файл с ограничениями OCL, на основе исходных данных формулирует задачу удовлетворения ограничений (CSP), а затем использует основанный на расширении языка Prolog решатель ограничений ECLiPSe [33] для нахождения ее решения. При этом проверяется выполнимость модели, а также наличие избыточных ограничений. Весь процесс полностью автоматизирован и не требует вмешательства пользователя. UMLtoCSP поддерживает основные элементы диаграммы классов UML за исключением зависимостей, агрегаций и стереотипов, а также инварианты, пред- и постусловия OCL.

2.6. Средства валидации

В качестве примера рассмотрим одно из наиболее распространенных приложений валидации. UML-based Specification Environment (USE) [34] — приложение генерации тестовых последовательностей для моделей, описываемых на UML/OCL. Формализм, лежащий в основе USE, базируется на оригинальном языке ASSL (A Snapshot Sequence Language). На основе заданного пользователем исходного состояния моделируемой системы и набора параметров USE генерирует последовательность состояний системы, описываемых на языке ASSL. Затем, используя метод «обход с возвратами», оно проверяет корректность каждого из сгенерированных состояний и либо находит хотя бы одно корректное, либо делает вывод о некорректности модели. Из всех рассматриваемых в данном обзоре решений USE наиболее широко охватывает конструкции UML/OCL: поддерживаются диаграммы классов, последовательностей и активностей UML, инварианты, пред- и постусловия языка OCL. При проверке постусловий выполнение операций UML моделируется вручную. Пользователь должен соответствующим образом изменить популяцию объектов, а затем запустить тест, проверяющий, удовлетворяет ли модифицированная популяция данному постусловию. Приложение USE неприменимо для валидации масштабных моделей, поскольку лежащий в его основе формализм основан на переборе всех возможных состояний и для нахождения решения требуется полный обход пространства поиска.

2.7. Сравнение и критика

Обсудим возможности применения вышеперечисленных методов для решения рассматриваемой задачи верификации масштабных индустриально значимых моделей данных. Методы, основанные на редукции к системам линейных неравенств или к дескрипционным логикам, не могут полноценно использоваться для ее решения по причине недостаточного покрытия семантики языков информационного моделирования. Поддержка только ограничений кардинальности не позволит решить поставленную задачу в полном объеме. Подходы, основанные на доказательстве теорем или методах

валидации, требуют ручного вмешательства, что представляется проблематичным, принимая во внимание высокую размерность исходных данных.

Сравним оставшиеся два подхода по производительности. Для этого обратимся к работе [35]. В ней описаны тесты производительности с использованием решений на основе редукции к задачам SAT и CSP на различных наборах данных. Естественно, что длительность верификации всецело зависит от особенностей используемого метода и набора тестовых данных. Однако можно заметить следующее: ни одно из решений не показывает приемлемого результата при работе с масштабными моделями, содержащими порядка 1000 классов и 1000 ассоциаций, что соответствует реальным индустриальным моделям, применяемым на практике. Метод CSP не может завершиться за разумное время уже на моделях, содержащих несколько десятков классов. Метод SAT, как правило, работает быстрее CSP. Однако и он не справляется с моделями, состоящими из нескольких сотен классов. Таким образом, вышерассмотренные решения не могут использоваться для верификации масштабных индустриально значимых моделей данных.

3. Предлагаемый комбинированный метод верификации масштабных объектно-ориентированных моделей данных

3.1. Основные определения

В настоящей статье остановимся на рассмотрении диаграмм классов UML, ограничения в которых формулируются на языке OCL, а также спецификаций на языке EXPRESS, поскольку эти два языковых средства наиболее часто используются для моделирования данных в различных индустриальных областях. Они предоставляют широкий спектр конструкций для описания структур данных и семантических ограничений, накладываемых на них. Несмотря на некоторую разницу в используемой терминологии и наборе поддерживаемых языковых конструкций, легко выделить общую часть, характерную для большинства языков моделирования, перейдя на уровень метамодели. Основываясь на работах [36, 37, 38], введем следующие определения.

Определение 1. *Объектно-ориентированной моделью* называется система $S = \langle T, \prec, \triangleright, R \rangle$, где:

- $T = T_D \cup C$ — множество простых типов данных T_D и объектных типов модели $C = C_C \cup C_A$, представимых абстрактными и конкретными классами C_A и C_C соответственно;
- $\prec$ — частичный порядок на C, отражающий отношения обобщения/специализации между классами и используемый в качестве основы для встраиваемого механизма полиморфизма. Отношение $\prec$ обладает свойством транзитивности $\forall c_1, c_2, c_3 \in C, c_1 \prec c_2 \wedge c_2 \prec c_3 \Rightarrow c_1 \prec c_3$;
- $\triangleright$ — отношения ассоциации, композиции и агрегации (связи), устанавливаемые между классами модели;
- $R = R_C \cup R_M \cup R_U \cup R_F \cup R_L \cup R_G$ — множество семантических ограничений модели, представимое ограничениями кардинальности R_C, кратности связей R_M, уникальности связей R_U, уникальности атрибутов классов R_F, а также локальными R_L и глобальными R_G правилами. Локальные и глобальные правила представляются произвольными логическими выражениями, определяемыми на отдельных типах данных либо их совокупности соответственно.

Определение 2. *Невырожденной* будем называть модель S, такую что $C_C \in S \mid \exists c \in C_C$.

Введем следующие обозначения: $o_i(c)$ — экземпляр класса $c \in C$ (объект), $O = \{o_1(c_1),...,o_{m_1}(c_1),...,o_1(c_n),...,o_{m_n}(c_n)\}$ — конечное множество (коллекция) объектов модели, O_c — подмножество ее объектов, являющихся экземплярами класса $c \in C$ и формирующих его простой экстент, $\overline{O_c}$ — расширенный экстент, объединяющий экземпляры класса и всех его специализаций, а также $\{a_i\}_c, i = \overline{1,n}$ — конечное множество атрибутов класса $c \in C$, $\{a_i\}_{o(c)}, i = \overline{1,n}$ — конечное множество значений атрибутов объекта $o(c)$.

Определение 3. *Семантически корректной* будем называть коллекцию объектов O, удовлетворяющих всем ограничениям модели R.

Фундаментальным свойством корректности объектно-ориентированной модели является ее выполнимость, т.е. возможность создания набора

экземпляров классов модели, не нарушающих определенных в ней ограничений. В работах [32, 35] даются определения слабой и сильной выполнимости модели. Переформулируем эти определения с использованием введенных обозначений.

Определение 4. Невырожденная модель S называется *слабо выполнимой* в том и только том случае, если существует семантически корректная коллекция O, такая что $\exists c \in C \mid o(c) \in O$.

Определение 5. Невырожденная модель S называется *сильно выполнимой* в том и только том случае, если существует семантически корректная коллекция O, такая что $\forall c \in C_C \exists o(c) \in O$.

Если определение слабой выполнимости не вызывает нареканий, то определение сильной выполнимости может привести к ложному утверждению о семантической некорректности модели из-за ограничений кардинальности вида «из подмножества классов $C_1 \subset C$ в корректную коллекцию O могут попасть объекты только заданного количества классов m, такого что $m < |C_1|$». Рассмотрим конкретные примеры, приведенные на рис. 1. Пусть для абстрактного класса *A* определено ограничение кардинальности $r_A \in R_C$ в виде интервала $I_{r_A} = [1,1]$. Тогда невозможно создать семантически корректную коллекцию объектов, в которой бы присутствовали экземпляры обоих специализаций данного класса *C1* и *C2* (рис. 1а). Аналогичным образом, пусть для класса *B* определено ограничение кардинальности $r_B \in R_C$ в виде интервала $I_{r_B} = [1,1]$. Тогда невозможно создать семантически корректную коллекцию объектов, в которой бы одновременно присутствовали экземпляры классов *D1* и *D2*, связанные с классом *B* взаимно исключающими отношениями агрегации (рис. 1б). Очевидно, что в обоих примерах нарушается вышеприведенное определение сильной выполнимости. Тем не менее, обе модели являются корректными.

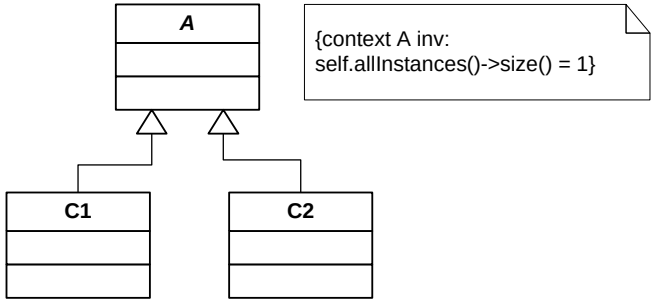


Рис. 1а. Отношения наследования, не допускающие наличия экземпляров одного из классов в коллекции.

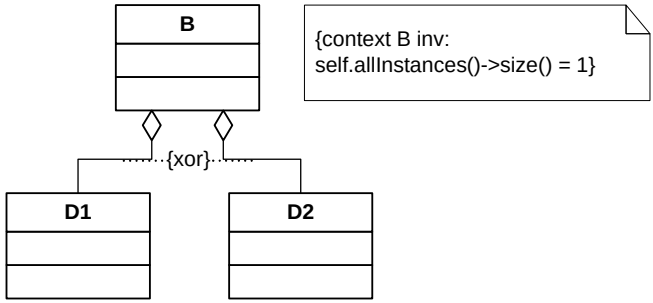


Рис. 1б. Отношения агрегации, не допускающие наличия экземпляров одного из классов в коллекции.

Дадим альтернативное определение сильной выполнимости, позволяющее избежать подобного ложного суждения о некорректности.

Определение 6. Невырожденная модель называется *сильно выполнимой* в том и только том случае, если существует множество семантически корректных коллекций $\{O\}$, такое что $\forall c \in C_C \exists o(c) \in O \mid O \subset \{O\}$.

3.2. Основные компоненты предлагаемого метода

Каждый из рассмотренных методов решения задачи верификации моделей работает по следующему сценарию: исходная задача редуцируется к той или иной математической постановке (к задаче линейного программирования, логического программирования, выполнимости булевых формул и т.п.), затем применяется известный метод решения поставленной математической задачи. При редукции каждому элементу верифицируемой модели ставится в соответствие одна или несколько переменных. Поиск решения осуществляется в области многомерного пространства, покрывающей все возможные

состояния модели. Однако размерность редуцируемой задачи становится настолько высокой, что с ней не справляются даже хорошо зарекомендовавшие себя подходы.

Анализ известных объектно-ориентированных моделей, применяемых в различных прикладных областях, показывает, что в них можно выделить группы сильно и слабо связанных между собой семантических ограничений. К первой группе можно отнести ограничения кардинальности, кратности связей и уникальности, определяющие в совокупности размер корректной популяции. Ко второй группе относятся локальные и глобальные правила, каждое из которых по отдельности влияет только на отдельный класс или небольшую группу взаимосвязанных классов.

В связи с этим для решения задачи верификации моделей предлагается использовать комбинированный метод, позволяющий сначала разрешить сильно связанные ограничения и определить необходимый размер семантически корректной коллекции объектов, а затем изолированно обрабатывать локальные и глобальные правила, определенные в модели, и установить корректные значения атрибутов объектов и ассоциаций между ними, удовлетворяющих каждому ограничению в отдельности. При этом формируются альтернативные наборы частных решений в виде набора элементарных операций инициализации атрибутов и ассоциаций. На заключительном этапе предполагается комбинировать полученные частные решения и согласовывать их между собой на основе оригинального метода семантической реконсиляции [39]. Данный метод подразумевает выявление зависимостей между операциями в параллельных транзакциях (применительно к рассматриваемой задаче это транзакции формирования данных для вышеупомянутых частных решений) и применение логического вывода для их семантически корректного слияния. В том случае, если в результате применения метода реконсиляции была сгенерирована коллекция данных, можно говорить о выполнимости модели и непротиворечивости в ней системы семантических ограничений. В случае отсутствия решения метод позволит определить несовместные операции и локализовать подмножество элементов исходной модели, в котором, возможно, допущена ошибка.

Таким образом, исходная задача верификации объектно-ориентированных моделей редуцируется к нескольким математическим постановкам: на этапе определения размера коллекции объектов — к задаче линейного программирования, на этапе формирования альтернативных частных решений — к задаче CSP, а на этапе их реконсиляции — к задаче выполнимости булевых формул (SAT). Методы решения данных математических задач хорошо известны, и для них имеется большое количество реализаций.

На наш взгляд, использование предлагаемого комбинированного метода позволит избежать проблем производительности, свойственных известным подходам. На начальном этапе размерность задачи будет относительно высокой. Однако известные методы редукции к системам линейных

неравенств и дальнейшему решению задачи линейного целочисленного программирования имеют наилучшую среди остальных полиномиальную оценку сложности [14, 15, 40]. На остальных этапах, когда область поиска решения будет локализована путем нахождения необходимого количества объектов, ограничения будут рассматриваться по отдельности и охватывать относительно небольшие группы объектов, что позволит серьезно сократить размерность задачи. С другой стороны, комбинированный метод позволит объединить достоинства отдельных методов, в частности обрабатывать подавляющее большинство известных языковых конструкций и достичь максимально возможной степени покрытия семантики языков информационного моделирования.

4. Определение размера корректной коллекции объектов

Ключевой проблемой, возникающей при генерации корректной коллекции объектов O, является определение допустимого количества экземпляров каждого класса из множества C, которое бы не нарушало семантические ограничения модели. Отметим, что упомянутые выше ограничения кардинальности, кратности и уникальности связей, уникальности атрибутов классов, а также отношения обобщения/специализации явным или неявным образом должны приниматься во внимание при решении данной задачи. Рассмотрим более подробно каждый из введенных видов ограничений и отношений в модели.

4.1. Ограничения кардинальности

Ограничения кардинальности задаются на классах модели в виде одного или нескольких интервалов и определяют допустимое число экземпляров соответствующих объектных типов. Например, в языке EXPRESS кардинальность объектных экстентов для некоторого класса c может быть определена явным образом с использованием логических выражений на основе функции `SIZEOF(c)`. Для класса UML ограничение кардинальности может быть специфицировано в виде инварианта OCL, в выражении которого участвует следующая последовательность операций: `self.allInstances()->size()`.

```
SCHEMA PersonOrganization;

  TYPE Label = STRING(255);
  END_TYPE;

  TYPE ActorRole = Label;
  END_TYPE;
```

```
  ENTITY Organization;
      Id          : INTEGER;
      Name        : Label;
      Description : OPTIONAL STRING;
      Roles       : LIST [1:?] OF UNIQUE ActorRole;
    INVERSE
      Engages     : SET [0:?] OF Person FOR EngagedIn;
    UNIQUE
      UR1 : Id;
    WHERE
      WR1 : SIZEOF( QUERY( temp <* Engages | 'Director'
in temp.Roles ) ) = 1;
  END_ENTITY;

  ENTITY Person;
      Id           : INTEGER;
      FamilyName   : OPTIONAL Label;
      GivenName    : OPTIONAL Label;
      MiddleNames  : OPTIONAL LIST [1:?] OF Label;
      PrefixTitles : OPTIONAL LIST [1:?] OF Label;
      SuffixTitles : OPTIONAL LIST [1:?] OF Label;
      Roles        : LIST [0:?] OF UNIQUE ActorRole;
      EngagedIn    : SET [0:?] OF Organization;
    UNIQUE
      UR1 : Id;
    WHERE
      WR1 : EXISTS(FamilyName) OR EXISTS(GivenName);
  END_ENTITY;

RULE DirectorRule FOR (Person);
  WHERE
    WR1 : SIZEOF( QUERY(temp <*Person | 'Director' in
temp.Roles AND
          SIZEOF( temp.EngagedIn ) <> 1 ) = 0;
END_RULE;

END_SCHEMA;
```

Рис 2. Представление схемы `PersonOrganization` *на языке EXPRESS*

Пусть $r \in R_C$ задает кардинальность соответствующего класса $c \in C$ в виде интервала I_r, тогда допустимое число экземпляров этого объектного типа должно лежать в данном интервале или:

$$low(I_r) \le \left|\overline{O_c}\right| \le high(I_r) \quad (1),$$

где функции $low(I_r)$ и $high(I_r)$ возвращают значения нижней и верхней границы интервала I_r соответственно.

Заметим, что переход от семантических ограничений кардинальности в языках информационного моделирования к линейным неравенствам носит необходимый, но не всегда достаточный характер. Рассмотрим это на примере фрагмента информационной схемы `PersonOrganization`, специфицированной на языке EXPRESS (см. рис. 2). Предположим, что каждую организацию возглавляет директор, который не может совмещать эту позицию с деятельностью в других организациях. В рамках информационной схемы это ограничение специфицировано с помощью глобального правила `DirectorRule`. Вместе с локальным правилом `WR1` в классе `Organization` оно приводит к очевидному условию для кардинальности: $\left|\overline{O_{PERSON}}\right| \ge \left|\overline{O_{ORGANIZATION}}\right|$.

Однако, чтобы удовлетворить исходные семантические ограничения, требуется не только создать необходимое количество объектов классов `Person` и `Organization`, но и задать правильным образом значения их атрибутов и ассоциаций. В рамках предложенной вычислительной стратегии это осуществляется на следующих этапах верификации, где размер генерируемой коллекции также может быть уточнен путем добавления или удаления некоторого количества объектов, необходимого для достижения ее корректного состояния.

Существенно, что если для класса c непосредственно определено n отношений специализации, $c \prec c_i, c_i \in C, i = \overline{1,n}$, то имеет место следующее равенство:

$$\left|\overline{O_c}\right| = \sum_{i=1}^{n} \left|\overline{O_{c_i}}\right| + \left|O_c\right| \quad (2),$$

и ограничение кардинальности неявно налагает дополнительные условия и для специализированных классов. В случае отсутствия специализаций у класса условие (2) трансформируется в простое и очевидное равенство: $\left|\overline{O_c}\right| = \left|O_c\right|$ (простой и расширенный экстенты в данном случае совпадают).

4.2. Множественное наследование

Остановимся более подробно на общей модели наследования классов, предусматриваемой языками информационного моделирования. Возьмем за основу схему наследования, применяемую в языке EXPRESS, как наиболее

общую. Все возможные случаи наследования в UML также покрываются данной схемой.

Итак, в языке EXPRESS схема наследования задается при определении суперкласса с помощью конструкции SUPERTYPE и выражения ограничения с использованием следующих спецификаторов:

— ONEOF устанавливает ограничение взаимоисключения для экземпляров подклассов, что соответствует ограничению disjoint в языке UML. Ни для одного из классов, входящих в список ONEOF, не может быть создан экземпляр, принадлежащий одновременно и любому другому классу из данного списка;

— ANDOR допускает любую комбинацию для конструируемых экземпляров подклассов, что соответствует ограничению overlapping в языке UML. Это означает, что экземпляр суперкласса может быть одновременно экземпляром более чем одной из его специализаций;

— AND определяет обязательные комбинации для экземпляров подклассов. Каждый экземпляр суперкласса в этом случае всегда формируется как составной объект и является одновременно экземпляром каждой из групп, разделенных в спецификации ограничения с помощью AND. Заметим, что непосредственный аналог данного ограничения в UML отсутствует, хотя его можно эмулировать путем комбинации ограничения overlapping и спецификаций перекрываемых классов как абстрактных.

Языком допускаются сложные вложенные выражения ограничений суперклассов. Однако в большинстве случаев используется простое наследование на основе конструкции ONEOF и сложное наследование на основе AND/ANDOR для групп, каждая из которых задается взаимоисключающим образом с помощью вложенной конструкции ONEOF. В случае простого наследования действует условие (2), приведенное в предыдущем разделе, где n — количество специализаций, указанное в конструкции ONEOF. Исследуем варианты множественного наследования и алгебраические условия для кардинальности, индуцируемые произвольными выражениями для ограничений суперкласса.

Введем оператор комбинации классов & для определения нового составного класса, экземпляры которого формируются как комбинации объектов перечисленных классов-операндов. Для оператора & имеют место следующие тождества:

```
A&A ≡ A
A&B ≡ B&A
A&(B&C) ≡ (A&B)&C = A&B&C
```

Первое тождество означает, что компонентами составного объекта могут являться лишь объекты различных классов. Второе и третье тождества

определяют свойства коммутативности и ассоциативности для оператора &, означая, что при составлении сложного объекта порядок классов не важен.

Определим оператор объединения классов как оператор задания множества перечисленных классов-операндов. Запись `[A,B,C]` означает задание множества классов `A,B,C`, а запись `[]` — пустое множество. Имеют место следующие тождества для введенного оператора объединения:

```
[A,B] ≡ [B,A]
[A,A,B] ≡ [A,B]
[A,[B,C]] ≡ [A,B,C]
[A]&[B,C] ≡ [A&B,A&C]
```

Отметим, что оператор комбинации классов аналогичен спецификатору `AND`, а оператор объединения классов — спецификатору `ONEOF`. Используя вышеописанный формализм, можно переписать выражения ограничений суперклассов в терминах множеств. Список `ONEOF` приводится к множеству, содержащему соответствующий набор классов. Выражение со спецификатором `AND` может быть заменено оператором комбинации классов & с соответствующими операндами. Наконец, спецификатор `ANDOR` задает множество, состоящее из классов-операндов по отдельности и их комбинации. Правила и типовые случаи приведения выражений для ограничений суперклассов даются ниже:

```
ONEOF(A,B,...)──> [A,B,...]
A AND B —> [A&B]
A ANDOR B —> [A,B,A&B]
A AND ONEOF(B1,B2) —> A&[B1,B2] = [A&B1,A&B2]
ONEOF(A1,A2) AND ONEOF(B1,B2) ──>
[A1,A2]&[B1,B2] =[A1&B1,A1&B2,A2&B1,A2&B2]
```

Применяя данные правила рекурсивно, можно «раскрыть» выражение ограничения суперкласса и получить требуемое представление для всех допустимых комбинаций классов, не содержащее ни одного терма `ONEOF`, `ANDOR` и `AND`. Тогда условие для кардинальности приобретает вид:

$$\left|\overline{O_c}\right| = \sum_{i=1}^{n}\left|O_{c_i}\right| + \left|O_c\right| \qquad (3),$$

где n — общее количество специализаций суперкласса, порождаемых всеми его простыми и составными подклассами рекурсивно. В отличие от равенства (2) здесь суммируются простые экстенты специализаций, поскольку составные объекты принадлежат одновременно расширенным экстентам нескольких специализаций и требуется исключить их повторный подсчет. Однако при этом следует рекурсивно учесть все возможные специализации суперкласса, включая не только его непосредственные, но и порождаемые ими.

Аналогичные условия могут быть получены для всех суперклассов объектно-ориентированной модели. Отметим, что формированию вектора

кардинальности должны предшествовать анализ отношений множественного наследования и идентификация всех составных классов в соответствии с описанными выше правилами раскрытия выражений для ограничений суперклассов.

4.3. Ограничения кратности и уникальности связей

Ограничения кратности связей R_M определяются на отношениях $\triangleright$ таким образом, что с каждым $r \in R_M$ ассоциировано два интервала I_r^f и I_r^b, задающих допустимые значения кратностей прямого и обратного отношений соответственно. Ограничения уникальности $r_U(c_1 \triangleright c_2) \in R_U$ определяются непосредственно на отношениях ассоциации между классами и исключают повторное вхождение одних и тех же объектов. Отношения композиции и агрегации автоматически предполагают наличие ограничений уникальности. Выразим данные семантические ограничения в виде алгебраических условий.

Для обязательных ассоциаций с неуникальными элементами должны выполняться следующие условия:

$$\begin{cases} low(I_r{}^f) \geq 1 \wedge \left|\overline{O_{c_1}}\right| \geq 1 \rightarrow \left|\overline{O_{c_2}}\right| \geq 1 \\ low(I_r{}^b) \geq 1 \wedge \left|\overline{O_{c_2}}\right| \geq 1 \rightarrow \left|\overline{O_{c_1}}\right| \geq 1 \end{cases} \quad (4),$$

поскольку задание нижней ненулевой границы кратности предполагает существование, по крайней мере, одного экземпляра ассоциируемого класса, как для прямого, так и для обратного отношения ассоциации.

Для обязательных ассоциаций с уникальными элементами должна выполняться следующая система условий:

$$\begin{cases} \left|\overline{O_{c_1}}\right| \geq 1 \rightarrow \left|\overline{O_{c_2}}\right| \geq low(I_r^f) \\ \left|\overline{O_{c_2}}\right| \geq 1 \rightarrow \left|\overline{O_{c_1}}\right| \geq low(I_r^b) \\ low(I_r^f)\left|\overline{O_{c_1}}\right| \leq high(I_r^b)\left|\overline{O_{c_2}}\right| \\ low(I_r^b)\left|\overline{O_{c_2}}\right| \leq high(I_r^f)\left|\overline{O_{c_1}}\right| \end{cases} \quad (5).$$

Первые два условия требуют наличие минимального количества экземпляров ассоциируемых классов, определяемого нижними границами кратностей соответствующих отношений. Следующие два условия позволяют исключить случаи невозможного разрешения ассоциаций в силу недостатка или избытка объектов соответствующих классов.

Выделим важные частные случаи формализации алгебраических условий по заданным ограничениям кратности. Для однонаправленной ассоциации

кратность обратного отношения можно считать равной $I_r^b = [0, \infty]$, и в этом случае последние два условия вырождаются. В случае задания обязательной ассоциации «один к одному» ($I_r^f = [1,1]$, $I_r^b = [1,1]$) система сводится к простому равенству $\left|\overline{O_{c_1}}\right| = \left|\overline{O_{c_2}}\right|$. Если два класса $c_1, c_2 \in C$ связаны отношением композиции, то кратность обратного отношения равна $I_r^b = [1,1]$ и система приобретает вид $low(I_r^f)\left|\overline{O_{c_1}}\right| \le \left|\overline{O_{c_2}}\right| \le high(I_r^f)\left|\overline{O_{c_1}}\right|$.

4.4. Ограничения уникальности атрибутов

Ограничения уникальности атрибутов $r \in R_F$ определяются для отдельных классов модели $c \in C$ путем задания группы атрибутов $\left\{c.a_i\right\}, i = \overline{1,n}$, комбинации значений которых не могут повторяться в семантически корректной коллекции данных. Заметим, что задание ограничений уникальности на атрибутах, имеющих конечную область определения значений, может налагать дополнительные условия на допустимое количество объектов соответствующего класса. К таковым, в частности, относятся атрибуты, определенные на перечислимых, логических типах или же на целых числах с дополнительными ограничениями области допустимых значений. Пусть в классе $c \in C$ задано ограничение уникальности $r \in R_F$ для группы атрибутов $\left\{c.a_i\right\}, i = \overline{1,n}$, определенных на конечных множествах $t_i \in T_D, i = \overline{1,n}$, тогда должно выполняться:

$$\left|\overline{O_c}\right| \le \left|t_1\right|\left|t_2\right|...\left|t_n\right| \qquad (6),$$

где $\left|t_i\right|$ — мощность соответствующего домена.

Обсудим более экзотические, но возможные варианты задания ограничения уникальности на коллекциях. Здесь интерес представляют случаи, когда тип элементов коллекции имеет ограниченное множество значений. К ним относятся рассмотренные выше перечислимые типы, логические типы, целочисленные типы с дополнительными ограничениями допустимой области значений, а также классы, участвующие в типизированных ассоциациях. Для краткости ограничимся наиболее часто применяемыми в языках информационного моделирования вариантами коллекций, определенных на упорядоченных и неупорядоченных множествах и мультимножествах.

Итак, пусть в классе $c \in C$ определено ограничение уникальности $r \in R_F$ для атрибута, являющегося коллекцией с кратностью, заданной интервалом

I_r, и типом элементов $t_j \in T_D$, являющимся конечным множеством значений. Пусть $k = \left| t_j \right|$ — мощность соответствующего домена (заметим, что если коллекция является ассоциацией с классом $c_j \in C$, то $k = \left| \overline{O_{c_j}} \right|$), $n = low(I_r)$, $m = high(I_r)$. Тогда данное семантическое ограничение приводит к следующим условиям кардинальности.

Для множества это — $\left| \overline{O_c} \right| \le \sum_{i=n}^{m} C_k^i$, где суммирование ведется по всем возможным сочетаниям значений из имеющегося набора размером k. Это довольно слабое условие. Для мультимножества аналогичное условие кардинальности приобретает вид $\left| \overline{O_c} \right| \le \sum_{i=n}^{m} C_{k+i-1}^{k-1}$, поскольку элементы в мультимножествах могут повторяться и это приводит к увеличению верхней оценки для кардинальности $\left| \overline{O_c} \right|$. Для упорядоченного мультимножества условие слабее, чем для множества, но сильнее, чем для мультимножества: $\left| \overline{O_c} \right| \le \sum_{i=n}^{m} k^i$. Наконец, для упорядоченного множества условие определяется общим количеством возможных размещений и имеет вид $\left| \overline{O_c} \right| \le \sum_{i=n}^{m} A_k^i$.

Следующая таблица объединяет полученные результаты для различных типов коллекций. Из полученных оценок можно выделить два случая, дающих линейные ограничения: первый, когда заранее известно, что $k = 1$ (например, это следует из другого ограничения модели), второй, когда ассоциация не является коллекцией, т.е. $n = m = 1$ (см. табл. 2). Дополнительный случай, когда вышеприведенные соотношения трансформируются в линейные неравенства, соответствует коллекциям необъектных типов. Тогда $k = \left| t_j \right|$ не является переменной задачи и комбинаторные выражения вырождаются в константы.

Тип коллекции	Условие кардинальности	Оценка
Мультимножество	$\left\|\overline{O_c}\right\| \le \sum_{i=n}^{m} C_{k+i-1}^{k-1}$	$k=1 \to \left\|\overline{O_c}\right\| \le m-n+1$ $n=m=1 \to \left\|\overline{O_c}\right\| \le k$
Упорядоченное мультимножество	$\left\|\overline{O_c}\right\| \le \sum_{i=n}^{m} k^i$	$k=1 \to \left\|\overline{O_c}\right\| \le m-n+1$ $n=m=1 \to \left\|\overline{O_c}\right\| \le k$
Множество	$\left\|\overline{O_c}\right\| \le \sum_{i=n}^{m} C_k^i$	$k=1 \to \left\|\overline{O_c}\right\| \le 1$ $n=m=1 \to \left\|\overline{O_c}\right\| \le k$
Упорядоченное множество	$\left\|\overline{O_c}\right\| \le \sum_{i=n}^{m} A_k^i$	$k=1 \to \left\|\overline{O_c}\right\| \le 1$ $n=m=1 \to \left\|\overline{O_c}\right\| \le k$

Табл. 2. Условия кардинальности, порождаемые уникальными коллекциями

4.5. Редукция к задаче линейного программирования

Для заданной объектно-ориентированной модели $S = \langle T, \prec, \triangleright, R \rangle$ поставим задачу определения мощностей объектных экстентов $\left|O_c\right|$ для всех конкретных классов $c \in C_C$, при которых коллекция объектов O будет являться семантически корректной. Решение задачи в данной постановке соответствует поиску необходимого количества объектов для генерации коллекции с целью проверки сильной выполнимости верифицируемой модели согласно определению 5. В предположении, что семантические ограничения исходной модели допускают одновременное наличие экземпляров каждого конкретного класса, сведем задачу определения мощностей к традиционной постановке линейного целочисленного программирования [40, 41].

Сформируем вектор неизвестных переменных $\{x_i\} \in \aleph$, $i = \overline{1,k}$, $k = \left|C_C\right|$, каждая из которых ассоциирована с соответствующим конкретным классом модели $c_i \in C_C$ и определяет мощность его простого экстента $x_i = \left|O_{c_i}\right|$. Рассмотрим задачу минимизации $\min f(x)$ с целевой функцией

$f(x) = \sum_{i=1}^{k} x_i$, что будет соответствовать поиску простейшей репрезентативной коллекции данных. Составим общую систему линейных неравенств путем включения в нее соотношений (1) для всех ограничений кардинальности, определенных для классов модели, условий (5) — для ассоциаций, агрегаций и композиций с уникальными элементами, неравенств (6) — для ограничений уникальности атрибутов, заданных на конечных доменах. Равенства (2) для простых или (3) для сложных отношений специализации выражаются в переменных x_i с учетом условия, что мощности простых экстентов абстрактных классов равны нулю, и подставляются в соотношения (1), (5), (6). Поскольку в рассматриваемой постановке предполагается, что $\left|\overline{O_{c_i}}\right| \geq 1,\ i = \overline{1,n},\ n = |C|$, то условия (4) вырождаются и не включаются в общую систему задачи линейного программирования, а условия (5) приобретают вид линейных неравенств. К системе добавляются дополнительные неравенства:

$$x_i \geq 1,\ i = \overline{1,k} \qquad (7),$$

формирующие необходимые условия наличия хотя бы одного экземпляра каждого конкретного класса в генерируемой коллекции. Что касается условий кардинальности, порождаемых уникальными коллекциями (см. табл. 2), то они включаются в систему в тех вышеупомянутых случаях, когда являются линейными неравенствами. Более слабые ограничения в остальных случаях предлагается проверять как постусловия.

Сформированная задача может быть решена известными методами, такими как симплекс-метод или метод угловых точек [42]. Найденное решение может быть взято за основу для генерации объектной коллекции с целью верификации сильной выполнимости модели. Факт отсутствия решения служит основанием для заключения о невыполнимости модели и наличии в ней избыточных или переопределенных ограничений. К сожалению, данный метод не позволяет локализовать множество данных ограничений. Для данных целей необходимо применять другие методы [14].

Заметим, что для проверки слабой выполнимости согласно определению 4 достаточно наличия одного объекта заданного класса $c_j \in C_C$. Однако формирование семантически корректного набора данных может потребовать задание и вспомогательных объектов в дополнение к основному, поскольку ограничения кардинальности носят нетривиальный характер. Тогда задача определения мощностей объектных экстентов сводится к аналогичной постановке линейного целочисленного программирования, где условия (7) трансформируются в следующие:

$$\begin{cases} x_j \geq 1 \\ x_i \geq 0 \end{cases}, i = \overline{1,k}, i \neq j \quad (8).$$

Для подтверждения сильной выполнимости модели согласно определению 6 (в случаях, когда семантические ограничения модели не допускают одновременное наличие экземпляров каждого конкретного класса) необходимо и достаточно сформировать множество из $k = |C_C|$ коллекций, каждая из которых включает хотя бы один объект класса $c_i \in C_C, i = \overline{1,k}$ и служит для проверки слабой выполнимости.

4.6. Вычислительный эксперимент

С целью апробации разработанного метода определения размера корректной коллекции был проведен вычислительный эксперимент. В качестве исходной модели взята последняя редакция IFC 4. Ее формальный статический анализ обнаружил 766 классов (из них 123 абстрактных и 643 конкретных), 207 отношений специализации, определенных в суперклассах данной модели, 63 бинарных ассоциации «один к одному», 1 обязательная бинарная ассоциация «многие ко многим» с уникальными элементами и невырожденными границами, 4 ограничения уникальности на символьных строках переменной длины. IFC использует простую модель наследования, где все специализации связаны соотношением `ONEOF` в рамках суперкласса. Таким образом, анализ сложной иерархии не требуется и параметры задачи линейного программирования могут быть сформированы за линейное время. Заметим, что количество ассоциаций в модели IFC значительно больше, но подавляющее большинство из них имеет вырожденные границы и не участвует в формировании задачи.

Вектор неизвестных задачи имеет размер 643, что соответствует числу конкретных классов. В систему включаются 63 равенства к которым сводится система условий вида (5) для ассоциаций «один к одному» и 2 неравенства для ассоциации «многие ко многим». Ограничения уникальности определяются на символьных строках переменной длины с максимальной границей 255, что соответствует случаю уникального упорядоченного мультимножества. Поскольку алфавит в IFC ограничивается печатными символами ASCII (то есть мощность домена $k = 95$), соответствующие условия кардинальности приобретают вид: $|\overline{O_c}| \leq \sum_{i=1}^{255} 95^i$. С учетом того, что решается задача нахождения минимальной репрезентативной коллекции данных, подобными условиями можно пренебречь. В систему также дополнительно включаются 643 неравенства вида (7) для каждой из переменных задачи. Проведенный вычислительный эксперимент показал, что для относительно сложной модели

данных метод демонстрирует производительность, приемлемую для практического использования.

5. Пример применения комбинированного метода

Рассмотрим применение предложенного комбинированного метода на примере простейшей модели. Сборник научных трудов может включать от 10 до 20 статей. Каждая статья имеет не более двух авторов и должна быть одобрена тремя рецензентами. Авторами статей могут быть как студенты (только в соавторстве со своим научным руководителем), так и научные сотрудники, а рецензентами — только научные сотрудники. Каждый автор и каждый рецензент участвует в написании либо, соответственно, обзоре только одной статьи. При этом научные сотрудники не имеют право рецензировать собственные статьи. Научные сотрудники могут руководить несколькими студентами, студент обязан иметь одного научного руководителя. Соответствующая UML модель с OCL ограничениями представлена на рис. 3.

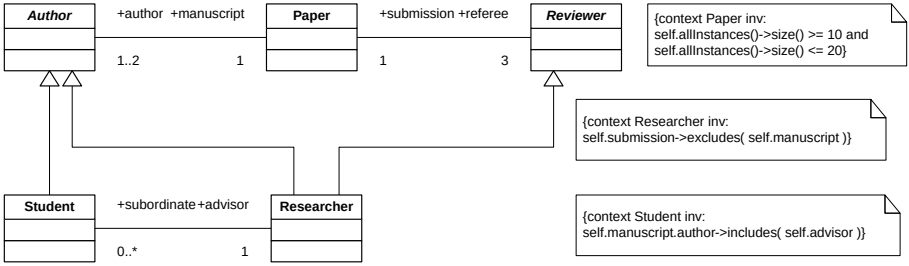


Рис. 3. Пример: UML модель «сборник научных трудов»

Множество $C = \{Paper, Author, Reviwer, Researcher, Student\}$ формирует систему классов модели, при этом $C_C = \{Paper, Researcher, Student\}$, а $C_A = \{Author, Reviwer\}$. Отношения специализации и ассоциации устанавливаются следующим образом: $Author \prec Student$, $Author \prec Researcher$, $Reviewer \prec Researcher$, $Paper \rhd Author$, $Paper \rhd Reviewer$, $Student \rhd Researcher$. Введем следующие переменные: $x_1 = |O_{PAPER}|$, $x_2 = |O_{RESEARCHER}|$, $x_3 = |O_{STUDENT}|$. Отношения специализации, установленные в данной модели, приводят к следующим равенствам: $|\overline{O_{AUTHOR}}| = x_2 + x_3$, $|\overline{O_{REVIEWER}}| = x_2$. Тогда задача определения размера коллекции сводится к следующей математической постановке:

$$f(x) = x_1 + x_2 + x_3 \to \min \quad (9)$$

$$x_1 \geq 10 \quad (10)$$

$$x_1 \leq 20 \quad (11)$$

$$x_2 \geq 3 \quad (12)$$

$$x_2 = 3x_1 \quad (13)$$

$$x_1 \leq x_2 + x_3 \quad (14)$$

$$x_2 + x_3 \leq 2x_1 \quad (15)$$

$$x_3 \geq 1 \quad (16)$$

Здесь неравенства (10) и (11) индуцированы явным ограничением кардинальности класса `Paper`, специфицированным на OCL, неравенство (12) и равенство (13) — ассоциативным отношением *Paper* ▷ *Reviewer*, неравенства (14) и (15) — ассоциативным отношением *Paper* ▷ *Author*. Дополнительное неравенство (16) требует наличия хотя бы одного экземпляра класса `Student` в коллекции, генерируемой с целью проверки сильной выполнимости исходной модели.

Заметим, что данная задача не имеет решения, что сигнализирует о невыполнимости модели и наличии в ней переопределенных ограничений. В самом деле, согласно установленному ассоциативному отношению *Paper* ▷ *Reviewer* рецензентов (в данном случае научных сотрудников) должно быть в три раза больше, чем статей. При этом, поскольку каждый научный сотрудник еще является автором, он также должен участвовать и в написании хотя бы одной статьи, что определяется кардинальностью роли manuscript $I_r = [1,1]$ ассоциативного отношения *Paper* ▷ *Author*. Но согласно установленной кардинальности $I_r = [1,2]$ роли `author` в данном отношении, общее количество авторов (включая как студентов, так и научных сотрудников) не может превышать количество статей больше, чем в два раза. Таким образом, одно из ограничений кардинальности, определяемое ассоциативными отношениями в модели, будет нарушено в любом случае.

Задача разрешима, если снять требование обязательности каждого автора участвовать в написании одной статьи, т.е. определить кардинальность роли `manuscript` как $I_r = [0,1]$. Тогда условие (15) вырождается и будет найдено следующее решение: $x_1 = 10$, $x_2 = 30$, $x_3 = 1$. Таким образом, для проверки сильной выполнимости модели необходимо сгенерировать коллекцию, содержащую 10 экземпляров класса `Paper`, 30 — класса `Researcher` и 1 — класса `Student`. Однако вывод о сильной выполнимости модели можно будет сделать только после корректной установки всех

ассоциативных отношений между сгенерированными экземплярами с учетом ограничений, что научный сотрудник не может рецензировать собственные статьи, а студенческая статья может быть выпущена только в соавторстве с научным руководителем.

Рассмотрим пример установки ассоциативных отношений в небольшом подмножестве сгенерированных объектов: $p_1 \in Paper$, $s_1 \in Student$, $r_1, r_2, r_3, r_4 \in Researcher$. Пусть при установке ассоциаций применяется следующий метод: ассоциированные экземпляры последовательно выбираются из множества допустимых объектов с их дальнейшим исключением из данного множества как уже задействованных. Тогда, учитывая кардинальности ассоциаций, соответствующие отношения могут быть расставлены следующим образом: (1) $s_1 \rhd r_1$, (2) $p_1 \rhd s_1$, (3) $p_1 \rhd (author) r_1$, (4) $p_1 \rhd (referee) r_1$, (5) $p_1 \rhd (referee) r_2$, (6) $p_1 \rhd (referee) r_3$. Анализ ограничения в контексте класса `Researcher` устанавливает следующие операции как взаимоисключающие: (3) $\oplus$ (4), а в контексте класса `Student` — выявляет следующее логическое соотношение: (2) $\rightarrow$ (1) $\wedge$ (3). Тогда один из возможных результатов реконсиляции выглядит следующим образом: (1) $s_1 \rhd r_1$, (2) $p_1 \rhd s_1$, (3) $p_1 \rhd (author) r_1$, (5) $p_1 \rhd (referee) r_2$, (6) $p_1 \rhd (referee) r_3$, (7) $p_1 \rhd (referee) r_4$, где операция (7) добавлена для установки нового отношения межу статьей и рецензентом взамен исключенного (4).

6. Заключение

Таким образом, рассмотрена проблема верификации масштабных моделей данных и предложен комбинированный метод для ее решения. Он основан на последовательной редукции к нескольким постановкам известных математических задач: линейного программирования, удовлетворения ограничений (CSP), выполнимости булевых формул (SAT). Проведенные эксперименты демонстрируют перспективность комбинированной вычислительной стратегии и эффективность предложенного метода для верификации масштабных моделей данных. В настоящее время он позволяет успешно разрешить следующие виды ограничений: ограничения кардинальности объектных экстентов, кратности и уникальности связей, уникальности атрибутов объектов. Однако для успешного разрешения ограничений, задаваемых произвольными выражениями, необходимо провести дальнейшие исследования и реализовать остальные его компоненты, связанные с поиском частных решений, удовлетворяющих отдельным ограничениям, и их семантической реконсиляции.

Литература

[1] Model Driven Architecture: The Architecture of Choice for a Changing World. Executive Overview, October 2013. http://www.omg.org/mda/executive_overview.htm

[2] Calafat A.Q. Validation of UML Conceptual Schemas with OCL Constraints and Operators. PhD Thesis, adviced by Dr. E. Teniente, Universitat Politecnica de Catalunya, Barcelona, 2009.

[3] ISO 10303-11:2004. Industrial automation systems and integration — Product data representation and exchange — Part 11: Description methods: The EXPRESS language reference manual.

[4] Unified Modeling Language (UML), V2.4.1, Release Date: August 2011. http://www.omg.org/spec/UML/2.4.1

[5] Object Constraint Language (OCL), V2.3.1, Release Date: January 2012. http://www.omg.org/spec/OCL/2.3.1

[6] IFC4 Release Summary, March 2013. http://www.buildingsmart-tech.org/specifications/ifc-releases/ifc4-release/ifc4-release-summary

[7] Khemlani L. The CIS/2 Format: Another AEC Interoperability Standard. // AECbytes Newsletter, July 27, 2005. http://www.aecbytes.com/buildingthefuture/2005/CIS2format.html

[8] ISO 10303-1:1994. Industrial automation systems and integration — Product data representation and exchange — Part 1: Overview and fundamental principles.

[9] ISO 15926-1:2004. Industrial automation systems and integration — Integration of life-cycle data for process plants including oil and gas production facilities — Part 1: Overview and fundamental principles.

[10] Чен П. Модель «сущность-связь» — шаг к единому представлению о данных. // СУБД, № 3, 1995. http://www.osp.ru/dbms/1995/03/271.htm

[11] Alloy 4: a language & tool for relational models, 2012. http://alloy.mit.edu/alloy

[12] Formica A. Finite Satisfiability of Integrity Constraints in Object-Oriented Database Schemas. // IEEE Transactions on Knowledge and Data Engineering, 14(1), 2002, pp. 123–139.

[13] Baader F., et al. The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press, 2003.

[14] Lenzerini M., Nobili P. On the Satisfiability of Dependency Constraints in Entity-Relationship Schemata. // Proceedings of the 13th VLDB Conference, 1987, pp. 147–154.

[15] Maraee A., Balaban M. Efficient Reasoning about Finite Satisfiability of UML Class Diagrams with Constrained Generalization Sets. // LNCS 4530, Proceedings of European Conference on Model Driven Architecture — Foundations and Applications, 2007, pp. 17–31.

[16] Jackson D. Alloy: a lightweight object modelling notation. // ACM Transactions on Software Engineering and Methodology, 11(2), 2002, pp. 266–290.

[17] Anastasakis K., Bordbar B., Georg G., Ray I. UML2Alloy: A challenging model transformation. // LNCS 4735, Model Driven Engineering Languages and Systems, 2007, pp. 436–450.

[18] Ramirez A., Vanpeperstraete P., Rueckert A., Odutola K., Bennett J., Tolke L., van der Wulp W. ArgoUML User Manual: A tutorial and reference description, 2011. http://argouml.tigris.org/ documentation/manual-0.34

[19] Soeken M., Wille R., Kuhlmann M., Gogolla M., Drechsler R. Verifying UML/OCL Models Using Boolean Satisfiability. // Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010, pp. 1341–1344.
[20] Turhan A.Y. Description Logic Reasoning for Semantic Web Ontologies. // WIMS'11 Proceedings of the International Conference on Web Intelligence, Mining and Semantics, Article 6, 2011.
[21] Berardi D., Calvanese D., de Giacomo G. Reasoning on UML Class Diagrams. // Artificial Intelligence, 168(1–2), 2005, pp. 70–118.
[22] The FaCT System, April 2003. http://www.cs.man.ac.uk/~horrocks/FaCT
[23] Haarslev V., Möller R. RACER system description. // LNAI 2083, Automated Reasoning, 2001, pp. 701–705.
[24] van der Straeten R., Mens T., Simmonds J., Jonckers V. Using Description Logic to Maintain Consistency between UML Models. // LNCS 2863, The Unified Modeling Language. Modeling Languages and Applications, 2003, pp. 326–340.
[25] Loom Project Home Page, July 2007. http://www.isi.edu/isd/LOOM
[26] PVS Specification and Verification System, January 2013. http://pvs.csl.sri.com
[27] Kyas M., Fecher H., de Boer F.S., Jacob J., Hooman J., van der Zwaag M., Arons T., Kugler H. Formalizing UML Models and OCL Constraints in PVS. // Electronic Notes in Theoretical Computer Science, 115, 2005, pp. 39–47.
[28] Brucker A.D., Wolff B. The HOL-OCL book. Technical Report 525, ETH Zurich, 2006.
[29] Nipkow T., Paulson L.C., Wenzel M. Isabelle/HOL — A Proof Assistant for Higher-Order Logic. // LNCS 2283, 2002.
[30] Tsang E. Foundations of constraint satisfaction. Academic Press Limited, London & San-Diego, 1993.
[31] Cabot J., Clariso R., Riera D. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. // Proceedings of the 22nd ACM/IEEE International Conference on Automated Software Engineering (ASE '07), 2007, pp. 547–548.
[32] Cabot J., Clariso R., Riera D. Verification of UML/OCL Class Diagrams using Constraint Programming. // Proceedings of the IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW'08), 2008, pp. 73–80.
[33] Apt K.R., Wallace M.G. Constraint Logic Programming using ECLiPSe. Cambridge University Press, 2007.
[34] Gogolla M., Bohling J., Richters M. Validating UML and OCL models in USE by automatic snapshot generation. // Software & System Modeling, 4(4), 2005, pp. 386–398.
[35] Shaikh A., Wiil U.K., Memon N. Evaluation of Tools and Slicing Techniques for Efficient Verification of UML/OCL Class Diagrams. // Advances in Software Engineering, 2011, Article ID 370198, 18 p.p.
[36] Breu R., Hinkel U., Hofmann C., Klein C., Paech B., Rumpe B., Thurner V. Towards a Formalization of the Unified Modeling Language. // LNCS 1241, ECOOP'97 — Object-Oriented Programming, 1997, pp. 344–366.
[37] Richters M., Gogolla M. On Formalizing the UML Object Constraint Language OCL. // LNCS 1507, Conceptual Modeling — ER'98, 1998, pp. 449–464.
[38] Семенов В.А., Морозов С.В., Тарлапан О.А. Инкрементальная верификация объектно-ориентированных данных на основе спецификации ограничений. // Труды Института системного программирования / под ред. В.П. Иванникова, т.8, ч.2, 2004, с. 21–52.

[39] Семенов В.А., Ерошкин С.Г., Караулов А.А., Энкович И.В. Семантическая реконсиляция прикладных данных на основе моделей. // Труды Института системного программирования / под ред. В.П. Иванникова, т.13, ч.2, 2007, с. 141–164.
[40] Vanderbei R.J. Linear Programming. Foundations and Extensions. Third edition. Princeton University, 2008.
[41] Шевченко В.Н., Золотых Н.Ю. Линейное и целочисленное линейное программирование. — Нижний Новгород: издательство Нижегородского госуниверситета им. Н.И. Лобачевского, 2004.
[42] Карманов В.Г. Математическое программирование: учебное пособие. 5-е издание. — М.: Физматлит, 2004.

A combined method for verification of large-scale data models

V.A. Semenov, S.V. Morozov, D.V. Ilyin
ISP RAS, Moscow, Russia
sem@ispras.ru, serg@ispras.ru, denis.ilyin@ispras.ru

Annotation. The paper is addressed to the actual problem of verification of large-scale data models applied in various industrial areas and specified using popular general-purpose object-oriented languages, such as EXPRESS, UML/OCL. Main benefits of information modeling languages (high expressiveness, declarative nature, advanced set of syntactic units) negatively affect the process of automatic verification of the specifications. The known approaches are based on reduction of the original complex problem to some well-known mathematical statement and its solution by existing methods. The performed analytical survey of the existing methods for model verification demonstrates that they cannot be used for solving the problem because of their high computational complexity. A combined method for verification of large-scale data models is proposed in the paper. The method is based on sequential reduction to the several mathematical problem statements: linear programming, constraint satisfaction problem (CSP), Boolean satisfiability (SAT). Usage of the combined method allows to avoid efficiency issues peculiar to the known approaches. At the first step the polynomial complexity methods of integer linear programming are applied to the original large-scale problem and localize the search region for solution by detection of the necessary amount of objects. At the next steps constraints imposed onto relatively small groups of objects can be considered individually, which allows to reduce significantly dimension of the problem. The key problem of estimation of the number of instances intended for generation of correct object collection and its reduction to an integer linear programming problem is investigated in detail. The performed experiments demonstrate prospectivity of the combined computational strategy and efficiency of the proposed method for verification of large-scale data models. The work is supported by RFBR (grant 13-07-00390).

Keywords: model verification, object-oriented modeling, UML/OCL, EXPRESS, constraint logic programming, linear integer programming, semantic reconciliation

References

[1] Model Driven Architecture: The Architecture of Choice for a Changing World. Executive Overview, October 2013. http://www.omg.org/mda/executive_overview.htm
[2] Calafat A.Q. Validation of UML Conceptual Schemas with OCL Constraints and Operators. PhD Thesis, adviced by Dr. E. Teniente, Universitat Politecnica de Catalunya, Barcelona, 2009.
[3] ISO 10303-11:2004. Industrial automation systems and integration — Product data representation and exchange — Part 11: Description methods: The EXPRESS language reference manual.
[4] Unified Modeling Language (UML), V2.4.1, Release Date: August 2011. http://www.omg.org/spec/UML/2.4.1

[5] Object Constraint Language (OCL), V2.3.1, Release Date: January 2012. http://www.omg.org/spec/OCL/2.3.1
[6] IFC4 Release Summary, March 2013. http://www.buildingsmart-tech.org/specifications/ifc-releases/ifc4-release/ifc4-release-summary
[7] Khemlani L. The CIS/2 Format: Another AEC Interoperability Standard. *AECbytes Newsletter*, July 27, 2005. http://www.aecbytes.com/buildingthefuture/2005/CIS2format.html
[8] ISO 10303-1:1994. Industrial automation systems and integration — Product data representation and exchange — Part 1: Overview and fundamental principles.
[9] ISO 15926-1:2004. Industrial automation systems and integration — Integration of life-cycle data for process plants including oil and gas production facilities — Part 1: Overview and fundamental principles.
[10] Chen P. The entity-relationship model — toward a unified view of data. *ACM Transactions on Database Systems*, vol. 1, no. 1, 1976, pp. 9–36.
[11] Alloy 4: a language & tool for relational models, 2012. http://alloy.mit.edu/alloy
[12] Formica A. Finite Satisfiability of Integrity Constraints in Object-Oriented Database Schemas. *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 1, 2002, pp. 123–139.
[13] Baader F., et al. The Description Logic Handbook: Theory, Implementation and Applications. *Cambridge University Press*, 2003.
[14] Lenzerini M., Nobili P. On the Satisfiability of Dependency Constraints in Entity-Relationship Schemata. *Proceedings of the 13th VLDB Conference*, 1987, pp. 147–154.
[15] Maraee A., Balaban M. Efficient Reasoning about Finite Satisfiability of UML Class Diagrams with Constrained Generalization Sets. *LNCS 4530, Proceedings of European Conference on Model Driven Architecture — Foundations and Applications*, 2007, pp. 17–31.
[16] Jackson D. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 2, 2002, pp. 266–290.
[17] Anastasakis K., Bordbar B., Georg G., Ray I. UML2Alloy: A challenging model transformation. *LNCS 4735, Model Driven Engineering Languages and Systems*, 2007, pp. 436–450.
[18] Ramirez A., Vanpeperstraete P., Rueckert A., Odutola K., Bennett J., Tolke L., van der Wulp W. ArgoUML User Manual: A tutorial and reference description, 2011. http://argouml.tigris.org/ documentation/manual-0.34
[19] Soeken M., Wille R., Kuhlmann M., Gogolla M., Drechsler R. Verifying UML/OCL Models Using Boolean Satisfiability. *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2010, pp. 1341–1344.
[20] Turhan A.Y. Description Logic Reasoning for Semantic Web Ontologies. *WIMS'11 Proceedings of the International Conference on Web Intelligence, Mining and Semantics*, Article 6, 2011.
[21] Berardi D., Calvanese D., de Giacomo G. Reasoning on UML Class Diagrams. *Artificial Intelligence*, vol. 168, no 1–2, 2005, pp. 70–118.
[22] The FaCT System, April 2003. http://www.cs.man.ac.uk/~horrocks/FaCT
[23] Haarslev V., Möller R. RACER system description. *LNAI 2083, Automated Reasoning*, 2001, pp. 701–705.
[24] van der Straeten R., Mens T., Simmonds J., Jonckers V. Using Description Logic to Maintain Consistency between UML Models. *LNCS 2863, The Unified Modeling Language. Modeling Languages and Applications*, 2003, pp. 326–340.
[25] Loom Project Home Page, July 2007. http://www.isi.edu/isd/LOOM

[26] PVS Specification and Verification System, January 2013. http://pvs.csl.sri.com
[27] Kyas M., Fecher H., de Boer F.S., Jacob J., Hooman J., van der Zwaag M., Arons T., Kugler H. Formalizing UML Models and OCL Constraints in PVS. *Electronic Notes in Theoretical Computer Science*, vol. 115, 2005, pp. 39–47.
[28] Brucker A.D., Wolff B. The HOL-OCL book. *Technical Report 525, ETH Zurich*, 2006.
[29] Nipkow T., Paulson L.C., Wenzel M. Isabelle/HOL — A Proof Assistant for Higher-Order Logic. *LNCS 2283*, 2002.
[30] Tsang E. Foundations of constraint satisfaction. Academic Press Limited, London & San-Diego, 1993.
[31] Cabot J., Clariso R., Riera D. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. *Proceedings of the 22nd ACM/IEEE International Conference on Automated Software Engineering (ASE '07)*, 2007, pp. 547–548.
[32] Cabot J., Clariso R., Riera D. Verification of UML/OCL Class Diagrams using Constraint Programming. *Proceedings of the IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW'08)*, 2008, pp. 73–80.
[33] Apt K.R., Wallace M.G. Constraint Logic Programming using ECLiPSe. *Cambridge University Press*, 2007.
[34] Gogolla M., Bohling J., Richters M. Validating UML and OCL models in USE by automatic snapshot generation. *Software & System Modeling*, vol. 4, no. 4, 2005, pp. 386–398.
[35] Shaikh A., Wiil U.K., Memon N. Evaluation of Tools and Slicing Techniques for Efficient Verification of UML/OCL Class Diagrams. *Advances in Software Engineering*, 2011, Article ID 370198, 18 p.p.
[36] Breu R., Hinkel U., Hofmann C., Klein C., Paech B., Rumpe B., Thurner V. Towards a Formalization of the Unified Modeling Language. *LNCS 1241, ECOOP'97 — Object-Oriented Programming*, 1997, pp. 344–366.
[37] Richters M., Gogolla M. On Formalizing the UML Object Constraint Language OCL. *LNCS 1507, Conceptual Modeling — ER'98*, 1998, pp. 449–464.
[38] Semenov V.A., Morozov S.V., Tarlapan O.A. Inkremental'naya verifikatsiya ob'ektno-orientirovannyh dannyh na osnove spetsifikatsii ogranichenij [Incremental verification of object-oriented data based on specification of constraints]. *Trudy ISP RAN* [*The Proceedings of ISP RAS*], vol. 8, no. 2, 2004, pp. 21–52 (in Russian).
[39] Semenov V.A., Eroshkin S.G., Karaulov A.A., Enkovich I.V. Semanticheskaya rekonsilyatsiya prikladnyh dannyh na osnove modelej [Model-based semantic reconciliation of applied data]. *Trudy ISP RAN* [*The Proceedings of ISP RAS*], vol. 13, no. 2, 2007, pp. 141–164 (in Russian).
[40] Vanderbei R.J. Linear Programming. Foundations and Extensions. Third edition. *Princeton University*, 2008.
[41] Shevchenko V.N., Zolotyh N.Yu. Linejnoe i tselochislennoe linejnoe programmirovanie [Linear and integer linear programming]. *Nizhniy Novgorod, N.I. Lobachevskiy State University Publ.*, 2004 (in Russian).
[42] Karmanov V.G. Matematicheskoe programmirovanie: uchebnoe posobie. 5-e izdanie [Mathematical programming: a tutorial. Fifth edition]. *Moscow, Fizmatlit Publ.*, 2004 (in Russian).

Снижение неоднозначности в оценке состояния объекта при управлении по прецедентам[1]

Л. Е. Карпов, В. Н. Юдин
ИСП РАН, Москва
{mak, yudin}@ispras.ru

Аннотация. Настоящая работа является продолжением ряда публикаций, посвященных исследованиям методов управления сложными объектами на основе прецедентов. В ситуации, когда трудно или невозможно получить точную математическую модель поведения объекта, целесообразно применять метод управления по прецедентам. Вместо модели мы можем опираться только на доступную информацию о состояниях объекта, управляющих воздействиях на него и результатах воздействий. Такой выбор в теории вывода по прецедентам соответствует трем составляющим понятия "прецедент": описанию проблемы, примененному решению и исходу – результату применения решения. Подход к управлению основан на разбиении состояний объектов управления на классы, состояния в каждом из которых эквивалентны друг другу с точки зрения управления. Состояние объекта управления сравнивается с прецедентами из заранее накопленной базы данных. Чтобы подобрать воздействие для текущего случая, на основе некоей меры близости в базе прецедентов отыскивается прецедент со схожим исходным и конечным состоянием. Из него заимствуется управляющее воздействие. В основе подходов к построению систем вывода по прецедентам лежит оценка схожести прецедента и текущего случая. В предыдущих публикациях был предложен метод оценки схожести, основанный на разбиении базы прецедентов на классы эквивалентности. Для оценки не полностью описанного случая используются проекции классов на его пространство признаков. Неполнота в описании объекта в условиях дефицита времени и ресурсов приводит к неоднозначности в оценке объекта. Классы, в пересечение которых попадает объект, образуют так называемый дифференциальный ряд объекта. Отсутствие разделяющего классы признака затрудняет выбор управляющего воздействия. Предлагаемая методика позволяет снизить эту неоднозначность, если принять во внимание предысторию объекта – его состояния и реакции на воздействия. Как следствие, ограничивается выбор управляющих воздействий. Описывается способ снижения неоднозначности оценки. Вводится понятие "прогнозируемый" дифференциальный ряд объекта. Итоговая оценка состояния выводится на основе сравнений текущего и прогнозируемого рядов.

[1] Работа поддержана грантами Российского фонда фундаментальных исследований № 12-01-00780, № 12-07-00214.

Ключевые слова: система управления, вывод по прецедентам, база прецедентов, сложный объект управления, неполнота описания, неоднозначность оценки, предыстория, дифференциальный ряд.

1. Введение

При управлении объектами часто возникает ситуация, принципиально отличающаяся от “классической”. Классические подходы к управлению строятся на том предположении, что можно получить аналитически заданную форму функциональной зависимости входных и выходных параметров управляемого объекта. Однако при всей изощренности наработанного математического аппарата, областью применения таких методов управления остаются сравнительно простые объекты с очевидными свойствами, то есть хорошо формализуемые объекты. На практике же часто встречаются объекты, которые формализуются плохо. Их свойства изначально плохо известны или изменяются в процессе функционирования. Однако управление такими объектами представляет не меньший интерес и является не менее важным, чем управление хорошо формализуемыми объектами.

При недостаточности наших знаний об объекте и среде, в которой он функционирует, невозможно получить точную модель поведения объекта. Вместо математической модели мы можем опираться только на доступную информацию о состояниях объекта, управляющих воздействиях на него и результатах воздействий, то есть на “прецедентах” управления. Такой выбор в теории вывода по прецедентам соответствует трем составляющим понятия прецедент: описанию проблемы, примененному решению и исходу - результату применения этого решения.

Вывод, основанный на прецедентах – это метод принятия решений, в котором используются знания о предыдущих ситуациях, или случаях. При таком выводе прецедент, если он признан схожим, часто является обоснованием решения. При рассмотрении новой проблемы (текущего случая) находится похожий прецедент в качестве аналога. Можно попытаться использовать его решение, возможно, адаптировав к текущему случаю, вместо того, чтобы вычислять решение каждый раз сначала. После того, как текущий случай будет обработан, он вносится в “базу прецедентов” вместе со своим решением для его возможного последующего использования.

Большая часть существующих подходов к построению систем вывода по прецедентам сосредоточена на одном аспекте: выборе наиболее подходящих прецедентов. В основе всех подходов к отбору лежит оценка схожести прецедента и текущего случая. Чаще всего – это подход на основе метрики, которая вводится в пространстве признаков. Однако в некоторых случаях ввести метрику не удается. Тогда вместо метрики используется так называемая мера близости.

На практике часто приходится принимать решение в условиях дефицита времени или ресурсов, когда текущий случай не полностью описан и оценивается неоднозначно. В медицине – это недостаток показателей больного (медицина катастроф, скорая помощь).

В [3] был впервые предложен алгоритм поиска наиболее подходящего прецедента, базирующийся на разбиении базы прецедентов на классы эквивалентности и оригинальной мере близости для не полностью описанных объектов. Для оценки используются проекции классов на пространство признаков текущего случая. Попадание случая в проекцию класса говорит о возможной принадлежности его к классу.

При оценке (распознавании) текущего случая часть его признаков по отношению к выбранным классам может отсутствовать. Одной из причин этого является недостаток информации в описании исследуемых объектов. Это приводит к тому, что объект может попасть в пересечение классов, т.е. неоднозначно оцениваться [1, 2].

2. Дифференциальный ряд

Исходным пунктом для разработки описанного метода послужило понятие “дифференциальная диагностика” в медицине. Недостаточно описанный случай может попасть в проекцию класса (заболевания), которому он не принадлежит, только потому, что у него не хватает признака, который дифференцировал бы его от этого класса. Классы, в пересечение которых попал объект, образуют так называемый “дифференциальный ряд” объекта. Искусство врача и заключается в том, чтобы найти недостающий признак, разделяющий классы при минимуме дополнительных исследований при ограничениях на время и ресурсы.

Разделяющие признаки можно позаимствовать у аналогов - прецедентов, которые в признаковом пространстве случая идентичны ему по принадлежности к классам, т.е. попадают в ту же область пересечения. Расстояние между текущим случаем и прецедентом определяется как разность количества классов, куда попал текущий случай, и количества классов из этого числа, в котором находится прецедент (*рис. 1*).

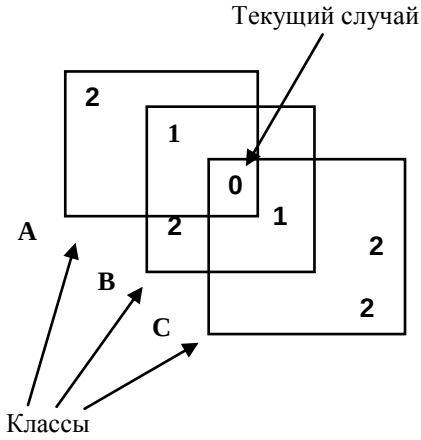


Рис. 1. Оценка близости для не полностью описанного объекта

Механизм вывода по прецедентам для управления объектами впервые был предложен в [4] и развит в дальнейших работах [16, 18]. Механизм основан на разбиении состояний объектов управления на классы, состояния в каждом из которых эквивалентны друг другу с точки зрения управления.

При таком подходе понятие цели управления неправильно было бы отождествлять с достижением конкретного состояния. Под целью здесь должно пониматься более широкое понятие: оптимальное поведение управляемого объекта, учитывающее переходы из одного класса состояний в другой, в частности, удержание в том же классе. Для этого необходимо найти алгоритм, обеспечивающий достижение цели за конечное число управляющих воздействий. В медицине при лечении хронических заболеваний обычно не ставится невыполнимая задача восстановления больного органа. Целью управления (лечебные процедуры, лекарственное воздействие, оперативное вмешательство) в подавляющем большинстве случаев является или замедление процесса дегенерации рабочей ткани – достижение ремиссии, или неотложные действия по сохранению жизни больного.

Предлагается следующая структура прецедента для управления (*рис. 2*):

1. Состояние объекта управления до воздействия. Дифференциальный ряд, образованный набором признаков этого состояния.
2. Управляющее воздействие. Описание воздействия. Как частный случай, возможно отсутствие воздействия.
3. Состояние после воздействия. Дифференциальный ряд, образованный новым набором признаков.

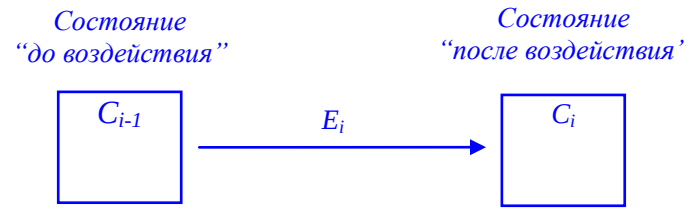


Рис. 2. Структура прецедента при управлении

Состояние объекта управления сравнивается с прецедентами из заранее накопленной базы данных (*рис. 3*). Чтобы подобрать воздействие для текущего случая, на основе описанной меры близости в базе прецедентов отыскивается прецедент со схожим исходным состоянием C_{i-1} (первая составляющая понятия “прецедент”), конечным состоянием C_i (третья составляющая понятия “прецедент”). Из него заимствуется управляющее воздействие (вторая составляющая), которое предположительно (по прогнозу) должно перевести наш объект в нужное состояние. Это используется напрямую или адаптируется к текущему случаю, исходя из степени близости прецедента. Результат воздействия также прогнозируется по прецеденту. Итог воздействия заносится в базу прецедентов для последующего использования.

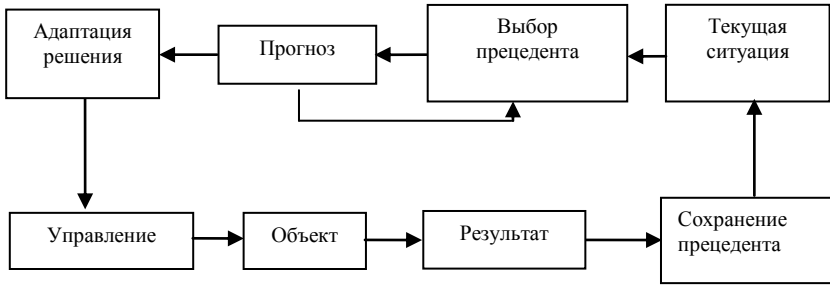


Рис. 3. Схема управления по прецедентам

Практика принятия решения, моделирующая человеческие рассуждения, применяется во многих областях человеческой деятельности. Человеческий организм как объект управления не может быть описан с помощью относительно простой математической модели, поэтому методы вывода по прецедентам в таких обстоятельствах могут рассматриваться как весьма перспективные в системах поддержки решений.

Медицина – прецедентная наука, но механизм прецедентов в явном виде не применяется при описании случаев в медицинской литературе. Примеры в медицине часто описываются с помощью правил “если-то”. Большое количество медицинской литературы в последнее время иллюстрируется ассоциативными правилами и деревьями решений. Последние являются частным случаем продукционных правил. Тем не менее, врач при постановке диагноза и выборе лечения в первую очередь использует прецедентный подход. Предложенный метод вывода по прецедентам был затем распространен на медицинские приложения и опробован в системе поддержки врачебных решений “Спутник Врача” [5-19].

2.1. Проблема неоднозначности и прогнозируемые ряды

Неоднозначность в оценке управляемого объекта делает невозможной гарантию того, что управляющее воздействие приведет объект в состояние, соответствующее цели управления на текущем шаге. На основе признаков текущего состояния можно построить дифференциальный ряд, и для каждого из классов этого ряда в базе прецедентов можно спрогнозировать воздействие, переводящее объект из этого класса в класс, соответствующий цели управления на текущем шаге. На самом же деле, приходится выбирать одно воздействие к объекту, исходя из предполагаемой, наиболее вероятной (по мнению исследователя), оценки – принадлежности к классу.

В медицине часто приходится действовать на свой страх и риск: при неполной идентификации состояния больного назначать лечение, которое может оказаться неадекватным с точки зрения стратегической цели лечения. Это, в первую очередь, воздействия, направленные на устранение опасности для жизни больного (например, интубация при обструкции дыхательных путей, ввод дофамина внутривенно при неудовлетворительной гемодинамике, лечение эпилептического статуса при подозрении на судорожный припадок). Во вторую – назначение лечения, когда точный диагноз предполагается, но не может быть пока установлен (например, противотуберкулезные назначения, несмотря на то, что диагностика этого заболевания часто затруднена). В последнем случае должно выбираться лечебное воздействие (в виде лекарств, процедур или хирургического вмешательства), которое, по прогнозу, применительно к любому из состояний дифференциального ряда, не должно привести к смерти или к серьезному ухудшению состояния больного.

После воздействия объект переходит в новое состояние, которое на основе его признаков можно описать новым дифференциальным рядом (*рис. 4*).

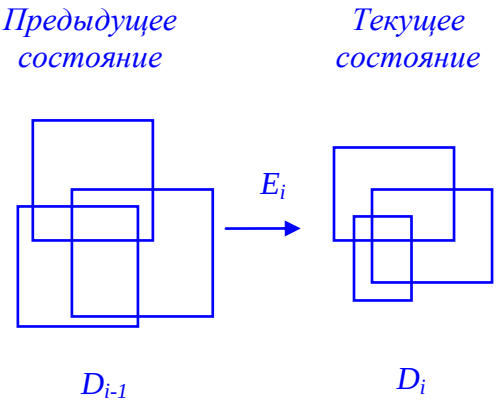


Рис. 4. Элементарный шаг управления с переходом в новый дифференциальный ряд

Обозначим текущее состояние объекта через i, а предыдущее – через $i–1$. Пусть

- D_{i-1} – дифференциальный ряд, основанный на признаках предыдущего, то есть $i–1$-го состояния объекта,
- D_i – дифференциальный ряд, основанный на признаках i-го состояния объекта,
- E_i – воздействие, которое перевело объект из $i–1$-го в i-е состояние.

На $i–1$–м шаге, несмотря на неоднозначную оценку состояния объекта, выбирается воздействие, которое исследователю показалось наиболее адекватным.

Казалось бы, какая связь этих двух состояний? Но следует учесть, что новое состояние, будь оно до конца не определено, все же вытекает из предыдущего. В этом случае даже переход объекта в состояние, которое не соответствует цели, может оказаться информативным, так как это может прояснить исходное состояние случая. Это можно сделать, сравнив предполагаемое состояние объекта после воздействия с реальным.

Смысл управления по прецедентам заключается в том, что выбор воздействия происходит на основе прогноза воздействия. Для каждого состояния, описываемого классом – элементом дифференциального ряда D_{i-1}, по базе прецедентов можно спрогнозировать результат виртуального применения этого воздействия (также класс). Если представить себе дифференциальный ряд, составленный из таких классов – это будет, вообще говоря, другой ряд. Назовём его “прогнозируемый” дифференциальный ряд i-го состояния и обозначим $D(E_i)$. (*рис. 5*).

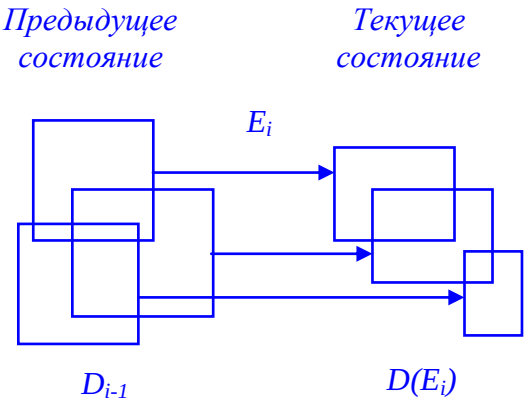


Рис. 5. Построение "прогнозируемого" дифференциального ряда

До воздействия мы можем рассчитать только прогнозируемый ряд $D(E_i)$. После воздействия – имеем также ряд D_i, построенный по признакам текущего, i-го состояния. Очевидно, что реальный класс, в котором находится объект после воздействия, находится в пересечении множеств $D(E_i) \cap D_i$. Если такую процедуру делать на каждом шаге управления объектом, начиная с первого, то вместо рядов D_k $(k = 1, \ldots i)$ будем иметь ряды $D(E_k) \cap D_k$, каждый из которых входит, либо (в крайнем случае) совпадает с рядом D_k. Если это не так, значит, база прецедентов недостаточно полна. Переход к прогнозируемым рядам позволит на каждом шаге управления в формуле $D(E_i) \cap D_i$ поддерживать всю предысторию состояния объекта, начиная с первого шага.

Новая методика управления позволяет снизить неоднозначность в оценке объекта. Если принять во внимание предысторию объекта – его состояния и реакции на воздействия – это может снизить степень неоднозначности оценки текущего состояния и ограничить выбор воздействий (достаточно вспомнить понятие "анамнез" в медицине). Особенность данной методики – в том, что управление объектом (оценка состояния и выработка управляющего воздействия) основывается не только на текущем состоянии, но и на предыстории поведения объекта.

Литература

[1] Юдин В.Н., «Система информационной поддержки врачебных решений, основанная на модифицированном методе динамического кластерного анализа», Труды Института системного программирования РАН (ИСП РАН), М., ИСП РАН, Т. 3, 2002, стр. 103-118, ISBN/ISSN: 2079-8156.

[2] V.N. Yudin, A.T. Bespaev, «Application of Cluster Analysis for Searching for Analogies in Diagnostics and Choice of Treatment in the "Doctor's Partner" System», Pattern Recognition and Image Analysis, vol. 13, No 2, 2003, pp. 387-390.
[3] В. Н. Юдин, «Мера близости в системе вывода на основе прецедентов», Доклады 12-й Всероссийской конференции Математические Методы Распознавания Образов (ММРО-12), МАКС Пресс, Москва 2005, стр. 241-244 .
[4] Л. Е. Карпов, В. Н. Юдин, «Адаптивное управление по прецедентам, основанное на классификации состояний управляемых объектов», Труды Института системного программирования РАН (ИСП РАН), т. 13, № 2, Институт системного программирования РАН, 2007, стр. 37-57, ISBN 5-89823-026-2. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), *http://www.ispras.ru/ru/proceedings/docs/2007/13/2/isp_2007_13_2_37.pdf, http://www.citforum.ru/consulting/BI/karpov/*
[5] Л. Е. Карпов, В. Н. Юдин, «Интеграция методов добычи данных и вывода по прецедентам в медицинской диагностике и выборе лечения», Математические методы распознавания образов. Сборник докладов 13-й Всероссийской конференции, октябрь 2007, МАКС Пресс, 2007, стр. 589-591, ISBN 978-5-317-02060-6, *http://www.mmro.ru/files/mmro13.pdf*
[6] В. Н. Юдин, Л. Е. Карпов, А. В. Ватазин, «Процесс лечения как адаптивное управление человеческим организмом в программной системе "Спутник врача"», Альманах клинической медицины, т. 17, № 1, МОНИКИ, 2008, стр. 262-265, ISBN 978-5-98511-032-6 (Т. XVII, ч. 1), ISBN 5-9900012-1-5, *http://www.isan.troitsk.ru/win/block1.pdf*
[7] В. Н. Юдин, Л. Е. Карпов, А. В. Ватазин, «Методы интеллектуального анализа данных и вывода по прецедентам в программной системе поддержки врачебных решений», Альманах клинической медицины, т. 17, № 1, МОНИКИ, 2008, стр. 266-269, ISBN 978-5-98511-032-6 (Т. XVII, ч. 1), ISBN 5-9900012-1-5, *http://www.isan.troitsk.ru/win/block1.pdf*
[8] Л. Е. Карпов, А. Н. Томилин, В. Н. Юдин, «Репликация и валидация в распределенной системе поддержки врачебных решений», Труды Всероссийской научной конференции "Научный сервис в сети Интернет: решение больших задач", МГУ, 2008, стр. 387-392, ISBN 978-5-211-05616-9, *http://agora.guru.ru/abrau2008/pdf/043.pdf*
[9] Л. Е. Карпов, В. Н. Юдин, А. В. Ватазин, «Виртуальная интеграция и консолидация знаний в распределенной системе поддержки врачебных решений», Научно-практическая конференция ЦФО РФ «Актуальные вопросы гемафереза, хирургической детоксикации и диализа», МОНИКИ, 2009, стр. 36. ISBN 978-5-98511-054-8.
[10] А. В. Ватазин, Л. Е. Карпов, В. Н. Юдин, «Виртуальная интеграция и консолидация знаний в распределенной системе поддержки врачебных решений», Альманах клинической медицины, т. 20, 2009, стр. 83-86. ISSN 2072-0505.
[11] А. В. Ватазин, Л. Е. Карпов, В. Н. Юдин, «Многопараметрическое управление сложным объектом в программной системе поддержки врачебных решений», III Евразийский конгресс по медицинской физике и инженерии "Медицинская физика – 2010", 21-25 июня 2010 г., т. 4, МОНИКИ, 2010, стр. 415-417.
[12] А. В. Ватазин, В. Н. Юдин, Л. Е. Карпов, «Многопараметрическое управление сложным объектом в программной системе поддержки врачебных решений», Ежегодная научно-практическая конференция Центрального Федерального округа РФ "Актуальные вопросы заместительной почечной терапии, гемафереза и

трансплантационной координации", МОНИКИ, 2010, стр. 8. ISBN 978-5-98511-091-3.
[13] Leonid Karpov, Valery Yudin, «The Case-Based Software System for Physician's Decision Support», Sami Khari, Lenka Lhotska, Nadia Pisanti (eds.), "Information Technology in Bio- and Medical Informatics, ITBAM 2010", Proceedings of the First International Conference, Bilbao, Spain. Lecture Notes in Computer Science Sublibrary: SL 3, Springer Verlag, Berlin, Heidelberg, 2010, pp. 78-85. ISSN 0302-9743.
[14] L. E. Karpov, V. N. Yudin, A. V. Vatazin, «Multi-Parametric Control of Complex Object in the Program System for Physician's Decision Support», Proceedings of the 12-th International Workshop on Computer Science and Information Technologies (CSIT'2010), Russia, Moscow – St. Petersburg, September 13-19, v. 1, Ufa State Aviation Technical University, 2010, pp. 28-30.
[15] Л. Е. Карпов, В. Н. Юдин, «Обмен данными в распределённой системе поддержки решений», Труды Института системного программирования, т. 19, 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*
[16] Л. Е. Карпов, В. Н. Юдин, «Многопараметрическое управление на основе прецедентов», Труды Института системного программирования, т. 19, Институт системного программирования РАН, 2010, стр. 81-93, 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_81.pdf*
[17] А. В. Ватазин, Л. Е. Карпов, Ю. Г. Сметанин, В. Н. Юдин, «Программная система поддержки врачебных решений с гибридной архитектурой на основе правил и прецедентов», V Троицкая конференция "Медицинская физика и инновации в медицине (ТКМФ-5)", Сборник материалов, том 2, стр. 425-427. 2012, РАН, Троицкий Научный Центр, ISBN 978-5-89513-272-2.
[18] Л. Е. Карпов, В. Н. Юдин, «Роль предыстории при оценке сложного объекта в управлении по прецедентам», Труды Института системного программирования РАН (ИСП РАН), т. 24, Институт системного программирования РАН, 2013, стр. 437-445, ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), *http://www.ispras.ru/ru/proceedings/docs/2013/24/isp_24_2013_437.pdf*
[19] В. Н. Юдин, Л. Е. Карпов, «Гибридный подход к построению систем поддержки решений», Труды Института системного программирования РАН (ИСП РАН), т. 24, Институт системного программирования РАН, 2013, стр. 447-456, ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), *http://www.ispras.ru/ru/proceedings/docs/2013/24/isp_24_2013_447.pdf*.

Lowing ambiguity level in object state estimation in a case-based control system[2]

L. E. Karpov, V. N. Yudin
ISP RAS, Moscow, Russia
{mak, yudin}@ispras.ru

Abstract. This work continues the publication series that is devoted to the investigation of methods of controlling complex objects basing on cases. In situation when it is hard or even impossible to use the exact mathematical model of object behavior the case-based reasoning control method becomes adequate. Instead of using the model we may use the accessible information about object under control state, controlling actions and their results. This means that we are using "cases". In the theory of deduction this idea corresponds to three constituents of the case-based reasoning: description of a problem, solution or action applied, and outcome that is the result of applying the solution. This approach is based on separating of object states into classes, in each of which all the states are equivalent to each other. After that the object under control state is comparing with cases in the case base which was collected in advance. In order to choose the controlling action for the current case some metrics of similarity is used to find a case with similar initial and final states in the case base. The controlling action is extracted from this case then. The way of estimating of similarity of the current case and its precedent is critical for a system that uses the case-based reasoning technique. In previous publications the method was presented that used to separate cases from the case base into classes of equivalence. To estimate not fully described case the projection of classes onto the features space was used. Incompleteness in object description brings to the ambiguity in object estimation especially while one has the lack of time and resources. Classes in the intersection of which the object appears, are forming so called differential set of the object. If there is no feature that may separate the classes the choice of controlling action is hardly being made. The method offered by authors helps to low the ambiguity level on the basis of object prehistory investigation (previous states found and actions used). As a result of taking the prehistory into account the number of possible choices of actions is decreasing. The way of lowing the ambiguity level is described in the article. The notion of "prognostic" differential set of object states is presented. The final state estimation is produced on the basis of comparison of the current and prognostic sets.

Key words: control system, case-based reasoning, case base, complex object, incompleteness of description, ambiguity of estimation, object behavior, prehistory, differential set.

References

[1] V. N. Yudin. Sistema informatsionnoj podderzhki vrachebnykh reshenij, osnovannaya na modifitsirovannom metode dinamicheskogo klasternogo analiza [The system of information support of physician's decisions based on modified method of dynamic

[2] This work is supported by Russian Fund for Basic Research, projects No 12-01-00780, No 12-07-00214.

cluster analysis], *Trudy ISP RAN* [*The Proceedings of ISP RAS*], vol. 3, 2002, pp. 103-118, ISBN/ISSN: 2079-8156 (in Russian).

[2] V.N. Yudin, A.T. Bespaev. Application of Cluster Analysis for Searching for Analogies in Diagnostics and Choice of Treatment in the "Doctor's Partner" System, Pattern Recognition and Image Analysis, vol. 13, No 2, 2003, pp. 387-390.

[3] V. N. Yudin. Mera blizosti v sisteme vyvoda na osnove pretsedentov. [The measure of closeness in the case-based reasoning system], Doklady 12-j Vserossijskoj konferentsii Matematicheskie Metody Raspoznavaniya Obrazov [Proc. of 12-th All-Russian conference Math. methods of pattern recognition], MAKS Press, 2005, pp. 241-244 (in Russian).

[4] L. E. Karpov, V. N. Yudin. Adaptivnoe upravlenie po pretsedentam, osnovannoe na klassifikatsii sostoyanij upravlyaemykh ob"ektov [Case-Based Reasoning adaptive control with classification of states of objects under control], *Trudy ISP RAN* [*The Proceedings of ISP RAS*], vol. 13, no. 2, 2007, pp. 37-57, ISBN 5-89823-026-2. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), *http://www.ispras.ru/ru/proceedings/docs/2007/13/2/isp_2007_13_2_37.pdf*, *http://www.citforum.ru/consulting/BI/karpov/* (in Russian).

[5] L. E. Karpov, V. N. Yudin. Integratsiya metodov dobychi dannykh i vyvoda po pretsedentam v meditsinskoj diagnostike i vybore lecheniya [Integration of data Mining and Case-Based Reasoning methods in medical diagnostics and treatment choosing], Sbornik dokladov 13-j Vserossijskoj konferentsii Matematicheskie metody raspoznavaniya obrazov [Proc. of 13-th All-Russian conference Math. methods of pattern recognition], October 2007, MAKS Press, 2007, pp. 589-591, ISBN 978-5-317-02060-6, *http://www.mmro.ru/files/mmro13.pdf* (in Russian).

[6] V. N. Yudin, L. E. Karpov, A. V. Vatazin. Protsess lecheniya kak adaptivnoe upravlenie chelovecheskim organizmom v programmnoj sisteme "Sputnik vracha" [Process of patient treatment as an adaptive control of human being organism in software system "Doctor's Partner"], Al'manakh klinicheskoj meditsiny [Almanac of Clinical Medicine], vol. 17, no. 1, МОНИКИ [Moscow Regional Scientific Research Clinical Institute], 2008, pp. 262-265, ISSN 2072-0505, ISBN 978-5-98511-032-6, ISBN 5-9900012-1-5, *http://www.isan.troitsk.ru/win/block1.pdf* (in Russian).

[7] V. N. Yudin, L. E. Karpov, A. V. Vatazin. Metody intellektual'nogo analiza dannykh i vyvoda po pretsedentam v programmnoj sisteme podderzhki vrachebnykh reshenij [Application of Data Mining and Case-Based Reasoning in software system for physician's decision support], Al'manakh klinicheskoj meditsiny [Almanac of Clinical Medicine], vol. 17, no. 1, MONIKI [Moscow Regional Scientific Research Clinical Institute], 2008, pp. 266-269, ISSN 2072-0505, ISBN 978-5-98511-032-6, ISBN 5-9900012-1-5, *http://www.isan.troitsk.ru/win/block1.pdf* (in Russian).

[8] L. E. Karpov, A. N. Tomilin, V. N. Yudin. Replikatsiya i validatsiya v raspredelennoj sisteme podderzhki vrachebnykh reshenij [Data replication and validation in distributed software system for physician's decision support], Trudy Vserossijskoj nauchnoj konferentsii "Nauchnyj servis v seti Internet: reshenie bol'shikh zadach" [Proc. All-Russian scientific conference "Scientific Service in Internet: solving of huge problems"], MGU [Moscow State University, 2008, pp. 387-392, ISBN 978-5-211-05616-9, *http://agora.guru.ru/abrau2008/pdf/043.pdf* (in Russian).

[9] L. E. Karpov, A. V. Vatazin, V. N. Yudin. Virtual'naya integratsiya i konsolidatsiya znanij v raspredelennoj sisteme podderzhki vrachebnykh reshenij [Virtual knowledge integration and consolidation in software system for physician's decision support], Trudy Nauchno-prakticheskaya konferentsiya TSFO RF «Aktual'nye voprosy gemafereza,

khirurgicheskoj detoksikatsii i dializa [Proc. of research and practical conference ‘Actual problems of hemapheresis, surgery detoxication and dialysis’], MONIKI [Moscow Regional Scientific Research Clinical Institute], 2009, pp. 36. ISBN 978-5-98511-054-8 (in Russian).

[10] A. V. Vatazin, L. E. Karpov, V. N. Yudin. Virtual'naya integratsiya i konsolidatsiya znanij v raspredelennoj sisteme podderzhki vrachebnykh reshenij [Virtual knowledge integration and consolidation in software system for physician's decision support], Al'manakh klinicheskoj meditsiny [Almanac of Clinical Medicine], vol. 20, 2009, pp. 83-86. ISSN 2072-0505 (in Russian).

[11] A. V. Vatazin, L. E. Karpov, V. N. Yudin. Mnogoparametricheskoe upravlenie slozhnym ob"ektom v programmnoj sisteme podderzhki vrachebnykh reshenij [Multiparametric object control in software system for physician's decision support], III Evrazijskij kongress po meditsinskoj fizike i inzhenerii "Meditsinskaya fizika – 2010" [Third Euro-Asia congress for medical physics], 21-25 of June 2010, vol. 4, MONIKI [Moscow Regional Scientific Research Clinical Institute], 2010, pp. 415-417 (in Russian).

[12] A. V. Vatazin, L. E. Karpov, V. N. Yudin. Mnogoparametricheskoe upravlenie slozhnym ob"ektom v programmnoj sisteme podderzhki vrachebnykh reshenij [Multiparametric object control in software system for physician's decision support], Ezhegodnaya nauchno-prakticheskaya konferentsiya TSentral'nogo Federal'nogo okruga RF "Aktual'nye voprosy zamestitel'noj pochechnoj terapii, gemafereza i transplantatsionnoj koordinatsii" [Proc. of Annual research and practical conference 'Actual problems of replacement therapy, hemapheresis, and transplantation coordination'], MONIKI [Moscow Regional Scientific Research Clinical Institute], 2010, стр. 8. ISBN 978-5-98511-091-3. (in Russian).

[13] Leonid Karpov, Valery Yudin. The Case-Based Software System for Physician's Decision Support. Sami Khari, Lenka Lhotska, Nadia Pisanti (eds.), "Information Technology in Bio- and Medical Informatics, ITBAM 2010", Proceedings of the First International Conference, Bilbao, Spain. Lecture Notes in Computer Science Sublibrary: SL 3, Springer Verlag, Berlin, Heidelberg, 2010, pp. 78-85. ISSN 0302-9743.

[14] L. E. Karpov, V. N. Yudin, A. V. Vatazin. Multi-Parametric Control of Complex Object in the Program System for Physician's Decision Support, Proceedings of the 12-th International Workshop on Computer Science and Information Technologies (CSIT'2010), Russia, Moscow – St. Petersburg, September 13-19, v. 1, Ufa State Aviation Technical University, 2010, pp. 28-30.

[15] L. E. Karpov, V. N. Yudin. Obmen dannymi v raspredelyonnoj sisteme podderzhki reshenij [Data exchange in distributed software system for decision support], *Trudy ISP RAN* [*The Proceedings of ISP RAS*], vol. 19, 2010, pp. 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* (in Russian).

[16] L. E. Karpov, V. N. Yudin. Case-based multi-parametric object control, *Trudy ISP RAN* [*The Proceedings of ISP RAS*], vol. 19, 2010, pp. 81-93, 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_81.pdf* (in Russian).

[17] A. V. Vatazin, L. E. Karpov, Y. G. Smetanin, V. N. Yudin. Programmnaya sistema podderzhki vrachebnykh reshenij s gibridnoj arkhitekturoj na osnove pravil i pretsedentov [Software system for physician's decision support with architecture based on rules and cases], V Troitskaya konferentsiya "Meditsinskaya fizika i innovatsii v meditsine (TKMF-5)", Sbornik materialov [Proc. of Fifth conference 'Medical physics

and innovations in medicine'], vol. 2, pp. 425-427. 2012, RAS, Troitsk Scienfic Centre, ISBN 978-5-89513-272-2 (in Russian).

[18] L. E. Karpov, V. N. Yudin. Rol' predystorii pri otsenke slozhnogo ob"ekta v upravlenii po pretsedentam [State prehistory for complex object estimation in a control system based on cases], *Trudy ISP RAN* [*The Proceedings of ISP RAS*], vol. 24, 2013, pp. 437-445, ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), *http://www.ispras.ru/ru/proceedings/docs/2013/24/isp_24_2013_437.pdf* (in Russian).

[19] V. N. Yudin, L. E. Karpov. Gibridnyj podkhod k postroeniyu sistem podderzhki reshenij [Hybrid approach to building decision support system], *Trudy ISP RAN* [*The Proceedings of ISP RAS*], vol. 24, 2013, pp. 447-456, ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), *http://www.ispras.ru/ru/proceedings/docs/2013/24/isp_24_2013_447.pdf* (in Russian).

Двусторонняя унификация программ и ее применение для задач рефакторинга

Т.А. Новикова
Казахстанский филиал МГУ им. М.В. Ломоносова, Астана, Казахстан
taniaelf@mail.ru
В.А.Захаров
ИСП РАН, Москва, Россия
zakh@cs.msu.su

Аннотация. Задача унификации пары подстановок θ_1 и θ_2 состоит в вычислении такой пары подстановок η' и η'', чтобы композиции $\theta_1\eta'$ и $\theta_2\eta''$ были равны. По существу, задача унификации подстановок равносильна задаче решения линейных уравнений вида $\theta_1 X = \theta_2 Y$ в полугруппе подстановок. Но некоторые линейные уравнения над подстановками также можно рассматривать как новые варианты задачи унификации. В этой статье мы вводим понятие двусторонней унификации как процесса преобразования одной заданной подстановки θ_1 к другой заданной подстановке θ_2 при помощи композиции, применяемой как справа, так и слева к подстановке θ_1. Иначе говоря, задача двусторонней унификации состоит в решении уравнений вида $X\theta_1 Y = \theta_2$. Двусторонняя унификация подстановок может быть использована при решении одной из задач реорганизации (рефакторинга) программ – выделения в заданном фрагменте кода тела библиотечной процедуры с целью последующей замены выделенного участка кода на вызов этой процедуры. В статье исследован вопрос о сложности задачи двусторонней унификации подстановок. Установлено, что эта задача является NP-полной. Доказательство NP-трудности задачи двусторонней унификации проводится путем сведения к ней NP-полной задачи правильного расположения домино в прямоугольной области плоскости. В статье также сформулирована и исследована задача двусторонней унификации программ в модели программ первого порядка с отношением логико-термальной эквивалентности. Доказано, что сформулированная задача двусторонней унификации программ также является NP-полной.

Ключевые слова: программа, рефакторинг, логико-термальная эквивалентность, подстановка, композиция, унификация, сложность, проблема домино.

1. Введение

Данная статья продолжает серию работ [1-4], в которых исследуется применимость методов и алгоритмов теории унификации [5] для решения задач оптимизации и верификации программ. В статьях [2] предложены

эффективные алгоритм проверки логико-термальной эквивалентности в ранее известной модели последовательных императивных программ [6,7], сложность которого существенно меньше сложности ранее известных алгоритмов (см. [8,9]). На основе этого алгоритма в статье [3] был предложен полиномиальный по времени алгоритм унификации программ, т.е. приведения двух программ к общему виду за счет выбора подходящих инициализаций входных переменных. Ранее задача унификации программ не исследовалась. В обоих случаях эффективность предложенных алгоритмов анализа и преобразования программ была обусловлена использованием быстрых алгоритмов унификации (вычисления наиболее общих примеров) и антиунификации (вычисления наиболее специальных шаблонов) заданных выражений (термов и атомов).

В данной работе рассматривается еще одна задача анализа и преобразования программ, относящаяся к области реорганизации (рефакторинга) программ [10]. Цель реорганизации программы состоит в том, чтобы за счет эквивалентных преобразований, сохраняющих функциональные свойства программы, привести ее к такому виду, который облегчает понимание программы, имеет более простую структуру, меньший размер, лишен избыточных конструкций и т. п. В частности, одной из задач реорганизации программ является задача обнаружения и устранения клонов. Содержательно, программный клон – это совокупность фрагментов программы, осуществляющих «похожие» преобразования данных. Корректное определение программного клона, сопровождаемое эффективным методом обнаружения клонов, позволяет проводить упрощения программного код путем замены нескольких больших фрагментов программы вызовом одной и той же процедуры или макроса [12]. Такое преобразование сокращает размер программы. Но оно также и облегчает понимание программы, поскольку понимание поведения нескольких разных фрагментов программы теперь сводится к пониманию поведения одной процедуры. Устранение клонов также упрощает тестирование и анализ программы, поскольку однородные ошибки, присущие фрагментам одного и того же клона, можно обнаружить и устранить при анализе поведения одной процедуры.

Попытки разработать и реализовать подходящие средства обнаружения и устранения клонов предпринимались во многих работах [13]. Главную трудность здесь составляют два вопроса: как определить понятие схожести вычислений фрагментов программ (определение клона), и как обнаружить фрагменты, имеющие сходное поведение (выявление клона). В частности, в статье [3] был предложен следующий подход к решению этих вопросов.

Предположим, что на множестве программ введено некоторое отношение эквивалентности и выделен некоторый класс «простых» программ Π. Тогда одно из возможных определений клона можно сформулировать, введя отношение Π-подобия программ. Заданная пара программ π_1 и π_2 считается Π-подобной, если существует такая программа π_0 и две такие пары программ

(ρ_1, λ_1) и (ρ_2, λ_2) из класса Π, для которых последовательная композиция программ $\rho_1; \pi_0; \lambda_1$ эквивалентна π_1, а последовательная композиция программ $\rho_2; \pi_0; \lambda_2$ эквивалентна π_2. Программу π_0 указанного вида, назовем Π-*ядром* пары программ π_1, π_2. Пары программ (ρ_1, λ_1) и (ρ_2, λ_2) в приведенном определении могут мыслиться как интерфейсы, преобразующие формат представления входных и выходных данных. Можно предполагать, что преобразования такого рода выполняются программами, имеющими сравнительно простое устройство. Тогда наличие ядра у пары программ означает, что эти программы вычисляют схожие функции и могут считаться подобными.

Аналогичным образом понятие ядра можно распространить и на целые семейства программ. Тогда Π-клоном называется семейство программ, имеющих Π-ядро. При обнаружении клона мы можем ввести в программе новую процедуру α, телом которой служит фрагмент π_0, и все фрагменты клона π_i , $1 \le i \le n$, заменить вызовом этой процедуры в составе композиций вида $\rho_i; call\ \alpha;\ \lambda_i$.

Для реализации этого подхода нужно уметь решать задачу проверки подобия программ: для произвольной заданной пары программ π_1 и π_2 выяснить существование ядро относительно заданного семейства интерфейсных программ Π. А решение этой задачи, в свою очередь, опирается на решение более простой задачи: для заданной пары программ π_1 и π_0 требуется выяснить, существует ли такая пара программ (ρ, λ) из класса Π, для которой программа π_1 эквивалентна композиции программ $\rho; \pi_0; \lambda$. Поскольку интерфейсные программы ρ, λ приводят программы π_1 и π_0 к общему виду, пару (ρ, λ) назовем *двусторонним унификатором* программ π_1 и π_0.

Понятие двусторонней унификации вводится нами впервые. Оно является естественным развитием широко известной концепции унификации для логических выражений, содержащих переменные. Задача унификации выражений (формул, атомов, термов) языка предикатов первого порядка состоит в том, чтобы для заданной пары выражений $E_1(x_1, \dots, x_n)$ и $E_2(x_1, \dots, x_n)$ отыскать такую подстановку $\theta = \{x_1/t_1, \dots, x_n/t_n)\}$, для которой выражения $E_1(x_1, \dots, x_n)\theta$ и $E_2(x_1, \dots, x_n)\theta$ становятся синтаксически одинаковыми. Впервые задачу унификации исследовал Дж. Робинсон в статье [13] в рамках разработки метода резолюций для систем автоматического доказательства теорем. В дальнейшем метод резолюций послужил отправной точкой для разработки концепции логического программирования, и алгоритмы унификации фактически стали основным средством вычисления логических программ. За прошедшие годы задача унификации была детально исследована. Было обнаружено, что унификация, помимо логического программирования, может быть использована при разработке систем автоматического доказательства теорем, обработке текстов естественных и

формальных языков, построении систем переписывания термов (см. обзор [14]). Был разработан широкий спектр эффективных алгоритмов унификации [15-20], имеющих почти линейную сложность, а также были найдены подходящие структуры данных для практической реализации этих алгоритмов.

Наиболее важным параметром в формулировке задачи унификации программ является отношение эквивалентности программ. Содержательный смысл задачи унификации требует, чтобы это отношение аппроксимировало отношение функциональной эквивалентности. В то же время, для эффективного решения задачи унификации необходимо, чтобы это выбранное отношение эквивалентности было разрешимым. Обоим требованиям удовлетворяет отношение логико-термальной (л-т) эквивалентности, введенное в статье [8]. Две программы π_1 и π_2 считаются л-т эквивалентными, если для любой синтаксически допустимой трассы $trace'$ в одной из программ существует такая трасса $trace''$ в другой программе, что в обеих трассах логические условия (предикаты) проверяются в одной и той же последовательности для одних и тех же наборов значений переменных. Как было установлено в [8], л-т эквивалентность программ влечет их функциональную эквивалентность в любой интерпретации базовых операций и предикатов. В статье [9] показано, что проверку л-т эквивалентности программ можно провести за время, полиномиально зависящее от размеров программ. Еще более быстрый алгоритм проверки л-т эквивалентности программ был предложен в статьях [2].

Цель настоящей статьи – оценить сложность задачи двусторонней унификации программ. Основным результатом статьи является теорема, показывающая, что задача проверки двусторонней унифицируемости программ относительно логико-термальной эквивалентности является NP-полной задачей.

Содержание статьи таково. Во втором разделе приведены определения основных понятий алгебры конечных подстановок [5,21], в терминах которых сформулирована задача двусторонней унификации для конечных подстановок. В этом же разделе показано, что эта задача может быть решена недетерминированным алгоритмом за полиномиальное время. Чтобы установить NP-полноту задачи двусторонней унификации подстановок, в третьем разделе рассмотрена задача о правильном расположении домино (мозаики) на ограниченной области плоскости. Эта задача, известная под названием Bounded Tiling Problem, является NP-полной задачей; она была сформулирована и исследована в статье [22]. Эта задача часто используется в качестве эталонной NP-полной задачи в теории сложности. В четвертом разделе показано, что задача о правильном расположении домино сводится к задаче проверки двусторонней унифицируемости подстановок; тем самым устанавливается NP-полнота исследуемой задачи. На основании этого результата в пятом разделе статьи показано, что задача двусторонней

унифицируемости в модели последовательных императивных программ с отношением логико-термальной эквивалентности также является NP–полной. В заключении статьи обсуждаются практические способы вычисления двусторонних унификаторов программ и методы выделения ядра у пар подобных программ.

2. Двусторонняя унификация конечных подстановок

Определим понятие конечной подстановки первого порядка и операции композиции подстановок. Пусть задано некоторое множество функциональных символов F. Символы U, X, Y, Z будем использовать для обозначения конечных множеств переменных. Множество термов $Term[X]$ над множеством переменных X определяется общепринятым для языка первого порядка образом.

Пусть $X = \{x_1, \dots, x_n\}$ и $Y = \{y_1, y_2, \dots\}$. Тогда *X-Y-подстановкой* называется всякое отображение $\theta: X \to Term[Y]$. Каждая подстановка может быть описана конечным множеством *связок* $\theta = \{x_1/\theta(x_1), \dots x_n/\theta(x_n)\}$. Переменные множества Y будем называть *входными переменными*, а переменные множества X – *выходными переменными* подстановки θ. Множество всех X-Y-подстановок обозначим записью $Subst[X, Y]$. Применение подстановки θ к терму $t(x_1, \dots, x_n)$ дает в результате терм $t\theta = t\big(\theta(x_1), \dots, \theta(x_n)\big)$, который получается из терма t одновременной заменой всех вхождений каждой переменной x_i, $1 \le i \le n$, на терм $\theta(x_i)$. *Композицией* X-Y-подстановки θ и Y-Z-подстановки η называется такая X-Z-подстановка ξ, которая удовлетворяет равенству $x\xi = (x\theta)\eta$ (или, иначе, $\xi(x) = \big(\theta(x)\big)\eta$) для каждой переменной x, $x \in X$. Для обозначения композиции подстановок θ и η будем использовать запись $\theta\eta$. Поскольку равенство $t(\theta\eta) = (t\theta)\eta$ справедливо для любого терма $t, t \in Term[X]$, операция композиции подстановок является ассоциативной. Будем называть X-X-подстановку θ *переименованием*, если отображение θ является биекцией на множестве переменных X. Две подстановки θ_1, θ_2 из семейства $Subst[Z, X]$ считаются эквивалентными, если $\theta_1 = \theta_2\rho$ для некоторого X-X-переименования ρ. Если подстановка θ_1 является композицией подстановок θ_2 и η, то подстановку θ_1 назовем *примером* подстановки θ_2, а подстановку θ_2 будем называть *шаблоном* подстановки θ_1.

Пусть заданы X-Y-подстановка θ_0 и Z-U-подстановка θ_1. Тогда пару η' и η'' из множеств $Subst[Z, X]$ и $Subst[Y, U]$ соответственно назовем *двусторонним унификатором* пары (θ_0, θ_1) тогда и только тогда, когда выполняется равенство $\eta'\theta_0\eta'' = \theta_1$. Задача двусторонней унификации состоит в том, чтобы для заданной пары подстановок указанного типа θ_0 и θ_1 вычислить двусторонний унификатор (η', η'') пары (θ_0, θ_1).

Отметим некоторые особенности задачи двусторонней унификации. Прежде всего, нетрудно увидеть, что эта задача фактически состоит в решении уравнения $\boldsymbol{X}'\theta_0\boldsymbol{X}'' = \theta_1$ относительно неизвестных подстановок $\boldsymbol{X}', \boldsymbol{X}''$ из семейств $Subst[Z, X]$ и $Subst[Y, U]$соответственно. Неявно заданные уравнения подобного вида ранее уже исследовались в теории унификации. Так, например, традиционная задача унификации выражений в языке первого порядка (см. [5]) может быть представлена как задача решения уравнений вида $\theta_0\boldsymbol{X}' = \theta_1\boldsymbol{X}''$. Задача односторонней унификации возникает во многих разделах математической логики, алгебры, теории вычислений, программирования, теории искусственного интеллекта. Подробнее с описанием прикладных возможностей односторонней унификации можно ознакомиться в статье [14]. Для ее решения было предложено немало эффективных алгоритмов (см. [15-20]), некоторые из которых вычисляют односторонний унификатор за почти линейное время относительно размера описания подстановок, представленных в виде размеченных ориентированных ациклических графов. Уравнения другого вида $\boldsymbol{X}'\theta_0 = \boldsymbol{X}''\theta_1$ также неявно возникали в работах [23], в которых исследовалась проблема эквивалентности в одном классе последовательных программ. Было показано, что решение этих уравнений можно вычислить за полиномиальное время. В связи с эти представляет интерес не только сама задача двусторонней унификации подстановок, но также и более общая задача проверки разрешимости и вычисления решения уравнений более общего вида над подстановками. Не исключено, что некоторые из этих уравнений могут иметь прикладное значение.

Следует заметить, что задача двусторонней унификации, в отличие от задачи односторонней унификации, является асимметричной, поскольку и сами исходные подстановки θ_0, θ_1, и подстановки двустороннего унификатора η', η'' относятся к разным типам и играют разную роль в содержательной интерпретации задачи унификации. Подстановки θ_0, θ_1 моделируют вычисления программ, а подстановки η'' и η' выполняют инициализацию входных данных и специализацию результатов вычисления. Поэтому, как следует из определения двусторонней унификации, термы подстановки η'' не могут непосредственно влиять на входные переменные подстановки η', но только через термы подстановки θ_0.

И, наконец, задача двусторонней унификации для подстановок θ_0, θ_1 может иметь несколько разных решений. Например, в случае $\theta_0 = \{x_1/f(y_1, y_2), x_2/y_3\}$, $\{\theta_1 = \{z/f(f(u,u), f(u,u))\}$ двусторонними унификаторами являются пары $(\eta' = \{z/f(x_1, x_1)\}, \eta'' = \{y_1/u, y_2/u\})$, $(\eta' = \{z/f(f(x_2, x_2), x_1)\}, \eta'' = \{y_1/u, y_2/u\})$ и $(\eta' = \{z/x_1\}, \eta'' = \{y_1/f(u,u), y_2/f(u,u)\})$. Однако множество попарно неэквивалентных двусторонних унификаторов заданной пары подстановок θ_0, θ_1 конечно, поскольку первая компонента η' каждого такого

унификатора является шаблоном подстановки θ_1, а число попарно неэквивалентных шаблонов всякой подстановки конечно.

Когда затрагиваются вопросы сложности задач и алгоритмов их решения, большое значение имеют способы описания задачи и структуры данных, с которыми работают алгоритмы. Например, задача односторонней унификации имеет экспоненциальную сложность, если для представления термов используются размеченные деревья, но эта же задача решается за почти линейное время, если для представления термов использовать размеченные ориентированные ациклические графы. Как будет видно из последующих разделов статьи, выбор представления термов не оказывает существенного влияния на сложность задачи двусторонней унификации. Для определенности мы будем в дальнейшем считать, что все термы в подстановках θ_0, θ_1 имеют древесное представление. Условимся обозначать записью T_θ множество размеченных деревьев (лес), представляющих все термы подстановки θ.

Очевидно, что в этом случае задача проверки двусторонней унифицируемости подстановок (θ_0, θ_1) может быть решена недетерминированным алгоритмом за полиномиальное время. Для решения этой задачи достаточно:

1. Провести недетерминированно образом по два сечения в каждом из деревьев множества T_{θ_1}, разделив лес T_{θ_1} на три части T', T_0 и T''; при этом листовые вершины каждой из частей T', T_0 совмещаются с корневыми вершинами частей T_0 и T'' соответственно;
2. Согласованно приписать переменные из множеств X и Y всем листовым вершинам фрагментов T' и T_0 соответственно (одинаковые переменные могут быть приписаны разным листовым вершинам, только если эти листья совместимы с корневыми вершинами одинаковых деревьев лежащих в следующем слое разреза).
3. Проверить, что все помеченные таким образом деревья из среднего фрагмента T_0, представляют термы из подстановки θ_0.

Очевидно, что проверить согласованность разметки листовых вершин, а также включение термов из фрагмента T_0 в древесное представление подстановки T_{θ_0} можно за время, полиномиальное относительно размеров подстановок T_{θ_0} и T_{θ_1}. Таким образом, справедлива

Лемма 1. Задача проверки двусторонней унифицируемости подстановок (θ_0, θ_1) принадлежит классу сложности NP.

NP-трудность этой задачи будет обоснована в разделе 4 путем сведения к ней ограниченного варианта проблемы домино (bounded tiling problem), которая формально описана в разделе 3.

3. Проблема ограниченного домино

Проблема домино состоит в том, чтобы покрыть заданную область плоскости четырехсторонними квадратными домино заданных типов так, чтобы смежные стороны двух соседних квадратов имели одинаковую окраску. Впервые эта задача была рассмотрена в статье [24]. Сложность этой задачи существенно зависит от формы покрываемой области. Например, задача покрытия всей плоскости посредством домино заданных типов алгоритмически неразрешима; в статье [25] было установлено, что к этому варианту проблемы домино сводится проблема (не)останова машин Тьюринга. В статье [22] было показано, что задача правильного покрытия прямоугольной области является NP-полной. В дальнейшем этот вариант проблемы домино под названием Bounded Tiling Problem широко использовался в математической логике и теории вычислений для доказательства NP-полноты многих задач. В частности, в монографии [26] проблема домино выступает в роли центральной задачи теории сложности вычислений. Мы также используем проблему домино для доказательства NP-полноты задачи двусторонней унифицируемости.

Неформальное описание ограниченного варианта проблемы домино таково. Предположим, что задано некоторое конечное множество домино $\{T_1, T_2, \dots, T_L\}$, представляющих собой квадраты единичного размера, каждая сторона которых окрашена в какой-либо цвет. Рассмотрим прямоугольник размера $n \times m$, стороны которого разделены на отрезки единичной длины. Предположим, что каждый из этих сегментов также окрашен какой-то цвет. Задача состоит в том, чтобы выяснить, можно ли расположить в указанном прямоугольнике домино заданных типов так, чтобы смежные стороны соседних домино имели одинаковую окраску, и чтобы стороны домино, прилегающие к границе прямоугольника, имели ту же окраску, что и сегменты границы, смежные с этими сторонами.

Опишем теперь эту задачу формально. Пусть задано конечное множество *цветов* $Colours = \{1,2, \dots, K\}$. *Домино* – это четверка цветов $tile = \langle a_1, a_2, a_3, a_4 \rangle$. Для выделения компонентов домино $tile$ (в указанном порядке) будем использовать записи $tile[0,-1], tile[-1,0], tile[0,1], tile[1,0]$, которые обозначают соответственно окраску северной, восточной, южной и западной сторон домино. Прямоугольная область размера $n \times m$ – это множество пар $Area = \{(i,j) : 0 \le i \le n+1, 0 \le j \le m+1\}$; элементы этого множества называются *квадратами*. Множество квадратов $Inter = \{(i,j) : 1 \le i \le n, 1 \le j \le m\}$ называется *внутренней областью* прямоугольника. *Граница* прямоугольника – это множество пар $Border = Area \setminus Interior$. Два квадрата (i_1, j_1) и (i_2, j_2) прямоугольника $Area$ считаются *соседними*, если $|i_1 - i_2| + |j_1 - j_2| = 1$. *Граничными условиями* называется всякое отображение $B: Border \to Colours$. Равенство $B(i,j) = a$ означает, что сторона граничного квадрата (i,j), прилегающая к внутренней области

прямоугольника, окрашена в цвет a. Пусть задано конечное множество типов домино $Tiles = \{tile_1, \dots, tile_L\}$. Тогда *покрытием* прямоугольника $Area$ называется всякое отображение $T: Inter \to Tiles$. Для заданного граничного условия B прямоугольника $Area$ покрытие T называется -*правильным*, если оно удовлетворяет следующим требованиям:

1. Для каждой пары соседних внутренних квадратов (i_1, j_1) и (i_2, j_2) справедливо равенство $T(i_1, j_1)[i_1 - i_2, j_1 - j_2] = T(i_2, j_2)[i_2 - i_1, j_2 - j_1]$: смежные стороны соседних домино одинаково окрашены;
2. Для каждого внутреннего квадрата (i_1, j_1), соседом которого является граничный квадрат (i_2, j_2), справедливо равенство $T(i_1, j_1)[i_1 - i_2, j_1 - j_2] = B(i_2, j_2)$: сторона домино, прилегающая к границе области, имеет ту же окраску, что и сегмент этой границы.

Примером ограниченной проблемы домино называется набор $BT = (n, m, Tiles, B)$. Этот пример считается *допустимым,* если существует -правильное покрытие прямоугольной области $Area$ размера $n \times m$ домино из множества $Tiles$. Задача Bounded Tiling Problem состоит в том, чтобы для произвольного примера ограниченной проблемы домино определить, является ли этот пример допустимым. В статье [22] показано, что проблема ограниченного домино является NP-полной.

4. NP-полнота проблемы двусторонней унифицируемости подстановок

Пусть задан произвольный пример ограниченной проблемы домино $BT = (n, m, Tiles, B)$, где $Tiles = \{tile_1, \dots, tile_L\}$, $Colours = \{1,2, \dots, K\}$. Для доказательства NP-полноты проблемы двусторонней унифицируемости подстановок покажем, что для примера BT ограниченной проблемы домино можно построить такую пару подстановок (θ_0, θ_1), которые двусторонне унифицируемы в том и только том случае, когда этот пример является допустимым. Связки X-Y-подстановки θ_0 представляют граничные условия B, а также все возможные примеры размещений домино в различных квадратах области $Area$. А Z-U-подстановка θ_1 состоит из единственной связки, которая представляет описание правильного покрытия прямоугольной области одноцветными домино, все стороны которых окрашены в цвет K. Двусторонний унификатор (η', η'') описывает решение примера BT: связки подстановки η' описывают некоторое покрытие T области $Area$, а вторая компонента унификатора η'' проверяет правильность предложенного

покрытия. Проверка правильности покрытия состоит в том, что подстановка η'' предпринимает попытку «перекрасить» стороны домино, увеличивая цвет каждой пары смежных сторон домино на одну и ту же величину. При таком способе «перекраски» достичь монохроматического покрытия удается в том и только том случае, когда исходное покрытие было правильным.

Чтобы определить формально подстановки θ_0 и θ_1 формально, мы введем множество функциональных символов F, включающее

- двухместный функциональный символ $g^{(2)}$; с его помощью будет построен терм, описывающий область покрытия $Area$;
- шестиместный функциональный символ $h^{(6)}$; с его помощью строятся термы, представляющие домино и граничные квадраты области покрытия;
- одноместный функциональный символ $f^{(1)}$; он позволяет строить термы (нумералы), представляющие натуральные числа, которыми обозначаются цвета и квадраты области покрытия.

Подстановки θ_0 и θ_1 оперируют над следующими множествами переменных

1. $X = X' \cup X''$, где
   - $X' = \{x'_{i,j} : (i,j) \in Border\}$: каждая переменная $x'_{i,j}$ привязана к граничному квадрату (i,j) области $Area$,
   - $X'' = \{x''_{i,j,\ell} : (i,j) \in Interior, 1 \le \ell \le L\}$: каждая переменная $x''_{i,j,\ell}$ ассоциируется с размещением домино $tile_\ell$ в квадрате (i,j) внутренней части области $Area$;
2. $Y = \{y_0\} \cup Y'$, где

$$Y' = \{y_{i_1,j_1,i_2,j_2} : 0 \le i_1 \le i_2 \le n+1, 0 \le j_1 \le j_2 \le m+1, |i_1 - i_2| + |j_1 - j_2| = 1\}$$

   - : каждая переменная y_{i_1,j_1,i_2,j_2} ассоциирована с парой соседних квадратов (i_1,j_1) и (i_2,j_2) области покрытия Area;
   - y_0 – вспомогательная переменная, используемая для нумерации координат квадратов;

3. $Z = \{z\}$, и $U = \{u\}$.

При помощи функционального символа $f^{(1)}$ определим рекурсивно *нумералы* – термы вида $f_n(y)$ – для каждого целого неотрицательного n следующим образом: $f_0(y) = y$ и $f_{n+1}(y) = f(f_n(y))$ для каждого n, $n \geq 0$. Очевидно, что для любых n, m верно $f_n(f_m(y)) = f_{n+m}(y)$. Число n будем называть *показателем* нумерала $f_n(y)$.

Термы, которые служат описаниями граничных условий B и размещений домино во внутренних квадратах области покрытия, определяются так.

Если граничный квадрат (i, j) располагается в одном из углов прямоугольной области покрытия, т.е. $(i, j) \in \{(0,0), (n+1,0), (n+1, m+1), (0, m+1)\}$, то этому квадрату в подстановке θ_0 сопоставляется связка $x_{i,j}/t_{i,j}$, где

$$t_{i,j} = h\left(f_i(y_0), f_j(y_0), f_K(y_0), f_K(y_0), f_K(y_0), f_K(y_0)\right).$$

Этот терм призван обозначить то, что все стороны указанного квадрата окрашены в цвет K.

Если граничный квадрат не лежит в углах прямоугольника, т.е. $(i, j) \in Border \setminus \{(0,0), (n+1,0), (n+1, m+1), (0, m+1)\}$, и при этом $B(i, j) = k$, то существует единственный соседний с ним квадрат (i', j') из внутренней области. Пусть y_{i_1,j_1,i_2,j_2} – переменная из множества Y', ассоциированная с парой соседних квадратов (i, j), (i', j'). Тогда граничные условия для квадрата (i, j) в подстановке θ_0 описываются связкой $x_{i,j}/t_{i,j}$, где

$$t_{i,j} = h\left(f_i(y_0), f_j(y_0), f_K(y_0), f_K(y_0), f_K(y_0), f_k\left(y_{i_1,j_1,i_2,j_2}\right)\right).$$

Терм $t_{i,j}$ призван обозначить тот факт, что сегмент границы, являющийся одной из сторон квадрата $t_{i,j}$, окрашен в цвет k, а все остальные стороны этого граничного квадрата окрашены в цвет K.

Если квадрат (i, j) лежит во внутренней области $Interior$ прямоугольника $Area$, то у этого квадрата есть в точности четыре соседних с ним квадрата, расположенных вверху, справа, внизу и слева от квадрата (i, j). Пусть y_{i_1,j_1,i'_1,j'_1}, y_{i_2,j_2,i'_2,j'_2}, y_{i_3,j_3,i'_3,j'_3}, и y_{i_4,j_4,i'_4,j'_4} - это четыре переменные из множества Y', которые ассоциированы с указанными четырьмя парами соседних квадратов, включающих квадрат (i, j). Тогда для каждого домино $tile_\ell = \langle k_1, k_2, k_3, k_4 \rangle$ из набора $Tiles$ введем в подстановку θ_0 связку $x_{i,j,\ell}/t_{i,j,\ell}$, где

$$t_{i,j,\ell} = h\left(f_i(y_0), f_j(y_0), f_{k_1}\left(y_{i_1,j_1,i'_1,j'_1},\right), f_{k_2}\left(y_{i_2,j_2,i'_2,j'_2}\right), f_{k_3}\left(y_{i_3,j_3,i'_3,j'_3}\right), f_{k_4}\left(y_{i_4,j_4,i'_4,j'_4}\right)\right).$$

Этот терм призван обозначать тот факт, что в квадрат (i,j) внутренней области может быть вставлено домино $tile_\ell$ с присущей этому типу домино раскраской сторон.

Располагая описанными выше тремя типами связок, определим подстановку θ_0 как множество всех указанных выше связок, т.е.

$$\theta_0 = \{x_{i,j}/t_{i,j} : (i,j) \in Border\} \cup \{x_{i,j,\ell}/t_{i,j,\ell} : (i,j) \in Interior, 1 \le \ell \le L\}.$$

Заметим, что каждая из переменных множества Y' присутствует сразу в нескольких термах подстановки θ_0, в качестве аргумента нумералов. Для каждого вхождения переменной $y, y \in Y'$, в некоторый терм подстановки θ_0 *глубиной* этого вхождения будем называть максимальный показатель тех нумералов, аргументом которых является это вхождение.

Используя только функциональный символ $g^{(2)}$, построим произвольный терм t_{area} местности $(n+2)(m+2)$ (т.е. имеющий $(n+2)(m+2)$ листовых вершин в древесном представлении). Каждый аргумент этого терма соответствует одному из квадратов в области покрытия $Area$. Для каждого квадрата (i,j) в области $Area$ введем терм $\hat{t}_{i,j} = h(f_i(u), f_j(u), f_K(u), f_K(u), f_K(u), f_K(u))$. Фактически, этот терм соответствует расположению в квадрате (i,j) домино со сторонами, монохроматически окрашенными в цвет K. Определим подстановку

$$\theta_1 = \{z/t_{area}(\hat{t}_{0,0}, \hat{t}_{0,1}, \dots, \hat{t}_{n+1,m+1})\}.$$

Эта подстановка означает, что вся область $Area$ заполнена монохроматически окрашенными домино указанного вида.

Лемма 2.

Пример ограниченной проблемы домино $BT = (n, m, Tiles, B)$ допустим тогда и только тогда, когда пара подстановок (θ_0, θ_1), определенных выше, двусторонне унифицируема.

Доказательство. 1) Пусть пример BT допустим. Тогда существует B-правильное покрытие T области $Area$ домино из семейства $Tiles$. Рассмотрим это правильное покрытие, и для каждой пары соседних квадратов $(i,j), (i',j')$ внутренней области прямоугольника, где $i \le i', j \le j'$, обозначим записью $c(i,j,i',j')$ тот цвет, которым окрашены смежные стороны домино $T(i,j)$ и $T(i',j')$, размещенных при покрытии T в указанных квадратах. Точно такой же записью обозначим тот общий цвет, в который окрашены сторона домино и смежный с этой стороной сегмент границы прямоугольника. Согласно описанию устройства термов вида $t_{i,j,\ell}$ каждое вхождение переменной $y_{i,j,i',j'}$ в термы $t_{i,j,T(i,j)}$ и $t_{i',j',T(i',j')}$ имеет одну и ту же глубину $c(i,j,i',j')$. Тогда двусторонним унификатором пары подстановок (θ_0, θ_1) является пара (η', η''), где

$$\eta' = \{z/t_{area}(x_{0,0}, x_{0,1}, \dots, x_{0,m+1}, x_{1,0}, x_{1,1,T(1,1)}, \dots, x_{1,m,T(1,m)}, x_{1,m+1}, \dots, x_{n+1,m+1})\},$$
$$\eta'' = \{y_0/u, y_{0,1,1,1}/f_{K-c(0,1,1,1)}(u), \dots, y_{i,j,i',j'}/f_{K-c(i,j,i',j')}(u), \dots\}.$$

В подстановке η' в терме t_{area} на место каждого аргумента, соответствующего квадрату (i,j) покрываемого прямоугольника, поставлена либо переменная $x_{i,j}$, если этот квадрат располагается на границе области, либо переменная $x_{i,j,T(i,j)}$, если этот квадрат располагается во внутренней части области. В последнем случае переменная $x_{i,j,T(i,j)}$ указывает, что в этот квадрат должно быть помещено домино $tile_{T(i,j)}$. В подстановке η'' вместо каждой переменной, соответствующей паре смежных сторон двух квадратов области $Area$, подставлен нумерал, дополняющий тот цвет, в который окрашены смежные стороны, располагающиеся в этих квадратах домино, до максимального цвета K.

Нетрудно убедиться, принимая во внимание B-правильность покрытия T, что имеет место равенство $\theta_1 = \eta'\theta_0\eta''$.

2) Пусть для некоторой пары подстановок (η', η'') справедливо равенство $\theta_1 = \eta'\theta_0\eta''$. Взглянув на устройство семейства размеченных деревьев, представляющих подстановку θ_1, можно заметить, что на каждой ветви вначале следуют функциональные символы g, затем функциональный символ h, и в заключении следуют функциональные символы f. Кроме того, для каждого квадрата (i,j) в терме подстановки θ_1 содержится единственный подтерм вида $h(f_i(y_0), f_j(y_0), \dots)$. В подстановке θ_0 все термы содержат только функциональные символы h и f. Таким образом, подстановка η' должна иметь вид

$$\eta' = \{z/t_{area}(x_{0,0}, x_{0,1}, \dots, x_{0,m+1}, x_{1,0}, x_{1,1,\ell_{1,1}}, \dots, x_{1,m,\ell_{1,m}}, x_{1,m+1}, \dots, x_{n+1,m+1})\},$$

а подстановка η'' должна иметь вид

$$\eta'' = \{y_0/u, y_{0,1,1,1}/f_{k_{0,1,1,1}}(u), \dots, y_{i,j,i',j'}/f_{k_{i,j,i',j'}}(u), \dots\}$$

Рассмотрим покрытие T, в котором для каждого квадрата (i,j) имеет место равенство $T(i,j) = \ell_{i,j}$ тогда и только тогда, когда в терме подстановки η' содержится переменная $x_{i,j,\ell_{i,j}}$. Покажем, что это покрытие является B-правильным.

Допустим противное. Тогда окраска двух смежных сторон каких-то двух соседних домино (или окраска стороны одного из домино и смежного с этой стороной сегмента границы) должны быть разными. Без ограничения общности ограничимся рассмотрением первого из этих двух вариантов нарушения правильности покрытия. Предположим, что несогласованность окраски сторон проявляется для домино, располагающихся в квадратах $(i_1, j_1), (i_2, j_2)$ внутренней области прямоугольника, где $i_1 \le i_2, j_1 \le j_2$. Это

означает, что в термах $t_{i_1,j_1,\ell_{i_1,j_1}}$ и $t_{i_2,j_2,\ell_{i_2,j_2}}$ вхождения разделяемой этими термами переменной y_{i_1,j_1,i_2,j_2} имеют разную глубину. Поэтому и в том терме, который представляет единственную связку композиции подстановок $\eta'\theta_0 = \{z/t_{area}(x_{0,0},\dots,x_{n+1,m+1})\theta_0\}$, оба вхождения переменной y_{i_1,j_1,i_2,j_2} имеют разную глубину. Но тогда и в композиции $\eta'\theta_0\eta''$ нумералы, располагающиеся в тех позициях подтермов $t_{i_1,j_1,\ell_{i_1,j_1}}$и $t_{i_2,j_2,\ell_{i_2,j_2}}$, которые соответствуют окраске сторон домино, будут разными. А это противоречит тому, что $\theta_1 = \eta'\theta_0\eta''$ и при этом в терме подстановки θ_1 показатели всех нумералов, обозначающих окраску сторон домино, одинаковы и равны K.

Таким образом, предложенное покрытие T является B-правильным. QED

Лемма 3. Проблема ограниченного домино $log - space$ сводима к проблеме двусторонней унифицируемости подстановок.

Доказательство. Из описания подстановок θ_0 и θ_1, приведенных в этом разделе, видно, что для всякого примера проблемы ограниченного домино $BT = (n, m, Tiles, B)$ соответствующую этому примеру пару подстановок (θ_0, θ_1) можно построить посредством детерминированного алгоритма, использующего объем памяти, пропорциональный логарифму размера описания примера. QED

Из лемм 1 и 3 вытекает

Теорема 1. Проблема двусторонней унифицируемости подстановок является NP-полной.

Здесь стоит отметить, что данная теорема является завершающим утверждением, дающим полную картину сложности проблемы разрешимости уравнений вида $\boldsymbol{X}_1^{\sigma_1}\theta\boldsymbol{X}_2^{\sigma_2} = \boldsymbol{X}_3^{\sigma_3}\eta\boldsymbol{X}_4^{\sigma_4}$ в полугруппе конечных подстановок первого порядка, где $\sigma_i \in \{0,1\}$ и при этом $\boldsymbol{X}^1 = \boldsymbol{X}$ и $\boldsymbol{X}^0 = \varepsilon$, а ε – тождественная подстановка (нейтральный элемент полугруппы). Действительно, уравнения вида $\boldsymbol{X_1\theta X_2 = X_3\eta X_4}$, $\boldsymbol{\theta X_2 = X_3\eta X_4}$ и $\boldsymbol{X_1\theta = X_3\eta X_4}$ имеют очевидные тривиальные решения вида $(\boldsymbol{X}_1 = \eta, \boldsymbol{X_2} = \boldsymbol{X}_3 = \varepsilon, \boldsymbol{X}_4 = \theta)$, $(\boldsymbol{X_2} = \eta, \boldsymbol{X}_3 = \theta, \boldsymbol{X}_4 = \varepsilon)$ и $(\boldsymbol{X}_1 = \eta, \boldsymbol{X}_3 = \theta, \boldsymbol{X}_4 = \varepsilon)$ соответственно. Уравнения вида $\boldsymbol{\theta X_2 = \eta X_4}$ и $\theta\boldsymbol{X_2} = \eta$ соответствуют проблеме унификации и, как показано в работах [16], разрешимы за почти линейное время. Уравнения вида $\boldsymbol{X_1\theta = X_3\eta}$ и $\boldsymbol{X_1\theta} = \eta$ были исследованы в статье [23] в связи с изучением проблемы эквивалентности в одном классе последовательных программ. Эти уравнения разрешимы за полиномиальное время. И, как установлено в теореме 1, лишь для уравнений вида $\boldsymbol{X_1\theta X_2} = \eta$ задача их разрешимости является NP-полной.

5. Двусторонняя унификация программ

Задача двусторонней унификации может быть обобщена и распространена на программы. В этом разделе статьи мы сформулируем проблему двусторонней

унификации для одной модели последовательных императивных программ и покажем, что эта задача также является NP-полной.

Мы будем рассматривать модель последовательных императивных программ, которая была введена в статье [6] и подробно исследована в монографии [7]. Программы в этой модели строятся из операторов ввода и вывода, операторов присваивания и тестов. Операторы ввода и вывода лишь обозначают множества входных и выходных переменных. Семантика каждого оператора присваивания e вида $x{:=}t$, где x – некоторая переменная, а t – терм, определяется подстановкой $\theta_e = \{x/t\}$. Эффект последовательной композиции операторов присваивания $e_1; e_2; \dots e_k$ описывается композицией соответствующих подстановок $\theta_{e_k} \cdots \theta_{e_2}\theta_{e_1}$. Тесты представляют собой атомарные формулы вида $P(t_1, \dots, t_m)$, где P – предикатный символ, а $t_1, \dots, t_m$ – термы.

В модели последовательных программ такого вида введено отношение логико-термальной эквивалентности программ (см. [8]). Это отношение эквивалентности аппроксимирует отношение функциональной эквивалентности [7,8] и при этом разрешимо за полиномиальное время [2,9]. Разрешимость логико-термальной эквивалентности позволяет использовать ее для построения эффективных алгоритмов верификации, оптимизации и реорганизации (рефактроинга) последовательных программ. В рамках этой модели можно сформулировать одну из важных задач реорганизации программ – задачу замены фрагментов программ вызовами заданных процедур. В этом разделе мы покажем, что эту задачу можно решить при помощи методов двусторонней унификации подстановок.

Определим формально класс последовательных программ в рассматриваемой модели. Для этого выделим три типа переменных: конечное множество входные переменных $Y = \{y_1, \dots, y_n\}$, конечное множество выходных переменных $X = \{x_1, \dots, x_m\}$, и конечное множество вспомогательных переменных Var. Пусть задано некоторое множество предикатных символов P. Обозначим записью $Atom[Var]$ множество атомарных формул, которые строятся обычным образом из предикатных символов и термов из множества $Term[Var]$. Также введем две специальные атомарные формулы $Input(y_1, \dots, y_n)$ и $Output(x_1, \dots, x_m)$ над множествами входных и выходных переменных.

Модель программы представляет собой размеченный ориентированный граф π. Две вершины этого графа – входная вершина v_{in} и выходная вершина v_{out} – особо выделены. Входной вершине v_{in} приписана атомарная формула $Input(y_1, \dots, y_n)$; из этой вершины исходит единственная дуга, которой приписана подстановка θ_{in} из множества подстановок $Subst[Var, Y]$. Выходной вершине v_{out} приписана атомарная формула $Output(x_1, \dots, x_m)$; из этой вершины не исходит ни одной дуги. Каждой внутренней вершине v этого графа, т.е. вершине, отличной от входной и выходной вершин, приписана

некоторая атомарная формула A_v из множества $Atom[Var]$. Из каждой такой вершины v исходят две дуги, одна из которых помечена символом 0, а другая – символом 1. Эти дуги могут вести в любую вершину графа за исключением входной вершины v_{in}. Каждой дуге, ведущей в графе π из внутренней вершины u во внутреннюю вершину v, приписана подстановка θ_{uv} из множества $Subst[Var, Var]$. Каждой дуге, ведущей в графе π из внутренней вершины u в выходную вершину v_{out}, приписана подстановка $\eta_{u,out}$ из множества $Subst[X, Var]$. Предполагается также, что через каждую вершину графа $\pi(Var)$ проходит некоторый маршрут, ведущий из входа программы в ее выход.

В модели программ такого вида вершины графа – это точки проверки логических условий, дуги графа соответствуют линейным участкам программы, вычислительный эффект которых описывается подстановками, приписанными этим дугам. Во входе программы осуществляет инициализация вспомогательных переменных программы, а в выходной вершине собираются результаты вычислений в виде подстановок в выходные переменные.

Существуют разные способы определения функциональных возможностей моделей программ такого рода (см. [7]). Однако, как было показано в статье [27], любой вид эквивалентности программ, опирающийся на понятия интерпретации языка первого порядка, оказывается неразрешимым. Поэтому для получения эффективных алгоритмов, разрешающих вычислительные свойства программ, приходится использовать структурные виды эквивалентности программ. Для этого приходится рассматривать и сравнивать всевозможные синтаксически допустимые трассы в анализируемых программах. Одним из таких видов эквивалентности является логико-термальная эквивалентность (л-т эквивалентность), введенная в статье [8] и подробно исследованная в работах [2,9]. Определение этой эквивалентности таково.

Трассой в программе π называется маршрут, ведущий из входной вершины в выходную вершину программы; в этом маршруте указываются все пометки, приписанные дугам. Пусть задана некоторая трасса в программе π

$$\alpha = v_0 \xrightarrow{\theta_0} v_1 \xrightarrow{\sigma_1, \theta_1} v_2 \xrightarrow{\sigma_2, \theta_2} \cdots v_{n-2} \xrightarrow{\sigma_{n-2}, \theta_{n-2}} v_{n-1} \xrightarrow{\theta_{n-1}, \sigma_{n-1}} v_n$$

,

где $v_0 = v_{in}, \sigma_i \in \{0,1\}$ для всех $i, 1 \le i \le n-1, \theta_i \in Subst[Var, Var]$ для всех $i, 1 \le i \le n-2$, $\theta_{n-1} \in Subst[Z, Var], v_n = v_{out}$. Тогда последовательность пар

$$lth(\alpha) = \\ \left(A_{v_1}\theta_0, \sigma_1\right), \left(A_{v_2}\theta_1\theta_0, \sigma_2\right), \\ \ldots, \left(A_{v_{n-1}}\theta_{n-1}\cdots\theta_1\theta_0, \sigma_{n-1}\right), \left(A_{v_n}\theta_{n-1}\theta_{n-2}\cdots\theta_1\theta_0, 1\right)$$

называется *логико-термальной историей* (л-т историей) трассы α. Двоичная последовательность $\sigma_1, \sigma_2, \dots, \sigma_{n-1}$ называется *логической характеристикой* трассы α. Нетрудно видеть, что разные трассы в программе имеют разные характеристики. *Детерминантом* программы π называется множество $Det(\pi)$=$\{lth(\alpha)$: α – трасса в программе $\pi\}$. Две программы π_1 и π_2 считаются *логико-термально* (*л-т*) *эквивалентными*, если справедливо равенство $Det(\pi_1) = Det(\pi_2)$. В статьях [2,9] показано, что задача проверки л-т эквивалентности программ разрешима за полиномиальное время.

Опишем операцию применения подстановок к программам. Пусть заданы программа π над множествами входных переменных X и выходных переменных Y, а также две подстановки η' из семейства $Subst[Z, X]$ и η'' из семейства $Subst[U, Y]$, где $U = \{u_1, \dots, u_k\}$, $Z = \{z_1, \dots, z_\ell\}$. Тогда *результатом применения пары подстановок* (η', η'') *к программе* π является программа $\eta'\pi\eta''$, которая получается из программы π в результате следующих преобразований:

- во входной вершине атом $Input(y_1, \dots, y_n)$ заменяется атомом $Input(u_1, \dots, u_k)$ и подстановка θ_0, помечающая исходящую из этой вершины дугу, замещается подстановкой $\theta_0\eta''$;
- в выходной вершине атом $Ouput(x_1, \dots, x_m)$ заменяется атомом $Output(z_1, \dots, z_\ell)$, и для каждой дуги, ведущей в выходную вершину, подстановка θ, помечающая эту дугу, замещается подстановкой $\eta'\theta$.

Программу $\eta'\pi\eta''$ можно истолковывать как вызов процедуры с телом π, в котором инициализация входных параметров осуществляется подстановкой η'', а специализация выходных параметров осуществляется подстановкой η'.

Задача двусторонней унификации программ формулируется так: для заданной пары программ π_0 и π_1 вычислить пару подстановок (двусторонний унификатор) (η', η''), для которой программы $\eta'\pi_0\eta''$ и π_1 будут л-т эквивалентными. Одно из возможных решений задачи двусторонней унификации программ опирается на следующую лемму.

Лемма 4. Пусть (η', η'') – двусторонний унификатор программ π_0 и π_1. Предположим, что в программе π_0 имеется трасса

$$\alpha = v_0 \xrightarrow{\theta_0} v_1 \xrightarrow{\sigma_1, \theta_1} v_2 \xrightarrow{\sigma_2, \theta_2} \cdots v_{n-2} \xrightarrow{\sigma_{n-2}, \theta_{n-2}} v_{n-1} \xrightarrow{\sigma_{n-1}, \theta_{n-1}} v_n$$

.

Тогда в программе π_1 существует трасса

$$\beta = v'_0 \xrightarrow{\mu_0} v'_1 \xrightarrow{\sigma_1, \mu_1} v'_2 \xrightarrow{\sigma_2, \mu_2} \cdots v'_{n-2} \xrightarrow{\sigma_{n-2}, \mu_{n-2}} v'_{n-1} \xrightarrow{\sigma_{n-1}, \mu_{n-1}} v_n$$

,

имеющая ту же самую логическую характеристику, что и трасса α, и удовлетворяющая следующему равенству $\eta'\theta_{n-1}\theta_{n-2}\cdots\theta_1\theta_0\eta''=\mu_{n-1}\mu_{n-2}\cdots\mu_1\mu_0$.

Доказательство: Если программы $\eta'\pi_0\eta''$ и π_1 л-т эквивалентны, то $Det(\eta'\pi_0\eta'') = Det(\pi_1)$. Поскольку $lth(\alpha) \in Det(\eta'\pi_0\eta'')$, в программе π_1 существует такая трасса β, что $lth(\alpha) = lth(\beta)$. Значит, трасса β должна иметь ту же самую логическую характеристику, что и трасса α, и при этом последние пары в последовательностях $lth(\alpha)$ и $lth(\beta)$ должны быть одинаковыми. Отсюда следует указанное в формулировке леммы равенство композиций подстановок. QED

Теорема 2. Задача двусторонней унифицируемости программ относительно л-т эквивалентности является NP-полной.

Доказательство. NP-трудность этой задачи следует из теоремы 1, поскольку задача унифицируемости подстановок является частным случаем задачи унифицируемости программ. О принадлежности задачи двусторонней унифицируемости программ классу сложности NP свидетельствует следующий недетерминированный алгоритм решения этой задачи за полиномиальное время. В программе π_0 нужно выбрать кратчайшую трассу

$$\alpha = v_0 \xrightarrow{\theta_0} v_1 \xrightarrow{\sigma_1, \theta_1} v_2 \xrightarrow{\sigma_2, \theta_2} \cdots v_{n-2} \xrightarrow{\sigma_{n-2}, \theta_{n-2}} v_{n-1} \xrightarrow{\sigma_{n-1}, \theta_{n-1}} v_n$$

и вычислить композицию подстановок $\theta = \theta_{n-1}\theta_{n-2}\cdots\theta_1\theta_0$. Очевидно, что это вычисление можно осуществить за время линейное относительно размера программы π_0. Затем в программе π_1 нужно выбрать трассу

$$\beta = v'_0 \xrightarrow{\mu_0} v'_1 \xrightarrow{\sigma_1, \mu_1} v'_2 \xrightarrow{\sigma_2, \mu_2} \cdots v'_{n-2} \xrightarrow{\sigma_{n-2}, \mu_{n-2}} v'_{n-1} \xrightarrow{\sigma_{n-1}, \mu_{n-1}} v_n$$

,

имеющую ту же самую логическую характеристику, что и трасса α. Согласно лемме 4 такая трасса существует и определяется однозначно. Затем для трассы β нужно вычислить композицию подстановок $\mu = \mu_{n-1}\mu_{n-2}\cdots\mu_1\mu_0$. Это вычисление также осуществимо за время, полиномиальное от размера программы π_1. Далее, опираясь на лемму 4, нужно вычислить двусторонний унификатор (η', η'') подстановок θ и μ. Согласно лемме 1 это вычисление (угадывание) можно осуществить недетерминированным алгоритмом за полиномиальное время. В заключение нужно проверить л-т эквивалентность программ $\eta'\pi_0\eta''$ и π_1. Согласно результатам статей [2,9] такую проверку можно провести за время, полиномиальное относительно размеров этих программ. QED

6. Заключение

Результаты решения задачи двусторонней унификации, представленные в этой статье, приводят к нескольким задачам, связанным с реорганизацией программ.

NP-полнота задачи двусторонней унификации подстановок означает, что для ее решения целесообразно привлечь практические средства решения вычислительно трудных задач. В связи с этим возникает вопрос о выборе подходящих средств такого рода. Вероятно, что для этой цели пригодны программные системы решения проблемы выполнимости булевых формул. В этом случае нужно разработать подходящий метод сведения задачи двусторонней унификации подстановок к проблеме SAT.

Разрешимость задачи двусторонней унификации программ позволяет приступить к исследованию более трудной, но, вместе с тем, и практически более значимой задачи выделения общей части двух программ. Для заданной пары программ π_1 и π_2 требуется выяснить, существует ли такая программа π_0 , которая имеет двустороннюю унификацию как с программой π_1, так и с программой π_2 . В случае существования такой программы π_0, ее можно использовать в качестве отдельной процедуры и заменить каждую из программ π_1 и π_2 вызовом этой процедуры для специализированных подходящим образом значений параметров. Решение этой задачи позволит значительно продвинуться в направлении автоматизации рефакторинга программ.

Список литературы

[1] Захаров В.А., Новикова Т.А.. Применение алгебры подстановок для унификации программ // Труды Института системного программирования РАН, – 2011. – т. 21 – с. 141-166.

[2] Захаров В.А., Новикова Т.А.. Полиномиальный по времени алгоритм проверки логико-термальной эквивалентности программ. // Труды Института системного программирования РАН, – 2012. – т. 22 – с. 435-455.

[3] Захаров В.А., Новикова Т.А.. Унификация программ. // Труды Института системного программирования РАН, – 2012. – т. 23 – с. 455-476.

[4] Novikova T.A., Zakharov V.A. Is it possible to unify programs? // Proceedings of the 27-th International Workshop on Unification, Epic Series. – 2013. – v. 19 – p. 35-45.

[5] Baader F., Snyder W. Unification theory // In J.A. Robinson and A. Voronkov, editors, Handbook of Automated Reasoning. — 2001. — v. 1 — p. 447-533.

[6] Luckham D.C., Park D.M., Paterson M.S., On formalized computer programs // Journal of Computer and System Science — 1970. — v.4, N 3. — p. 220-249.

[7] Котов В.Е., Сабельфельд В.К. Теория схем программ. — М.:Наука, 1991. — 348 с.

[8] Иткин В.Э. Логико-термальная эквивалентность схем программ // Кибернетика. — 1972. — N 1. — с. 5-27.

[9] Сабельфельд В.К. Полиномиальная оценка сложности распознавания логико-термальной эквивалентности // ДАН СССР. — 1979. — т. 249, N 4. — с. 793-796.

[10] Фаулер М. Рефакторинг. Улучшение существующего кода. — Символ-Плюс, 2008. — 432 с.
[11] Komondoor R., Horwitz S. Using slicing to identify duplication in source code // Proceedings of the 8th International Symposium on Static Analysis. — Springer-Verlag, 2001. — p. 40–56.
[12] Roy C. K., Cordy J. R. A survey on software clone detection research // Technical report TR 2007-541, School of Computing, Queen's University. — 2007. — v. 115.
[13] Robinson J.A. A machine-oriented logic based on the resolution principle // Journal of the ACM. 1965 — v. 12, N 1. — p. 23-41.
[14] Knight K. Unification: a multidisciplinary survey // ACM Computing Surveys — 1989. — v. 21 — N 1 — p. 93-124.
[15] Baxter L.D. An efficient unification algorithm // Technical Report CS-73-23, Dep. of Analysis and Comp. Sci., University of Waterloo, Ontario, Canada, 1973.
[16] Paterson M.S., Wegman M.N. Linear unification // The Journal of Computer and System Science. — 1978. — v. 16, N 2 — p. 158-167.
[17] Martelli A., Montanari U. An efficient unification algorithm //ACM Transactions on Program, Languages and Systems. — 1982. — v. 4, N 2 — p. 258-282.
[18] Stickel E.M. A unification algorithm for associative-commutative functions // Journal of the association for Computing Machinary. — 1981. — v. 28, N 5 — p. 423-434.
[19] Herold A., Sieckmann J. Unification in Abelean semigroups // Jornal of Automated Reasoning. — 1983. — p. 247-283.
[20] Lincoln P., Christian J. Adventures in associative-commutative unification //Journal of Symbolic Computation. — 1989. — v.8. — p. 393 — 416.
[21] Eder E. Properties of substitutions and unifications // Journal of Symbolic Computations. — v. 1. — 1985. — p. 31-46.
[22] Lewis C.H. Complexity of solvable cases of the decision problem for predicate calculus // Proceedings of the 19-th Annual Symposium on Foundations of Computer Science – 1978 - p. 35-47.
[23] Zakharov V.A. On the decidability of the equivalence problem for orthogonal sequential programs // Grammars. – 2000. – v. 2 - N 3. - p. 271-281.
[24] Wang Hao. Proving theorems by pattern recognition // Bell System Technical Journal. - 1961. - v. 40. - N 1. - p. 1-41.
[25] Berger R. The undecidability of domino problem // Memoirs of American Mathematical Society – v. 66.
[26] Lewis C.H., Papadimitriou C.H. Elements of the Theory of Computation – Prentice Hall, Englewood Cliffs, 1981.
[27] Itkin V.E., Zwinogrodski Z. On program schemata equivalence // Journal of Computer and System Science. – 1972. - v.6. - N 1. - p. 88-101.

Two-sided program unification and its application to program refactoring

T.A. Novikova
Kazakhstan Branch of Lomonosov Moscow State University, Astana, Kazakhstan
taniaelf@mail.ru
V.A.Zakharov
ISP RAS, Moscow, Russia
zakh@cs.msu.su

Annotation. It is generally accepted that to unify a pair of substitutions θ_1 and θ_2 means to find out a pair of substitutions η' and η'' such that the compositions $\theta_1\eta'$ and $\theta_2\eta''$ are the same. Actually, unification is the problem of solving linear equations of the form $\theta_1 X = \theta_2 Y$ in the semigroup of substitutions. But some other linear equations on substitutions may be also viewed as less common variants of unification problem. In this paper we introduce a two-sided unification as the process of bringing a given substitution θ_1 to another given substitution θ_2 from both sides by giving a solution to an equation $X\theta_1 Y = \theta_2$. Two-sided unification finds some applications in software refactoring as a means for extracting instances of library subroutines in arbitrary pieces of program code. In this paper we study the complexity of two-sided unification for substitutions and programs. In Section 1 we discuss the concept of unification on substitutions and programs, outline the recent results on program equivalence checking that involve the concept of unification and anti-unification, and show that the problem of library subroutine extraction may be viewed as the problem of two-sided unification for programs. In Section 2 we formally define the concept of two-sided unification on substitutions and show that the problem of two-unifiability is NP-complete (in contrast to P-completeness of one-sided unifiability problem). NP-hardness of two-sided unifiability problem is established in Section 4 by reducing to it the bounded tiling problem which is known to be NP-complete; the latter problem is formally defined in Section 3. In Section 5 we define two-sided unification of sequential programs in the first-order model of programs supplied with strong equivalence (logic&term equivalence) of programs and proved that this problem is NP-complete. This is the main result of the paper.

Keywords: program, strong equivalence, substitution, complexity, tiling problem, NP-completeness.

References

[1] Zakharov V.A., Novikova T.A. Primininie algebry podstanovok dlya unifikacii program [On the application of substitution algebra to program unification]. *Trudy ISP RAN* [The Proceedings of ISP RAS], 2011, vol. 21, p. 141-166 [in Russian].
[2] Zakharov V.A., Novikova T.A. Polynomialniy po vremeni algoritm proverki logiko-termalnoy ekvivalentnosti program [Polynomial time algorithm for checking strong

equivalence of program]. *Trudy ISP RAN* [The Proceedings of ISP RAS], 2012, vol. 22, p. 435-455 [in Russian].
[3] Zakharov V.A., Novikova T.A. Unifikaciya program [Program unification]. *Trudy ISP RAN* [The Proceedings of ISP RAS], 2012, vol. 23, p. 455-476 [in Russian].
[4] Novikova T.A., Zakharov V.A. Is it possible to unify programs? // Proceedings of the 27-th International Workshop on Unification, Epic Series. – 2013. – v. 19 – p. 35-45.
[5] Baader F., Snyder W. Unification theory. In J.A. Robinson and A. Voronkov, editors, Handbook of Automated Reasoning, 2001, v. 1, p. 447-533.
[6] Luckham D.C., Park D.M., Paterson M.S. On formalized computer programs // Journal of Computer and System Science. 1970, vol. 4, N 3, p. 220-249.
[7] Kotov V.E. Sabelfeld V.K. Teoriya skhem program [Theory of program schemata]. Moscow, "Nauka", 1991, 348 p.
[8] Itkin V.E. Logical-termal equivalence of program schemata. Proceedings of the International Sympoisum on Theoretical Programming, 1972, p. 127-143.
[9] Sabelfeld V.K. The Logic-Termal Equivalence is Polynomial-Time Decidable. Information Processing Letters, 1980, vol. 10, N 2, p. 57-62.
[10] Fauler M. Refactoring: Improving the design of existing code. 1999, Addison Wesley.
[11] Komondoor R., Horwitz S. Using slicing to identify duplication in source code // Proceedings of the 8th International Symposium on Static Analysis. Springer-Verlag, 2001, p. 40–56.
[12] Roy C. K., Cordy J. R. A survey on software clone detection research. Technical report TR 2007-541, School of Computing, Queen's University, 2007, vol. 115.
[13] Robinson J.A. A machine-oriented logic based on the resolution principle. Journal of the ACM, 1965, vol. 12, N 1, p. 23-41.
[14] Knight K. Unification: a multidisciplinary survey. ACM Computing Surveys. 1989, vol. 21, N 1, p. 93-124.
[15] Baxter L.D. An efficient unification algorithm. Technical Report CS-73-23, Dep. of Analysis and Comp. Sci., University of Waterloo, Ontario, Canada, 1973.
[16] Paterson M.S., Wegman M.N. Linear unification. The Journal of Computer and System Science. 1978, vol. 16, N 2, p. 158-167.
[17] Martelli A., Montanari U. An efficient unification algorithm. ACM Transactions on Program, Languages and Systems. 1982, vol. 4, N 2, p. 258-282.
[18] Stickel E.M. A unification algorithm for associative-commutative functions. Journal of the association for Computing Machinary. 1981, v. 28, N 5, p. 423-434.
[19] Herold A., Sieckmann J. Unification in Abelean semigroups. Journal of Automated Reasoning. 1983, p. 247-283.
[20] Lincoln P., Christian J. Adventures in associative-commutative unification. Journal of Symbolic Computation. 1989, vol. 8, p. 393-416.
[21] Eder E. Properties of substitutions and unifications. Journal of Symbolic Computations. vol. 1, 1985, p. 31-46.
[22] Lewis C.H. Complexity of solvable cases of the decision problem for predicate calculus. Proceedings of the 19-th Annual Symposium on Foundations of Computer Science, 1978, p. 35-47.
[23] Zakharov V.A. On the decidability of the equivalence problem for orthogonal sequential programs. Grammars. 2000, vol. 2, N 3, p. 271-281.
[24] Wang Hao. Proving theorems by pattern recognition. Bell System Technical Journal. 1961, vol. 40, N 1, p. 1-41.
[25] Berger R. The undecidability of domino problem. Memoirs of American Mathematical Society, v. 66.

[26] Lewis C.H., Papadimitriou C.H. Elements of the Theory of Computation – Prentice Hall, Englewood Cliffs, 1981.
[27] Itkin V.E., Zwinogrodski Z. On program schemata equivalence. Journal of Computer and System Science. 1972, vol. 6, N 1, p. 88-101.

Методы пороговой криптографии для защиты облачных вычислений[1]

Варновский Н.П., Мартишин С.А., Храпченко М.В., Шокуров А.В.

Аннотация. Защита информации в облачных вычислениях активно исследуется мировым научным сообществом. Эти исследования показали, что обозначенная проблема намного сложнее тех задач защиты информации, которые решаются известными криптографическими средствами. Так, например в работе [1] рассмотрена математическая модель организации облачных вычислений и доказано, что уже в случае двух пользователей защита информации невозможна. Предлагается альтернативная модель организации облачных вычислений, в которой указанный отрицательный результат не имеет места. Исследуются методы защиты информации в этой новой модели.

Ключевые слова: защита информации, облачные вычисления, вычисления над зашифрованными данными, гомоморфные вычисления.

Проблемы защиты информации в облачных вычислениях стали активно исследоваться слишком поздно, когда эти вычисления уже стали фактически реализованной технологией. Только практические применения рассеяли ранее бытовавшие иллюзии, что в этом случае для защиты информации достаточно уже имеющихся криптографических средств.

Сложность возникающих проблем можно понять уже на простейших примерах. Обратимся к следующей модели. Облако рассматривается как единая сущность. Обозначим ее через S. В системе имеются также пользователи и клиенты. Для простоты будем считать, что множество пользователей $P_1,\ldots,P_n$ неизменно. Количество клиентов не регламентируется. У пользователя P_i имеются конфиденциальные данные x_i, хранящиеся на облаке (такой метод использования облака в литературе называют «базы данных как сервис»). В число пользователей может входить также само облако с данными x_S. Любой клиент C может обратиться к облаку с запросом на вычисление значения некоторой функции f, зависящей от конфиденциальных данных. Некоторые пользователи облака могут быть его клиентами. Множества пользователей и клиентов могут пересекаться. Запрос

[1] Работа поддержана грантом РФФИ 12-07-00206-а.

состоит из описания функции f, идентификатора клиента и его открытого ключа k_C. Облако должно проверить полномочия клиента C на вычисление $f(x_S, x_1, \ldots, x_n)$. Такая проверка реализуется с помощью стандартных криптографических средств и в данной статье не обсуждается. Если клиент C имеет право вычислять функцию f, то облако должно вычислить значение $E(k_C, f(x_S, x_1, \ldots, x_n))$ и отправить его клиенту. Здесь E - функция шифрования какой-либо криптосистемы с открытым ключом, например, RSA.

Описанная система может рассматриваться и как модель вычислений над конфиденциальными данными, и как модель базы данных с конфиденциальной информацией. В последнем случае функции f представляют собой запросы к базе данных.

Пользователь, который помещает в базу данных свои конфиденциальные данные, не доверяет облаку и должен обеспечить криптографическую защиту данных. Криптография подсказывает, казалось бы, надежное решение: данные следует зашифровать. Но здесь возникает следующая проблема. Если на облаке хранятся не данные x_i, а их криптограммы, то как вычислить функцию f?

В литературе можно найти описания подходов к организации вычислений над конфиденциальными данными. Но при этом совершенно упускается важнейший научный вопрос: а такая защита информации возможна?

Предполагаем, что противник имеет возможность обращаться к базе данных с конфиденциальной информацией с запросами двух типов:

- запросы на добавление в базу нового элемента;
- запросы на сравнение значений двух элементов.

Если интересующий противника элемент данных является целым числом из известного диапазона, то хорошо известным методом деления пополам значение этого элемента будет определено вне зависимости от используемой системы защиты информации.

Подобную угрозу можно пытаться предотвратить, вводя запреты на некоторые типы запросов. Мы далее рассмотрим предельный случай статичной базы данных – значения $x_1, \ldots, x_n$, хранящиеся на облаке, не меняются. Таким образом, речь идет именно о вычислениях над конфиденциальными данными.

В статье [4] для описанной выше модели облачных вычислений дано формальное определение стойкости системы защиты информации и доказано, что уже в случае двух пользователей стойкая защита конфиденциальных данных невозможна.

Этот результат опровергает прочно укоренившееся представление, что недавно предложенная [2] криптосистема гомоморфного шифрования решает, по крайней мере, теоретически все проблемы защиты информации в облачных вычислениях.

Мы предлагаем исследовать альтернативную модель облачных вычислений. Она отличается от вышеописанной тем, что каждый субъект, заинтересованный в обеспечении конфиденциальности, создает свой криптосервер. Обозначим эти криптосерверы $S_1,\ldots,S_m$. С точки зрения пользователей криптосерверы являются частью облака.

Для защиты данных, хранящихся на облаке, используется пороговая гомоморфная криптосистема с открытым ключом. Для ее инициализации криптосерверы выполняют специальный протокол, который создает пару ключей (k_1,k_2). Ключ k_1 - открытый: он доступен для всех пользователей. Ключ k_2 - секретный, он неизвестен никому. В результате выполнения протокола сервер S_i получает долю k_2^i секретного ключа k_2. Параметром криптосистемы является число t, $t<m$, которое называется порогом. Смысл этого параметра таков. Пусть E^t и D^t функции соответственно шифрования и дешифрования криптосистемы, и пусть $E^t(k_1,m)$ - криптограмма элемента данных m. Тогда любое подмножество из не менее чем t серверов может, используя свои доли секретного ключа, вычислить функцию D^t, то есть извлечь данные из криптограммы. А любая коалиция из не более, чем $(t-1)$ сервера, сложив свои доли k_2^i секретного ключа, не сможет этого сделать. Другими словами, криптосистема остается стойкой в присутствии противника, который контролирует не более $(t-1)$ серверов.

Необходимо подчеркнуть, что в процессе выполнения протокола, вычисляющего функцию D^t, секретный ключ k_2 никогда не восстанавливается из его долей. Все данные пользователей $x_1,\ldots,x_n$ шифруются все на одном ключе k_1. На облаке хранятся криптограммы $E^t(k_1,x_i)$.

Стойкость предлагаемой системы защиты информации основывается на стойкости пороговой гомоморфной криптосистемы и на следующем предположении: никакое подмножество из не менее чем t серверов не может вступить в сговор с целью нарушения конфиденциальности данных пользователей.

Благодаря этому предположению отрицательный результат [1] на нашу модель не распространяются.

Запрос клиента C выполняется следующим образом. Сначала облако, используя гомоморфность криптосистемы, вычисляет значение $E^t(k_1, E(k_C, f(x_S, x_1, \ldots, x_n)))$. Затем серверы выполняют протокол дешифрования, извлекая из этой криптограммы значение $E(k_C, f(x_S, x_1, \ldots, x_n))$.

Полное и математически строгое изложение будет приведено в отдельной статье.

Литература

[1] C. Gentry, Fully homomorphic encryption using ideal lattices , in Proceedings of the 41st ACM Symposium on Theory of Computing|STOC 2009, ACM, New York (2009), 169-178.

[2] C. Gentry and S. Halevi, Implementing Gentry's Fully-Homomorphic Encryption Scheme , in Advances in Cryptology|EUROCRYPT 2011, Lect. Notes in Comp. Sci. 6632 , (2011), Springer, 129-148.

[3] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, Fully Homomorphic Encryption over the Integers , in Advances in Cryptology|EUROCRYPT 2010, Lect. Notes in Comp. Sci. 6110 (2010), Springer, 24-43.

[4] M. v. Dijk and A. Juels. On the impossibility of cryptography alone for privacy-preserving cloud computing. In Hot topics in Security (HotSec'10), pages 1-8. USENIX Association, 2010.

A Threshold Cryptosystem in Secure Cloud Computations

Varnovskij N.P. (ISI), Martishin S.A. (ISP RAS), Khrapchenko M.V. (ISP RAS), Shokurov A.V. (ISP RAS)

Abstract. Information security in cloud computing technology is actively investigated by the world scientific community. They uses the internet and the central remote servers to provide and maintain data as well as applications. This users' data files can be accessed and manipulated by any other users. So the problem of secure data storage and computation is actual. The modern studies in this field shows that the indicated problem is much more complex than any of the other information security problems, which are solved by well-known cryptographic methods. So, for example M. van Dijk and A. Juels in the paper "On the impossibility of cryptography alone for privacy-preserving cloud computing" described a mathematical model of the organization of cloud computing and proved that in the case of two users information protection is impossible. This result refutes the well-established point of view that the recently proposed by C. Gentry construction for fully homomorphic encryption solves at least theoretically, all the problems of information security in cloud computing. We offer an alternative model of cloud computing, in which the specified negative result does not holds. It differs from the above in the point that each subject interested in privacy, creates his own crypto server. From the point of view of users these cryptoservers are the part of the cloud. The methods of information protection, using threshold cryptosystem in this new model are investigated.

Keywords: Secure Cloud Computing, Cryptographic Protocols, fully homomorphic encryption

References

[1] C. Gentry, Fully homomorphic encryption using ideal lattices , in Proceedings of the 41st ACM Symposium on Theory of Computing|STOC 2009, ACM, New York (2009), 169-178.

[2] C. Gentry and S. Halevi, Implementing Gentry's Fully-Homomorphic Encryption Scheme , in Advances in Cryptology|EUROCRYPT 2011, Lect. Notes in Comp. Sci. 6632 , (2011), Springer, 129-148.

[3] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, Fully Homomorphic Encryption over the Integers , in Advances in Cryptology|EUROCRYPT 2010, Lect. Notes in Comp. Sci. 6110 (2010), Springer, 24-43.

[4] M. v. Dijk and A. Juels. On the impossibility of cryptography alone for privacy-preserving cloud computing. In Hot topics in Security (HotSec'10), pages 1-8. USENIX Association, 2010.

О синтаксическом определении класса языков, распознаваемых недетерминированными машинами Тьюринга на логарифмической памяти

Д.А. Носов
ООО «Яндекс», Москва
dmitrytvr@gmail.com

Аннотация. В работе М. А. Тайцлина и А. П. Столбоушкина было введено понятие недетерминированной программы, и определен класс языков, распознаваемых недетерминированными программами. Доказана теорема, свидетельствующая о близости этого класса, и $\mathbf{NL}$. Мы исследуем класс языков, определяемый в некоторой модификации указанной вычислительной модели, и доказываем, что рассматриваемый класс языков равен $\mathbf{NL}$.

Ключевые слова: машина Тьюринга; недетерминированные программы; класс сложностей; логарифмическая память; сложность вычислений; универсальная алгебра.

1. Введение

Мы предполагаем, что читатель знаком с понятиями машины Тьюринга, входа, времени работы, и памяти, в которой работает машина Тьюринга, а также со сложностными классами $\mathbf{P}$ и $\mathbf{PSPACE}$. Говорят, что язык L принадлежит сложностному классу $\mathbf{NL}$, если существует недетерминированная машина Тьюринга, которая распознает язык L в логарифмической от длины входа памяти.

Изначально проблема, совпадают ли сложностные классы $\mathbf{NL}$ и $\mathbf{P}$, была сформулирована в работах С. Кука [1], [2], [3]. В этих работах доказаны утверждения, называемые, соответственно, первой, и второй теоремой Кука.

В обеих теоремах Кука рассматривается машина Тьюринга, работающая на входе длины n, и $S(n)$ — функция, которая мажорирует двоичный логарифм от n.

Первая теорема Кука утверждает, что класс языков, распознаваемых на детерминированной машине Тьюринга за время $2^{cS(n)}$, где c — любая

положительная константа, распознается также детерминированными многоленточными машинами Тьюринга со стэком, память которых ограничена $S(n)$. Эта теорема не доказывает совпадение классов $\mathbf{NL}$ и $\mathbf{P}$, именно из-за наличия стэка неограниченного размера. Не получается доказать никаких соотношений между классами языков, распознаваемых машинами Тьюринга со стэком, классов $\mathbf{NL}$ и $\mathbf{P}$.

Вторая теорема Кука, утверждает, что язык, который распознается недетерминированной односторонней многоленточной машиной Тьюринга со стэком, память которой ограничена $S(n)$, распознается также недетерминированной односторонней многоленточной машиной Тьюринга без стэка, за время $2^{cS(n)}$, где константа c не зависит от длины входа машины Тьюринга.

Иммерман доказал, что $\mathbf{NL} = \mathbf{coNL}$ (см. [4]).

Теорема Савича утверждает, что класс $\mathbf{NL} \subseteq \mathbf{L^2}$, где $\mathbf{L^2}$ - класс языков, распознаваемых машинами Тьюринга, работающими на квадратичной от логарифма длины входа памяти (см. [5]).

Подробнее о сложностных классах можно узнать в [6] и [7].

Определение класса недетерминированных программ PR^+ было дано в работе [8]. Авторы доказали, что класс языков, распознаваемых недетерминированными программами из PR^+ совпадает с $\mathbf{NL} \cap L_M$, где L_M — все языки, являющиеся кодами универсальных алгебр. Насколько нам известно, полное доказательство этой теоремы не опубликовано. Данная теорема свидетельствует о том, что сложностной класс $\mathbf{NL}$ близок к классу языков, распознаваемых недетерминированными программами из PR^+. Однако вопрос, можно ли модифицировать модель вычислений основанную на недетерминированных программах таким образом, чтобы сложностной класс $\mathbf{NL}$ совпадал с классом языков, распознаваемых недетерминированными программами, оставался открытым. В данной работе на этот вопрос дается положительный ответ. В классе PR^+ выделяется специальный подкласс недетерминированных программ. Для этого подкласса дается новое определение языка, распознаваемого недетерминированной программой. Пусть $\mathbf{Z}^+_{\Psi}$ — класс всех таких языков. В данной работе доказано, что $\mathbf{NL} = \mathbf{Z}^+_{\Psi}$.

Автор выражает глубокую благодарность научному руководителю, Варновскому Н. П., благодаря которому эта работа обрела законченный облик, а также рецензенту Анохину М.И. за ряд ценных замечаний.

2. Недетерминированные программы

2.1. Сигнатура недетерминированных программ

Каждая недетерминированная программа из класса PR^+ работает на некоторой конечной стандартно заданной универсальной алгебре A. Носителем данной универсальной алгебры является начальный отрезок натуральных чисел $A = \{0,1,\ldots,LAST\}$. Сигнатура рассматриваемых универсальных алгебр имеет вид $\Omega = \langle c_1, c_2, \ldots, c_{n_1}, 0, LAST, ', f_1^{k_1}, f_2^{k_2}, \ldots, f_{n_2}^{k_{n_2}} \rangle$, где $c_1, c_2, \ldots, c_{n_1}$ — константы, каждая из функций $f_j^{k_j}$ — k_j-арная операция на A. Заметим, что в сигнатуре всегда есть унарная операция $'$, возвращающая следующий элемент универсальной алгебры. Другими словами, на носителе универсальной алгебры задана некоторая нумерация κ. 0 — константа, всегда равная первому (по κ) элементу; $LAST$ — константа, всегда равная последнему (по κ) элементу; унарная операция ‘ всегда возвращает следующий элемент универсальной алгебры. Так как $LAST$ — последний элемент, то $LAST'$ всегда равен $LAST$.

Сигнатура каждой недетерминированной программы Λ включает в себя Ω, а также конечное множество X — набор используемых в недетерминированной программе переменных. $\Lambda = \langle \Omega, X \rangle$.

2.2. Синтаксис класса недетерминированных программ PR^+

Недетерминированные программы класса PR^+ сигнатуры Λ задаются по индукции.

- **x = f(x$_1$,..,x$_k$);** — программа, если $x, x_1, .., x_k$ — переменные из X, $f :| A |^k \to A$; f из Ω. Это оператор присваивания.
- **x = x+1;** — программа, если x — переменная из X. Это оператор инкремента.
- **x = y;** — программа, если x, y — переменные из X. Это оператор присваивания.
- **x = c;** — программа, если c — константа из Ω, x — переменная из X. Это оператор присваивания.
- **Y** — программа, если Y — тест.

• $\mathbf{P_1;P_2}$ — программа, если P_1 и P_2 — программы. Это последовательная композиция программ.

• **if(Y) S endif** — программа, если Y — тест, S — программа. Это ветвление.

• $\mathbf{P_1} \cup \mathbf{P_2}$ — программа, если P_1, P_2 — программы. Это параллельная композиция программ.

• **while(Y) S endwhile** — программа, если Y — тест, S — программа. Это цикл.

Тесты:

• $\mathbf{x = y?}$ — программа, если x, y — переменные из X. Это тест.

• $\mathbf{x \neq y?}$ — программа, если x, y — переменные из X. Это тест.

Количество переменных в программе конечно, фиксировано, и равно мощности множества X.

Заметим, что $\mathbf{x = x + 1;}$ — это оператор недетерминированной программы, который в результате выполнения меняет значение переменной x на x'. Семантика всех операторов будет рассмотрена ниже.

Определение 1 *Для теста* $\mathbf{x = y?}$*, где* x, y *— переменные из* X, $\mathbf{x \neq y?}$ *— это обратный тест. Мы будем обозначать обратный тест к* T *через* $!T$. *Аналогично, для теста* $\mathbf{x \neq y?}$, $\mathbf{x = y?}$ *— это обратный тест.*

Определение 2 *Класс недетерминированных программ* PR^+ *— это все недетерминированные программы всех сигнатур.*

2.3. Семантика недетерминированных программ

Определение 3 *Путь вычислений — конечная последовательность операторов и тестов.*

По каждой недетерминированной программе можно построить не более чем счетное множество путей вычислений, которое однозначно определяет работу этой недетерминированной программы.

Построение будем осуществлять по индукции.

Через $\alpha = \{\alpha_1, \alpha_2, \ldots\}$ обозначим множество путей вычислений недетерминированной программы P_1; $\beta = \{\beta_1, \beta_2, \ldots\}$ — это множество путей вычислений недетерминированной программы P_2. α_i и β_i в данных обозначениях — пути вычислений.

Определение 4 *Множество путей вычислений программы* P.

• Программе P, состоящей из одного оператора присваивания, инкремента или теста, соответствует множество путей вычислений программы P, в котором ровно один путь вычислений, состоящий из одного этого оператора(теста).

• Программе P, являющейся последовательной композицией программ, $P = P_1; P_2$ соответствует множество путей вычислений, состоящее из всех путей вычислений вида $\alpha_i; \beta_j$, для каждого α_i из α и β_j из β. Множества путей вычислений α и β недетерминированных программ P_1 и P_2 уже построены по индукции. Через $\alpha_i; \beta_j$ мы обозначаем конкатенацию путей вычислений.

• Программе P, являющейся параллельной композицией программ, $P = P_1 \cup P_2$ соответствует множество путей вычислений являющееся объединением (в теоретико-множественном смысле) множеств α и β. Как и в предыдущем случае, множества путей вычислений α и β недетерминированных программ P_1 и P_2 уже построены по индукции.

• Программе P, являющейся условным оператором вида **if(Y) P_1 endif** соответствует объединение (в теоретико-множественном смысле) двух множеств путей вычислений:

- $Y; \alpha_i$ — конкатенация теста Y с каждым α_i из α.

- $!Y; \alpha_i$ — конкатенация теста, обратного к Y с каждым α_i из α.

• Программе P, являющейся оператором цикла **while(Y) P_1 endwhile** соответствует счетное множество путей вычислений:

- $!Y;$ — тест, обратный к Y.

- $Y; \alpha_i; !Y$ — конкатенация теста Y с каждым α_i из α, далее тест, обратный к Y.

- $Y; \alpha_{i_1}; Y; \alpha_{i_2}; !Y$ — дважды повторенная конкатенация теста Y с каждым α_{i_1}, и с каждым α_{i_2} из α, далее тест, обратный к Y.

- $\{Y; \alpha_{i_k};\}_j !Y$ — j раз повторенная конкатенация теста Y с каждым α_{i_k} из α, далее тест, обратный к Y. Пути вычисления α_{i_k} на разных

итерациях принимают все возможные значения из α независимо друг от друга.

В этом случае α — не более чем счетное по индукции, а значит, мы имеем счетное объединение не более чем счетных множеств, которое, также, не более чем счетно.

3. Класс Z^+

Мы рассматриваем недетерминированные программы произвольной сигнатуры $\Lambda = \langle \Omega, X \rangle$.

Определение 5 *Вход недетерминированной программы сигнатуры* $\Lambda = \langle \Omega, X \rangle$ *— это универсальная алгебра* A *сигнатуры* Ω.

Другими словами, вход недетерминированной программы — это конкретные значения всех констант и функций сигнатуры Λ недетерминированной программы.

Пусть $\alpha_i = (h_1, h_2, \ldots, h_k)$ — это некоторый путь вычислений из множества путей вычислений программы P; каждый из h_t — это оператор или тест; универсальная алгебра A — вход программы P.

Определение 6 *Состояние пути вычислений* α_i *при работе на универсальной алгебре* A *— это функция* $s(t,x)$*, действующая в* $|A|$. $t = 0,1,2\ldots k$ *— это номер текущего оператора (теста),* $x \in X$ *— переменная.*

• В начальный момент времени все переменные принимают значение 0, то есть $s(0,x) = 0$, для каждой переменной x.

• Пусть в момент времени t выполняется оператор $x = f(x_1,..,x_k)$. В этом случае $s(t+1,x) = f(s(t,x_1),..,s(t,x_k))$; $s(t+1,y) = s(t,y)$ для любой переменной y, не совпадающей с x; значение функции f определяется универсальной алгеброй A.

• ?$x = y$. В этом случае $s(t+1,x) = s(t,x)$ для любой переменной x.

• ?$x \neq y$. В этом случае $s(t+1,x) = s(t,x)$ для любой переменной x.

• $x = y$. В этом случае $s(t+1,x) = s(t,y)$; $s(t+1,z) = s(t,z)$ для любой переменной z не совпадающей с x.

• $x = c$. В этом случае $s(t+1, x) = c$; $s(t+1, y) = s(t, y)$ для любой переменной y, не совпадающей с x. Значение константы c определяется универсальной алгеброй A.

• $x = x+1$. В этом случае $s(t+1, x) = s(t, x)'$. Данное определение корректно в силу того, что операция ‘ по определению есть в каждой сигнатуре. Соответственно, если $s(t, x)$ есть $LAST$, то $s(t+1, x) = s(t, x)$. В обоих случаях, $s(t+1, y) = s(t, y)$ для любой переменной y, не совпадающей с x.

Каждый из тестов в рассматриваемом пути вычислений, на входе A, может быть либо истинным, либо ложным.

Определение 7 *Тест* $?x = y$ *истинен в рассматриваемом пути вычислений, на входе* A*, тогда и только тогда, когда* $s(t, x) = s(t, y)$.

Определение 8 *Тест* $?x \neq y$ *ложен в рассматриваемом пути вычислений, на входе* A*, тогда и только тогда, когда* $s(t, x) = s(t, y)$.

Недетерминированная программа, также как и машина Тьюринга–распознаватель либо принимает вход, или не принимает.

Определение 9 *Недетерминированная программа* P *принимает вход* A*, если существует путь вычислений* α_i*, из множества путей вычислений* P*, при работе которого на* A*, все тесты в* α_i *истинны.*

Код универсальной алгебры A сигнатуры Ω — слово в конечном алфавите, однозначно соответствующее A.

Пусть мы кодируем универсальную алгебру A сигнатуры

$$\Omega = \langle c_1, c_2, \ldots, c_{n_1}, 0, LAST, f_1^{k_1}, f_2^{k_2}, \ldots, f_{n_2}^{k_{n_2}} \rangle.$$

Слово, кодирующее эту систему:

• разделитель $\#$;

• N символов $|$, где $N = LAST + 1$ — количество элементов в носителе $|A|$ универсальной алгебры A;

• разделитель $\#$;

• значения констант $c_1, c_2, \ldots, c_{n_1}$, в унарной системе счисления, разделенные специальным символом $\%$;

• разделитель $\#$;

• таблицы функций $f_1^{k_1}, f_2^{k_2}, \ldots, f_{n_2}^{k_{n_2}}$, разделенные специальным символом $\%$;

• разделитель $\#$.

Таблица функции $f_i^{k_i}$ — это последовательность из $N^{k_i}+1$ символов 0 и 1, где на месте $s = x + j_1 * N + j_2 * N^2 + \ldots + j_{k_i} * N^{k_i}$ (где каждое из чисел $x, j_1, \ldots, j_{k_i}$ меньше N) стоит 1 тогда и только тогда, когда $f(j_1, \ldots, j_{k_i}) = x$.

Определение 10 $L_{M\Omega}$ *— это множество всех слов, кодирующих какую-либо универсальную алгебру сигнатуры* Ω.

Определение 11 L_M *— это множество, элементами которого являются все возможные подмножества всех множеств* $L_{M\Omega}$, *для всех возможных сигнатур* Ω.

Определим класс языков, распознаваемых недетерминированными программами. Итоговый класс зависит от принятого кодирования универсальных алгебр.

Определение 12 *Недетерминированная программа* P *сигнатуры* $\Lambda = \langle \Omega, X \rangle$ *распознает язык* L *в алфавите* $\{|, 0, \#, 1\}$, *если выполнено следующее условие:* $w \in L$ *тогда и только тогда, когда*

• w является кодом некоторой конкретной универсальной алгебры A сигнатуры Ω, то есть $w \in L_{M\Omega}$;

• недетерминированная программа P принимает вход A.

Определение 13 *Класс языков* Z^+ *— это все языки, распознаваемые недетерминированными программами из класса* PR^+.

Справедлива следующая теорема, авторами которой являются М. А. Тайцлин и А. П. Столбоушкин, [8].

Теорема 14 $Z^+ = \mathbf{NL} \cap L_M$.

4. Класс Z^{+}_{Ψ}

В классе $\mathbf{Z}^{+}_{\Psi}$, в отличие от предыдущего класса, рассматриваются недетерминированные программы фиксированной сигнатуры. Всюду далее мы будем считать, что сигнатура $\Psi = \langle 0, LAST, a, b, err, ', f^{(1)} \rangle$. $\Lambda = \langle \Psi, X \rangle$. X, как и ранее конечное множество переменных недетерминированной программы; $0, LAST, a, b, err$ — константы.

Определение 15 *Класс недетерминированных программ* PR^{+}_{Ψ} *— это все недетерминированные программы сигнатуры* $\Lambda = \langle \Psi, X \rangle$.

Определение 16 *Универсальная алгебра* A *принадлежит классу универсальных алгебр* $\mathsf{A}(\Psi)$*, если*

- сигнатура A — это $\Psi = \langle 0, LAST, a, b, err, ', f^{(1)} \rangle$;
- носитель $A = \{err, a, b, 0, \ldots, N\}$;
- $f(i) \in \{a, b\}$, если $i \in 0,..,N$;
- $f(i) = err$ в противном случае;
- значения констант err, a, b это, соответственно, элементы err, a, b носителя универсальной алгебры.

Определение 17 *Код универсальной алгебры* $A \in \mathsf{A}(\Psi)$ *— это слово* w *длины* $N+1$ *в алфавите* $\{a, b\}$*, такое, что на i-ой позиции в слове* w *стоит значение* $f(i)$. N *— это число из носителя* $|A| = \{err, a, b, 0, \ldots, N\}$.

Утверждение 18 *Существует взаимно однозначное соответствие кодов* $\mathsf{A}(\Psi)$ *и слов в алфавите* $\{a, b\}$.

Доказательство. Очевидно следует из того, что каждая универсальная алгебра из $\mathsf{A}(\Psi)$ однозначно характеризуется значениями своей функции f.

Утверждение 19 *Множество кодов универсальных алгебр* $\mathsf{A}(\Psi)$ *— это все слова в алфавите* $\{a, b\}$.

Доказательство. Это утверждение очевидно, так как выше доказано, что существует взаимно однозначное соответствие кодов $\mathsf{A}(\Psi)$ и слов в алфавите $\{a, b\}$.

Определение 20 *Недетерминированная программа* $P \in PR^{+}_{\Psi}$ *распознает язык L, если* $w \in L$ *тогда и только тогда, когда*

• w является кодом некоторой конкретной универсальной алгебры A из $\mathsf{A}(\Psi)$;

• недетерминированная программа P принимает вход A из пункта 1.

Определение 21 *Класс языков* $\mathbf{Z}^{+}_{\Psi}$ *— это все языки, распознаваемые недетерминированными программами из класса* PR^{+}_{Ψ}.

Теорема 22 $\mathbf{Z}^{+}_{\Psi} = \mathbf{NL}$.

Доказательство. Всюду далее, без ограничения общности, будем считать, что рассматриваемая машина Тьюринга начинает работу в состоянии q_0, и имеет два заключительных состояния — q_{acc} и q_{rej}. В случае, если машина Тьюринга принимает слово, записанное на входной ленте, то существует конечная последовательность команд, применяя которые машина перейдет в состояние q_{acc}. В противном случае машина Тьюринга зацикливается, или переходит в q_{rej}. У нескольких команд машины Тьюринга может быть одинаковая левая часть, в этом случае на исполнение будет недетерминированно выбрана одна из этих команд.

В доказательстве мы не будем рассматривать случай, когда слово w пусто. В этом, и только в этом случае входная лента машины Тьюринга пуста. В этом, и только в этом случае носитель универсальной алгебры A состоит только из $\{err, a, b\}$; значения констант $0, LAST$ программы P равны err; а также для любого i $f(i) = err$. Длины всех рабочих лент в этом случае ограничены константой.

1. Докажем, что $\mathbf{Z}^{+}_{\Psi} \subseteq \mathbf{NL}$.

Пусть $L \in \mathbf{Z}^{+}_{\Psi}$, P — недетерминированная программа из класса PR^{+}_{Ψ}, распознающая этот язык. Это значит, что для каждого слова $w \in L$ существует универсальная алгебра A из $\mathsf{A}(\Psi)$, такая что w является кодом A, и P принимает вход A. Нам необходимо доказать, что $L \in \mathbf{NL}$. Для этого достаточно построить недетерминированную машину Тьюринга M, работающую в логарифмической памяти с одной входной и несколькими рабочими лентами, которая принимает все слова $w \in L$, и только их.

Количество рабочих лент машины M в данном доказательстве зависит от количества переменных в недетерминированной программе. Точное число рабочих лент:

• по одной рабочей ленте для каждой переменной недетерминированной программы,

• одна рабочая лента для выполнения оператора функционального присваивания,

• по одной ленте для каждой константы.

До начала работы машины M на входной ленте записано слово w, все рабочие ленты пусты. На всех лентах машины Тьюринга мы будем использовать заранее фиксированный алфавит.

Утверждение 23 *Значение любой переменной программы P может быть записано на одной рабочей ленте.*

Доказательство. Рассмотрим слово w длины $N+1$, записанное на входной ленте машины M. Носитель универсальной алгебры $A=\{err,a,b,0,\ldots,N\}$. Каждая из переменных недетерминированной программы P может принимать значение от 0 до N, а также значения a,b,err. На входной ленте машины M записано ровно $N+1$ символов, а длина рабочей ленты ограничена логарифмом от длины входа, следовательно, любое значение переменной недетерминированной программы может быть записано в двоичной системе счисления на любой рабочей ленте.

Пусть в рассматриваемой недетерминированной программе k переменных, тогда текущее состояние всех переменных может быть записано на k рабочих лентах машины Тьюринга.

До начала выполнения программы, все переменные принимают значение 0, соответственно на каждой рабочей ленте, отвечающей за соответствующую переменную, записано значение, соответствующее 0. В начале работы машина Тьюринга M запишет на специальные рабочие ленты значения констант недетерминированной программы, в том числе констант $0,LAST,err$.

В ходе доказательства мы сначала построим машины Тьюринга, которые соответствуют операторам недетерминированной программы. Эти "базисные" машины изменяют состояние рабочих лент, и переходят в свое состояние q_{acc}, когда моделирование оператора недетерминированной программы завершено. Когда моделируются тесты недетерминированной программы, то соответствующая "базисная" машина Тьюринга переходит в q_{acc}, или в q_{rej}.

Далее мы рассмотрим, как построить машины Тьюринга, соответствующие более сложным программам, по индукции. Эти построения будут осуществляться, в основном, изменением состояния машины Тьюринга, и не будут менять ее рабочие ленты.

Следующее утверждение является базисом индукции по сложности программы.

Утверждение 24 *Если программа* P *состоит из одного оператора, то соответствующая "базисная" машина Тьюринга* M *существует.*

Доказательство. Рассмотрим различные программы из одного оператора.

- $P =$ **x = f(y);** — оператор присваивания. В этом случае, соответствующая машина M должна записать на ленту, соответствующую переменной x, значение функции f от y. Напомним, что слово, кодирующее функцию f записано на входной ленте. Для выполнения данного оператора нам понадобится дополнительная рабочая лента, которая, как и все ленты, ограничена логарифмом от длины входа. Работа данного оператора происходит следующим образом: машина M смещается по входной ленте, соответствующей данной функции, на y ячеек. Для этого используется дополнительная лента. Далее необходимо присвоить значение a или b, в зависимости от обозреваемого символа на входной ленте, переменной x. Для этого надо скопировать значение, соответствующее a или b, хранящееся на специальной рабочей ленте, на ленту соответствующую переменной x. После этого машина Тьюринга M переходит в заключительное состояние q_{acc}.

- $P =$ **x = x + 1;** — оператор инкремента. В этом случае, соответствующая машина M должна прибавить единицу в двоичной системе счисления на ленте, соответствующей переменной x, и перейти в заключительное состояние q_{acc}.

- $P =$ **x = y;** — оператор присваивания. В этом случае, соответствующая машина M должна скопировать ленту, соответствующую переменной y, на ленту, соответствующую переменной x, и перейти в заключительное состояние q_{acc}.

- $P =$ **x = c;** — оператор присваивания. В этом случае, соответствующая машина M должна скопировать ленту, соответствующую константе c, на ленту, соответствующую переменной x, и перейти в заключительное состояние q_{acc}.

• P = **x = y?** — положительный тест. В этом случае, соответствующая машина M должна удостовериться, что ленты, соответствующие переменным x и y совпадают. Иными словами, если ленты совпадают, соответствующая машина M переходит в состояние q_{acc}, а если не совпадают — то в q_{rej}.

• P = **x ≠ y?** — отрицательный тест. В этом случае, соответствующая машина M должна удостовериться, что ленты, соответствующие переменным x и y различны хотя бы в одном символе. Иными словами, если ленты совпадают соответствующая машина M переходит в состояние q_{rej}, а если не совпадают — то в q_{acc}.

Следующее утверждение является шагом индукции по сложности программы.

Утверждение 25 *Если программа P является композицией программ, ветвлением, или циклом, и машины Тьюринга, моделирующие составные части программы P, существуют, то существует машина Тьюринга M, моделирующая программу P.*
Доказательство.

Определение 26 *Безусловный переход из состояния q_1 в состояние q_2 машины Тьюринга M — это переход из состояния q_1 в состояние q_2, при котором не происходит движения головок машины Тьюринга по лентам, не происходит изменения рабочих лент, и который происходит при любых символах, которые читаются в текущий момент головками машины Тьюринга.*
Введем условные обозначения: машину Тьюринга, соответствующую недетерминированной программе P_1 мы обозначим через M_1; машину Тьюринга, соответствующую недетерминированной программе P_2 мы обозначим через M_2; машину Тьюринга, соответствующую тесту Y мы обозначим через M_T; результирующую машину Тьюринга мы будем обозначать через M. Здесь недетерминированные программы P_1 и P_2 — это составные части программы P.

• $P = \mathbf{P_1; P_2}$ — последовательная композиция программ P_1 и P_2. В этом случае результирующая машина M делает безусловный переход из q_{acc} машины Тьюринга M_1 в начальное состояние машины Тьюринга M_2, и безусловный переход из q_{rej} машины Тьюринга M_1 в q_{rej} машины Тьюринга, соответствующей P_2 (M_2). Начальным состоянием результирующей машины M будет начальное состояние M_1; заключительными — заключительные состояния M_2. Далее необходимо преобразовать полученную машину, чтобы в ней было одно входное состояние, и два заключительных — q_{acc} и q_{rej}.

• $P = \mathbf{if(Y)\ P_1\ e\ n}$ — ветвление. Начальным состоянием результирующей машины M будет начальное состояние машины M_T. заключительными состояниями — заключительные состояния M_1. Также в результирующей машине M есть безусловный переход из q_{acc} машины M_T в начальное состояние M_1, и безусловный переход из q_{rej} машины M_T в q_{acc} машины M_1.

• $P = \mathbf{P_1} \cup \mathbf{P_2}$ — параллельная композиция программ. Введем новые состояния — начальное и два заключительных для машины M. В результирующей машине M будут следующие безусловные переходы:

- из начального состояния машины M в начальное состояние M_1;
- из начального состояния машины M в начальное состояние M_2;
- из q_{acc} машины M_1 в q_{acc} машины M;
- из q_{acc} машины M_2 в q_{acc} машины M;
- из q_{rej} машины M_1 в q_{rej} машины M;
- из q_{rej} машины M_2 в q_{rej} машины M;

• Пусть программа P это **while(Y) $\mathbf{P_1}$ endwhile** — цикл. Мы вводим новые состояния — начальное и два заключительных для машины M. Мы добавляем безусловные переходы:

- из начального состояния машины M в начальное состояние M_T;

- из q_{rej} машины M_T в q_{acc} машины M;

- из q_{rej} машины M_1 в q_{rej} машины M;

- из q_{acc} машины M_T в начальное состояние машины M_1;

- из q_{acc} машины M_1 в начальное состояние машины M_T;

На этом построение машины Тьюринга M завершено.

Лемма 27 *Если программа P приняла вход w, то машина Тьюринга M принимает вход w.*

Доказательство. Если программа P приняла вход w, то в ней существует путь вычислений α_i, при работе которого на входе w, все тесты в нем истинны. Машина Тьюринга M принимает вход w тогда и только тогда, когда в ней есть ветвь вычислений, завершающаяся в состоянии q_{acc}.

Рассмотрим ветвь вычислений y, соответствующую этому пути вычислений α_i. В этой ветви вычислений все циклы отработают в точности столько же раз, как и в α_i; все операторы объединения будут работать так же, как и в α_i. В результате ветвь вычислений y, после выполнения каждого блока команд, соответствующих одному оператору, будет моделировать **состояние** α_i. Но это значит, что, так как все тесты в α_i истинны, в ветви вычислений y все блоки команд, соответствующих одному оператору, будут завершаться в промежуточном состоянии q_{acc}. А значит, и вся ветвь y целиком завершится в состоянии q_{acc}, что и требуется.

Лемма 28 *Если программа P не принимает вход w, то и машина Тьюринга M не принимает вход w.*

Доказательство. Машина Тьюринга M не принимает вход w тогда и только тогда, когда в M нет ни одной ветви вычислений, завершающаяся в состоянии q_{acc}. Если программа P не приняла вход w, значит каждый путь вычислений в ней либо бесконечен, либо содержит ложный тест.

Будем проводить доказательство от противного — пусть некоторая ветвь вычислений y машины M завершается в состоянии q_{acc}. Заметим, что, по построению машины M, переход из промежуточного состояния q_{rej} в состояние q_{acc} возможен **только** в цикле. Построим путь вычисления α_y по

ветви вычислений y. Он конечен. Так как y закончилась в состоянии q_{acc}, то в α_y нет ни одного теста, который ложен. Но существование такого пути вычислений невозможно по исходному предположению. Из этого следует, что если программа P не принимает вход w, то и машина Тьюринга M не принимает вход w.

2. Докажем, что $\mathbf{NL} \subseteq \mathbf{Z}_{\Psi}^{+}$.

Пусть язык $L \in \mathbf{NL}$. Это значит, что язык L распознается машиной Тьюринга M, работающей в логарифмической памяти. Нам необходимо построить недетерминированную программу P из класса PR_{Ψ}^{+}, которая распознает этот язык L.

Без ограничения общности можно считать, что моделируемая машина Тьюринга M имеет одну входную ленту, одну рабочую ленту, каждая лента машины Тьюринга является полубесконечной, и длина рабочей ленты ограничена константой, умноженной на логарифм от длины входной ленты. Благодаря этой константе нам не важно основание логарифма.

Вход машины Тьюринга — слово в алфавите $\{a,b\}$, которому однозначно соответствует универсальная алгебра из $\mathsf{A}(\Psi)$, на которой работает недетерминированная программа.

При длине входа N, количество различных вариантов заполнения рабочей ленты равно $d^{\log_{c'} N} = N^{v}$, для некоторой константы v. Учитывая, что каждая переменная недетерминированной программы принимает значения от 0 до $N-1 = LAST$, а также a, b, err, то с помощью v переменных недетерминированной программы можно закодировать рабочую ленту.

Количество состояний в машине Тьюринга конечно, следовательно их можно закодировать с помощью конечного числа переменных.

Позиция на входной ленте — это число от 0 до N^{u}, где u — максимальная арность функции подающейся на вход, а значит также может быть закодирована с помощью u переменных.

Определение 29 *Кортеж переменных — это последовательность переменных недетерминированной программы, кодирующая*

- состояние машины Тьюринга;
- или состояние одной рабочей ленты машины Тьюринга;
- или позицию на входной ленте.

Обозначим кортеж, соответствующий состоянию q через x_q. Кортеж, соответствующий текущему состоянию обозначим через x_{curQ}. Кортеж, соответствующий рабочей ленте обозначим $work$. Подчеркнем, что в программе нет кортежа, соответствующего входной ленте (но есть кортеж с позицией на входной ленте). Будем обозначать кортеж — текущую позицию на входной ленте за $posV$. Позиция на рабочей ленте — число, меньшее $\log_{c'} N$ и также может быть записана в одной переменной. Для единообразия, можно считать ее кортежем из одной переменной. Будем обозначать позицию на рабочей ленте за $posW$.

Так как все операции осуществляются с кортежами переменных (разной длины), операции сравнения кортежей, добавления и вычитания единицы к кортежу являются элементарными подпрограммами.

Элементарные подпрограммы

При моделировании машин Тьюринга мы будем использовать следующие подпрограммы:

• Программа $Input(i)$, вычисляющая i-ый символ на входной ленте машины M.

• Программа $WorkSymbol(w,i)$, вычисляющая i-ый символ на рабочей ленте, соответствующей кортежу w.

• Программа $WriteSymbol(w,s,i)$, моделирующая запись символа s на i-ую позицию на рабочей ленте, соответствующей кортежу w.

• Программа $ShiftInput(posV,d)$, выполняющая оператор $i = i + d$ для счетчика, соответствующего входной ленте. $d \in \{-1,0,1\}$ — это сдвиг.

• Программа $ShiftWork(posW,d)$, выполняющая оператор $i = i + d$ для счетчика, соответствующего рабочей ленте. $d \in \{-1,0,1\}$ — это сдвиг.

Определение 30 *Подпрограмма — это набор операторов недетерминированной программы.*

Лемма 31 *Каждая команда машины Тьюринга* M *может быть промоделирована подпрограммой.*

Доказательство. Рассмотрим произвольную команду com машины Тьюринга. Она имеет вид:

$com = \langle cq, ca, cb \rangle \to \langle cq', cw, cs_1, cs_2 \rangle$, где

• cq и cq' — состояния;

• ca — символ, который машина Тьюринга наблюдает в текущий момент на входной ленте;

• cb — символ, который машина Тьюринга наблюдает в текущий момент на рабочей ленте;

• cw — символ, который машина Тьюринга запишет на рабочую ленту в текущую позицию;

• cs_1 — сдвиг на входной ленте;

• cs_2 — сдвиг на рабочей ленте.

Как уже говорилось выше, несколько команд машины Тьюринга могут иметь одинаковую левую часть.

Каждая команда машины Тьюринга преобразуется в собственную подпрограмму.

В подпрограмме, соответствующей команде com машины Тьюринга M вначале проверяется, что

• кортеж текущего состояния x_{curQ} равен x_{cq}, соответствующему состоянию cq из левой части команды com;

• на входной ленте в текущей позиции записан правильный символ, то есть $Input(posV)$ совпадает с ca;

• на рабочей ленте в текущей позиции записан правильный символ, то есть $WorkSymbol(work, posW)$ совпадает с cb.

Все эти проверки являются вложенными условными операторами.

Если все эти условия выполнены, то подпрограмма далее

• меняет кортеж состояния x_{curQ} на $x_{cq'}$;

• меняет кортеж, кодирующий рабочую ленту, выполняя $WriteSymbol(work, cw, posW)$;

• меняет текущие позиции на входной и рабочей лентах (выполняя элементарные подпрограммы $ShiftInput(posV, cs_1)$ и $ShiftWork(posW, cs_1)$).

Заметим, что подпрограмма, соответствующая одной команде машины Тьюринга, обладает следующим свойством: в этой программе все тесты выполнены тогда и только тогда, когда данная команда может быть применена, то есть выполнены все условия левой части команды. Это и означает, что подпрограмма моделирует команду машины Тьюринга.

Построение программы P, моделирующей машину Тьюринга M.

Далее все полученные таким образом программы $s_1, \ldots, s_d$, где d — количество команд машины Тьюринга объединяются следующим образом: Пока текущее состояние моделируемой машины M не заключительное (то есть не q_{acc} или q_{rej}), мы выполняем **объединение** всех подпрограмм, соответствующих командам машины Тьюринга.

(p = 0)

while(p == 0)

$\mathbf{P_1 \cup P_2 \cup \ldots \cup P_d}$

if(x_{curQ} == $x_{q_{acc}}$)p = err;

endwhile.

На этом построение программы P завершено.

По определению, машина Тьюринга распознает некоторое слово тогда и только тогда, когда существует последовательность команд машины Тьюринга, переводящая ее в заключительное состояние q_{acc}. В случае, если существует последовательность команд машины Тьюринга, переводящая ее в заключительное состояние, будет существовать и конечный путь вычислений, соответствующий данной последовательности команд.

Лемма 32 *Если машина Тьюринга M принимает вход w, то и программа P принимает вход w.*

Доказательство. Если машина Тьюринга M принимает вход w, то в ней есть ветвь вычислений, завершающаяся в состоянии q_{acc}. Если в программе P при работе на входе w есть путь вычислений, на котором все тесты истинны, то программа P принимает вход w. Соответственно, нам необходимо доказать, что если в исходной машине Тьюринга M была ветвь вычислений, завершающаяся в состоянии q_{acc}, то в программе P при работе на входе w будет путь вычислений, на котором все тесты истинны.

Рассмотрим ветвь вычислений y, завершающаяся в состоянии q_{acc}. Ей соответствует набор подпрограмм $P_{i_1}, P_{i_2}, \ldots, P_{i_k}$, таких что P_{i_j} соответствует j-ой команде ветви y. Рассмотрим путь вычислений r, полученный последовательной композицией этих подпрограмм, однозначно определяемый по одной ветви вычислений M. В конец r добавим тест

$x_{curQ} = x_{q_{acc}}$. Дополним путь вычислений r тестами $p == 0$, $x_{curQ} \neq x_{q_{acc}}$, между каждой из подпрограмм P_{i_j} и $P_{i_{j+1}}$. В результате r будет путем вычислений из множества путей вычислений итоговой программы P.
Докажем, что на r все тесты истинны.

• Все тесты в подпрограммах P_{i_j} истинны, так как они соответствуют условиям, при которых применялись команды ветви вычислений y машины Тьюринга M.

• Все добавленные тесты $p == 0$, $x_{curQ} \neq x_{q_{acc}}$ истинны, так как при выполнении ветви вычислений y состояние машины Тьюринга равно q_{acc} только после выполнения последней из команд ветви вычислений y, но не ранее.

• Добавленный тест $x_{curQ} == x_{q_{acc}}$ истинен, так как в конце выполнения ветви вычислений y состояние машины Тьюринга действительно равно q_{acc}.

Никаких других тестов в рассматриваемом пути вычислений нет, а, значит, программа P действительно принимает вход w.

Лемма 33 *Если машина Тьюринга M не принимает вход w, то и программа P не принимает вход w.*

Доказательство. Если машина Тьюринга M не принимает вход w, то в ней нет ни одной ветви вычислений, завершающаяся в состоянии q_{acc}. Программа P не принимает вход w тогда и только тогда, когда в ней при работе на входе w нет ни одного пути вычислений, на котором все тесты истинны. Соответственно, нам необходимо доказать, что если в исходной машине Тьюринга M не было ветви вычислений, завершающейся в состоянии q_{acc}, то в программе P при работе на входе w не будет ни одного пути вычислений, на котором все тесты истинны.

Пусть в исходной машине Тьюринга M не было ветви вычислений, завершающейся в состоянии q_{acc}. Будем доказывать утверждение от противного. Предположим, что существует путь вычислений программы P, на котором все тесты истинны. По построению итоговой программы, чтобы выйти из цикла результирующей программы P, должен был быть истинным тест $x_{curQ} == x_{q_{acc}}$. Однако это значит, что в результате выполнения

некоторого конечного набора подпрограмм $P_{i_1}, P_{i_2}, \ldots, P_{i_k}$ верно, что $x_{curQ} == x_{q_{acc}}$. По построению подпрограмм P_{i_j}, это значит, что в машине Тьюринга M существует ветвь вычислений, завершающаяся в состоянии q_{acc}, что неверно. Полученное противоречие доказывает утверждение.

Список литературы.

[1] Cook S.A. Variations on push-down machines In Proc. 1st ACM Symp. on Theory of Computing, pp. 229–232, 1969.

[2] Cook S.A. Characterizations of push–down machines in terms of time–bounded computers Journal of the ACM, 18(1), pp. 4–18, 1971.

[3] Cook S.A. Senti R. Storage requirements for deterministic polynomial time recognizable languages Journal of Computer and System Sciences, 13(1), pp. 25–37, 1976.

[4] Immerman N. Nondeterministic space is closed under complementation, SIAM Journal on Computing 17, pp. 935–938, 1988.

[5] Savitch W.J. Relationships between nondeterministic and deterministic tape complexities, Journal of Computer and System Sciences 4 (2), pp. 177–192, 1970.

[6] Papadimitriou C. Computational Complexity. Boston: Addison-Wesley. ISBN 0-201-53082-1, 1994.

[7] Arora S., Barak B. Computational complexity. A modern approach. Cambridge University Press. ISBN 978-0-521-42426-4, 2009.

[8] Столбоушкин А.П., Тайцлин М.А. Динамические логики. Кибернетика и вычислительная техника под ред. В.А. Мельникова, том 2, Наука, Москва, стр. 180–230, 1986.

Syntactical characterization of nondeterministic logspace complexity class

D.A. Nosov
Yandex, Moscow, Russia
dmitrytvr@gmail.com

Abstract. $\mathbf{NL}$ is defined as the class of languages recognizable by logspace nondeterministic Turing machines. One of the main unsolved problems in complexity theory is that of relation between classes $\mathbf{P}$ and $\mathbf{NL}$. It is known that $\mathbf{NL}$ is contained in $\mathbf{P}$, since there is a polynomial-time algorithm for 2-satisfiability, but it is not known whether $\mathbf{NL} = \mathbf{P}$ or whether $\mathbf{NL} = \mathbf{L}$. A possible approach to these problems can be based on searching for an alternative suitable definition of the class $\mathbf{NL}$. Taitslin et al. propose such a definition in terms of nondeterministic programs. The syntax of such programs is similar to that of usual computer programs. Each nondeterministic program takes a finite universal algebra as input. Taitslin et al. defined a class of languages recognizable by such programs and proved that this class is a subclass of $\mathbf{NL}$. In the present paper, we slightly modify their syntactical definition. Namely, we modify a definition of nondeterministic program input and give a new definition of a language recognized by a given program. We prove that the class of languages recognizable by nondeterministic programs according to our definition is just $\mathbf{NL}$.

Keywords: Turing machine; nondeterministic program; complexity class; logspace complexity; universal algebra

References

[1] Cook S.A. Variations on push-down machines In Proc. 1st ACM Symp. on Theory of Computing, pp. 229–232, 1969.
[2] Cook S.A. Characterizations of push–down machines in terms of time–bounded computers Journal of the ACM, 18(1), pp. 4–18, 1971.
[3] Cook S.A. Senti R. Storage requirements for deterministic polynomial time recognizable languages Journal of Computer and System Sciences, 13(1), pp. 25–37, 1976.
[4] Immerman N. Nondeterministic space is closed under complementation, SIAM Journal on Computing 17, pp. 935–938, 1988.
[5] Savitch W.J. Relationships between nondeterministic and deterministic tape complexities, Journal of Computer and System Sciences 4 (2), pp. 177–192, 1970.
[6] Papadimitriou C. Computational Complexity. Boston: Addison-Wesley. ISBN 0-201-53082-1, 1994.
[7] Arora S., Barak B. Computational complexity. A modern approach. Cambridge University Press. ISBN 978-0-521-42426-4, 2009.
[8] Stolboushkin A.P., Taitslin M.A. Dinamicheskie logiki. [Dynamic logics] Kibernetika i vychislitel'naya tekhnika [Cybernetics and computer technology] V.A. Melnikov (editor). Moscow, Nauka, pp. 180–230, 1986 (in Russian).