ИСП РАН

ТРУДЫ ИНСТИТУТА СИСТЕМНОГО ПРОГРАММИРОВАНИЯ РАН

PROCEEDINGS OF THE INSTITUTE FOR SYSTEM PROGRAMMING OF THE RAS

ISSN PRINT
2079-8156
ТОМ 31
ВЫПУСК 4

ISSN ONLINE
2220-6426
VOLUME 31
ISSUE 4

ИНСТИТУТ СИСТЕМНОГО ПРОГРАММИРОВАНИЯ
ИМ. В.П. ИВАННИКОВА РАН

МОСКВА, 2019

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

Труды Института системного программирования РАН
Proceedings of the Institute for System Programming of the RAS

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

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

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

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

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

Труды ИСП РАН реферируются и/или индексируются в:

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

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

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

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

Open access to the journal content allows to provide public access to the research results and to support global exchange of knowledge. **Proceedings of ISP RAS** is abstracted and/or indexed in:

УДК004.45

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

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

Содержание

Table of Contents

DOI: 10.15514/ISPRAS-2019-31(4)-1

Средства трассировки ОС РВ семейства «Багет»

А.Н. Годунов, ORCID: 0000-0001-5952-9185 <nkag@niisi.ras.ru>
Ф.Н. Чемерев, ORCID: 0000-0002-6494-716X <nkfedor@niisi.ras.ru>
ФГУ ФНЦ Научно-исследовательский институт системных исследований РАН,
117218, Россия, г. Москва, Нахимовский пр., д. 36, к. 1

Аннотация. Статья посвящена проблемам построения средств трассировки систем жесткого реального времени. В настоящее время практически в каждой операционной системе реального времени (ОС РВ) имеются программные средства трассировки событий. Их задача состоит в поиске «обычных» программных ошибок (с которыми не справляются традиционные отладчики) и ошибок реального времени. При этом приходится анализировать не только последовательности событий, но и «утечки» памяти, динамику состояний процессора и потоков управления (профилирование), состояний семафоров, мьютексов и других средств синхронизации, а также очереди потоков управления, ожидающих освобождения необходимых им ресурсов. Рассматривается методология проектирования программ просмотра и анализа журналов событий (трасс), сформированных приложениями реального времени. Рассмотрены особенности отображения (в том числе, в виде деревьев) журналов событий и временных диаграмм состояний объектов анализируемой системы, представленных наборами данных, содержащими большое количеством записей. Предложено формальное описание моделей данных трассировки, механизмов их визуализации и механизмов управления запросами к записям трассы и состояниям объектов. Эффективность указанных моделей и механизмов подтверждена опытом эксплуатации программы просмотра и анализа протоколов событий ОС РВ семейства «Багет», реализованной с помощью библиотеки графических элементов GTK+.

Ключевые слова: ОСРВ; операционная система реального времени; средства трассировки; журнал событий; профилирование; Model/View/Controller; индексация записей.

Для цитирования: Годунов А.Н., Чемерев Ф.Н. Средства трассировки ОС РВ семейства «Багет». Труды ИСП РАН, том 31, вып. 4, 2019 г., стр. 7-28. DOI: 10.15514/ISPRAS-2019-31(4)-1

Благодарности: Публикация выполнена в рамках государственного задания по проведению фундаментальных научных исследований по теме «Исследование и реализация программной платформы для перспективных многоядерных процессоров» (№ 0065-2019-0002).

Tracing Tools for «Baget» Family RTOS

A.N. Godunov, ORCID: 0000-0001-5952-9185 <nkag@niisi.ras.ru>
F.N. Chemerev, ORCID: 0000-0002-6494-716X <nkfedor@niisi.ras.ru>
Scientific Research Institute for System Analysis of the Russian Academy of Sciences,
GSP-1, 36/1, Nakhimovsky pr., Moscow, 117218, Russia

Abstract. The paper deals with the problems of developing tracing software for hard real-time systems. Currently, almost every real-time operating system (RV OS) has event tracking software. The goal of this software is to search for «ordinary» software errors (which traditional debuggers cannot handle) and real-time errors. In this case, it is necessary to analyze not only the sequence of events, but also the «memory leak», the dynamics of the processor states and control flows (profiling), the states of semaphores, mutexes, and other synchronization tools, as well as the queue of control flows waiting to release the resources they need. The methodology for designing programs for viewing and analyzing event logs (traces) generated by RTOS-based

software systems is regarded. Specifics of visualizing RTOS events and time diagrams of states of objects in the analyzed systems, represented by data sets containing a large number of records are discussed. A formal specification is proposed to the tracing data models, the methods for their visualization and for filter management of trace records and object states. The effectiveness of these models and methods is confirmed by the operating experience of the Tool for Viewing and Analyzing the Event Logs for RTOS for «Baget» family developed with the toolkit GTK+ for creating graphical user interfaces.

Keywords: RTOS; events; logging; run-time behavior; tracing tool; Model/View/Controller; MVC

For citation: Godunov A.N., Chemerev F.N. Tracing Tools for «Baget» Family RTOS. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 4, 2019, pp. 7-28 (in Russian). DOI: 10.15514/ISPRAS-2019-31(4)-1

Acknowledgements. This work was supported by the State Research Program of the Russian Federation (Project No. 0065-2019-0002).

1. Введение

В настоящее время практически в каждой операционной системе реального времени (ОС РВ) имеются программные средства трассировки событий. Под трассировкой принято понимать порождение, накопление и анализ данных о событиях, имевших место при выполнении пользовательского приложения [1]. В 2001 году в стандарт POSIX 1003.1 внесены дополнения, специфицирующие интерфейс прикладной программы со средствами трассировки [1].

Задача трассировки — поиск «обычных» программных ошибок (с которыми не справляются традиционные отладчики) и ошибок реального времени. При этом приходится анализировать не только последовательности событий, но и «утечки» памяти, динамику изменения состояний процессора и объектов ОС – потоков управления, семафоров, мьютексов и других средств синхронизации, а также очереди потоков, ожидающих освобождения необходимых им ресурсов.

Средства трассировки должны предусматривать возможность представления информации о зарегистрированных событиях в виде, приемлемом для непосредственного восприятия человеком – в виде текста, графиков, диаграмм и т.п. В зависимости от задач, возложенных на средства трассировки, результаты визуализации событий могут выводиться в обычный текстовый файл, но могут быть представлены содержимым окон в приложениях, реализующих графический пользовательский интерфейс (GUI).

1.1 Средства трассировки в существующих ОС РВ

В настоящее время большинство операционных систем оснащено собственными инструментами трассировки. Имеется также положительный опыт создания единого инструментария для визуализации трасс, сформированных под управлением различных ОС (*Tracealyzer* фирмы *Percepio* [3]).

События, связанные с вызовами ядра, работой планировщика, обработкой прерываний, созданием, уничтожением и изменением состояний потоков и процессов, а также события прикладной программы может генерировать большинство используемых в настоящее время ОС РВ, например, *QNX Neutrino* [4], *INTEGRITY-178* [5], *ThreadX* [6] , *RTOS-32* [7], *VxWorks* [8] и др. ОС РВ *ThreadX* допускает генерацию и последующую обработку событий промежуточного программного обеспечения из стеков *NetX*, *FileX* и *UsbX*.

В ходе регистрации содержимое буферов, в которых накапливаются результаты генерации событий, направляется на устройство вывода (например, в файл инструментальной ЭВМ). При этом на данные трассировки могут быть наложены фильтры – как при генерации, так и в процессе регистрации. Управление регистрацией данных трассировки может обеспечиваться как непосредственно средствами соответствующей ОС РВ (такими, например, как *tracelogger* в *QNX*, средства ведения журнала аудита в ОС РВ *INTEGRITY-178*, средства трассировки, включенные в разделы (*partition*), в ОС РВ *PikeOS* [9]), так и с

помощью функций, доступных прикладной программе (таких, например, как *TraceEvent*() в *QNX*). В первом случае разработчики ОС РВ предоставляют средства конфигурирования (обычно на основе *GUI*). В *QNX*, например, при конфигурировании инструментированного ядра обычно задаются режимы регистрации, размер буфера, а также фильтры, указывающие, какие события и классы событий должны регистрироваться. При конфигурировании ОС РВ *PikeOS* записи трассы могут быть, кроме того, расширены и использованы в качестве триггеров.

Наличие в различных ОС РВ функций управления регистрацией данных трассировки является предпосылкой создания единого инструментария для визуализации трасс, сформированных под управлением этих ОС. Таким инструментом является *Tracealyzer* фирмы *Percepio* [3]. Он либо использует данные, полученные регистратором событий трассировки, входящим в соответствующую библиотеку регистратора трасс ОС РВ (*VxWorks, µC/OS-III, FreeRTOS, ThreadX*), либо экспортирует данные трассировки, как в случае *On Time RTOS-32*, – с помощью инструмента *Trace Exporter*, который поставляется в исходном коде на *C* и должен быть включен в приложение реального времени.

Собранные данные могут быть обработаны как в режиме реального времени (*online*-анализ), так и в автономном режиме (*offline*-анализ). В качестве инструмента обработки чаще всего предлагаются инструменты, использующие графический пользовательский интерфейс (*GUI*), как автономные, например, система мониторинга *CODEO* данных трассировки ОС РВ *PikeOS* [9] или *Tracealyzer* фирмы *Percepio* [3], так и включенные в интегрированную среду разработки (*IDE*) приложений реального времени, как, например, в *QNX*. Графический инструмент для анализа, проверки и профилирования приложений реального времени *OpenTracer* фирмы *Altreonic* [10] интегрирован со средой разработки *OpenComRTOS Designer*. Фирмой *Percepio* поддерживаются несколько ведущих *IDE*, в которые интегрирован *Tracealyzer*. Среди них – *IAR Embedded Workbench, Keil µVision (MDK), Atmel Studio, Microchip MPLAB X IDE, Wind River Workbench* и большинство *IDE* на основе *Eclipse* (например, *Atollic TrueStudio, SW4STM32, Code Composer Studio (TI CCS), NXP LPCxpresso/MCUxpresso* и т.д. [3]).

Наряду с интерактивными средствами, использующими *GUI*, некоторые разработчики ОС РВ предлагают инструменты на основе программного интерфейса приложения (*API*). В *QNX* [4], например, имеется набор функций, в том числе, *traceparcer*() и *traceprinter*(), с помощью которых пользователь может реализовать собственные процедуры обработки данных трассировки.

1.2 Обзор средств трассировки в ОС РВ семейства «Багет»

Положения стандарта POSIX 1003.1 учтены при разработке операционных систем семейства «Багет», ориентированных на использование в целевых (встраиваемых) ЭВМ, работающих в системах жёсткого реального времени. Предназначенные для функционирования на целевых ЭВМ прикладные программы (приложения реального времени) разрабатываются на инструментальной ЭВМ. Трассы, сформированные средствами трассировки ОС РВ семейства «Багет», просматриваются и анализируются на инструментальной ЭВМ после завершения работы приложения на целевой ЭВМ (*offline*-анализ).

Интерактивным средством просмотра и анализа протоколов событий, сформированных системой протоколирования ОС РВ семейства «Багет», является Трассировщик – программа, реализующая графический пользовательский интерфейс. Существенной особенностью программы является то, что основным средством отображения данных трассировки, в том числе объединенных в иерархическую структуру, являются таблицы, которые могут содержать большое количество строк.

Важным показателем эффективности Трассировщика, является время реакции на действия пользователя при работе с программой:

- время открытия (первичной обработки) трассы – от момента выбора файла протокола событий до появления на экране содержимого трассы, которую пользователь может просматривать и анализировать;
- время вторичной обработки трассы (при этом формируются трассы состояний – агрегаты данных, отражающие динамику изменения состояния процессора и объектов ОС);
- время, необходимое для перерисовки содержимого отображаемых больших таблиц при их прокрутке (скроллинге);
- время, затрачиваемое на формулирование условий отбора записей трассы, состояний процессора и объектов трассировки;
- время выполнения процедур отбора записей протокола событий и трасс состояний.

Снижение затрат времени предполагает отказ от тотальной обработки трассы. Высокую скорость отбора записей трассы и доступа к ним обеспечивает специальная модель трассы. При открытии трассы обрабатываются лишь те данные, которые необходимы для построения этой модели – при этом идентифицируются объекты ОС, а записи трассы индексируются. Остальные данные Трассировщик обрабатывает по мере необходимости – например, когда требуется отобразить их на экране. Минимизация затрат времени на формулирование условий отбора обеспечивается предельно лаконичным механизмом задания этих условий в наиболее важных типичных случаях.

В данной статье рассматриваются подходы и технические решения, лежащие в основе моделей, используемых средствами трассировки ОС РВ семейства «Багет».

2. Основные понятия

2.1 События трассировки и режимы протоколирования

Сведения о действиях, производимых при выполнении приложения, фиксируются в виде агрегатов данных, называемых событиями трассировки. События записываются в потоки трассировки, которые содержат также служебные данные, необходимые для интерпретации событий. Для получения копии потока для последующего анализа, содержимое буфера сбрасывается в журнал трассировки (протокол событий, трасса) – файл, располагающийся в долговременной памяти.

Одним из главных требований к системе трассировки является минимизация накладных расходов. Чтобы удовлетворить этому требованию, потоки трассировки размещают в оперативной памяти – при этом, как правило, используется кольцевой буфер. Перенос данных из буфера в файл может производиться автоматически по мере заполнения буфера, когда процессор не занят выполнением «штатных» функций приложения реального времени, или по явному требованию.

Действия при переполнении буфера определяются настройками системы протоколирования. В частности, возможна остановка при переполнении буфера. Имеется также возможность циклической записи, когда при достижении конца буфера запись продолжается в начало буфера – новые записи замещают старые. В последнем случае, копия содержимого буфера (самые «свежие» записи) создается при закрытии потока трассировки.

В процессе выполнения прикладной программы могут иметь место фатальные ошибки, в результате которых система окажется неработоспособной. В подобных случаях полезным источником информации является содержимое буфера на момент краха системы. В ОС РВ семейства «Багет» имеется возможность создать копию его содержимого при последующем запуске приложения.

2.2 Системные и пользовательские события

В соответствии со стандартом POSIX 1003.1 события трассировки подразделяются на две категории:

- системные – генерируются в ответ на действия ОС, в том числе, при обращении прикладной программы к функциям ОС;
- пользовательские (они же события прикладной программы) – генерируются при вызове функции $posix\_trace\_event()$.

Пользовательские события могут использоваться как маркеры наиболее важных участков прикладного кода – их можно ассоциировать с точками входа в функции и отдельными ветвями реализующих эти функции алгоритмов. События прикладной программы могут генерироваться средствами ее самоконтроля [11]. Наконец, с пользовательскими событиями в трассе можно связать реальные события, произошедшие на объекте управления: срабатывания реле, закрытие и открытие клапанов, факт выхода значений критически важных параметров (температура, давление) за пределы допуска.

2.3 Служебные записи трассы

Наряду с событиями трасса может содержать служебные записи. Если трасса имеет блочную структуру (как в ОС РВ семейства «Багет»), то служебной может быть, например, запись, содержащая номер блока, или «пустая» запись, заполняющая его «хвост». Некоторые служебные записи (назовем их существенными служебными) содержат важную информацию, необходимую для интерпретации событий трассировки.

Заголовок трассы, например, содержит сведения о версии ОС, под управлением которой трасса была сгенерирована. Другие служебные записи содержат характеристики объектов ОС (семафоров, мьютексов, потоков управления, очередей сообщений и т.п.).

Источником этих данных являются события, регистрируемые в момент создания этих объектов. Система протоколирования ОС РВ семейства «Багет» дублирует эти характеристики, помещая их в служебные блоки системы протоколирования. В отличие от содержимого циклического буфера, служебные блоки не затираются. В момент завершения протоколирования они вместе с записью представления (заголовком трассы) отписываются в файл трассы.

2.4 Типы записей трассы

Каждая запись трассы относится к некоторому типу t, принадлежащему множеству T_r типов записей ($t \in T_r$). Их номенклатура, т.е. множество T_r определяется версией ОС РВ. Типы событий и типы существенных служебных записей составляют множество T_e типов существенных записей ($T_e \subset T_r$). Номенклатура типов событий и служебных записей определяется системой протоколирования конкретной версии ОС.

Каждый тип записи характеризуется уникальными именем и идентификатором. Имена используются при визуализации записей, тогда как в потоке трассировки хранятся числовые идентификаторы их типов. Соответствие имен и идентификаторов типов обеспечиваются средствами просмотра и анализа трассы

2.5 Объекты трассировки

В записях трассы, соответствующих системным событиям, могут присутствовать поля, значения которых идентифицируют объекты ОС (потоки управления, семафоры, мьютексы и т.п.). Поля записей, соответствующих пользовательским событиям, могут содержать тексты или целые числа, используемые в качестве идентификаторов пользовательских объектов.

В модели данных Трассировщика номенклатура типов объектов трассировки наряду с типами объектов ОС включает в себя прерывания и сигналы, которые в совокупности составляют множество типов объектов трассировки $T_{(obj)}$. В целом эта номенклатура отражает наиболее общие представления об операционных системах семейства «Багет». Целочисленные идентификаторы i типов объектов трассировки могут быть представлены набором констант, задаваемых перечисляемым типом (в языке С – $enum$).

Два типа объектов «прерывание» и «поток управления» составляют множество T_c типов контекстов ($T_c \subset T_{(obj)}$).

Множество объектов трассировки типа $t \in T_{(obj)}$ обозначим через O_t, $|O_t|$ – зарегистрированное в трассе количество объектов типа t. Множество всех зарегистрированных объектов трассировки обозначим через $O = \bigcup_{t \in T_{(obj)}} O_t$.

2.6 Состояния процессора и объектов ОС

Наряду с событиями, представленными в трассе, приходится анализировать состояния процессора (*CPU*) и объектов ОС.

При анализе загрузки процессора и поведения объектов ОС РВ удобно оперировать состояниями с разной степенью детальности. Состояния, для которых не предусмотрена детализация, будем называть атомарными или просто состояниями. Состояния, допускающие детализацию будем называть агрегированными (укрупненными). Примером агрегированного состояния процессора является состояние «*выполняются потоки (не важно какие)*», детализируемое состоянием «*выполняется конкретный поток*»

Процессор может находиться либо в состоянии «*простой*», либо в одном из состояний, когда выполняются «*обработчики прерываний (не важно какие)*», «*потоки (не важно какие)*», «*конкретный обработчик прерывания*», «*конкретный процесс*», «*конкретный поток*». Эти состояния (как атомарные, так и агрегированные) могут быть представлены множеством:

$$S_{(CPU)} = \{cpuIdle, cpuInterrupts, cpuThreads, cpuIntr, cpuProc, cpuThr\}$$

Множество состояний объектов типа t обозначим через S_t. Например, поток управления может пребывать в следующих состояниях: «*не исполняется*», «*исполняется*», «*не готов*», «*готов*», «*прерван*». Соответственно множество состояний потока управления (атомарных и агрегированных) может быть представлено следующим образом:

$$S_{(Thread)} = \{thrNotRun, thrRun, thrNotReady, thrReady, thrInterrupted\}.$$

2.7 Отображение данных, имеющих иерархическую структуру

В основе наиболее популярных современных средств отображения данных, имеющих иерархическую структуру, лежит концепция *MVC* (*Model/View/Controller*) [12]. Реализации этой концепции в разных фреймворках – кроссплатформенных библиотеках графических элементов (таких, например, как *GTK+* [13] и *Qt* [14]), предназначенных для создании пользовательского интерфейса (*GUI*), – могут иметь существенные различия. Однако, обсуждаемые в рамках данной статьи подходы и технические решения не касаются аспектов концепции, связанных с редактированием данных. Важным является лишь то обстоятельство, что в рамках концепции *MVC* модель данных (*Model*) отделена от представления (*View*), а собственно данные (более точно, источник данных – *Data*) отделены от модели (*Model*), которая по отношению к элементам данных выступает в роли навигатора.

В дальнейшем будет показано, что временны́е характеристики используемых Трассировщиком средств отображения данных могут быть существенно улучшены именно благодаря тому, что в основе их реализации лежит концепция *MVC*. Ниже приводится формальное описание наиболее важных с этой точки зрения ее механизмов. (Приведенные в

данном разделе функции являются условными обозначениями методов классов, реализующих модели *Model*, представления *View,* или комбинаций этих методов.)

Интересующее нас представление (*View*) отображает данные в виде таблицы. Каждой ее строке соответствуют позиция в дереве, представляющая собой последовательность $P = (p_i)_{i=1}^{|P|}$, ($p_i \in \mathbb{N}$). Элементы p_i этой последовательности задают логические координаты узла дерева – путь от p_1-го элемента верхнего уровня дерева к p_n-му элементу ($n = |P|$), соответствующему уровню $|P|$ дерева. В случае одноуровневого дерева (списка) позиция $P = \{p_1\}$ любой строки определяется ее номером, равным p_1. Логические координаты используются представлением *View*, с одной стороны, для адресации к отображаемым данным, с другой – для позиционирования в пространственных координатах виджета, реализующего представление *View*. В *GTK+*, например, эти координаты представлены структурами типа *GtkTreePath*.

Модель *Model* обеспечивает связь представления *View* с источником данных *Data*. Взаимодействие модели с этими данными обеспечиваются классом, реализующим методы интерфейса, определяемого конкретным фреймворком. В случае *GTK+*, например, этой цели служит абстрактный интерфейс *GtkTreeModel* [13], в случае *Qt* – интерфейс базового класса *QAbstractItemModel* [14]. Эти методы, с одной стороны, должны обеспечивать навигацию внутри источника данных (привязку узла модели *Model* к элементу данных источника *Data*), а с другой – соотнесение узла модели с позицией P в дереве представления *View*.

Узел модели в *GtkTreeModel* описывается структурой типа *GtkTreeIter*, в *QAbstractItemModel* – объектом класса *QModelIndex*. Введем условное обозначение $node$ для описателя узла любой модели *Model* (в том числе, для типа *GtkTreeIter* или *QModelIndex)*. Несмотря на то, что номенклатура методов модели и их сигнатура могут быть разными у различных фреймворков, эти методы обеспечивают:

- определение типа (с помощью функции $type(node)$) и значения ($value(node)$) данного из хранилища *Data*;
- установление соответствия – с помощью функция $node(P)$ – между узлом $node$ модели *Model* и позицией P в дереве представления *View* (тем самым реализуется доступ методов представления к данным из хранилища *Data*), а также обратного соответствия – с помощью функция $P(node)$;
- навигацию по узлам модели *Model* (с помощью функций $next(node)$, $prev(node)$, $parent(node)$, $nth\_child(node,\ n)$);
- доступ к характеристикам узла, таким, например, как количество дочерних узлов.

Столбцы таблицы, реализуемой представлением *View*, могут быть представлены в виде последовательности c_i описателей столбцов $C = (c_i)_{i=1}^{|C|}$, где $|C|$ – их количество. С каждым столбцом c_i связан объект $render_i$, отображающий содержимое ячейки из этого столбца с помощью низкоуровневых графических средств, и функция $paint(render_i, c_i, node)$, переопределяющая значения внутренних параметров объекта $render_i$, используемых им при «отрисовке» ячейки (таких как текст, цвет шрифта и т.п.) – исходя из интерпретации. Реализация функции $paint()$ возлагается на разработчика *GUI*-приложения. Ему же предоставляется выбор класса, которому принадлежит объект $render_i$, из некоторого фиксированного набора. Связь между логическими координатами P строки, и узлом $node$ модели *Model*, необходимые для позиционирования области рисования, предоставляются соответствующими методам *Model*.

3. Модель трассы событий

Современные ЭВМ, используемые в качестве инструментальных при просмотре и анализе протоколов событий, обладают достаточным объемом оперативной памяти и механизмом

виртуальной памяти, что позволяет помещать протоколы событий в виртуальную память инструментальной ЭВМ целиком. В дальнейшем, если специально не оговаривается обратное, считается, что все описываемые крупные агрегаты данных (трассы, индексные массивы) размещены в виртуальной памяти. В частности, адреса записей трассы и их полей – это адреса в виртуальной памяти инструментальной ЭВМ.

Трасса представляет собой агрегат данных, моделью которого является последовательность записей $L = (r_i)_{i=1}^{|L|}$, где $|L|$ – количество записей в трассе. В свою очередь, запись трассы r_i – это агрегат данных, которому можно поставить в соответствие его адрес $a_i = Adr(i)$ в оперативной памяти, размер $l_i = Size(i)$ и идентификатор типа записи $t = Eid(i)$, где $t \in T_r$.

3.1 Индексация записей трассы

Эффективная реализация функций $Adr()$, $Size()$ и $Eid()$ предполагает наличие того или иного механизма индексации записей трассы. Решение этой задачи существенно упрощаются, если в начале каждой записи r_i присутствует заголовок (дескриптор), содержащий длину записи и идентификатор ее типа, а сами записи удовлетворяют условию связности трассы:

$$a_{i+1} = a_i + l_i \# (1)$$

где a_i – адрес i-й записи, l_i – ее размер.

В ОС РВ семейства «Багет» дескриптор записи представлен двумя полями *type_record* и *size_record*, что позволяет – в процессе создания новых версий ОС – в широких пределах варьировать как номенклатуру типов записи, так и их размер.

Первичным индексом записи r_i трассы будем называть ее порядковый номер i. Для быстрого поиска записи по ее первичному индексу используется индексный массив I_L. С учетом условия связности (1) индексация (назовем ее первичной) сводится к построению индексного массива беззнаковых целых чисел, реализующего последовательность $I_L = (\Delta_i)_{i=1}^{|L|}$, где $\Delta_i = a_i - a_1$ – смещение записи r_i относительно начала трассы L.

Концепция Трассировщика предполагает, что размер трассы не превышает 4 ГБ. Поэтому использование 32-разрядных смещений Δ_i для индексации записей позволяет в условиях 64-разрядной инструментальной ОС в два раза уменьшить размер индексного массива по сравнению с непосредственным использованием адресов.

Рассмотрим последовательность E, полученную из последовательности L путем исключения из нее несущественных служебных записей. Каждому номеру j элемента последовательности E соответствует первичный индекс i, такой что $\overline{r}_j = r_i, r_i \in L$.

3.2 Описание структуры записи трассы на языке *C*

В ОС РВ семейства «Багет» записи трассы формируются с помощью функций, реализованных на языке *C*. Поэтому структурные типы этого языка программирования являются наиболее органичным средством описания записей, поля которых могут быть как простыми (описываемыми типами *short, unsigned int* и т.п.), так и составными: структурами (*struct*) и объединениями (*union*). В записях трассы ОС РВ семейства «Багет» агрегату данных типа *union* предшествует простое поле, значение которого определяет поле объединения *union*, в соответствии с которым следует интерпретировать содержимое этого агрегата данных.

Любое поле f (простое или составное) любой существенной записи имеет тип, заданный в соответствии со стандартом языка *C*. Все типы полей, встречающиеся в описаниях существенных записей всех типов образуют множество T_d, определяемое версией ОС РВ.

В концепции Трассировщика выделяется множество базовых типов T_b – единое для всех трасс, вне зависимости от того, какой именно версией ОС РВ семейства «Багет» трасса была

получена. Это множество включает в себя стандартные типы данных языка *C*, а также некоторые структурные типы данных (например, *struct timespec* стандарта POSIX 1003.1). В последних версиях ОС РВ семейства «Багет» полям записи, содержащим идентификаторы объектов ОС, назначаются типы, имена которых однозначно определяют тип объекта ОС (например, «*thread_ptr*», «*chan_id*»). Эти типы (и им подобные) также являются элементами множества базовых типов T_b. Множество базовых типов, применимых к отдельно взятой версии ОС РВ, является подмножеством множества T_b.

Для каждого базового типа $b \in T_b$ создается набор функций, используемых при интерпретации и визуализации значений полей записей трассы, принадлежащих этому типу. Указатели на эти функции – вместе с данными о размере поля типа b и способе его выравнивания в памяти ЭВМ находятся в описателе базового типа b. В Трассировщике поддерживается механизм, устанавливающий соответствие между именами базовых типов и их описателями.

Структурный тип данных – элемент множества T_d – характеризуется именем типа. В языке C с этим именем связана либо структура (*struct*), либо объединение (*union*).

Описания записей и описания их полей содержатся в заголовочных файлах (*h*-файлах) операционной системы. Каждое *C*-описание структуры записи типа e содержит исчерпывающую информацию, необходимую для получения значения любого поля f простого типа любой записи трассы.

Если программа просмотра и анализа трасс в своей существенной части написана на языке *C*, в принципе возможно непосредственное использование заголовочных файлов в ходе анализа. В этом случае необходимые *h*-файлы ОС РВ могут быть использованы в процессе компиляции программы просмотра как ее собственные заголовочные файлы. Но, во-первых, такая программа просмотра не будет адекватно работать с трассами, созданными другими версиями ОС РВ, *h*-файлы которых отличаются от тех, что были использованы в проекте программы просмотра. А во-вторых, даже в рамках одной и той же версии ОС РВ в заголовочных файлах присутствуют зависимости от архитектуры процессора, используемого на целевой машине. Это означает, что для каждой версии ОС и для каждой архитектуры придется собирать отдельную программу просмотра (или, по крайней мере, существенную ее часть, ответственную за интерпретацию записей трассы).

В Трассировщике ОС РВ семейства «Багет» реализован иной подход, в рамках которого описание записей и их полей представлено в виде файла на языке *XML*, формируемого на основании *C*-описаний. При этом *XML*-описание трассы дополняется сведениями, отсутствующими в *h*-файлах ОС РВ, например, спецификациями форматов, используемых при отображении записей трассы.

3.3 Описание структуры записи трассы на языке *XML*

В *XML*-описании трассы множество T_r типов записей представлено последовательностью *XML*-элементов $\langle event \rangle$. Атрибут $Name$ элемента $\langle event \rangle$ определяет имя записи, атрибут id – значение ее идентификатора. Каждый элемент $\langle event \rangle$ состоит из элементов $\langle field \rangle$ с атрибутами $Name$, dim и $type\_name$. Порядок элементов $\langle field \rangle$ совпадает с порядком элементов *C*-описания структуры. Имя элемента в *C*-структуре соответствует значению атрибута $Name$ элемента $\langle field \rangle$, размерность – значению атрибута dim, тип данных – значению атрибута $type\_name$.

Множество структурных (составных) типов данных ($T_d \setminus T_b$) представлено в описании трассы *XML*-элементом $\langle Common \rangle$, содержанием которого является *XSD*-последовательность (*xs:sequence*) *XML*-элементов $\langle dt \rangle$, различающихся значениями атрибута $Name$.

Совпадение значения атрибута $type\_name$ с именем базового типа $b \in T_b$ свидетельствует о том, что поле, описываемое элементом $\langle field \rangle$ относится к базовому типу b. В противном

случае этому полю соответствует *XML*-элемент $\langle dt \rangle$ с атрибутом $Name$, значение которого равно значению атрибута $type\_name$ элемента $\langle field \rangle$. Структура элемента $\langle dt \rangle$, также как и структура элемента $\langle event \rangle$, состоит из *XML*-элементов $\langle field \rangle$ с атрибутами $Name$, dim и $type\_name$.

XML-описания типов записей трассы и составных типов данных является частью общего описания трассы, которое для Трассировщика является основным источником исходных данных, необходимых для интерпретации трассы. Каждой версии ОС РВ соответствует определенный *XML*-файл.

В настоящее время используется простой и надежный способ генерации *XML*-описаний трасс, не предполагающий лексического разбора заголовочных файлов ОС РВ. Программа, использующая кросс-отладчик *rdp-gdb*, автоматически генерирует *XML*-описание трассы – свое для каждой из поддерживаемых ОС РВ аппаратных платформ целевой ЭВМ. При запуске отладчика, ему в качестве аргумента передается целевой образ – файл, сконфигурированный под определенную процессорную архитектуру целевой ЭВМ.

Основное достоинство этой программы, наряду с ее очевидной простотой и надежностью, – это точность, с которой сгенерированное *XML*-описание отражает особенности процессорной архитектуры.

3.4 Дерево описания записи трассы

Непосредственное использование *XML*-описания трассы сопряжено со значительными затратами времени как в процессе первичной обработки трассы, так и при визуализации ее записей.

В Трассировщике реализована модель записи трассы, источником данных для которой является *XML*-описание трассы. Эта модель применима к трассам, сформированным различными версиями ОС РВ. Она, во-первых, обеспечивает приемлемую скорость обработки трассы, а, во-вторых, непосредственно используется инструментарием *GTK+* при визуализации ее записей. Эта модель формируется в самом начале обработки трассы – после того, как из заголовка трассы (записи представления) извлечена информация о версии ОС РВ и архитектуре процессора.

Любая существенная запись трассы имеет, вообще говоря, иерархическую структуру, описание которой может быть представлено в виде последовательности описателей непосредственно содержащихся в ней полей:

$$R_e = (d_i)_{i=1}^{|R_e|},$$

где e – тип записи, $|R_e|$ – количество полей записи (элементов последовательности R_e), d_i – описатель -го поля.

Среди описателей полей последовательности R_e может присутствовать один или более описатель d составного (структурного) поля, который, аналогично описанию структуры самой записи R_e, может быть представлен последовательностью описателей полей:

$$d = (\tilde{d}_i)_{i=1}^{|d|},$$

где $|d|$ – количество элементов в последовательности d.

В Трассировщике описатель d является структурой, содержащей, в частности, имя поля и указатели на «соседние» описатели: «*родительский*», «*следующий*», «*предыдущий*», «*первый подчиненный*». Система этих указателей реализует иерархию дерева описания записи типа e, узлами которого являются описатели полей. Описатели d_i – узлы верхнего уровня, объединенные посредством указателей «*следующий*» в список, началом которого является элемент d_1, – определяют последовательность R_e. Таким образом, R_e описывает древовидную структуру записи типа $e \in T_e$.

Дерево описания записи R_e формируется в процессе последовательного обхода *XML*-описания записи e. При обнаружении в *XML*-описании очередного элемента $\langle field \rangle$

создается соответствующий ему описатель d. Этот описатель встраивается в существующую часть дерева R_e (обновляются указатели на «соседей» у самого описателя d и у смежных узлов).

В ходе формирования дерева R_e используются элементы структуры описателя d: dt («*указатель на тип поля*») и $offset$ («*смещение относительно начала записи*»). Если значение атрибута $type\_name$ элемента $\langle field \rangle$ совпадает с именем базового типа $b \in T_b$, элементу dt структуры присваивается значения указателя на описатель базового типа b. Если значение атрибута $type\_name$ элемента $\langle field \rangle$ совпадает с именем атрибута $Name$ элемента $\langle dt \rangle$, описывающего составное поле, узел d «достраивается» в соответствии с «содержимым» *XML*-элемента $\langle dt \rangle$. Эта процедура рекурсивна, поскольку «содержимое» $\langle dt \rangle$ может включать поля структурных типов. При этом всякий раз для описателя d рассчитывается значение смещения ($offset$) описываемого им поля относительно начала записи – с учетом размеров и способов выравнивания полей базовых типов.

Размер простого поля f в записи $\overline{r}_j \in E$ трассы и способ его выравнивания в оперативной памяти инструментальной ЭВМ определяется его типом b ($b \in T_b$), тогда как адрес – индексом i записи r_i и позицией P (полным путем от корня) в структуре R_e, описывающей запись r_i типа e трассы L. Каждому базовому типу b соответствует функция, возвращающая значение поля f по его адресу: значение v_f задаваемое путем P простого поля f записи r_i, может быть получено с помощью функции $v_f = Value(i, P)$. Таким «значением» может быть, например, строка, число, значение указателя на структуру в адресном пространстве целевой ЭВМ, содержащую данные об объекте ОС. С типом b можно связать также функцию отображения $s = Show(v_f, F)$, преобразующую значение v_f в строку s в соответствии с форматом F.

3.5 Идентификаторы объектов трассировки в записи трассы

Идентификатор oid объекта трассировки, обнаруженный в поле f_P, записи $\overline{r}_i$ типа e, определяется значением поля f_P, позиция которого в записи задана путем P: $oid = Value(i, P)$. Однако, для идентификации объекта этого значения недостаточно: необходимо знание идентификатора процесса, в котором, участвует объект. Как правило, идентификатор pid процесса является значением одного из полей записи $\overline{r}_i$. В общем случае для идентификации объекта o в записи, описываемой деревом R_e, используется пара описателей полей записи $\overline{r}_i$: d_o (описатель поля идентификатора объекта) и d_p (описатель поля идентификатора процесса). В случае «однопроцессных» ОС РВ, а также в случае объектов трассировки, не привязанных к конкретному процессу, используется нулевое значение идентификатора процесса.

Каждому объекту o соответствует свое описание – структура, содержащая, в частности, имя объекта, его тип, и ссылку на служебную запись трассы, содержащую его характеристики. В целом, эту структуру можно рассматривать как программную реализацию объекта o. Тип объекта $t \in T_o$ может быть получен с помощью функции $t = Type(o)$.

В Трассировщике для идентификации объектов трассировки используется хеш-таблица H, из которой объект o типа t может быть извлечен с помощью функции $o = GetObject(H, t, pid, oid)$. В случае отсутствия в таблице объекта с такими характеристиками, описание объекта создается, указатель на него заносится в таблицу H.

Описание R_e записи типа e, может содержать описатели (образующие множество O_e) нескольких полей, значения которых интерпретируются как идентификаторы объектов. Описатель одного из полей может быть объявлен описанием поля идентификатора главного объекта. Выбор этот, в принципе, произволен и определяется лишь тем, насколько актуален отбор записей трассы, в которых фигурирует объект, выбранный в качестве главного. Для события «захват семафора», например, главным объектом естественно считать семафор. Программной реализацией множества O_e в Трассировщике является список *GList* (из

библиотеки *GLib*). Описатель поля, содержащего идентификатор объекта, объявленного главным, помещается в начало списка.

3.6 Описание записи трассы как модель данных для отображения в таблице

Используемое в ходе анализа дерево описания записи R_e ориентировано на обработку записей как в рутинном (автоматическом) режиме, так и в интерактивном, когда записи просматриваются в скроллируемых таблицах.

В *GTK+* данные, имеющие иерархическую структуру, отображаются с помощью виджета *GtkTreeView*, реализующего представление *View* концепции *MVC* (см. 2.7) Для взаимодействия с этими данными всегда используется класс, реализующий методы абстрактного интерфейса *GtkTreeModel*.

При заполнении ячеек таблицы, отображаемой виджетом *GtkTreeView*, используются функции обратного вызова с сигнатурой *GtkTreeCellDataFunc* (см. 2.7, функция $paint()$). В зависимости от того, какие именно данные модели требуется отобразить, эти функции переопределяют значения внутренних параметров рендерера – объекта класса *GtkCellRenderer*, аналога объекта $render_i$ (см. 2.7), связанного с i-м столбцом таблицы.

В Трассировщике для отображения детальной информации, содержащейся в записи трассы, реализован класс *DiTree*, дочерний по отношению к классу *GObject* и реализующий методы интерфейса *GtkTreeModel*. Виджет *GtkTreeView* отображает существенную запись $\overline{r}_j$ типа e, воспроизводя структуру дерева описания записи R_e, узлы которого снабжены именами, совпадающими с именами полей в *C*-описании (*h*-файлах ОС РВ). Тип записи определяется первичным индексом i $(e = Eid(i))$, соответствующим номеру j элемента последовательности E существенных записей. В каждом случае, когда требуется отобразить содержимое записи типа e, виджету *GtkTreeView* в качестве модели данных D_e предъявляется объект класса *DiTree*. Единственным атрибутом (свойством) этого объекта является R_e – дерево описания записи типа e.

Привязка модели данных D_e записи типа e к виджету *GtkTreeView* происходит в момент выбора существенной записи $\overline{r}_j$. Функции, реализующие интерфейс *GtkTreeModel*, используют указатели на «смежные» узлы дерева, содержащиеся в структуре описателей полей. Значением в узле дерева R_e, полученным с помощью функции $get\_value()$, является указатель на описатель d этого узла. Значения полей, определяемые адресом записи $a_i = Adr(i)$ и смещениями $offset$ из описателей полей записи, вычисляются в теле функций, задающих параметры рендерера.

Наличие у модели D_e интерфейса *GtkTreeModel*, позволяет инкапсулировать детали ее реализации. Функции интерфейса можно использовать для перемещения по структуре записи – независимо от того, как именно этот интерфейс реализован.

4. Статистика событий и вторичная индексация трассы

Анализ работы приложения реального времени, предполагает наличие хотя бы минимальной статистики, позволяющей оценить, какие именно события происходили, в каких контекстах, какие объекты ОС в них участвовали.

4.1 Тройки $\langle e,\ c,\ o\rangle$

В фазе первичной обработки записей трассы определяется тип e очередной существенной записи $\overline{r}_j$, идентифицируются объекты трассировки и пополняется хеш-таблица H, используемая для поиска объектов по их идентификаторам (см. 3.5). Среди объектов трассировки, обнаруженных в трассе, выделяется контекст. Для любой существенной

записи $\overline{r}_j$ можно указать тройку $\langle e,\ c,\ o\rangle$ (иначе – eco-тройку), где $e \in T_e$, $c \in O$, $Type(c) \in T_c$, главный объект $o \in O$, $Type(o) \in T_{(obj)}$.

В ходе последовательной обработки существенных записей трассы формируется хеш-таблица – для поиска e-троек по значениям типа события и указателей на описатели контекста и главного объекта. При обнаружения в трассе новой (отсутствующей в хеш-таблице) тройки, создается ее описатель – структура, содержащая тип события и ссылки (указатели) на описатели контекста и главного объекта, а также счетчик, в котором ведется учет «появлений» тройки в записях трассы. Каждой новой тройке назначается (в порядке обнаружения) идентификатор – очередной номер $ecoId$ тройки, после чего указатель на ее описатель заносится в хеш-таблицу $H_{(eoc)}$. Таким образом, каждая запись $\overline{r}_j$ оказывается снабжена идентификатором $ecoId$ соответствующей e-тройки.

На заключительном этапе первичной обработки трассы, когда все e-тройки занесены в хеш-таблицу $H_{(eoc)}$, формируется массив $ecoD$ троек, упорядоченный по идентификаторам $ecoId$, после чего доступ к тройке может быть осуществлен как по ее идентификатору $ecoId$, так и по совокупности значений ее компонент e, c и o (с помощью таблицы $H_{(eoc)}$).

4.2 Вторичная индексация трассы

Средства анализа трассы должны обеспечивать эффективный доступ в первую очередь к существенным записям трассы (несущественные записи, как правило, должны быть скрыты от пользователя). Другим требованием, предъявляемым к средствам анализа трассы, является наличие эффективных механизмов задания условий отбора и фильтрации записей, необходимых для решения конкретных задач анализа.

Как показывает опыт реализации средств трассировки семейства «Багет», подавляющее большинство задач анализа может успешно решаться в рамках процедур отбора записей, условия которых накладываются исключительно на компоненты e–троек. Это означает, что для проверки условий отбора нет необходимости применять их к каждой записи $\overline{r}_j \in E$. Для начала (в первой фазе отбора) достаточно проверить на соответствие этим условиям все e–тройки трассы, «помечая» в битовом массиве те из них, которые этим условиям удовлетворяют. (В дальнейшем – при обходе существенных записей трассы – можно воспользоваться тем, что с каждой записью связана eco-тройка, уже прошедшая проверку.)

В основе доступа к существенным записям трассы – наличие соответствия между номером j существенной записи $\overline{r}_j$ (элемента последовательности E) и определяющим адрес записи $\overline{r}_j$ первичным индексом i – номером элемента индексного массива I_L (см. 3.1). С другой стороны, каждому номеру j существенной записи $\overline{r}_j \in E$ соответствует e–тройка с идентификатором $ecoId$. Это позволяет завершить отбор записей $\overline{r}_j$ процедурой их обхода (вторая фаза отбора), помечая те из них, «чьи» eco–тройки были «помечены» в первой фазе отбора.

Первичный индекс i существенной записи $\overline{r}_j$ вместе с идентификатором $ecoId$ соответствующей этой записи e–тройки образует пару $\langle i, ecoId\rangle$. Компонента i этой пары обеспечивает быстрый доступ к адресу существенной записи, компонента $ecoId$ – к описателю eco–тройки, используемому при отборе записей.

Вторичным индексом существенной записи $\overline{r}_j$ трассы будем называть ее порядковый номер j. Для быстрого поиска записи $\overline{r}_j$ по ее индексу и проверки ее соответствия условиям отбора используется индексный массив, реализующий последовательность пар $\langle i, ecoId\rangle$:

$$I_E = (\langle i, ecoId\rangle)_{j=1}^{|E|},$$

такую что $\overline{r}_j = r_i, r_i \in L, e = Eid(i), e \in T_e$, $|E|$ – количество существенных записей.

4.3 Дерево событий – результат переупорядочивания массивов троек $\langle e,\ c,\ o\rangle$

Располагая информацией о количестве «появлений» в трассе каждой из троек $\langle e,\ c,\ o\rangle$, можно, например, получить статистику реализации события типа e в различных сочетаниях контекстов и главных объектов. Для этого достаточно упорядочить соответствующим образом тройки и просуммировать счетчики тех из них, которые этому сочетанию удовлетворяют. То же самое можно сказать о любой другой компоненте тройки. В конечном счете, достаточно создать три массива Eco, Coe, Oec, являющиеся переупорядоченными копиями массива $ecoD$.

В случае массива Eco, имеется в виду лексикографическая упорядоченность $\langle e_1,\ c_1,\ o_1\rangle < \langle e_2,\ c_2,\ o_2\rangle$ означающая, что выполнено условие

$$(e_1 < e_2) \vee (e_1 = e_2) \wedge ((c_1 < c_2) \vee (c_1 = c_2) \wedge (o_1 < o_2)) \qquad (2)$$

Над массивом Eco можно надстроить массив Ec упорядоченных пар $\langle e,\ c\rangle$, для которых упорядоченность $\langle e_1,\ c_1\rangle < \langle e_2,\ c_2\rangle$, означающая, что $(e_1 < e_2) \vee (e_1 = e_2) \wedge (c_1 < c_2)$, прямо следует из (2). С любым i-м элементом $\langle e_i,\ c_i\rangle$ этого массива можно связать диапазон номеров $[j_1, j_2]$ элементов массива Eco, которому соответствует последовательность троек $D_i = \langle e_i,\ c_i,\ o_j\rangle_{j=j_1}^{j_2}$.

Из массива Ec можно, в свою очередь, сформировать переупорядоченную копию Ce, в котором порядок пар будет определяться условием $\langle e_1,\ c_1\rangle < \langle e_2,\ c_2\rangle$, означающим, что $(c_1 < c_2) \vee (c_1 = c_2) \wedge (e_1 < e_2)$. При этом, однако, -му элементу $\langle e_i,\ c_i\rangle$ массива Ec и j-му элементу $\langle e_j\ ,\ c_j\rangle$ массива Ce соответствует один и тот же диапазон D_k элементов массива Eco – при условии, что $(e_i = e_j) \wedge (c_i = c_j)$. Над массивом Ec аналогичным образом надстраивается массив E событий e_i, а над массивом Ce – массив C контекстов c_j. В процессе формирования массивов Ec и E для каждого события e_i могут быть подсчитаны количества его появлений в трассе. Аналогичные расчёты могут проведены для каждого из контекстов, в котором фиксируется событие e_i.

В целом же E-упорядоченность троек $\langle e,\ c,\ o\rangle$ порождает два варианта иерархии массивов: $E.Ec.Eco$ (события | контексты | объекты) и $C.Ce.Eco$ (контексты | события | объекты). Аналогичным образом C-упорядоченность порождает иерархии массивов: $C.Co.Coe$ (контексты | объекты | события) и $O.Oc.Coe$ (объекты | контексты | события), а Oec-упорядоченность – $O.Oe.Oec$ (объекты | события | контексты) и $E.Eo.Oec$ (события | объекты | контексты). Каждой из этих шести иерархий соответствует свой набор статистических данных, каждый из которых (по-своему) представляет интерес для анализа. Эти иерархии, образуют дерево статистики событий (дерево событий), обладающее способностью менять порядок иерархии в зависимости от режима упорядочивания троек $\langle e,\ c,\ o\rangle$, назначаемого пользователем.

4.4 Дерево событий – модель для отображения. Интерфейс *GtkTreeModel*

В подразделе 2.7 подчеркивалось, что комфортный просмотр данных, имеющих иерархическую структуру, обеспечивается тем, что над этим данными строится специальный интерфейс, которому в *GTK+*, например, соответствует *GtkTreeModel*. Дерево статистики событий (в своих шести «ипостасях») является, пожалуй, наиболее сложным из всех иерархических объектов, с которыми имеет дело Трассировщик.

Элементом любого из массивов Eco, Coe, Oec является указатель на структуру, описывающую терминальный узел дерева событий. Эта структура, наряду с данными, определяющими соответствующую тройку $\langle e,\ c,\ o\rangle$, содержит информацию о типе узла (зависящем от его положением в иерархии, в данном случае – указывающем на то, что содержимым узла является *eco*-тройка), а также ссылки на описатель трассы и на

потенциальных родителей (их два – согласно 4.3). Описатель трассы объединяет всю информацию, необходимую для интерпретации трассы, в частности, содержит общий для трассы параметр настройки – вариант иерархии.

Современная версия Трассировщика допускает работу с несколькими трассами. Деревья статистики событий, относящиеся к различным трассам, представлены в общей таблице (*GtkTreeView*), допускающей выделение нескольких строк. Эта возможность используется для задания списка трасс, сформированных взаимодействующими между собой целевыми ЭВМ. Такие списки обычно используются для объединения трасс в общей таблице, где их записи упорядочены по времени, что позволяет анализировать поведение многомодульной системы в целом.

Каждой трассе соответствует своя ветвь дерева, отображаемого виджетом *GtkTreeView*, в отношении которой используется свой вариант иерархии. Кроме того, в каждой из ветвей вводятся (для большей наглядности) дополнительные уровни – групп событий $\langle G \rangle$, типов контекстов $\langle T_c \rangle$ и типов главных объектов $\langle T_{(obj)} \rangle$. Как следствие всей совокупности узлов дерева событий конкретной трассы соответствуют иерархии типов:

- $\langle G \rangle|\langle e \rangle|\langle e,c \rangle|\langle e,c,o \rangle$,
- $\langle G \rangle|\langle e \rangle|\langle e,o \rangle|\langle e,c,o \rangle$,
- $\langle T_c \rangle|\langle c \rangle|\langle e,c \rangle|\langle e,c,o \rangle$,
- $\langle T_c \rangle|\langle c \rangle|\langle c,o \rangle|\langle e,c,o \rangle$,
- $\langle T_{(obj)} \rangle|\langle o \rangle|\langle e,o \rangle|\langle e,c,o \rangle$,
- $\langle T_{(obj)} \rangle|\langle o \rangle|\langle c,o \rangle|\langle e,c,o \rangle$.

Каждый из моно-узлов $\langle e \rangle$, $\langle c \rangle$ и $\langle o \rangle$ ссылается на соответствующие ему диапазоны двух вариантов дочерних массивов – $\{Ec,\ Eo\}$, $\{Ec,\ Co\}$ и $\{Eo,\ Co\}$, упорядоченных в соответствии с иерархией. Каждый из узлов-пар $\langle e,c \rangle$, $\langle e,o \rangle, \langle c,o \rangle$ ссылается на две родительские вершины $\{\langle e \rangle,\ \langle c \rangle\}$, $\{\langle e \rangle,\ \langle o \rangle\}$ и $\{\langle c \rangle, \langle o \rangle\}$, ссылка определяется иерархией. Остальным узлам, отличным от терминальных, всегда соответствует один дочерний и один родительский массив.

В предыдущих версиях Трассировщика дерево событий каждой трассы было представлено шестью независимыми деревьями, реализованными как объекты класса *GtkTreeStore*. Следствием такого решения была дорогостоящая процедура формирования из массивов Eco, Coe, Oec списков *GList*, лежащих в основе модели *GtkTreeStore*. Современная модель, основанная на реализации интерфейса *GtkTreeModel*, лишена этих недостатков. Модель внутреннего представления дерева событий едина, представляющие ее массивы ни во что не конвертируются и не меняются при переходе от одного варианта иерархии к другому. Меняется лишь единственный параметр трассы, соответствующий существенному варианту иерархии. Выбирая соответствующее значение этого параметра, пользователь практически мгновенно меняет иерархию отображаемого на экране дерева.

4.5 Дерево событий – инструмент для задания условий отбора записей трассы

Как и в прежних версиях Трассировщика, узлы дерева событий используются для задания условий отбора записей трассы. Они помечаются маркерами «*отобрать*» и «*исключить*» – в любой иерархии. Отбор записей осуществляется в два этапа. На первом этапе обрабатываются *eco*-тройки – либо помеченные непосредственно, либо подчиненные помеченным узлам. При этом маркер любого узла «сильнее» маркера его родителя: он переопределяет условие отбора. На втором этапе – в процессе обхода вторичного индексного массива I_E (см. 0) обновляются значения элементов битового массива – таким образом, что ненулевые биты соответствуют отобранным записям.

5. Отображение агрегатов данных в скроллируемых таблицах

Задачу отображения файла (набора данных в оперативной памяти), записи которого не имеют фиксированной длины, в виде одноуровневой скроллируемой таблицы – достаточно часто приходится решать при разработке *GUI*-приложений. При наличии большого числа записей особую актуальность приобретает скорость вертикальной прокрутки таблицы. Высокая скорость прокрутки обеспечивается в том случае, если в процессе скроллинга вычисление пространственных координат любой строки требует минимальных затрат времени. В самом общем случае, средства разработки графического интерфейса, такие, например, как *GTK+*, рассчитывает поправку к значениям координат строк по оси *Y* всякий раз, когда меняется их высота (например, при изменении ширины колонок, допускающих перенос текста). Перед первым выводом содержимого таблицы на экран рассчитываются высоты всех без исключения строк, что сопровождается явным преобразованием данных отображаемой модели в тексты – в соответствии с выбором шрифтов.

Скорость формирования таблицы значительно повышается, если фиксировать высоту ее строк – поскольку при этом процедура разметки предельно упрощается, в частности, не требует обращения к функциям, определяющим параметры рендерера (см. 3.6). (Из этого условия, в частности, вытекает требование недопустимости переноса текста внутри ячеек «большой» таблицы.)

5.1 Основная таблица и виджет выделенной записи

Для увеличения скорости скроллинга таблиц с большим числом строк (не только таблицы с трассой событий) может быть применен единый подход, суть которого сводится к следующему:

- записи индексируются, индекс (номер записи) ассоциируется с ее адресом (длина либо задается непосредственно, либо – в случае выполнения условия связности записей – может быть определена как разность адресов соседних записей;
- в ячейках строки таблицы отображается не вся информация, содержащаяся в конкретной записи, а лишь та ее часть, которая присуща всем записям (например, время регистрации записи), что позволяет использовать фиксированную высоту строк таблицы;
- оставшаяся часть записи (либо ее полное содержание) отображается отдельным виджетом, причем лишь тогда, когда она выделена курсором в основной таблице (в мультиселектных таблицах – на последнем шаге выделения).

Реализация этого подхода применительно к трассе событий может быть проиллюстрирована следующей схемой (включающей дерево событий):

Рис. 1. Отображение детальной информации выделенной записи в отдельном виджете
Fig. 1. Visualization of selected record details by standalone widget

5.2 Линейная модель агрегатов данных

В общем случае наборы данных, отображаемые в скроллируемых одноуровневых таблицах, могут быть представлены последовательностью $D = (a_i)_{i=1}^{|D|}$ агрегатов данных a_i, где $|D|$ – количество агрегатов. Рассмотрим – в рамках концепции *MVC* (раздел 2.7) – одноуровневую таблицу, являющуюся представлением (*View*) последовательности D. Модель (*Model*), реализующую доступ к элементу a_i этой последовательности по его номеру $i = value(node)$, где $node$ – узел *Model*, назовем линейной.

Позиция P любой строки таблицы определяется ее номером row. В отсутствие фильтрации записей этот номер совпадает с номером i агрегата данных a_i: $row = i$. Такая модель может быть применена к источникам данных любой природы, лишь бы они были представлены последовательностью D. В этом случае *Model* не нуждается в каких либо данных, отличных от количества агрегатов $|D|$.

Под фильтром номер строки не совпадает с номером записи агрегата данных, что побуждает строить массивы, в которых номерам строк, удовлетворяющих фильтру, соответствуют номера записей. Компактной реализацией такого соответствия является битовый массив $BitArray$, доступный для редактирования со стороны функций, внешних по отношению к *Model*. Включение $BitArray$ непосредственно в *Model*, позволяет свести навигацию (с помощью функций $next(node)$, $prev(node)$, $parent(node)$, $nth\_child(node, n)$) по отобранным агрегатам данных, к навигации по массиву $BitArray$.

Реализация функции $paint(render_i, c_i, node)$, определяющей параметры объекта $render_i$, отображающего данные, связанные с узлом $node$, в ячейке столбца c_i, обеспечивается тем, что в реализациях функции $node(P)$ и $P(node)$ также, как и в функциях навигации, используется массив $BitArray$.

В Трассировщике линейная модель представлена классом $Lines$, дочерним по отношению к классу *GObject* и реализующим методы интерфейса *GtkTreeModel*.

Эта модель может быть применена к агрегатам данных различной природы: к трассе, текстовому файлу, записи которого разделяет символ конца строки, к любому бинарному файлу, который отображается как последовательность записей, длина которых не фиксирована и т.п. Для любых типов агрегатов данных могут быть определены условия отбора, присущие только этому типу. От реализующих эти условия алгоритмов требуется одно: результатом отбора должно быть содержимое битового массива линейной модели.

6. Трассы состояний

В каждом из состояний объект ОС (или процессор) непрерывно пребывает в течение некоторого отрезка времени. Последовательность таких отрезков времени есть трасса состояний объекта (иначе – временная диаграмма состояний объекта). Переход из одного состояния в другое всегда есть результат некоторого события. Множество типов событий, обусловливающих переход, в сочетании с условиями перехода (в том числе, учитывающими предшествующее состояние объекта) определяют алгоритм построения последовательности смены состояний объекта – трассу состояний.

Детализация (классификация) состояний (см. 2.6) может иметь несколько уровней агрегирования.

6.1 Агрегированные состояния

Каждому уровню агрегирования соответствует номер l ($l = 1, \dots, N_L$, где N_L – максимальное число уровней агрегирования), а уровню атомарных состояний соответствует номер $l = 1$.

Состояния объекта o операционной системы образуют множество S_t, определяемое типом t объекта o:

$$S_t = \{\sigma_i^t | i = 1, \dots, N_t, t = Type(o), t \in T_{(obj)}\},$$

где N_t – количество возможных состояний объектов типа t.

Множество S_t включает в себя как атомарные, так и агрегированные состояния. Каждому агрегату уровня l соответствует подмножество множества S_t, состоящее из агрегатов уровня $(l-1)$. Для объектов типа t может быть определена функция $Aggregate(\sigma, t, l)$, результатом выполнения которой является состояние $a \in S_t$, являющееся -агрегатом по отношению к состоянию σ:

$$a = Aggregate(\sigma, t, l),$$

где $t = Type(o)$, l – номер уровня агрегирования.

6.2 Модель трассы атомарных состояний (временная диаграмма)

Пусть $\tau_1, \dots, \tau_n$ – последовательность моментов времени, в которых происходит смена состояний некоторого объекта o типа $t \in T_{(obj)}$. В течение промежутка времени (τ_i, τ_{i+1}), характеризуемого длительностью $\Delta\tau_i = \tau_{i+1} - \tau_i$, объект пребывает в состоянии $\sigma_i \in S_t$. Обычно (например, при отображении) трасса (временная диаграмма) состояний объекта o представляется в виде последовательности троек $s_i = \langle \tau_i, \sigma_i, \Delta\tau_i \rangle$:

$$\Im_o = (s_i)_{i=1}^{|\Im_o|},$$

где $|\Im_o|$ – количество переходов объекта o из одного состояния в другое, i – номер перехода.

Поскольку каждому -му моменту перехода соответствует определяющее его событие (существенная запись $\overline{r}_{j_i} \in E$, где $j_i \in I_E$ – вторичный индекс записи), то компоненты тройки s_i полностью определяются парой (j_i и j_{i+1}) вторичных индексов, соответствующих соседним переходам из одного состояния в другое:

$$\tau_i = Time(j_i),\ \sigma_i = State(j_i, o), \tau_{i+1} = Time(j_{i+1}), \Delta\tau_i = \tau_{i+1} - \tau_i,$$

При этом предполагается, что описатель объекта o содержит необходимые ссылки и к трассе событий, и к ее описанию.

Нетрудно видеть, что хранить временную диаграмму в памяти инструментальной ЭВМ в виде троек $\langle \tau_i, \sigma_i, \Delta\tau_i \rangle$ нет необходимости. Временная диаграмма «атомарных» состояний полностью определяется массивом целых чисел, реализующих последовательность вторичных индексов $I_o = (j_i)_{i=1}^{|\Im_o|}$ $(I_o \subset I_E)$, где i – номер перехода. Получение значений, соответствующих «каноническому» представлению можно отложить до момента, когда они реально требуются – например, при их отображении.

6.3 Агрегированные состояния и многослойная модель агрегатов данных

Трасса агрегированных состояний объекта o уровня (слоя) l может быть представлена в виде последовательности $A_o^l = (\alpha_i)_{i=1}^{|A_o^l|}$, где α_i – i-й переход объекта в одно из возможных агрегированных состояний, $|A_o^l|$ – количество переходов. В то же время агрегированная трасса A_o^l – в соответствии с определением агрегирования – может быть представлена последовательностью целых чисел $I_o^l = (j_i^{l-1})_{i=1}^{|A_o^l|}$, где i – номер перехода внутри слоя l, а j_i^{l-1} – соответствующий ему номер перехода внутри слоя $(l-1)$. Отношения между уровнями агрегирования порождают естественную иерархию реализованных состояний.

Массивы, соответствующие слоям агрегирования, никак не связаны с природой данных, которые они объединяют в иерархическую структуру. При использовании *GTK+* они являются основой универсальной многослойной иерархической модели с интерфейсом *GtkTreeModel*. По отношению к этим массивам могут быть применены те или иные процедуры отбора. Для фильтрации отобранных записей (и навигации по ним в режиме отключенного фильтра) в модель встроен механизм, аналогичный тому, что реализован в

линейной модели (см. 4.5 и 5.2): с каждым слоем связан битовый массив, ненулевые биты которого соответствуют отобранным элементам слоя.

В предыдущих версиях Трассировщика трассы состояний отображались в трех режимах: «детальная», «промежуточная» и «укрупненная», реализованных с помощью модели *GtkListStore*. Отчасти это решение было продиктовано опасениями, что иерархическая модель окажется «медленнее» линейной.

Многослойная модель лежит в основе всех трасс состояний, с которыми работает современный Трассировщик. При этом скорость прокрутки и позиционирования в дереве, отображаемом на экране, скорость раскрытия родительских узлов не вызывает никакого дискомфорта: опасения, что иерархическая модель окажется существенно медленнее линейной, не подтвердились.

В предыдущих версиях Трассировщика трассы состояний отображались – наряду с табличной формой – в виде прямоугольников, размещенных в специальной «области рисования» (виджете *GtkDrawingArea*). В современной версии этой цели служат ячейки виджета *GtkTreeView*, в которых прямоугольники отображаются с помощью объектов класса *GtkCellRendererPixbuf*.

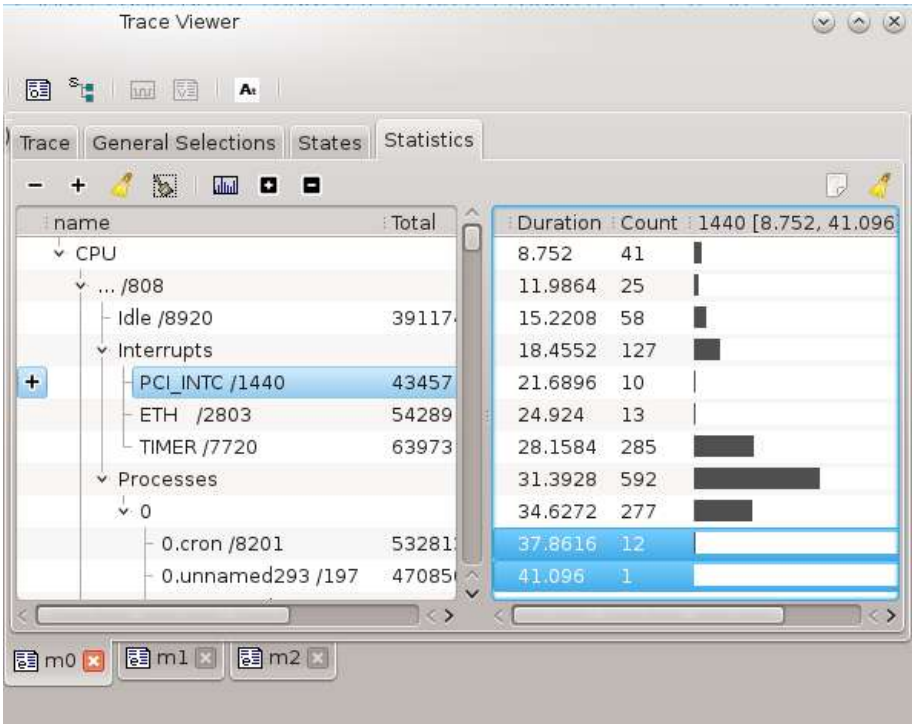


Рис. 2. Гистограмма состояния процессора
Fig. 2. Histogram of CPU state

6.4 Статистика состояний и условия отбора записей трасс состояний

В процессе формирования Трассировщиком трассы состояний *CPU* или объекта ОС рассчитываются его статистические характеристики: общая, средняя и максимальная длительность реализации состояний. Статистические данные по всем объектам, для которых строилась трасса состояний, накапливаются в дереве статистики состояний. Его верхний уровень образуют вершины, соответствующие процессору и тем типам объектов ОС, для которых построены трассы состояний. Каждый тип объединяет вершины второго уровня — соответствующие этому типу объекты ОС.

Содержимое ветви, подчиненной вершине, соответствующей конкретному объекту ОС или процессору (*CPU*) отражает иерархию состояний объекта с точки зрения «степени детальности» трассы, при этом «детали», конкретизирующие «менее подробное» состояние, находятся на более глубоких уровнях.

Условия отбора, накладываемые на записи трасс состояний, можно задавать, помечая узлы дерева статистики состояний маркерами «*отобрать*» и «*исключить*».

Моделью дерева является представление *GtkTreeStore*. С каждым его узлом по запросу пользователя может быть связана таблица (*GtkTreeView)*, отображаемая при выделении узла дерева статистики состояний. Колонки этого виджета служат для представления гистограммы реализаций связанного с узлом состояния. В них представлены количества реализаций состояний в данном диапазоне длительностей – в числах (в ячейках *GtkCellRendererText*), и в виде прямоугольников (в ячейках, поддерживаемых объектами *GtkCellRendererPixbuf*), ширина каждого из которых соответствует этому количеству.

Выделение одной или нескольких строк в таблице, отображающей гистограмму, задает дополнительные условия отбора записей трассы состояний: отобраны будут лишь те из них, у которых длительности реализации будут лежать в диапазонах, соответствующих выделенным «столбцам» гистограммы.

7. Заключение

Рассмотренные в статье модели трасс (протоколов событий), формируемых в процессе работы приложений реального времени, и трасс состояний, формируемых в ходе анализа событий, а также механизмы их интерпретации и визуализации реализованы в современных средствах трассировки ОС РВ семейства «Багет». Опыт их эксплуатации дает основания считать удачным использование предложенных подходов. К числу очевидных положительных результатов следует отнести значительное сокращение объема программного кода, повышение эффективности и надежности реализации основных функций средств трассировки.

Предпринятая в статье попытка представить предлагаемые решения в максимально формализованном виде преследовала единственную цель – показать, что эти решения никак не связаны ни с использованием языка программирования, ни со спецификой ОС РВ семейства «Багет», ни с особенностью реализации функций просмотра и анализа трасс. Несколько более детальное описание моделей и механизмов, реализованных с помощью *GTK+*, не означает, на наш взгляд, что подобные механизмы не могут быть реализованы с помощью иных инструментов, используемых при создании графического пользовательского интерфейса.

Список литературы / References

[1]. А.Н. Годунов, Л.В. Жихарский, П.Е. Назаров, Ф.Н. Чемерев. Средства протоколирования в ос2000. Программные продукты и системы, no. 3, 2007, стр. 22-27 / A.N. Godunov, L.V. Zhikharsky, P.E. Nazarov, F.N. Chemerev. Logging tools in oc2000. Software Products and Systems, no. 3, 2007, pp. 22-27 (in Russian).

[2]. IEEE 1003.1-2001 – IEEE Standard for IEEE Information Technology - Portable Operating System Interface (POSIX(R)). Available at: https://standards.ieee.org/standard/1003_1-2001.html.

[3]. Percepio Tracealyzer. Available at: https://percepio.com/tz/.

[4]. BlackBerry QNX. Available at: http://blackberry.qnx.com/.

[5]. Green Hills INTEGRITY-178. Available at: https://www.ghs.com/products/safety_critical/integrity-do-178b.html.

[6]. Express Logic ThreadX. Available at: https://rtos.com/.

[7]. On Time RTOS-32. Available at: http://www.on-time.com/rtos-32-docs/.

[8]. Wind River VxWorks. Available at: https://www.windriver.com/products/vxworks/.

[9]. SYSGO PikeOS. Available at: https://www.sysgo.com/products/pikeos-hypervisor.

[10]. Altreonic OpenComRTOS. Available at: http://www.altreonic.com/.

[11]. К.А. Костюхин. Средства самоконтроля программ и их применение при отладке систем со сложной архитектурой. В сборнике «Информационная безопасность. Микропроцессоры. Отладка сложных систем». М., НИИСИ РАН, 2004, стр. 151-160 / K.A. Kostyukhin. Tools for program self-control and their application in debugging of systems with complex architecture. In «Information Security. Microprocessors. Debugging complex systems.» M., SRISA/NIISI RAS, 2004, pp. 151-160 (in Russian).

[12]. С. Рогачев. Обобщённый Model-View-Controller. Каркас на основе шаблона проектирования MVC в исполнении Generic Java и C# / S. Rogachev. Generalized Model-View-Controller. Framework based on the MVC design pattern and implemented with Generic Java and C#. Available at: http://rsdn.org/article/patterns/generic-mvc.xml (in Russian).

[13]. GTK+ 3 Reference Manual. Available at: https://developer.gnome.org/gtk3/stable/.

[14]. Qt Documentation. Available at: https://doc.qt.io/.

Информация об авторах / Information about authors

Александр Николаевич ГОДУНОВ – заведующий отделом системного программирования, кандидат физико-математических наук.

Alexander Nikolayevich GODUNOV – head of the system programming department, candidate of physical and mathematical sciences.

Федор Николаевич ЧЕМЕРЕВ, ведущий инженер отдела системного программирования.

Fedor Nikolaevich CHEMEREV, Leading Engineer of the System Programming Department.

DOI: 10.15514/ISPRAS-2019-31(4)-2

Автоматизация обнаружения и анализа ошибок в гиперконвергентных системах

Д.В. Силаков, ORCID: 0000-0001-9175-6943 <dsilakov@virtuozzo.com>
ООО «Виртуоззо Рисерч»,
127273, Россия, г. Москва, ул. Отрадная, д. 2Б, стр. 9

Аннотация. Статья посвящена проблеме выявления и оперативного анализ ошибок, возникающих при эксплуатации гиперконвергентных систем. Одним из подходов к организации гиперконвергентных систем является установка на каждый физический сервер отдельного экземпляра операционной системы (ОС), несущей в себе средства виртуализации и инструментарий для администрирования и использования распределенного хранилища данных. Возникновение ошибок возможно как на уровне отдельного экземпляра ОС, так и на уровне всего кластера, Например, некорректные команды управляющих элементов с одного узла инфраструктуры могут вызвать сбой ПО на другом узле. Кроме того, ошибки со стороны подсистем кластера могут спровоцировать нештатные ситуации внутри виртуальных машин. Сложность архитектуры гиперконвергентных систем обуславливает сложность анализа возникающих в них ошибок. Для упрощения такого анализа и повышения его эффективности необходима автоматизация процесса обнаружения проблем и сбора данных, необходимых для их изучения и исправления. Рассматриваются подходы к автоматизации подобных процессов в существующих ОС и предлагаются способы их адаптации к системам, использующим распределенное хранилище данных и виртуализацию. Описывается опыт применения адаптированных решений в продуктах Virtuozzo.

Ключевые слова: обнаружение ошибок; виртуализация; хранилище данных

Для цитирования: Силаков Д.В. Автоматизация обнаружения и анализа ошибок в гиперконвергентных системах. Труды ИСП РАН, том 31, вып. 4, 2019 г., стр. 29-38. DOI: 10.15514/ISPRAS-2019-31(4)-2

Automated Error Detection and Analysis in Hyperconverged Systems

D.V.Silakov, ORCID: 0000-0001-9175-6943 <dsilakov@virtuozzo.com>
Virtuozzo,
Seattle, USA (HQ) 110 110th Ave NE #410, Bellevue, WA 98004

Abstract. The paper is devoted to the problem of early error detection and analysis in hyperconverged systems. One approach to organizing hyperconverged systems is to install on each physical server a separate instance of an operating system (OS) that carries virtualization tools and tools for administering and using a distributed data warehouse. Errors can occur both at the level of a single OS instance and at the level of the entire cluster. For example, incorrect control element commands from one infrastructure node can cause software failure on another node. In addition, errors from the subsystems of the cluster can provoke abnormal situations inside virtual machines. The complexity of the architecture of hyperconverged systems makes it difficult to analyze the errors that occur in them. To simplify such an analysis and increase its effectiveness, it is necessary to automate the process of detecting problems and collecting data necessary for their study and correction. Existing approaches for automation of error detection are described and various improvements are suggested to adopt them for systems where distributed storage and virtualization technologies are actively used. Improvements include log collection from the whole cluster just after the error occurred, additional analysis of guest operating system behaviour inside virtual machines, usage of a knowledge base for automated crash recovery and

duplicate detection. Finally, a real-life scenario of error handling process in Virtuozzo company products is described starting from error detection and ending with fix deployment.

Keywords: error detection; virtualization; data storage

For citation: Silakov D.V. Automated Error Detection and Analysis in Hyperconverged Systems. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 4, 2019. pp. 29-38 (in Russian). DOI: 10.15514/ISPRAS-2019-31(4)-2

1. Введение

Выявление проблем в ходе эксплуатации ПО и оперативное реагирование на их возникновение – задача, которая вряд ли когда-нибудь потеряет актуальность. Мониторинг и анализ нештатных ситуаций необходимы как при реальной эксплуатации, так и при различного вида тестировании на всех стадиях разработки.

К настоящему времени создано немало средств автоматизации подобного мониторинга, однако новые тенденции в миру ПО предъявляют новые требования к инструментарию. К одной из таких тенденций относится создание гиперконвергентных инфраструктур, которые подразумевают использование программных средств для объединения ресурсов множества серверов в кластер, являющийся с точки зрения системных администраторов и пользователей единой средой с централизованным управлением.

Ресурсы серверов включают прежде всего вычислительные мощности и дисковое пространство. В гиперконвергентной системе устройства хранения данных множество машин объединяется в сетевое хранилище (возможно, с дублированием данных в целях повышения отказоустойчивости), а вычислительные ресурсы используются для создания и запуска виртуальных машин. Именно последние занимаются выполнением задач конечных пользователей. Такой подход позволяет гибко и в то же время эффективно распределять физические ресурсы между различными прикладными задачами.

Одним из подходов к организации гиперконвергентных систем является установка на каждый физический сервер отдельного экземпляра операционной системы (ОС), несущей в себе средства виртуализации и инструментарий для администрирования и использования распределенного хранилища данных.

Возникновение ошибок возможно как на уровне отдельного экземпляра ОС, так и на уровне всего кластера – например, некорректные команды управляющих элементов с одного узла инфраструктуры могут вызвать сбой ПО на другом узле. Кроме того, ошибки со стороны подсистем кластера могут спровоцировать нештатные ситуации внутри виртуальных машин.

Сложность архитектуры гиперконвергентных систем обуславливает сложность анализа возникающих в них ошибок. Для упрощения такого анализа и повышения его эффективности необходима автоматизация процесса обнаружения проблем и сбора данных, необходимых для их изучения и исправления.

2. Существующие подходы к автоматизации обработки нештатных ситуаций

Существующие подходы к обнаружению нештатных ситуаций можно разделить на две основных категории:

- выявление ошибок непосредственно в момент их возникновения;
- выявления факта возникновения проблемы на основе анализа системных журналов.

2.1 Оперативное выявление проблем

Применение первого способа позволяет произвести анализ системы и сбор потенциально полезной для анализа проблемы информации «по горячим следам». Например, можно сделать снимок памяти проблемного процесса, сохранить контекст его выполнения и так далее.

Однако реализация такого подхода сильно зависит от того, какого рода ошибки необходимо выявлять. Каждый тип нештатной ситуации может потребовать разработки отдельного инструментального средства для его оперативного обнаружения.

Например, широко применяемые в Linux инструментальные наборы ABRT [1] и Apport [2] позволяют отслеживать следующие виды нештатных ситуаций:

- ошибки сегментации («Segmentation Fault») в бинарных программах – как правило, вызванные некорректной работой с памятью;
- возникновение исключений в программах на Python, Ruby и Java;
- аварийное завершение работы ядра Linux с последующей перезагрузкой системы, а также сообщения «OOPS» от ядра, сигнализирующие о нештатной ситуации, выявленной самим ядром (которая могла и не повлечь остановки машины);
- возникновение проблем при работе графической подсистемы (X-сервера).

Для обнаружения каждой из указанных проблем реализована отдельная утилита, использующая специфические средства ОС и системного ПО. В частности:

- ошибки работы с памятью регистрируются с помощью ядра Linux, посредством добавления обработчика в файл /proc/sys/kernel/core_pattern [3];
- для обнаружения исключений в Java-программах в среде исполнения Java (Java Runtime Environment) регистрируется соответствующий обработчик, использующий Java Native Interface;
- схожий механизм используется для обнаружения проблем в скриптах на Python и Ruby – их среды исполнения также позволяют регистрировать сторонние обработчики исключений;
- аварийное завершения работы ядра Linux, приведшие к перезагрузке машины, определяется по наличию файла vmcore (образа памяти ядра на момент завершения работы, автоматически создаваемого при возникновении нештатной ситуации);
- прочие нештатные ситуации, связанные с ядром Linux (к таковым также относятся потенциальные проблемы с оборудованием, о которых сообщает ядро), отслеживаются путем мониторинга файла /proc/kmsg, предоставляющего прямой доступ к сообщениям ядра;
- для обнаружения проблем графической подсистемы ведется постоянный мониторинг файла журнала /var/log/Xorg.0.log и анализ добавляемой в него информации.

К достоинствам данного подходя относится возможность собрать информацию, которая доступна только в течение короткого промежутка времени после возникновения инцидента, но которая будет крайне полезна для его анализа. К подобным данным относится снимок памяти упавшего процесса, перечень открытых им файлов и сетевых соединений и прочая информация, которая находится только в оперативной памяти машины.

Архитектура инструментальных средств наподобие ABRT и Apport является модульной и позволяет добавлять обработчики произвольных событий – необходимо только разработать соответствующее инструментальное средство. Необходимость такой разработки и является основным фактором, ограничивающим применение подхода.

Для многих приложений единственный способ определить нештатную ситуацию в их работе (не вызывающую аварийного останова программы, но критическую для ее функционирования) заключается в анализе журналов – как общесистемных, так и их собственных. Разрабатывать и запускать инструменты такого анализа для каждого приложения в системе может оказаться слишком накладно (особенно если стоит цель мониторинга журналов в реальном времени) – и тогда может быть использован альтернативный подход.

2.2 Выявление ошибок на основе анализа журналов

Другое популярное решение основано на том, что в большинстве программных систем ведутся журналы различных событий, в число которых обычно входят и любые нештатные ситуации. Анализируя журналы событий, можно определить и факт возникновения тех или иных инцидентов, требующих внимания разработчиков и системных администраторов. Можно осуществлять как мониторинг журналов в реальном времени, реагируя на каждую новую запись, так и периодический анализ данных, появившихся за определенный промежуток времени.

В настоящее время существует множество программных средств, занимающихся анализом и фильтрацией журналов с целью выявления заданных событий. При этом многие решения (например, Graylog [4]) допускают обработку журналов со множества машин и имеют расширяемую архитектуру, позволяющую оперативно добавлять формальные описания интересующих пользователя системы событий.

Наряду с однозначными формальными правилами фильтрации событий (например, с использованием регулярных выражений для выделения сообщений об ошибках), для выявления нештатных ситуаций могут применяться нечеткие правила и правила, выведенные на основе машинного обучения [5].

Помимо гибкости в добавлении описаний и масштабируемости, к достоинствам таких систем стоит отнести возможность повторного анализа уже имеющихся журналов при добавлении или обновлении описания события для анализа. Основным же недостатком в большинстве случаев является отсутствие в журналах информации, необходимой для детального изучения проблемы.

Например, аварийное завершение приложений в дистрибутивах Linux зачастую может остаться вне поля зрения системных журналов. Но даже если факт аварийной остановки будет отмечен, то в журналы редко помещается детальная информация – такая, как снимок памяти процесса. Впрочем, для приложений, использующих для работы специфическую среду исполнения (в частности – программ, написанных на языках Go, Java, Python и ряде других), в системные журналы может помещаться трасса вызова функций, приведших к возникновению нештатной ситуации.

Тем не менее, в общем случае подход с анализом журналов в плане полноты информации сильно уступает целенаправленному сбору данных об инциденте в момент его возникновения.

Ниже мы рассмотрим возможные усовершенствования существующих подходов к обработке инцидентов «по горячим следам», которые позволят повысить эффективность анализа проблем.

3. Обход кластера для сбора информации

Большинство автоматических средств сбора информации о нештатных ситуациях в момент их возникновения ограничиваются рамками ОС, в которой они работают. В гиперконвергентных системах такой подход может оказаться недостаточен. Такие системы являются распределенными, и ОС на отдельно взятой машине не является изолированной и самодостаточной – в ней могут выполняться служебные процессы, контролируемые управляющими элементами с других узлов кластера.

Для воспроизведения полной картины инцидента, связанного с подобными служебными процессами, может потребоваться информация и от управляющих узлов. Например, в системах, реализующих отказоустойчивость сервисов посредством их дублирования, некорректное поведение служебной программы может быть вызвано поступлением противоречивых команд от разных управляющих процессов.

Обход узлов кластера и сбор данных с удаленных машин – более дорогая операция, чем анализ в рамках одного физического сервера. Поэтому перед обращением на удаленные машины необходимо определить, есть ли в этом смысл.

Одним из возможных способов выяснить целесообразность обхода является составление списка процессов, ошибки в которых могут быть вызваны удаленными узлами кластера. В этот список могут входить служебные сервисы, обслуживающие инфраструктуру кластера, а также любые программы, обращающиеся напрямую к удаленным узлам системы. Если инцидент произошел с процессом, не входящим в этот список, то собирать данные с удаленных узлов не надо.

Еще большей точности можно достигнуть, проанализировав, с какими узлами и сервисами кластера взаимодействовал проблемный процесс непосредственно перед возникновением аварийной ситуации.

4. Анализ ошибок виртуализации

Помимо распределенной природы многих сервисов, важным аспектом гиперконвергентных систем является использование виртуальных машин (ВМ). В идеале каждая ВМ с точки зрения пользователя равнозначна отдельной физической машине. Однако изоляция процессов одной ВМ от другой реализуется с использованием программных средств (гипервизора либо ядра ОС при использовании контейнерной виртуализации), в которых могут содержаться ошибки, приводящие либо к нарушению изоляции, либо дезориентации гостевой операционной системы.

Проблемы в подсистеме виртуализации могут фатально сказаться как на виртуальных окружениях, так и на хост-системе, в которой они запущены. Ввиду возможности таких ошибок отдельной обработки заслуживают следующие ситуации:

- аварийный останов виртуальной машины;
- ошибка в сервисе, входящем в систему виртуализации на хост-системе.

В обоих случаях необходимо собрать данные как о событиях, происходивших на сервере в приложениях виртуализации, так и о том, что происходило в этот момент внутри ВМ. К потенциально полезным данным изнутри ВМ относятся:

- сведения о нештатных ситуациях внутри гостевой ОС (в частности, аварийное завершение работы приложений и ядра ОС);
- список процессов, которые были активны на момент инцидента;
- список установленного ПО.

Список процессов и приложений может помочь в определении потенциально вредоносного ПО, работавшего внутри ВМ, которое и могло стать причиной инцидента [6].

Информация изнутри ВМ может быть извлечена либо посредством гостевого агента (при условии, что ВМ запущена) либо непосредственно с образа ее жесткого диска (при условии, что в гостевой системе не используется шифрование дискового пространства). В случае, если в результате инцидента произошло аварийное выключение ВМ, система виртуализации может предоставить снимок памяти всех процессов гостевой ОС для дальнейшего анализа.

5. Выявление одинаковых ошибок

Перед сдачей программного обеспечения в эксплуатацию большинство разработчиков проводят его тщательное тестирование. Тем не менее, определенный процент ошибок в программах остается, но для их проявления нужны некоторые специфические условия - определенное сочетание переменных среды и аргументов, нехватка оперативной памяти или дискового пространства и тому подобное. Как следствие, подобные ошибки встречаются относительно редко, но при массовом использовании продукта общее число инцидентов может стать значительным.

В частности, в гиперконвергентных системах многие машины зачастую имеют идентичную конфигурацию и занимаются схожими задачами. Велика вероятность того, что если ошибка возникла на одном из серверов, то она может воспроизвестись и на других серверах. Знание о том, что проблеме подвержено множество машин, полезно, однако проводить тщательный анализ каждой из них излишне.

Проверка того, являются ли две ошибки дубликатами или нет, – отдельная задача, для которой существует множество возможных решений. Одно и наиболее простых – это прямое сравнение трасс вызова функций. Этот метод может быть не очень эффективен в случае сложных приложений (в частности – многопоточных), но существует немало решений, производящих нечеткое сравнение трасс, в том числе использующих машинное обучение для поиска дубликатов – см., например, работы [7] и [8].

Большинство методов определения дубликатов применимо независимо от того, принадлежат ли трассы ошибкам, случившимся на одной физической машине или на разных машинах. Однако в настоящее время они применяются на приемнике, куда поступают отчеты об ошибках. Если же мы хотим избежать сбора излишних данных на стороне пользовательской системы (и не посылать излишнего количества отчетов), то выявление дубликатов должно производиться на стороне пользователя.

В ряд существующих инструментов (в частности, ABRT) уже встроен механизм отсеивания дубликатов – для этого в системе хранится история уже обработанных ранее ошибок, и при обнаружении новой проблемы она первым делом сравнивается с теми, что есть в истории.

В случае наличия кластера взаимосвязанных машин логичным усовершенствованием данного подхода будет создание единого (в рамках кластера) хранилища с историей ошибок. При возникновении нового инцидента на одном из узлов кластера инструментарий анализа проблем сможет сверяться с этим хранилищем и принимать решение – производить детальный анализ и сбор данных о проблеме либо проигнорировать ее как уже обработанную.

Помимо истории ошибок конкретного кластера, для обнаружения дубликатов может быть задействована и централизованная база знаний об ошибках на стороне разработчиков или системных администраторов, о которой пойдет речь в следующем разделе.

6. Обратная связь

Большинство программ, отслеживающих возникновение нештатных ситуаций, играют исключительно пассивную роль – они только отправляют извещение о проблеме и никак не вмешиваются в работу системы. Однако с точки зрения пользователя важно не только обнаружить проблему и изучить ее причины, но и оперативно ее устранить.

Современные программные комплексы предоставляют набор примитивов, позволяющих в ряде случаев автоматически восстанавливать функционал системы. Например, при аварийном останове некоторого сервиса он может автоматически перезапускаться средствами ОС, при нехватке места на диске могут удаляться наименее ценные файлы и так далее.

Такие средства, предоставляемые ОС, претендуют на универсальность и, как следствие, во многих случаях не могут помочь, поскольку не учитывают специфику конкретной проблемы. Так, если сервис аварийно останавливается из-за поврежденного файла конфигурации, то стартовать сервис до исправления этого файла бессмысленно.

Какие именно действия необходимо произвести для возвращения системы к штатному режиму работы, зависит от конкретной программы и от того, что с ней произошло. Заложить все возможные сочетания проблем и их решений непосредственно в систему при ее развертывании не представляется возможным – ведь множество таких сочетаний растет в ходе жизненного цикла ПО как за счет выявления новых потенциальных проблем, так и за счет разработки новых путей для их решений.

Перспективным решением этой проблемы является использование базы знаний, содержащей информацию о нештатных ситуациях, которые могут возникнуть в ходе эксплуатации системы, а также о методах их решения.

Подобные базы уже поддерживаются многими проектами, однако описания проблем и их решений в них содержится в человекочитаемой, зачастую – неформальной форме. Для использования этих баз в программных средствах обнаружения ошибок необходимо выделить формальные признаки, которые характеризуют каждую проблему и наличие которых можно определить автоматически.

К подобным признакам можно отнести:

- имя процесса, вызвавшего нештатную ситуацию;
- имя и версию приложения, частью которого является процесс;
- трассу вызова функций, приведших к аварийной ситуации;
- значение переменных среды, которые могли повлиять на ход работы приложения.

Какие именно признаки учитывать – зависит от каждого конкретного случая. Например, в случае ошибок ядра имя приложения всегда одинаково, а вот версия может иметь значение.

Отметим, что все перечисленные признаки являются строками либо наборами строк. В качестве уникального идентификатора проблемы в базе знаний можно использовать хэш-сумму всех признаков, характеризующих проблему.

Решение проблемы также должно быть представлено в виде, пригодном для автоматического применения. Наиболее простым подходом к этому является оформление решения в виде исполнимого файла, который достаточно запустить на целевой системе.

При наличии формализованной базы знаний автоматизированные средства обнаружения нештатных ситуаций могут использовать ее для поиска и применения решения возникшего инцидента. Доступ к базе может осуществляться как удаленно через Интернет, так и посредством использования локальной и регулярно обновляемой копии.

В любом случае особое внимание должно быть уделено проверке подлинности базы – ведь решение проблемы подразумевает выполнение ряда действий на целевой системе и компрометация базы знаний может быть использована злоумышленниками с целью выполнения вредоносного кода.

7. Жизненный цикл ошибки в системах Virtuozzo

Флагманские продукты компании Virtuozzo – Virtuozzo 7 и Virtuozzo Infrastructure Platform – являются типичными представителями гиперконвергентных систем, предоставляющих конечным пользователям виртуальные машины и распределенное отказоустойчивое хранилище для размещения данных этих ВМ.

Для обнаружения и обработки нештатных ситуаций в указанных продуктах используется связка программ ABRT и libreport, которая применяется во многих дистрибутивах Linux, использующих для управления ПО инструментарий RPM. ABRT предоставляет набор средств для обнаружения определенных видов проблем. libreport дополняет их средствами составления отчетов об обнаруженных проблемах для их последующего анализа, а также доставки этих отчетов.

При выявлении нештатной ситуации ABRT в совокупности с libreport собирают сведения, потенциально полезные для анализа (имя программы, ее опции, снимок памяти и прочее), и оповещают о них заинтересованных лиц – например, системных администраторов или разработчиков.

Связка ABRT и libreport хорошо зарекомендовала себя во многих дистрибутивах Linux, однако потребовала доработки для эффективного использования в решениях Virtuozzo. В частности, уже добавлены либо планируются к реализации следующие функции, описанные в данной статье:

- сбор необходимых данных не только с машины, где обнаружена проблема, но и со всего кластера;
- обнаружения проблем в гостевых ОС виртуальных машин, которые могут быть вызваны ошибками виртуализации;
- анализ ошибок подсистемы виртуализации, которые могли быть вызваны гостевой ОС;
- динамический выбор информации, помещаемой в журнал об ошибке, в зависимости от деталей инцидента (например, включение журналов тех или иных сервисов в зависимости от того, в каком компоненте возник инцидент);
- использование единого для кластера перечня уже случавшихся ошибок с целью определения дубликатов.

Кроме того, рассматривается возможность реализации «обратной связи», когда приемник отчетов не просто получает сведения об ошибке, но и возвращает список действий, которые можно в автоматическом режиме выполнить на проблемной машине для устранения последствий (например, удалить либо воссоздать необходимые файлы, перезапустить сервис и так далее).

Для реализации функционала обратной связи планируется провести предварительное исследование инцидентов, обнаруженных за время работы инструментов автоматического сбора ошибок. В рамках исследования необходимо выяснить, для каких случаев была необходима процедура восстановления системы после ошибки и в каком проценте ситуаций эта процедура могла бы быть проведена автоматически. Эти данные позволят оценить целесообразность дальнейшей разработки.

Сочетание ABRT и libreport является лишь одним из звеньев цепочки анализа ошибок в Virtuozzo. Полностью процесс обработки проблемы, возникшей в работающей системе, выглядит следующим образом.

1) ABRT обнаруживает возникновение проблемы.
2) libreport совместно с ABRT собирают информацию для отладки. На это стадии также работают дополнительные утилиты Virtuozzo, собирающие сведения, специфичные для продуктов компании.
3) Для обнаруженной проблемы строится идентификатор, который является одним и тем же для одинаковых проблем, возникших на разных машинах или на одной и той же машине в разное время. Например, в случае аварийного останова бинарной программы таким идентификатором может быть хэш-сумма от конкатенации имени упавшего процесса и стека вызовов функций, приведших к падению.
4) Идентификатор проблемы отправляется на сервер-приемник отчетов, который смотрит, приходили ли уже отчеты о проблемах с таким идентификатором или нет.
5) Если подобные отчеты уже были, то приемник отчетов на своей стороне отмечает появление еще одной машины, на которой возникла проблема.
6) Если отчет новый, то на приемник дополнительно отсылается вся информация, собранная на шаге 2. Идентификатор проблемы запоминается в базе данных об ошибках, а информация о проблеме передается на следующий шаг – автоматическому анализатору отчетов.
7) Анализатор строит уточненный отчет об ошибке с использованием данных, не доступных на пользовательских машинах. Например, для бинарных программ на основе отладочной информации и исходных кодов строятся трассы вызовов функций с указанием значения параметров и с привязкой к коду. Построить такой отчет на стороне пользователя невозможно ввиду отсутствия доступа к исходному коду приложений.
8) На основе уточненного отчета автоматически оформляется сообщение об инциденте в системе контроля ошибок.

Таким образом, разработчики получают сообщения об ошибках в привычной им среде и с данными, достаточными для первичного анализа проблемы. Для более глубокого изучения разработчикам предоставляется сервис, создающий контейнер с окружением, максимально приближенным к тому, где возникла проблема (в частности, с точно совпадающим набором установленных приложений и их версий при условии, что в системе не устанавливались сторонние компоненты). Такой контейнер может быть использован для воспроизведения ошибки и для отладки программы.

Указанная схема используется как для мониторинга ошибок на реальных системах клиентов компании, так и на тестовых установках при проведении различного рода проверок. За 1 год использования была накоплена следующая статистика:

- среднее количество отчетов в день: 25000 (99.9% из них поступает с тестовых стендов);
- средний размер отчета (в сжатом виде): 100 Мб;
- общий размер накопленных отчетов (без учета сообщений, признанных дубликатами и не сохраненных в базе): 2,5 Тб
- сообщений об ошибках, автоматически созданных на основе отчетов в системе учета ошибок: 3000 (что составляет ~10% от общего количества инцидентов, заведенных в системе за тот же период);
- исправлено ошибок в продуктах: 400 (~5% от общего количества исправлений, сделанных на основе сообщений об ошибках).

На основе собранной статистики был сделан вывод о целесообразности дальнейшего использования и развития инструментария. Одним из основных направлений развития представляется более скрупулезный выбор информации, добавляемый в отчет об ошибке. В частности, предлагается для каждого ключевого приложения составить свой перечень данных, которые необходимо собрать при возникновении в нем ошибки. Также возможны дальнейшие улучшения определения дубликатов – например, за счет использования машинного обучения.

8. Заключение

С ростом сложности программного обеспечения растет и сложность анализа возникающих в нем ошибок. Упростить анализ можно путем его автоматизации – в частности, посредством автоматического сбора данных, необходимых для изучения проблемы.

Для получения наиболее полной информации сбор данных необходимо проводить непосредственно после возникновения инцидента. В настоящее время уже существуют хорошо зарекомендовавшие себя подходы и инструментальные средства для решения этой задачи, но большинство из них нацелено на работу в рамках одной операционной системы на одном компьютере.

В то же время современные тенденции состоят в развитии гиперконвергентных систем, объединяющих множество физических машин в единое целое и предоставляющей конечным пользователям виртуализированные ресурсы. В рамках таких систем ошибка на одной машине может быть вызвана некорректными действиями других частей системы. Поэтому для эффективного использования в гиперконвергентных средах имеющиеся решения должны быть доработаны с учетом особенностей этих сред.

В данной статье мы предложили ряд таких усовершенствований, большинство из которых уже реализовано нами в продуктах Virtuozzo 7 и Virtuozzo Infrastructure Platform. За более чем годовой период использования реализованные инструментальные средства доказали свою эффективность, позволив выявить и исправить заметное количество ошибок, не перегружая при этом разработчиков ложными срабатываниями и дублирующимися сообщениями.

Список литературы / References

[1]. Doleželová M., Muehlfeld M. et al. Automatic Bug Reporting Tool (ABRT). Deployment, Configuration, and Administration of Red Hat Enterprise Linux 7. Chapter 25 (online). Available at: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/system_administrators_guide/ch-abrt.

[2]. Apport. Ubuntu Wiki (online). Доступно по ссылке: https://wiki.ubuntu.com/Apport

[3]. How to set process core file names. Red Hat Customer Portal (online). Available at: https://access.redhat.com/solutions/901293

[4]. Силаков Д.В. Открытое решение Graylog. Сбор и анализ событий в сетях промышленных масштабов. Системный администратор, № 3, 2019 г., стр. 24-29 / Silakov D.V. Open Graylog Solution. Collection and analysis of events in networks of industrial scale. System Administrator, № 3, 2019, pp. 24-29 (in Russian).

[5]. Du Min, Li Feifei, Zheng Guineng, and Srikumar Vivek. DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning. In Proc. of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017, pp. 1285-1298.

[6]. P. Garcia-Teodoro, J. Diaz-Verdejo, G. Macia-Fernandez, and E. Vazquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. Computers & Security, vol. 28, issues 1-2, 2009, pp. 18–28.

[7]. K.K. Sabor, A. Hamou-Lhadj, and A. Larsson. DURFEX: A Feature Extraction Technique for Efficient Detection of Duplicate Bug Reports. In Proc. of the 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), 2017, pp. 240-250.

[8]. R.P. Gopalan and A. Krishna. Duplicate Bug Report Detection Using Clustering. In Proc. of the 2014 23rd Australian Software Engineering Conference, 2014, pp. 104-109.

Информация об авторе / Information about the author

Денис Владимирович СИЛАКОВ – кандидат физико-математических наук, старший системный архитектор компании Virtuozzo, отвечает за автоматизацию процессов разработки и сопровождения продуктов компании. Сфера научных интересов: автоматизация тестирования, управление требованиями, автоматизация разработки ПО.

Denis Vladimirovich SILAKOV – Ph.D. in Physical and Mathematical Sciences, Senior system architect in the Virtuozzo company, responsible for automation of development and maintanence of company products. Research interests: test automation, requirements, software development automation.

DOI: 10.15514/ISPRAS-2019-31(4)-3

Designing robust quadcopter software based on a real-time partitioned operating system and formal verification techniques

S.M. Staroletov, ORCID: 0000-0001-5183-9736 <serg_soft@mail.ru>
M.S. Amosov, ORCID: 0000-0002-2056-2794 <faystmax@gmail.com>
K.M. Shulga, ORCID: 0000-0003-1422-4681 <kirsh.ru@yandex.ru>
Polzunov Altai State Technical University,
Lenin avenue 46, Barnaul, 656038, Russia

Abstract. Currently, the creation of reliable unmanned aerial vehicles (drones) is an important task in science and technology because such devices can have a lot of use-cases in digital economy and modern life, so we need to ensure their reliability. In this article, we propose to assemble a quadcopter from low-cost components in order to obtain a hardware prototype, and also to develop a software solution for the flight controller with high-reliability requirements, which will meet avionics software standards, using existing open-source software solutions. We apply the results as a model for teaching courses «Components of operating systems» and «Software verification». In the study, we analyze the structure of quadcopters and flight controllers for them, and present a self-assembly solution. We describe Ardupilot as open-source software for unmanned aerial vehicles, the appropriate APM controller and methods of PID control. Today's avionics standard of reliable software for flight controllers is a real-time partitioned operating system that is capable to respond to events from devices with an expected speed, as well as to share processor time and memory between isolated partitions. A good example of such OS is the open-source POK (Partitioned Operating Kernel). In its repository, it contains an example design of a system for a quadcopter using AADL language for modeling its hardware and software. We apply such a technique with Model-driven engineering to a demo system that runs on real hardware and contains a flight management process with PID control as a partitioned process. Using a partitioned OS brings the reliability of flight system software to the next level. To increase the level of control logic correctness we propose to use formal verification methods. We also provide examples of verifiable properties at the level of code using the deductive approach as well as at the level of the cyber-physical system using Differential dynamic logic to prove the stability.

Keywords: quadcopter; partitioned OS; ARINC 653; formal verification; Cyber-physical system

For citation: Staroletov S.M., Amosov M.S., Shulga K.M. Designing robust quadcopter software based on a real-time partitioned operating system and formal verification techniques. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 4, 2019. pp. 39-60. DOI: 10.15514/ISPRAS-2019-31(4)-3

Разработка программного обеспечения квадрокоптера с повышенными требованиями к надёжности на основе партицированной ОС и технологий формальной верификации

С.М. Старолетов, ORCID: 0000-0001-5183-9736 <serg_soft@mail.ru>
М.С. Амосов, ORCID: 0000-0002-2056-2794 <faystmax@gmail.com>
К.М. Шульга, ORCID: 0000-0003-1422-4681 <kirsh.ru@yandex.ru>
Алтайский государственный технический университет им. И.И. Ползунова,
656038 Барнаул, проспект Ленина, 46

Аннотация. Создание надежных беспилотных летательных аппаратов является важной задачей для науки и техники, потому что необходимо обеспечивать надежность таких решений. В этой статье предлагается использование аппаратного прототипа квадрокоптера и разработка программного решения для полетного контроллера с высокими требованиями к надежности, которое будет соответствовать новым стандартам для программного обеспечения авионики и будет использовать существующие программные решения с открытым исходным кодом. В ходе исследования мы анализируем состав квадрокоптеров и полетных контроллеров для них. Мы описываем открытое программное обеспечение Ardupilot для беспилотных летательных аппаратов, контроллер APM и методы ПИД-регулирования. Сегодняшним стандартом надежного программного обеспечения для бортовых контроллеров являются партицированные операционные системы реального времени, которые способны реагировать на события от оборудования с ожидаемой скоростью, а также разделять процессорное время и память между изолированными разделами. Хорошим примером такой ОС с открытым исходным кодом является POK (Partitioned Operating Kernel). В репозитории она содержит пример описания системы для дронов с использованием языка AADL с моделированием аппаратного и программного обеспечения решения. Мы применяем такую технику к демонстрационной системе, которая работает на реальном оборудовании и содержит процесс управления полетом с PID-регулятором в виде изолированного процесса. Использование партицированной ОС выводит надежность программного обеспечения полетного контроллера на новый уровень. Для того, чтобы повысить уровень корректности логики управления, мы предлагаем использовать формальные методы верификации и демонстрируем примеры проверяемых свойств на уровне кода, используя дедуктивный подход, а также проводя моделирование на уровне киберфизической системы с использованием динамической дифференциальной логики для доказательства устойчивости.

Ключевые слова: квадрокоптер; операционная система; партицирование; ARINC 653; формальная верификация

Для цитирования: Старолетов С.М., Амосов М.С., Шульга К.М. Разработка программного обеспечения квадрокоптера с повышенными требованиями надёжности на основе партицированной ОС и технологий формальной верификации. Труды ИСП РАН, том 31, вып. 4, 2019 г., стр. 39-60 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(4)-3

1. Introduction

Unmanned aerial vehicles (UAVs) also called drones are becoming a big part of our digital life. In this paper, we consider *quadcopters* – vehicles with four software-controllable motors. They can be used for taking nice videos, for delivering parcels from Internet stores to end-customer by air, and some people even proposed to use them on the FIFA World Cup to deliver the balls.

However, in some countries, the use of UAVs is prohibited in public places, mostly because of fears of a poorly functioning machine falling from the air onto people. An example of last year's public testing of Russian Post delivering drone, which crashed after a few seconds from the start, shows us that we need to build highly reliable software for it, which should prevent the drone from failure and fall in the most of cases (breakages of hardware or software, incorrect and contradictory commands from a pilot).

Also, we should take in mind that a drone is a representation of a normal air vehicle. Most of the technologies to build the quadcopter, to create the software for it, to control safety and liveness

properties, to model physical properties of flight are the same as for big air vehicles, so it could be a cheap model to design of highly reliable aircraft.

The scope of the work includes the following.

- Assembling a flying quadcopter for the purpose of obtaining a hardware prototype (from components that are mass-marketed);
- Developing a flight controller software solution with high-reliability requirements, which will coincide with the standards of avionics software (to some extent);
- Obtaining a model for teaching courses on the design of operating system components and software verification.

The first task covers selection of existing hardware components including passive (i.e., frame, battery, etc.) and active parts (their state could be read and changed by software, i.e., GPS sensor, accelerometer, controllable motors, etc.) with maximum compatibility; and selection of a basic flight controller, which manages the whole hardware system; we are going to write software for it.

The second task is devoted to creating an operating system for the drone, which will be based on ARINC 653 [1] – an industrial standard interface of applied software for avionics and existing open-source solutions in a free code for flight controllers to produce control logic and interoperation with the hardware.

And the last task is to apply the project achievements in the study process of future software engineers to improve the interest of students in system programming and to offer different tasks as lab works.

2. Related work

The task of building reliable software for UAVs (and also for aircraft) is strongly connected to developing an operating system that is robust according to the initial design. Today's OS for flight machines should be real-time and should offer memory space and time division capabilities. There exists an avionics industry standard for these requirements, created by the Aeronautical Radio, Inc., ARINC 653 [1]. BSD-licensed open-source POK OS, which satisfies this standard with some limitations, was created in France by Julien Delange [2]. It uses Model-Based Engineering approach [3, 4] for describing the system configuration, and its source code is available in [5].

In Russia, JetOS, a certified operating system for aircraft, was created by GosNIIAS and ISP RAS [6, 7] as a fork of POK with advanced debugging capabilities, rewritten scheduler, system partition feature and different platforms support. The code is partially available on GitHub under the GPLv3 licence [8].

Building robust software encompasses not only the proper design of the OS for the flight controller. It should also include formal verification of developed code and proof of software correctness to satisfy some requirements. ISP RAS is the leader in this field in Russia, the recent results were published in [9, 10, 11]. Their principal result is producing testing and deductive verification techniques and an adequate memory model, mainly used in checking of correctness of Linux modules.

The task of building software and hardware for a drone is being solved in project Crazyflie [12] and the models are given in [13]. It is used in some universities, e.g., at Chalmers University of Technology where students build dynamic models for Crazyflie drone using the modeling language Modelica and then design control algorithms based on them [14].

3. The drone: components and terminology

Consider a drone that is assembled from scratch based on components available on the market. It consists of the following components (see fig. 1):

- **frame** connects all the components and provides electrical routing of high-current energy to the motors (here we use a 450mm frame);

- **brushless motors** (4x for a quadcopter) provide lifting force to the propellers, which allows the drone to fly or rotate;
- **ESC** (Electronic Speed Controller) provides a high-level interface to control a corresponding motor, translates required RPM to electric voltage and controls the results of motors rotation;
- **battery** (usually of the LiPo type) provides electricity to the drone, and the time of drone operation depends on it; a 11.1V, 3000mAh 3S battery provides about 10-15 mins of operation and requires a special balancing charger;
- **power module** connected to the battery provides two electrical circuits: a low-current circuit to the flight controller and a high-current one to the motors, also it provides a controlling signal to the flight controller with the current state of the battery;
- external **GPS/compass** module is used for getting current coordinates to use in waypoint algorithms, it should be separated from the flight controller and frame to minimize the effects of electrical noise;
- **telemetry** is a radio transceiver that works at the frequency of 915Mhz and allows to make a channel to a ground control module to send the current state and receive commands by providing radio-transparent UART (communication port);
- **additional periphery** is the useful load of a drone, in the simplest case it may be a camera, in more complex cases it may be for example a fire extinguishing device; such load can be operated by an additional controller or even by the main flight controller with special isolated processes;
- **flight controller** is responsible for controlling the whole drone state (the states of all internal hardware components) and operating it like the car ECU; it has some sensors on-chip and software for flying either with operator's control and stand-alone. Our goal is to create reliable software for it.

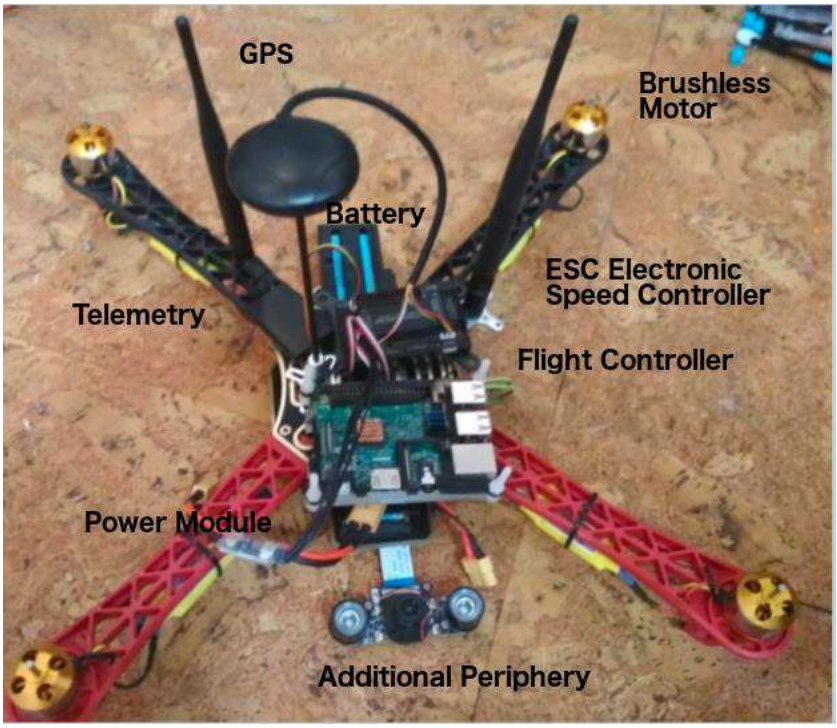


Fig. 1. The parts of the quadcopter

Consider a question, how does the quadcopter fly? According to [15], in the stable state the drone has following properties (fig. 2):

- the equivalence of forces: $\sum T_i = -mg$;
- the equivalence of moments: $\sum M_i = 0$;
- the equivalence of directions: $T_{1,2,3,4} || g$;
- the sum of rotation speeds: $(w_1 + w_3) - (w_2 + w_4) = 0$.

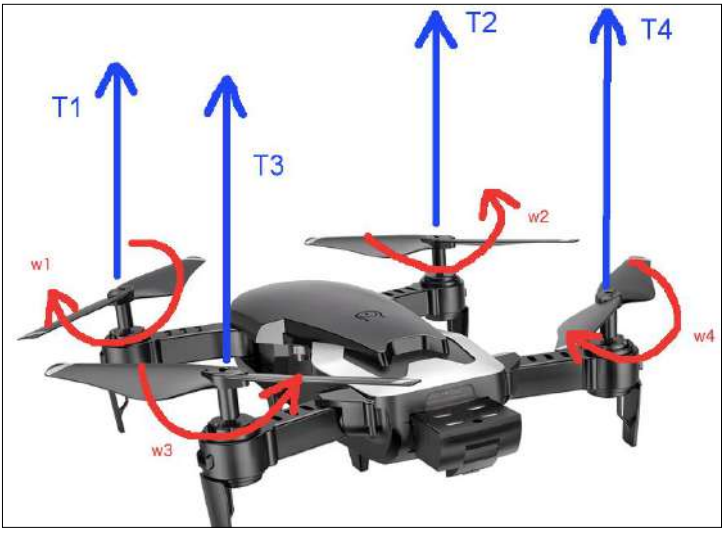


Fig. 2. Quadcopter flight state

Every violation of the properties leads to move of the drone among one of three axes, and so-called Euler's angles are defined (fig. 3):

- pitch;
- roll;
- yaw.

Fig. 3. Quadcopter angles

Movement or rotation of the quadcopter is done by means of different rotation speeds of its propellers controlled by the flight controller.

4. Today's software solution

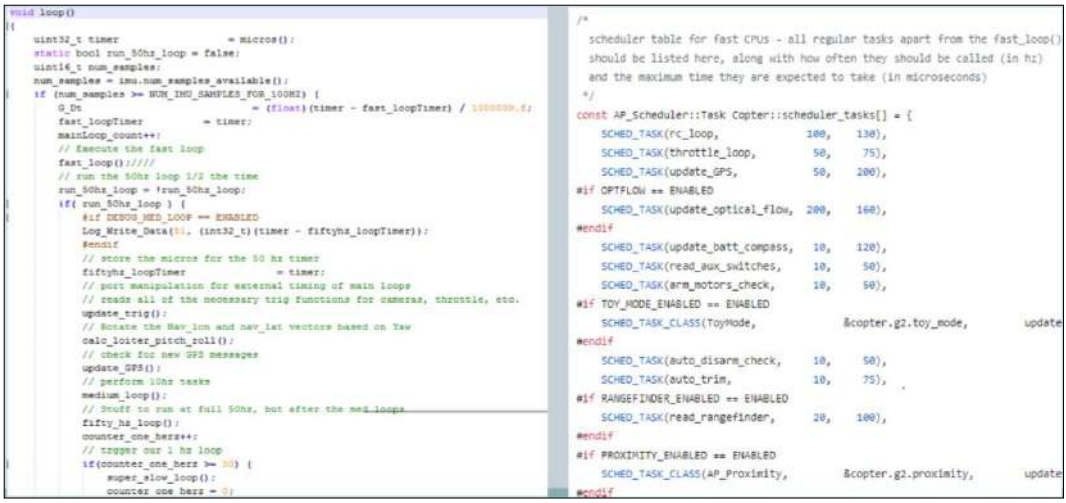
4.1 Ardupilot / Adrucopter

Ardupilot project [16, 17] is the most known open-source project to provide free software to control the drones with its Arducopter codebase (as well as rovers and planes, refer to Ardurover and Arduplane codebases respectively) and high-level abstraction to operate hardware components for them. The source code of control is based on the recent results of the UAV research community, so it is a good base for tracking recent achievements. Inside, it consists of HAL (Hardware Abstraction Layer) that allows to run the system on different hardware platforms, main scheduling loop and a set of different libraries that provide some code to communicate with a particular device (get and store data), mathematical calculations, control algorithms. We are mostly interested in the following libraries:

- AC_PID: a library to implement an algorithm of PID-control;
- AC_AttitudeControl: attitude position control of a drone using PID-control algorithms;
- AC_WPNav: pre-defined trajectory flight using waypoints and PID-control.
- APM_Control: stabilisation on the pitch, roll, yaw axes;
- AP_Math: a module with internal mathematical routines;
- AP_Techs: combined energy, speed and altitude control;
- AP_Arming: check the preconditions of equipment performance before starting drone control;
- AP_Compas: a module to work with compass hardware;
- AP_Baro: a module to work with barometer hardware;
- AP_BattMonitor: a module of work with the battery controller, including the detection of its discharge;
- AP_GPS: a module to work with GPS hardware;
- AP_Motors: a module to work with drone's motors, supporting statuses, control and their testing;
- AP_Frsky_Telem: a module to work with the radio telemetry;
- GCC_MAVLink: support for MAVLink telemetry transmission protocol.

These libraries allow to develop a custom software solution for a drone based on hardware described above.

Consider the main working procedure in the flight controller in Ardupilot software. It has to get current hardware state using HAL-abstraction and libraries to work with devices, then construct a high-level state of the drone (for example, the orientation of it in the space), run algorithms of control based on the current mode, and after then apply some commands to hardware. Due to various types of devices and different protocols of communication (for example, SPI or UART), the code should use a different speed for devices polling or priorities. Fig. 4 shows an evolution of Ardupilot software depending of properties of corresponding hardware: up to branch 3.1 [17], Ardupilot used loops with different frequencies in the main loop code (some of them executed at each iteration, some – at each two iterations, etc.), then the code was upgraded to use the concept of tasks with priorities (depending on hardware platform it can be executed like loops of early versions or run inside processes of a real-time OS).



```
void loop()
{
    uint32_t timer              = micros();
    static bool run_50hz_loop = false;
    uint16_t num_samples;
    num_samples = imu.num_samples_available();
    if (num_samples >= MIN_IMU_SAMPLES_FOR_100HZ) {
        G_Dt                    = (float)(timer - fast_loopTimer) / 1000000.f;
        fast_loopTimer          = timer;
        mainLoop_count++;
        // Execute the fast loop
        fast_loop();////
        // run the 50hz loop 1/2 the time
        run_50hz_loop = !run_50hz_loop;
        if( run_50hz_loop ) {
            #if DEBUG_MED_LOOP == ENABLED
            Log_Write_Data(51, (int32_t)(timer - fiftyhz_loopTimer));
            #endif
            // store the micros for the 50 hz timer
            fiftyhz_loopTimer           = timer;
            // port manipulation for external timing of main loops
            // reads all of the necessary trig functions for cameras, throttle, etc.
            update_trig();
            // Rotate the Nav_lon and nav_lat vectors based on Yaw
            calc_loiter_pitch_roll();
            // check for new GPS messages
            update_GPS();
            // perform 10hz tasks
            medium_loop();
            // Stuff to run at full 50hz, but after the med_loops
            fifty_hz_loop();
            counter_one_herz++;
            // trigger our 1 hz loop
            if(counter_one_herz >= 50) {
                super_slow_loop();
                counter_one_herz = 0;
```

```
/*
  scheduler table for fast CPUs - all regular tasks apart from the fast_loop()
  should be listed here, along with how often they should be called (in hz)
  and the maximum time they are expected to take (in microseconds)
 */
const AP_Scheduler::Task Copter::scheduler_tasks[] = {
    SCHED_TASK(rc_loop,              100,    130),
    SCHED_TASK(throttle_loop,         50,     75),
    SCHED_TASK(update_GPS,            50,    200),
#if OPTFLOW == ENABLED
    SCHED_TASK(update_optical_flow,  200,    160),
#endif
    SCHED_TASK(update_batt_compass,   10,    120),
    SCHED_TASK(read_aux_switches,     10,     50),
    SCHED_TASK(arm_motors_check,      10,     50),
#if TOY_MODE_ENABLED == ENABLED
    SCHED_TASK_CLASS(ToyMode,              &copter.g2.toy_mode,         update
#endif
    SCHED_TASK(auto_disarm_check,     10,     50),
    SCHED_TASK(auto_trim,             10,     75),
#if RANGEFINDER_ENABLED == ENABLED
    SCHED_TASK(read_rangefinder,      20,    100),
#endif
#if PROXIMITY_ENABLED == ENABLED
    SCHED_TASK_CLASS(AP_Proximity,         &copter.g2.proximity,        update
#endif
```

Fig. 4. Evolution of the main loop code of flight controller: from simple loops with given frequency (left) to RT OS with scheduling tasks (right)

In the beginning, Ardupilot / Arducopter software was designed to support APM controller hardware (APM stands for ArduPilot Mega and Ardu stands for Arduino), and the project goal was to use Arduino Mega board as a flight controller. Later, the code was abstracted from hardware and now it can be used in open-source platforms (APM, Pixhawk, Pixhack, see fig. 5) as well in proprietary solutions (Intel Aero, SnapDragon).

Actually, controllers may use the following types of system software:

- firmware without OS;
- real-time OS;
- Linux.

Fig. 5. Ardupilot/Arducopter compatible hardware on the market

The difference of controllers is based on their purposes: for amateur or commercial use, operate with slow or high speed, the necessity of having additional logic that implies using special sensors. Modern flight controllers act like ECU (Electronic Control Unit) for cars and even contain CAN (Controller Area Network) bus for the periphery. Some controllers also include a «safety co-processor» [18] to ensure the robustness of the flight state.

Note that when we move from the simple loop with device polling to real-time scheduling algorithms, we should take in account that process switching can add some delays to normal handling of real physical process control data, so such algorithms must be formally verified.

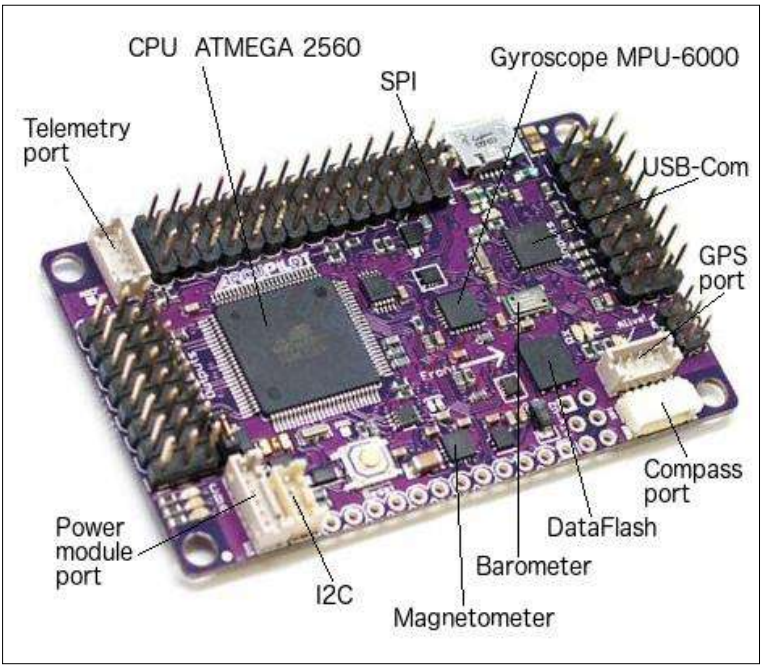


Fig. 6. Components of APM 2.6

The APM controller was built in 2011-2012 [18] as a result of initial Ardupilot project development by 3D Robotics company. Later, the non-profit organisation ardupilot.org was established. Today the controller is slightly outdated but it is cheap and available on market. The components of the controller are shown in fig.6:

- ATMEGA 2560 processor (8bit);
- barometer MS5611, SPI connection;
- 3-axes magnetometer (compass) HMC5843, I2C connection;
- 6-axes gyroscope and accelerometer MPU-6000, SPI connection;
- GPS: UART external interface;
- FrSky telemetry: UART external interface;
- external SPI interface.

In our current research work, we learned to work with this hardware and provided a transfer of necessary data to another controller to execute control algorithms. For us following control algorithms are most interesting:

- altitude (position) control;
- waypoint flight or curve-based flight;
- smoothing pilot's commands.

All algorithms are based on the PID-controller approach based on the Control Theory.

4.2 PID control

The PID (Proportional, Integral, Differential) controller is a feedback system for correcting the state of the control object (fig. 7). When controlling the object we calculate an error between current and desired state (for example, between current and set drone altitude), then based on current error we calculate the impact based on three parts with given coefficients.

- The proportional part (P) is responsible for the proportional reduction of the error (present).
- The integral part (I) is a statistical change of the error (past).
- The differential part (D) is the change of the error, its tendency to 0 (future).

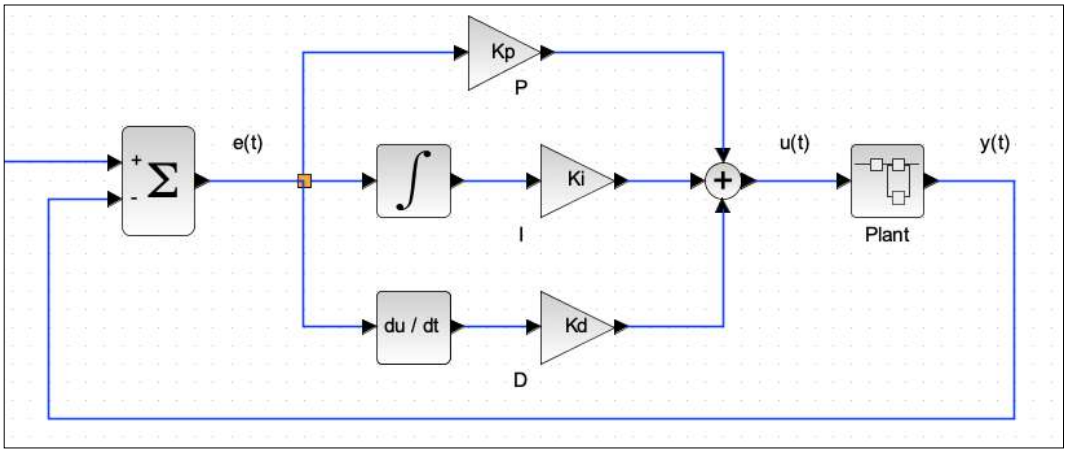


Fig. 7. PID-control scheme.

PID-control is some kind of abstraction when the control process is primarily based on the difference (error) but not on attempts to describe the exact physical model of the system, because during flight a variable wind can blow, the rotation of the propellers may be unstable, the center of mass is shifted and so on, but in such situations PID controller will detect the deviation and will try to make an impact to change it.

The analytic equation for the PID-control scheme:

$$u(t) = P + I + D = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de}{dt} \quad (1)$$

where K_p, K_i and K_d are the PID coefficients.

Presently, the coefficients are set up in the user interface of ground control software, for example, using Mission Planner [20]. Some guidance on coefficients' choice exists, and their values are known for standard drone assemblies but for custom quadcopter (for example, a heavyweight one) they should be identified as a result of experiments.

If a value of some coefficient is zero the controlled system may lose stability or smoothness of behaviour and the controller can be called P-controller, PD-controller or PI-controller.

4.3 PID control implementation in Ardupilot

According to [21], the current implementation of Ardupilot/Arducopter for altitude control uses a combined scheme of P- and PID-controller. The orientation for each axis is controlled with a special P-controller to convert the angle error (the difference between the specified angle and the actual one) to the desired rotational speed. Then a PID-controller converts the rotational speed error to a high-level motor command. A special part of the P-controller – «square root controller» – at first represents the angle function linearly and then uses its square root approximation. [22] provides a discussion of the overall control scheme. This scheme and its current refactoring are shown in fig. 8.

The scheme demonstrates that all the axes have an influence on each other and the construction of the whole analytical equation like (1) is a very challenging task. Also, it shows some input transformation blocks that help to control smoothly [22].

- User's actions to control the drone (for example, commands to move it) are not translated to drone motors directly because they are not regular, may be conflicting and have different

duration. They are buffered and divided into small pieces of actions, and an internal scheduler sends them with some constant frequency.

- Based on the collected set of user's actions and parameters, a final action may be calculated in advance and then posted to the scheduler.

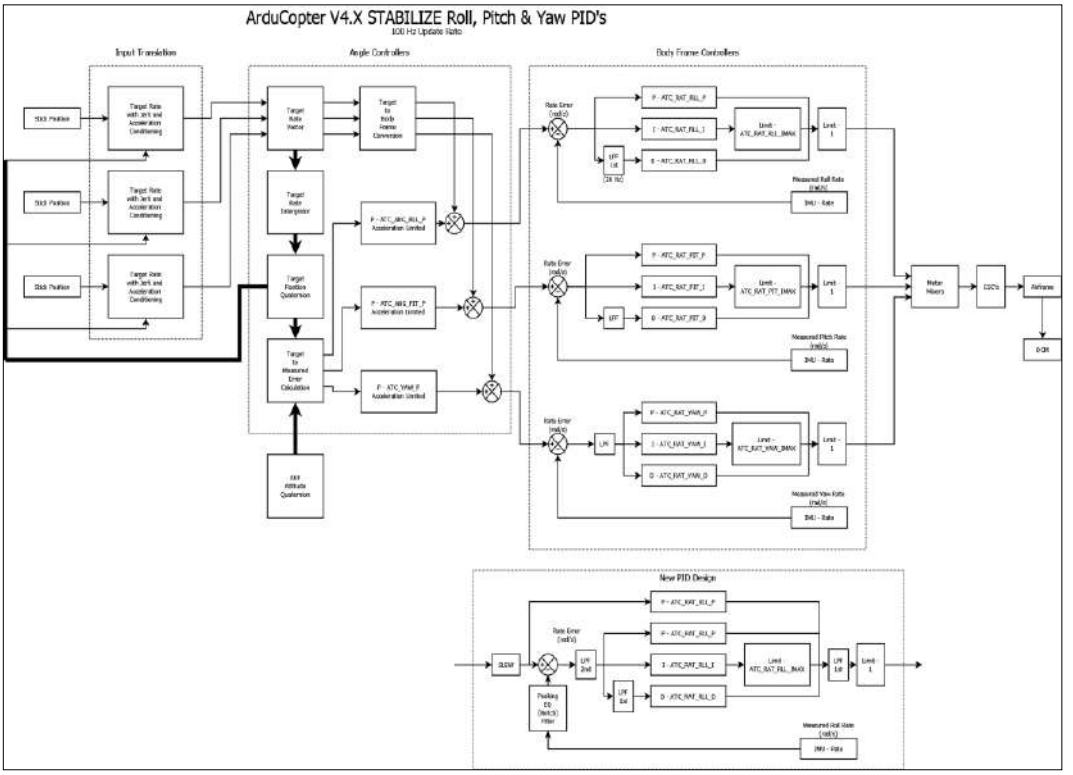


Fig. 8. Overall PID-control scheme (taken from the ArduCopter project)

To provide flight along a given trajectory it is necessary to carry out maneuvers in advance, otherwise the drone will have to inefficiently turn in sharp corners. The approach of constructing smooth curves for unmanned air vehicles is called L1 [23]. For drones, this method was changed because of other physical characteristics and implemented in the AC_WPNav library.

5. Towards a partitioned Ardupilot code

5.1 POK

POK (Partitioned Operating System Kernel) is a name of OS created in France by Julien Delange during his PhD research.

The main features of this work are the following:

- MDE (Model-Driven Engineering) approach: initial OS kernel configuration was defined with the AADL language, which allowed to generate code and represent the configuration graphically;
- it is a good proof-of-concept of working models and examples;
- the system partially conforms to the ARINC 653 standard for real-time onboard aviation systems;
- the system uses protected partitions with time and memory space resources isolation;

- two types of real-time processes schedulers with different strategies exist – the partition planner and the process planner in each partition;
- controllable message exchange between processes or using BlackBoard concept.

The use of OS designed according to avionics standards and providing isolation of processes and verifiable interprocess communications increases the robustness of the solution at the system level. By browsing source code [5], we have created a scheme of internal POK architecture (see fig. 9). It consists of three layers: Arch with platform-dependent code (open-source repository includes implementations for three platforms: x86-qemu, PowerPC, and Spark), Core as the internal code of kernel and syscalls, and libpok that can be used as an API.

The ARINC 653-compatible API provides a possibility to work with partitions, processes, locks, ports, queries, and messages in a standardized way.

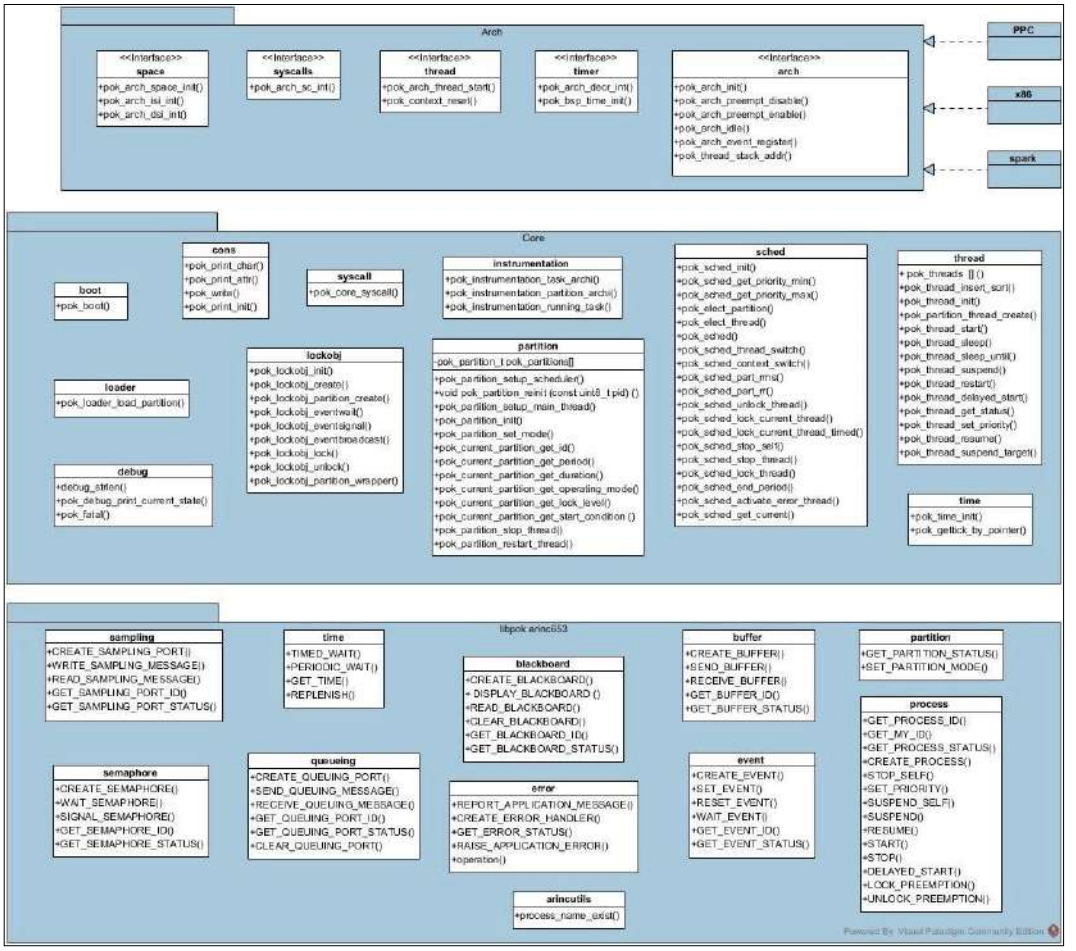


Fig. 9. Overall scheme of POK architecture

```
processor ATMEGA2560 extends ATMEGA328
properties
  Data_Sheet::UUID
     => "https://ww1.microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-
         microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf";
  Processor_Properties::Processor_Frequency => 16 Mhz;
end ATMEGA2560;

processor implementation ATMEGA2560.impl extends ATMEGA8.impl
properties
  Memory_Size => 262_144 Bytes applies to Flash_Memory;
  Memory_Size =>  8_192 Bytes applies to SRAM;
  Memory_Size =>  4_096 Bytes applies to EEPROM;
end ATMEGA2560.impl;
```

Fig. 10. Defining a simple processor model as an extension in AADL

5.2 AADL

AADL (Architecture Analysis & Design Language) [24] provides system modeling engineers with following capabilities:

- to develop a top-level system design;
- to think in abstractions and then write implementations;
- to see a graphical representation of the code;
- to generate code using integrated tools such as Ocarina;
- to validate properties of developed systems automatically;
- to create certified solutions.

In essence, AADL acts like «executable UML» but at the top level and it is designed not for expressing algorithms, but for describing systems with additional safety requirements.

The language allows to describe systems and components as sets of extensible properties to model both hardware and software parts. For example, if we need to define a model for the APM board (already introduced in fig. 6) we may first define a model for ATMEGA 2560 processor as an extension of existing processor ATMEGA 328 (fig. 10) with another frequency and memory sizes and then use this model in a board model definition (fig. 11) additionally specifying connections by ports with use of SPI buses.

In our work, AADL is used to model the whole structure of the partitioned OS for the flight controller with the possibility to generate and validate its kernel configuration.

```
system implementation Ardupilot.impl
    Subcomponents
    ATM2560         : processor Processors::ATMEGA::ATMEGA2560.impl
    {
    Deployment::Execution_Platform => Native;
    Scheduling_Protocol => (RMS);
    Thread_Limit        => 4;
    };
    SPI5            : device devices::spi::spi_pins;
    SPI6            : device devices::spi::spi_pins;
    connections
    -- See https://docs.google.com/spreadsheets/d/1Jq6nc5VG22dpqr7eraIv_TtWPPhRgmtyl8KQxbI3lF8
    -- for more details for this mapping

    -- interface SPI5=JP5 / ATM2560
    A1 : port ATM2560.PB4 -> SPI5.MISO;
    A2 : port ATM2560.PB5 -> SPI5.SCLK;
    A3 : port ATM2560.PC6 -> SPI5.RST;
    A4 : port SPI5.MOSI  -> ATM2560.PB3;
```

Fig. 11. Defining an implementation of APM board with a processor and SPI buses in AADL

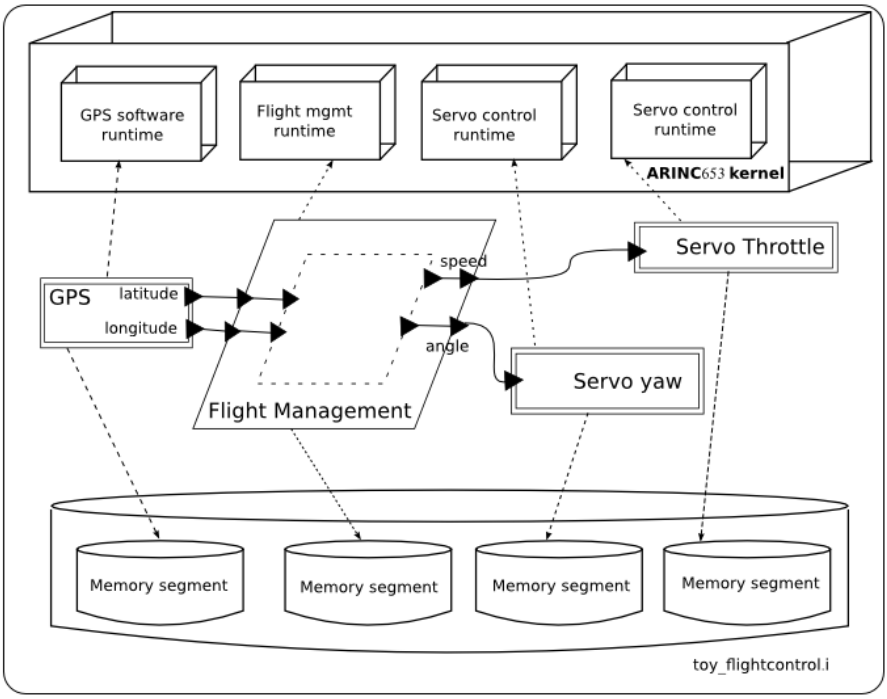


Fig. 12. Flight controller case study in a POK way [25]

5.3 Flight controller system in a POK way

In this subsection, we define a structure of real-time OS for the flight controller with increased reliability requirements using POK way. In [25], POK authors give some tips on such structure definition (see fig. 12).

```
system implementation ardupilot.i
subcomponents
   kernel         : processor poklib::pok_kernel.demo_four_partitions;
   mem            : memory    ardupilot_platform::mem.i;
   prs_gps        : process   ardupilot_software::process_gps.i;
   prs_mgmt       : process   ardupilot_software::process_mgmt.i;
   prs_throttle   : process   ardupilot_software::process_throttle.i;
   prs_yaw        : process   ardupilot_software::process_yaw.i;
connections
  c1: port prs_gps.altitude    -> prs_mgmt.altitude;
  c2: port prs_gps.latitude    -> prs_mgmt.latitude;
  c3: port prs_gps.longitude   -> prs_mgmt.longitude;
  c4: port prs_mgmt.speed      -> prs_throttle.speed;
  c5: port prs_mgmt.angle      -> prs_yaw.angle;
properties
   Actual_Memory_Binding => (reference (mem.segment1)) applies to prs_gps;
   Actual_Memory_Binding => (reference (mem.segment2)) applies to prs_mgmt;
   Actual_Memory_Binding => (reference (mem.segment3)) applies to prs_throttle;
   Actual_Memory_Binding => (reference (mem.segment4)) applies to prs_yaw;

   Actual_Processor_Binding => (reference (kernel.partition1)) applies to prs_gps;
   Actual_Processor_Binding => (reference (kernel.partition2)) applies to prs_mgmt;
   Actual_Processor_Binding => (reference (kernel.partition3)) applies to prs_throttle;
   Actual_Processor_Binding => (reference (kernel.partition4)) applies to prs_yaw;

   POK::Additional_Features => (console, libc_stdio, libc_stdlib) applies to kernel.partition1;
   POK::Additional_Features => (libmath, console, libc_stdio, libc_stdlib) applies to kernel.partition2;
   POK::Additional_Features => (console, libc_stdio, libc_stdlib) applies to kernel.partition3;
   POK::Additional_Features => (console, libc_stdio, libc_stdlib) applies to kernel.partition4;
end ardupilot.i;
```

Fig. 13. System structure definition for flight controller software in AADL [25]

```
process process_mgmt
features
   altitude    : in data port poklib::integer;
   latitude    : in data port poklib::float;
   longitude   : in data port poklib::float;
   speed       : out data port poklib::integer;
   angle       : out data port poklib::integer;
end process_mgmt;

process implementation process_mgmt.i
subcomponents
   thr : thread thr_mgmt.i;
connections
  p1: port altitude     -> thr.altitude;
  p2: port latitude     -> thr.latitude;
  p3: port longitude    -> thr.longitude;
  p4: port thr.speed    -> speed;
  p5: port thr.angle    -> angle;
end process_mgmt.i;

thread thr_mgmt extends poklib::thr_periodic
features
   altitude    : in data port poklib::integer;
   latitude    : in data port poklib::float;
   longitude   : in data port poklib::float;
   speed       : out data port poklib::integer;
   angle       : out data port poklib::integer;
properties
   Initialize_Entrypoint => classifier (ardupilot_software::spg_flt_mgmt_init);
end thr_mgmt;

thread implementation thr_mgmt.i
calls
   call1 : { pspg : subprogram spg_flt_mgmt_simulation;};
connections
   c1: parameter altitude   -> pspg.altitude;
   c2: parameter latitude   -> pspg.latitude;
   c3: parameter longitude  -> pspg.longitude;
   c4: parameter pspg.speed -> speed;
   c5: parameter pspg.angle -> angle;
end thr_mgmt.i;

subprogram spg_flt_mgmt_simulation extends poklib::spg_c
features
   altitude    : in parameter poklib::integer;
   latitude    : in parameter poklib::float;
   longitude   : in parameter poklib::float;
   speed       : out parameter poklib::integer;
   angle       : out parameter poklib::integer;
properties
   Source_Name => "flt_mgmt_simulation";
   Source_Text      => ("../../../flt-mgmt.o");
end spg_flt_mgmt_simulation;

subprogram spg_flt_mgmt_init extends poklib::spg_c
properties
   Source_Name => "flt_mgmt_init";
   Source_Text      => ("../../../flt-mgmt.o");
end spg_flt_mgmt_init;
```

Fig. 14. Linking a process with variables and ports to a thread and a compiled C-code [25]

The scheme is generated from AADL code and represents the partitioning: four partitions and four memory segments for GPS interoperation, runtime support for throttle servo control, and runtime support for yaw and flight management to provide a PID-style control.

Initial AADL code is shown in fig. 13 where *demo_four_partitions* defines a processor with four partitions, a major time frame, time slices for the partitions, and scheduling policy; *partitions 1..4* define partitions with given individual schedulers and additional user libraries; *segments 1..4* define memory segments of given types and sizes; *connections* section defines ports to support interprocess

communication between parts of the system; *prs_gps*, *prs_mgmt*, *prs_throttle* and *prs_yaw* define actually working threads within partitions.

Fig. 14 illustrates the AADL approach to forwarding system variables to processes through ports and then to features in threads inside these processes, as well as to binding those to external object code that will run in these threads. We see that the stabilizing PID controller gets a tuple (altitude, latitude, longitude) and adjusts speed and angle of a quadcopter.

So, configuring OS using this approach provides a Model-Driving approach to design software for the flight controller. The process interoperations are clear and the configuration is verifiable, therefore this increases the reliability of the solution.

5.4 Current state of our solution

To obtain a model for developing OS and testing PID controllers we propose an architecture shown in fig. 15.

We took the 3.2.1 branch of Ardupilot software and modified its code to send current state data of a quadcopter through SPI from the APM controller to a different controller that executes a partitioned OS and provides a flight control based a stabilizing PID controller. Fig.16 shows the current hardware connection.

This solution is the first step to migrate the whole flight controller logic to a fully-partitioned code – we start from using the APM controller as a gateway to quadcopter hardware. The patched code of the APM transfers input data through SPI to a partitioned process (prc_gps in fig.12 and 13) that provides flight-control and sends control data through SPI back to APM. The CRC (cyclic redundancy check) algorithm is used to ensure the correctness of the transaction. We plan to add partitions with additional monitoring processes to ensure dynamic properties of safety and stability.

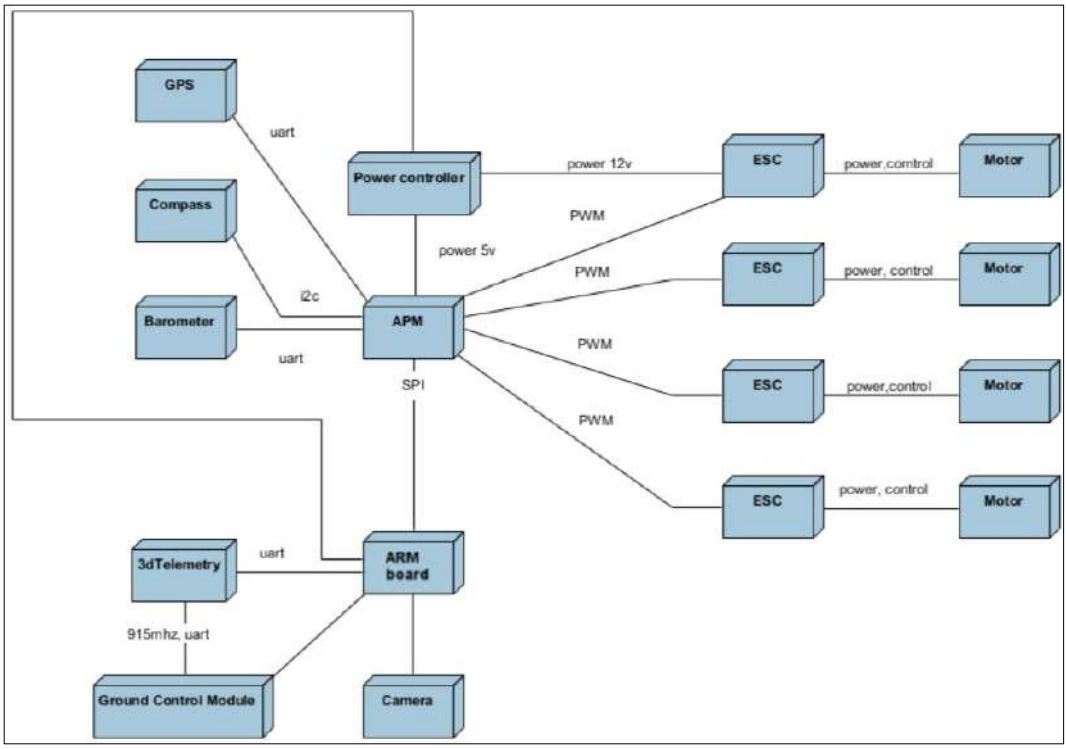


Fig. 15. Our intermediate architecture: APM controller, its sensors, motors and ARM board connection using SPI bus

Fig. 16. Connection of the controllers using SPI bus

Fig.17 shows two windows of USB-Com port devices monitoring the APM and the new controller with partitioned OS that provides PID-control based on real sensors data.

We used the ARM M3-based board STM32f103 as the controller hardware. The source code of FreeRTOS was used to study some platform-related stuff. The ARM M7-based board STM32F746 was used to work with ARM MPU regions [26] with debugging process based on OpenOCD and STM HAL library to deal with periphery. We are working to deploy the solution on the Raspberry Pi board as some open source programs for it [27] allow to work with internal Broadcom hardware. We are planning to describe POK porting issues in further papers.

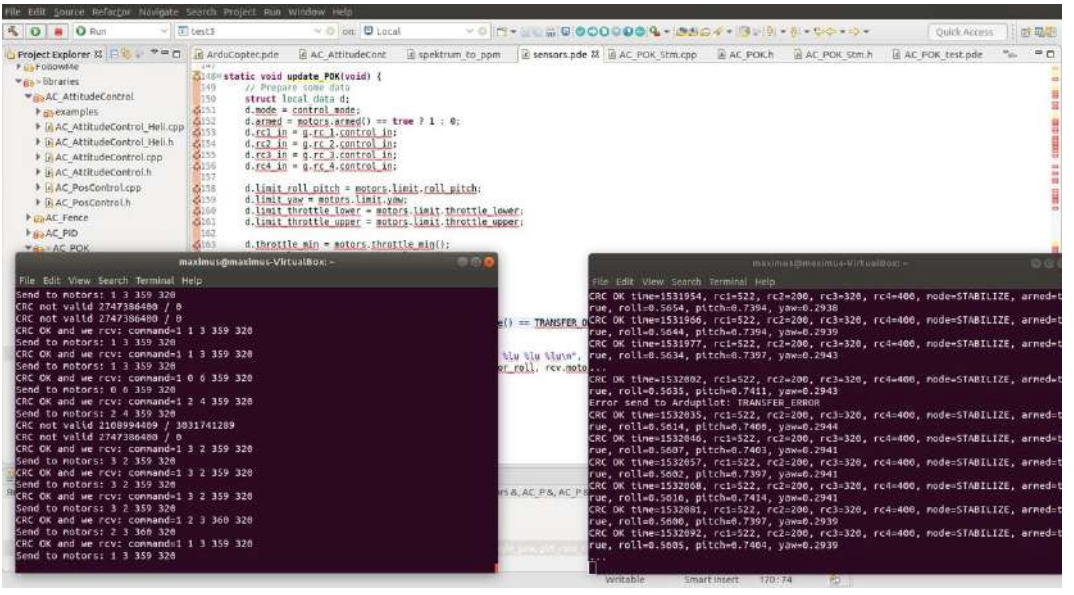


Fig. 17. USB-Com port monitoring of two interconnected controllers

```
\problem {
 \[
 R p, v, a, S, Kp, Kd, c, r
 \] (   (  (  (  (((v) >= (0)) & ((Kp) = (2)))
 & ((Kd) = (3)))
 & ((c) >  (0)))
 & (   (  (  (  ((5) / (4))
 * (((p) - (r)) ^ (2)))
 + ((((p) - (r)) * (v)) / (2)))
 + (((v) ^ (2)) / (4)))
 <  (c)))
 -> (\[
 {(p') = (v), (v') = (((-(Kp)) * ((p) - (r))) - ((Kd) * (v))),
 (v) >= (0)}
 \] (   (  (  (  ((5) / (4))
 * (((p) - (r)) ^ (2)))
 + ((((p) - (r)) * (v)) / (2)))
 + (((v) ^ (2)) / (4)))
 <  (c))))
}
```

Fig. 18. Hybrid program in KeYmaera syntax to check the stability

6. Formal verification

6.1 Verification at the cyber-physical system level

Cyber-physical system is a computer science abstraction of controllable physical process. It consists of two parts:

- a Cyber part – discrete controller;
- a Physical part – continuous model of the system usually expressed in ordinary differential equations.

These systems can be modeled as Hybrid automata with discrete-time and continuous-time transitions. Such models are known as Hybrid ones and are specified using the Dynamic Differential Logic. The approach is a kind of sliding state changing [28]. According to A. Platzer [29], the syntax of hybrid programs is defined as follows:

$$\alpha ::= x := e | ?Q | x' = f(x)Q | \alpha \cup \alpha | \alpha; \alpha | \alpha^* \quad (2)$$

where α is a meta-variable for a hybrid program; x is a meta-variable for the program variables; e is a meta-variable for first-order real-valued terms; f is a meta-variable for continuous real functions; and Q is a meta-variable for first-order formulas over real numbers. The construct «;» means sequential composition, «∪» is non-deterministic choice, «?» is the condition operator, and «α^*» is non-deterministic iteration (like Kleene-star) [30].

Here we discuss a formal verification of a simple PD-controller (the simplification of PID-controller) given from example [31]. The model of the system is represented as a Hoare's triple:

$$init \Longrightarrow [controller](req) \quad (3)$$

where *init* – a precondition; *controller* – a hybrid model; *req* – requirements that are invariants.

Then, we decompose the system into precondition, continuous PD-controller and requirements. Precondition:

$$init :== v \geq 0 \wedge c > 0 \wedge K_p = 2 \wedge K_d = 3 \wedge V(p, p_r, v) < c \quad (4)$$

where v is the velocity; c – a number greater than zero; K_p – a proportional part coefficient; K_d – a differential part coefficient; $V(p, p_r, v)$ – is a Lyapunov function.

The continuous state:

$$controller :== p' = v, v' = -K_p \cdot (p - p_r) - K_d \cdot v \quad (5)$$

where p is a current position; p_r – a resulting position.

For the requirement, it is proposed to try a stability check using the Lyapunov method in the form:

$$req :== V(p, p_r, v) < c. \quad (6)$$

The stability means here that the UAV during control will be stabilized around a given point in space. The Lyapunov function is defined in a quadratic form as follows:

$$V(p, p_r, v) = 5/4 \cdot (p - p_r)^2 + (p - p_r) \cdot v/2 + v^2/4 \quad (7)$$

Using KeYmaera tool (see fig.18 for the system (2)-(7) representation in code in the form of Hybrid program) it is possible to automatically verify the stability of the CPS that modeling the PD-controller. For the real PID-controller of the drone (see fig. 8) it is hard to obtain an analytical form of the whole model and to find a Lyapunov function to prove the stability. Possibly, a special kind of linearisation is required here. So it is a very challenging task and a subject of further research. Moreover, additional research is required to find ways to generate proof obligations with preconditions, postconditions and invariants in hybrid automaton states to help automatically prove stability of a system (our initial results are desrcribed in [32]).

6.2 Verification at the code level

Let discuss a verification technique for a PID controller using the Weakest Precondition (WP) method and adding ACSL annotations [33] into C code. The WP approach as an extension of the Hoare's logic was proposed by Dijkstra. It requires a precondition to be as simple as possible to surely reach the corresponding postcondition. In this case, the further program verification will be as follows: first, calculate $W = wp(f, Q)$, go from the end of Q to the start of the function, and then post a task to prove $P => W$ to a theorem prover (we use Frama-C tool with WP plugin and its internal Alt-Ergo prover).

For example, below we deductively prove several functions from the code that performs the PID control based on the Ardupilot code. In this code, the intermediate values are not calculated every time but are stored in the PID structure (fig. 19).

```
typedef struct {
    float kp;       ///< coefficient for P
    float ki;       ///< coefficient for I
    float kd;       ///< coefficient for D
    int imax;       ///< maximum i value
    float integrator;       ///< integrator value
    float last_input;       ///< last input for derivative
    float last_derivative; ///< last derivative for low-pass filter
    float d_lpf_alpha;      ///< alpha used in D-term LPF
} PID;
```

Fig. 19. PID structure (from Ardupilot code)

To prove the code, we write annotations for postconditions, preconditions, and side effects of functions in ISO-standardized ACSL language. Consider first the simplest code with annotations for the function *float get_p (PID * pid, float error*) (fig. 20).

This specification is fairly obvious: *\valid* defines the requirement of a non-NULL pointer to the PID structure, *\result* – the return value, *requires* means the precondition, *ensures* – the post-

condition. However, even such a simple function cannot be proven because multiplying of two real numbers can cause overflow with an unexpected result. Therefore. the verification tool cannot guarantee the correctness of this code without explicitly using «infinite» Real logical type. To do this, we run the proof tool with the parameter *"-wp-model typed+real"* and (for this moment) we abandon possible floating-point errors with loss of precision.

```
/*@
requires \valid(pid);
ensures \result == error * (float)pid->kp;
*/
float get_p(PID *pid , float error)
{
    return error * pid->kp;
}
```

Fig. 20. A simple function with annotations

To prove the code of function *float get_i (PID * pid, float error, float dt)*, it is necessary to construct lemmas describing the verifying code in terms of logic, similar to the examples in [34].

Firstly, we note that the function can change the value of *pid->integrator* and there are three cases:

- pid-> integrator <-pid->imax: it is limited to -pid-> imax;
- pid-> integrator> pid->imax: it is limited to pid-> imax;
- otherwise, that is, (integrator> = -max) and (integrator <= max): no change of pid-> integrator.

At the same time, there must first be a change of *pid-> integrator* to (*error * pid-> ki) * dt*. Therefore, the solution is to create a set of lemmas in ACSL terminology and a logic that will be used as a function in the *ensures* section.

```
float get_i(PID *pid, float error, float dt) {
        if ((pid->ki != 0) && (dt != 0)) {
                pid->integrator += ((float) error * pid->ki) * dt;

                if (pid->integrator < -pid->imax) {
                        pid->integrator = -pid->imax;
                } else
                if (pid->integrator > pid->imax) {
                        pid->integrator = pid->imax;
                }
                return pid->integrator;
        }
        return 0;
}
```

Fig. 21. A function to prove

Secondly, we note that the function returns 0 if the first condition does not hold and it does not change the value of *pid->integrator*. To describe the postcondition, we provide a description of the guard conditions in a form of implications. In fig. 22 we show a specification for the function in fig. 21. Here *\old* is the memory state before calling the function and *\at(..., Post)* – after calling it.

Fig 23 shows that the specification often takes even more space than the code itself, and writing it significantly changes the way of development; it ensures the quality of the code by coding the algorithms twice: in programming and logic languages.

```
axiomatic CheckAxiomatic {
logic float CheckUp{L}(float integrator, integer max);
lemma CheckUpMin{L}: \forall float integrator, integer max;
     (integrator < -max) ==> CheckUp(integrator, max) == (float)-max;
lemma CheckUpMax{L}: \forall float integrator, integer max;
      integrator >  max ==> CheckUp(integrator, max) == (float)max;
lemma CheckUpNorm{L}: \forall float integrator, integer max;
     (integrator >= -max) && (integrator <= max) ==> CheckUp(integrator, max) == integrator;
}

requires \valid(pid);
requires pid->imax > 0;
assigns pid->integrator;

ensures ((pid->ki != 0) && (dt != 0)) ==> \at(pid->integrator, Post)  ==
CheckUp((float) (\old(pid->integrator) + ((float) error * pid->ki) * dt), (int) pid->imax);
ensures !((pid->ki != 0) && (dt != 0)) ==> \at(pid->integrator, Post)  ==  \old(pid->integrator);
ensures  ((pid->ki != 0) && (dt != 0)) ==> \result ==  \at(pid->integrator, Post);
ensures !((pid->ki != 0) && (dt != 0)) ==> \result ==  0;
```

Fig. 22. A specification for the last function

```
Check that partitions declare their criticality level..... validated
Check that partition component share the same memory level..... validated
Check compliance of Health Monitoring service for partitions processes validated
Check for permanent errors between partitions..... validated
Check Threads Memory requirements..... validated
Check for transient errors between partitions..... validated
Check that each virtual bus provides protection mechanisms..... validated
Check Biba security policy..... validated
Check Partitions Memory requirements..... validated
Check compliance of Health Monitoring service for modules..... validated
Check that each virtual bus with a different security level has a different cipher key validated
Check Major Time Frame compliance..... validated
Check error coverage..... validated
Check that connections support appropriate security levels (MILS)..... validated
Check Bell-Lapadula security policy..... validated
Check that AADL model contain memory components..... validated
Check compliance of Health Monitoring service for partitions..... validated
Check that buses provides virtual buses..... validated
Check that virtual processors contain virtual buses..... validated
Check that each partition is executed at least one time by the module. validated
```

Fig. 23. Validation of AADL code

6.3 Validation of OS config

Thanks to AADL, a language with formal semantics, it is possible to analyze the code in it, create a formal model and then validate it. Some ideas of AADL code checking are given in [4] and [24].

In fig. 23 we demonstrate the result of automatic code validation for the model presented in Section 5 at the phase of the code generation process.

7. Conclusion

During the research, we studied information on modern drones with open-source software and commodity hardware. We did a detailed analysis of the Ardupilot software and the APM board. We touched some modern approaches to the organization of operating systems for such devices using partitions, AADL language and MDE applied to the OS configuration, code generation and validation.

We have developed a demo system, in which a quadcopter state is transferred from the APM board to a different ARM-based board with the control logic implemented in parallel partitioned processes using ports for interprocess communication.

As a result, we propose a design of software solution for UAVs with enhanced reliability requirements. This solution should ensure robustness at the following five levels.

- OS Level. Using a partitioned real-time OS will provide low-level scheduling and process isolation.
- Validation level. The level of interaction between the processes through ports and messages will be described in AADL. Additional model checking is available.
- Code level. Source code annotations and its deductive proof.
- Level of processes-monitors (dynamic testing). Monitoring processes can ensure that the safety conditions of the running system are maintained.
- Level of cyber-physical system (static verification). Proof of correctness of mathematical models of physical processes (safety and stability).

References / Список литературы

[1]. Avionics application software standard interface, part 1 – required services, ARINC specification 653P1-3, November 15, 2010. Aeronautical radio, inc. (ARINC).
[2]. Delange J., Lec L. POK, an ARINC653-compliant operating system released under the BSD license. In Proc. of the 13th Real-Time Linux Workshop, 2011.
[3]. Delange J., Gilles O., Hugues J., and Pautet L. Model-Based Engineering for the Development of ARINC653 Architectures. SAE International Journal of Aerospace, vol. 3, no. 1, 2010, pp. 79-86.
[4]. Hugues J., Delange J. Model-based design and automated validation of ARINC653 architectures using the AADL. In Cyber-Physical System Design from an Architecture Analysis Viewpoint, Springer, 2017, pp. 33-52.
[5]. POK kernel repository. Available at: https://github.com/pok-kernel/pok.
[6]. Mallachiev K.M., Pakulin N.V., Khoroshilov A.V. Design and architecture of real-time operating system. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 2, 2016, pp. 181-192. DOI: 10.15514/ISPRAS-2016-28(2)-12.
[7]. Solodelov Yu.A., Gorelits N.K. Certifiable onboard real-time operation system JetOS for Russian aircrafts design. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 3, 2017, pp. 171-178 (in Russian). DOI: 10.15514/ISPRAS-2017-29(3)-10 / Солоделов Ю.А., Горелиц Н.К. Сертифицируемая бортовая операционная система реального времени JetOS для российских проектов воздушных судов. Труды ИСП РАН, том 29, вып. 3, 2017 г.. стр. 171-178.
[8]. JetOS. Available at: https://github.com/yoogx/forge.ispras.ru-git-chpok.
[9]. Khoroshilov A.V. On formalization of operating systems behaviour verification. In Proc. of the Eleventh International Conference on Computer Science and Information Technologies, Revised Selected Papers, 2017, pp. 168-172.
[10]. Kulyamin V.V., Lavrischeva E.M., Mutilin V.S., Petrenko A.K. Verification and analysis of variable operating systems, Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 3, 2016, pp. 189–208 (in Russian). DOI: 10.15514/ISPRAS-2016-28(3)-12 / Кулямин В.В., Лаврищева Е.М., Мутилин В.С., Петренко А.К. Верификация и анализ вариабельных операционных систем. Труды ИСП РАН, том 28, вып. 3, 2016 г., стр. 189-208.
[11]. Khoroshilov A.V., Kuliamin V.V., Petrenko A.K. Verification of Operating System Components. System Informatics, No. 10, 2017, pp. 11-22.
[12]. Klein G. Operating system verification – An overview. Sadhana, vol. 34, no. 1, 2009, pp. 27-69
[13]. Giernacki W. et al. Crazyflie 2.0 quadrotor as a platform for research and education in robotics and control engineering. In Proc. of the 22nd International Conference on Methods and Models in Automation and Robotics (MMAR), 2017, pp. 37-42.
[14]. Crazyflie AADL Case Study. Available at: https://github.com/OpenAADL/Crazyflie.
[15]. Santoro C. How does a Quadrotor fly? A journey from physics. Available at: https://www.slideshare.net/corradosantoro/quadcopter-31045379.
[16]. Ardupulot project. Available at: http://ardupilot.org/copter.
[17]. Ardupulot project on Github. Available at: https://github.com/ArduPilot/ardupilot.
[18]. The CUAV Pixhack V3 flight controller board. Available at: https://docs.px4.io/en/flight_controller/pixhack_v3.html.
[19]. History of Ardupilot. Available at: http://ardupilot.org/planner2/docs/common-history-of-ardupilot.html.

[20]. Ardupilot. Advanced Tuning. Available at: http://ardupilot.org/copter/docs/tuning.html.
[21]. Copter Attitude Control. Available at: http://ardupilot.org/dev/docs/apmcopter-programming-attitude-control-2.html.
[22]. Hall Leonard. Practical PID implementation and the new Position Controller. ArduPilot UnConference, 2018. Available at: https://www.youtube.com/watch?v=-PC69jcMizA.
[23]. Park S., Deyst J., How J. A new nonlinear guidance logic for trajectory tracking. In Proc. of the AIAA guidance, navigation, and control conference and exhibit, 2004.
[24]. Delange J. AADL in Practice: Become an expert in software architecture modeling and analysis. Reblochon Development Company, 2017, 252 p.
[25]. POK. Examples. Case Study Ardupilot. Available at: https://github.com/pok-kernel/pok/tree/master/examples/case-study-ardupilot
[26]. Joseph Yiu. The Definitive Guide to the ARM Cortex-M3, 2nd Edition. Newnes, 2009, 479 p. / Джозеф Ю. Ядро Cortex-M3 компании ARM. Полное руководство. Пер. с англ. АВ Евстифеева. Додэка-XXI, 2012, 552 стр.
[27]. RaspberryPi-FreeRTOS. Demo. Drivers. Available at: https://github.com/jameswalmsley/RaspberryPi-FreeRTOS/tree/master/Demo/Drivers
[28]. Brandtstädter H. Sliding mode control of electromechanical systems. PhD Thesis, Technische Universität München, 2009.
[29]. Platzer A. Logical foundations of cyber-physical systems. Springer, 2018, 639 p.
[30]. Sergey Staroletov, Nikolay Shilov et al. Model-Driven Methods to Design of Reliable Multiagent Cyber-Physical Systems. In Proc. of the Conference on Modeling and Analysis of Complex Systems and Processes (MACSPro 2019), 2019.
[31]. Jan-David Quesel, Stefan Mitsch et al. How to model and prove hybrid systems with KeYmaera: a tutorial on safety. International Journal on Software Tools for Technology Transfer, vol. 18, №. 1, 2016, pp. 67-91.
[32]. Baar T., Staroletov S.M. A control flow graph based approach to make the verification of cyber-physical systems using KeYmaera easier. Modeling and Analysis of Information Systems, vol. 25, №. 5, 2018, pp. 465-480.
[33]. Patrick Baudin, Jean-Christophe Filliâtre et al. ACSL: ANSI C Specification Language. Frama-C, 2008, 81 p.
[34]. Jochen Burghardt, Andreas Carben et al. ACSL by Example. DEVICE-SOFT project publication. Fraunhofer FIRST Institute, 2010, 134 p.

Информация об авторах / Information about authors

Сергей Михайлович СТАРОЛЕТОВ – кандидат физико-математических наук, доцент кафедры прикладной математики. Сфера научных интересов: формальная верификация, Model checking, киберфизические системы, операционные системы.

Sergey Michailovich STAROLETOV – Candidate of Physical-Mathematical Sciences (PhD), Associate Professor at the department of Applied Mathematics. Research interests: formal verification, Model checking, cyber-physical systems, operating systems.

Максим Станиславович АМОСОВ – магистрант по специальности «Программная инженерия». Сфера научных интересов: программирование, формальная верификация, системы контроля версий.

Maxim Stanislavovich AMOSOV – Master degree student of Software Engineering. Research interests: programming, formal verification, version control systems.

Кирилл Михайлович ШУЛЬГА – бакалавр по специальности «Программная инженерия». Сфера научных интересов: программирование, квадрокоптеры, блокчейн.

Kirill Mikhailovich SHULGA – Bachelor of Software Engineering. Research interests: programming, quadcopters, blockchain.

DOI: 10.15514/ISPRAS-2019-31(4)-4

Технология и методы отложенного синтеза 4K-стереороликов для сложных динамических виртуальных сцен

П.Ю. Тимохин, ORCID: 0000-0002-0718-1436 <webpismo@yahoo.de>
М.В. Михайлюк, ORCID: 0000-0002-7793-080X <mix@niisi.ras.ru>
Е.М. Вожегов, ORCID: 0000-0003-2676-1206 <vozhegovem@icloud.com>
К.Д. Пантелей, ORCID: 0000-0001-9281-2396 <kpanteley@mail.ru>
ФГУ «ФНЦ Научно-исследовательский институт системных исследований РАН», 117218, Россия, г. Москва, Нахимовский просп., 36, к.1

Аннотация. В статье рассматривается задача записи управляемой исследователем стереовизуализации сложной динамической виртуальной сцены в видеопоследовательность стереопар (стереоролика) сверхвысокого разрешения. Предлагается технология отложенного синтеза стереороликов, которая позволяет создавать такие стереоролики, не нарушая масштаб реального времени визуализации. Технология включает в себя построение в масштабе реального времени сценария процесса визуализации и офлайн-преобразование сценария в стереоролик. В работе рассматриваются методы реализации этих этапов на примере задачи стереовизуализации изоповерхности насыщенности вытесняющей жидкости. В исследовании предлагается разработанный оригинальный файловый формат «scr» сценария визуализации, основанный на чанковой структуре данных, который реализует компактное представление соседних одинаковых кадров. Преобразование файла сценария в последовательность 4K-стереопар выполняется с помощью технологии внеэкранного рендеринга виртуальной сцены, а добавление стереопар в стереоролик – с помощью набора открытых библиотек FFmpeg обработки цифровых видеозаписей. В основе стереоролика используется медиаконтейнер MP4 и стандарт H.264 сжатия видео (оба являются частями международного стандарта MPEG-4). Предложенные технология и методы отложенного синтеза 4K-стереороликов реализованы в программном комплексе визуализации результатов моделирования неустойчивого вытеснения нефти из пористых сред. С помощью данного программного комплекса был синтезирован 4K-стереоролик, иллюстрирующий процесс изменения изоповерхности насыщенности вытесняющей жидкости на стадии развития процесса неустойчивого вытеснения нефти. Проведенная апробация подтвердила адекватность созданных решений поставленной задаче. Разработанные решения могут быть использованы в виртуальных лабораториях, при построении систем виртуального окружения, систем научной визуализации, в образовательных приложениях и др.

Ключевые слова: стерео; визуализация; виртуальная сцена; сценарий; кодирование видео; захват видео; реальное время; внеэкранный рендеринг; 4K

Для цитирования: Тимохин П.Ю., Михайлюк М.В., Вожегов Е.М., Пантелей К.Д. Технология и методы отложенного синтеза 4K-стереороликов для сложных динамических виртуальных сцен. Труды ИСП РАН, том 31, вып. 4, 2019 г., стр. 61-72. DOI: 10.15514/ISPRAS-2019-31(4)-4

Благодарности: Исследование выполнено при финансовой поддержке РФФИ в рамках научного проекта № 16-29-15099-офи_м.

Technology and methods for deferred synthesis of 4K stereo clips for complex dynamic virtual scenes

P.Yu. Timokhin, ORCID: 0000-0002-0718-1436 <webpismo@yahoo.de>
M.V. Mikhaylyuk, ORCID: 0000-0002-7793-080X <mix@niisi.ras.ru>
E.M. Vozhegov, ORCID: 0000-0003-2676-1206 <vozhegovem@icloud.com>
K.D. Panteley, ORCID: 0000-0001-9281-2396 <kpanteley@mail.ru>
Federal State Institution «Scientific Research Institute for System Analysis» of RAS,
36, build. 1, Nakhimovskiy Avenue, Moscow, 117218, Russia

Abstract. The paper considers the task of capturing controlled by a researcher stereo visualization of a complex dynamic virtual scene into a stereopair videosequence (stereoclip) of ultrahigh resolution. An efficient technology of deferred synthesis of stereoclips is proposed. It allows to create stereoclips without violating a real-time visualization. The technology includes real-time constructing of scenario of visualization process and offline-transforming the scenario to stereoclip. In the paper, methods to realize these stages are considered for the task of stereovisualization of saturation isosurface of displacing liquid. For this, original file format «scr» of visualization scenario is developed, based on «chunk» data structures. The format developed provides a compact representation of neighboring repeated frames. Transforming scenario file to a sequence of 4K-stereopairs is carried out by means of an offscreen rendering of virtual scene, and adding stereopairs to a stereoclip is performed using a number of open-source FFmpeg libraries designed for processing digital video content. For video recording media container MP4 and video compressing standard H.264 are used. Proposed technologies and methods of 4K-stereoclips deferred synthesis are implemented in a program complex for visualization of simulation results of unstable oil displacement from porous media. By means of the program complex a 4K-stereoclip is created, which illustrates the evolution of the isosurface during the process of unstable oil displacement. The approbation results confirmed the adequacy of the proposed solution to the task. Developed solutions can be used in virtual laboratories, in constructing of virtual environment systems and scientific visualization systems, in educational applications etc.

Keywords: stereo; visualization; virtual scene; scenario; video encoding; video capture; real-time; offscreen rendering; 4K

For citation: Timokhin P.Yu., Mikhaylyuk M.V., Vozhegov E.M., Panteley K.D. Technology and methods for deferred synthesis of 4K stereo clips for complex dynamic virtual scenes. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 4, 2019. pp. 61-72 (in Russian). DOI: 10.15514/ISPRAS-2019-31(4)-4

Acknowledgements. The reported study was funded by RFBR, project number 16-29-15099.

1. Введение

В настоящее время во многих научных экспериментах востребована трехмерная визуализация сложных динамических виртуальных сцен [1, 2] в масштабе реального времени (с частотой смены кадров не менее 25 раз в секунду). Это особенно актуально в тех областях, где исследование реального объекта сопряжено с его разрушением, а цена нового образца крайне высока. Примером является нефтегазовая сфера, в частности, виртуальные эксперименты по неустойчивому вытеснению нефти из пористых сред [3-5].

Значительно повысить информативность и качество виртуальных экспериментов позволяет стереовизуализация виртуальных сцен [6] в сверхвысоком разрешении (4K, Ultra HD). Выполнение такой визуализации в реальном времени является сложной вычислительной задачей, для решения которой необходимы видеокарты high-end класса и специализированное программное обеспечение, что препятствует обмену полученными результатами в научном сообществе. Эффективным выходом является запись процесса визуализации виртуальной сцены в виде видеопоследовательности из стереопар (*стереоролика*). Стереороликами можно легко обмениваться между исследователями и воспроизводить их с помощью программ-стереоплееров [7] на персональных компьютерах и мобильных устройствах. Проблема состоит в том, что при использовании внешних программ

видеозахвата [8] визуализация каждого кадра фактически приостанавливается на некоторое время, необходимое для считывания и кодирования изображения, что приводит к нарушению режима реального времени синтеза изображений и появлению рывков в видеоролике. Особенно это заметно при выполнении исследователем различных управляющих воздействий на процесс визуализации – вращении и приближении виртуальной сцены, переключении между ракурсами и др. С целью уменьшения временных затрат на захват кадров активно развиваются аппаратные кодировщики видео (NVidia NVENC [9]), в которых алгоритмы сжатия распараллеливаются на многоядерных графических процессорах (GPU). По сравнению с кодированием видео на центральном процессоре (CPU) это дает существенный прирост скорости в задачах сжатия готовых видеопоследовательностей [10], однако, в задачах визуализации сложных динамических виртуальных сцен, где интенсивно используется GPU [4], наблюдается эффект взаимного торможения процессов кодирования и визуализации. В этой связи возникает задача разработки технологий синтеза стереороликов *in situ*, т.е. непосредственно в системе визуализации, основанных на захвате первичной информации (динамических параметров визуализации виртуальной сцены), объем которой значительно меньше, чем объем вторичных данных (синтезированных изображений).

В данной работе предлагается технология *отложенного* синтеза 4K-стереороликов, основанная на построении в масштабе реального времени сценария управляемой исследователем визуализации виртуальной сцены и преобразовании сценария в стереоролик в офлайн режиме. Предлагаемое решение реализуется на языке C++ с использованием открытого комплекса Qt средств разработки приложений, графической библиотеки OpenGL и набора открытых библиотек FFmpeg обработки цифровых видеозаписей.

2. Технология отложенного синтеза стереороликов

Пусть имеется некоторая система визуализации, которая может выполнять в масштабе реального времени рендеринг виртуальных сцен в моно и стерео режимах (горизонтальная стереопара, side-by-side [6]). Исходно в систему загружена необходимая виртуальная сцена и установлен моно режим. Предлагаемая технология синтеза стереороликов включает в себя два этапа: а) захват параметров визуализации и б) непосредственно синтез 4K-стереоролика.

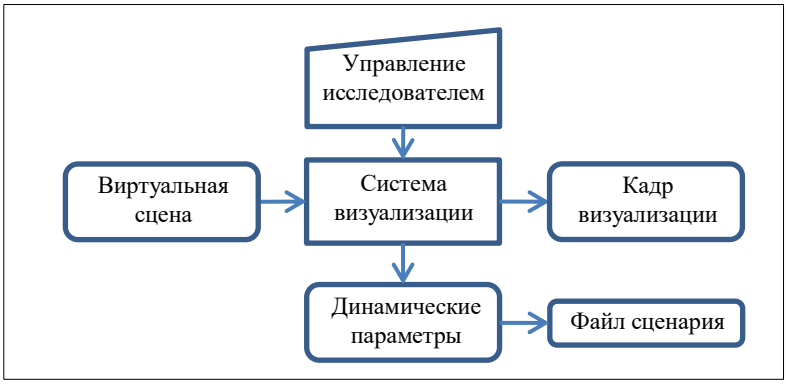


Рис. 1. Схема построения сценария стереоролика
Fig.1. The scheme of stereoclip scenario construction

На ***первом этапе*** из системы визуализации в масштабе реального времени считывается набор *динамических параметров* визуализации виртуальной сцены (см. рис. 1). По каждому такому набору можно однозначно восстановить состояние виртуальной сцены на момент синтеза соответствующего кадра. Совокупность всех захваченных неповторяющихся подряд наборов динамических параметров в данной работе называется *сценарием* визуализации. По окончании этапа захвата формируется файл с записанным в него сценарием.

На ***втором этапе*** (синтеза 4K-стереоролика) в системе визуализации выполняется специальное покадровое воспроизведение файла сценария, при котором в отдельно созданном внеэкранном (невидимом) буфере кадра выполняется синтез 4K-стереопары, а в экранный (видимый) буфер кадра помещается ее уменьшенная копия, вписанная по размерам в главное окно визуализации (см. рис. 2). Каждая синтезированная 4K-стереопара считывается из внеэкранного буфера, кодируется и добавляется в файл стереоролика, а уменьшенная копия выводится на экран для визуального контроля процесса синтеза стереоролика. Данный процесс может выполняться уже не в масштабе реального времени в зависимости от выбранного кодировщика видео и вычислительной мощности CPU и GPU компьютера.

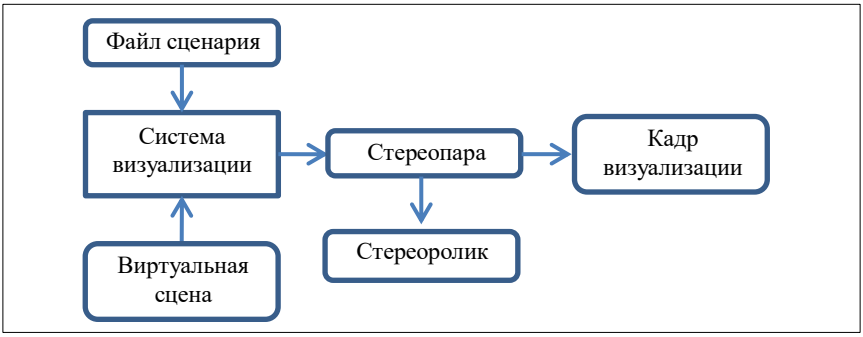


Рис. 2. Схема преобразования сценария в стереоролик
Fig. 2. The scheme of transforming scenario to a stereoclip

Далее рассмотрим методы реализации описанных этапов на примере задачи визуализации результатов неустойчивого вытеснения нефти из пористых сред [6], в частности, визуализации поверхности постоянного значения поля насыщенности вытесняющей жидкости (*изоповерхности*).

2.1 Метод построения сценария визуализации

В рассматриваемой задаче параметрами процесса визуализации являются:

- положение и ориентация модели изоповерхности;
- параметры виртуальной камеры;
- параметры источника освещения;
- параметры материала модели изоповерхности;
- постоянное значение поля насыщенности (изоуровень);
- номер визуализируемого шага моделирования.

Для захвата этих параметров в данной работе разработан формат файла сценария визуализации *(scr-файл)*, который включает в себя заголовок (с общей информацией) и блок данных со значениями параметров визуализации (рис. 3). Заголовок scr-файла содержит следующие данные:

- идентификатор *scrId* = 0x00726373 («scr» в кодировке ASCII);
- число *numFrames* кадров в сценарии;
- интервал *queryInterval* опроса значений параметров визуализации, в мс;
- размер *scrSize* блока данных *scr*-файла, в байтах.

В *блоке данных scr-файла* хранятся значения параметров визуализации только для неповторяющихся подряд кадров. Если кадр повторяется несколько раз, то параметры визуализации записываются только для первого кадра, а последующие повторные кадры просто подсчитываются. Для этого используется структура «пакет кадра», включающая в себя:

- размер *blockSize* блока данных пакета, в байтах;
- число *numRepeats* повторов подряд текущего кадра;
- блок данных пакета со значениями параметров визуализации.

Блок данных пакета состоит из «чанков». «Чанк» – это структура данных, которая содержит идентификатор *id* объекта, размер *size* (в байтах) и поле *data* значений параметров визуализации. В поле *data* «чанка» мы записываем значения параметров визуализации (одного или группы параметров, объединенных по смыслу). Использование таких структур данных дает возможность эффективно добавлять в *scr*-формат новые параметры визуализации, модифицировать и удалять их, сохраняя обратную совместимость между форматами (если система визуализации не может распознать какой-то идентификатор «чанка», то его поле *data* просто пропускается).

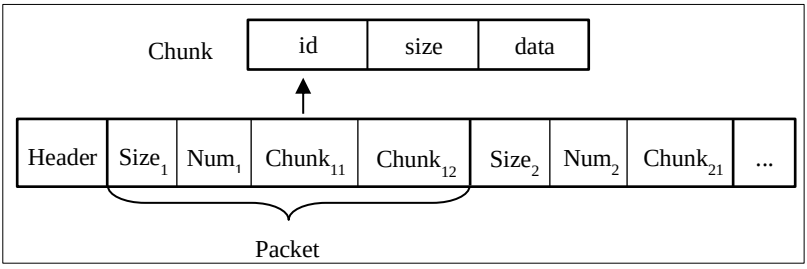


Рис. 3. Структура scr-файла
Fig. 3. Scr-file structure

Для рассматриваемой задачи реализованы следующие типы «чанковых» структур:

- *SMVChunk* – модельно-видовой матрицы, задающей положение и ориентацию модели изоповерхности в системе координат камеры (id = 0, size = 64, data - 16 элементов модельно-видовой матрицы);
- *SCamChunk* – камеры (id = 1, size = 16, data – вертикальный угол раствора камеры, отношение ширины кадра к высоте (*аспект*), расстояния до ближней и дальней плоскостей отсечения);
- *SLightChunk* – направленного источника освещения (id = 2, size = 64, data – направление источника освещения, интенсивности фоновой, диффузной и зеркальной составляющих освещения);
- *SMaterialChunk* – материала модели изоповерхности (id = 3, size = 52, data – цвета фоновой, диффузной и зеркальной составляющих материала, а также коэффициент блеска);
- *SIsoLevelChunk* – изоуровня (id = 4, size = 4, data – постоянное значение скалярного поля насыщенности вытесняющей жидкости);
- *SStepChunk* – визуализируемого шага моделирования (id = 5, size = 4, data – номер визуализируемого шага моделирования).

Захват параметров визуализации осуществляется с некоторой частотой ν_1 опроса (в разах в секунду), где $25<\nu_1\leq\nu_2$, а ν_2 – наименьшая частота синтеза изображений виртуальной сцены, которая зависит от вычислительной мощности видеокарты. Во время каждого опроса считываются значения всех параметров визуализации и сравниваются со значениями предыдущего опроса. Если текущее значение какого-то параметра визуализации повторяется,

то оно не записывается в файл сценария. На рисунке 4 изображена схема опроса на примере модельно-видовой матрицы модели изоповерхности, которая рассчитывается для каждого кадра на GPU. Отметим, что в более сложных многообъектных виртуальных сценах для сокращения размера файла сценария положения и ориентации виртуальных объектов можно записывать в виде кватернионов.

В данной работе опрос значений параметров визуализации реализуется с помощью таймера (класс *QTimer*) из библиотеки Qt. Сигнал периодического срабатывания таймера связывается с разработанной функцией формирования пакета с помощью оператора *connect*. Получаемые в результате работы этой функции пакеты кадров мы будем накапливать в динамический массив *V* байтов.

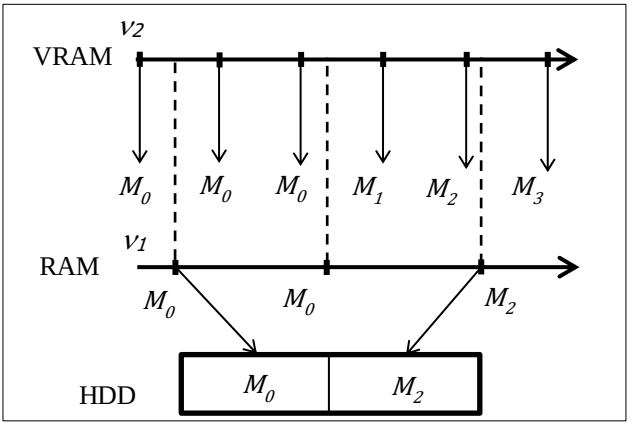


Рис. 4. Опрос модельно-видовой матрицы модели изоповерхности
Fig.4. Querying of modelview matrix of isosurface model

Построение сценария визуализации реализует *Алгоритм 1*.

1. Запишем: *scrSize* = 0, *numFrames* = 0, *blockSize* = 0, *numRepeats* = 0, *PACKET_HEADER* = *sizeof(blockSize)* + *sizeof(numRepeats)*, где *sizeof* возращает размер переменной в байтах.
   Инициализируем флаг нового неповторяющегося кадра: *gotFrame* = *false*.
2. Создадим «чанки» *SMVChunk*, *SCamChunk*, *SLightChunk*, *SMaterialChunk*, *SIsoLevelChunk*, *SStepChunk* с пустыми полями *data*.
3. Запустим таймер опроса с помощью функции *start()*.
4. Цикл опроса с интервалом *queryInterval*
   blockSize = 0.
   Считаем значение *isoLevel* изоуровня из системы визуализации.
   Если *isoLevel* ≠ значению *data* «чанка» *SIsoLevelChunk*, то:
   Если *gotFrame* ≠ *true*
   Добавим в массив *V* *blockSize* и *cntFrame*.
   gotFrame = true.
   Запишем *isoLevel* в поле *data* «чанка» *SIsoLevelChunk*.
   Добавим «чанк» *SIsoLevelChunk* в массив *V*.
   blockSize = blockSize + sizeof(SIsoLevelChunk).
   Проверим значения остальных параметров визуализации аналогично *isoLevel*.
   gotFrame = false.
   Если *blockSize* ≠ 0, то:
   numRepeats = 0.
   Обновим в массиве *V* значения *blockSize* и *numRepeats* текущего пакета.
   scrSize = scrSize + PACKET_HEADER + blockSize.
   В противном случае:
   numRepeats = numRepeats + 1.

Обновим в массиве *V* значение *numRepeats*.
numFrames = numFrames + 1.
Конец цикла.

5. Остановим таймер опроса с помощью функции *stop()*.
6. Создадим scr-файл.
7. Запишем в scr-файл *scrId*, *numFrames*, *queryInterval*, *scrSize* и массив *V*.
8. Закроем файл.

Алгоритм 1. Построение сценария визуализации
Algorithm 1. Construction of the visualization scenario

Чтобы просмотреть захваченный процесс визуализации, в данной работе также создан режим воспроизведения файла сценария. Как и в *Алгоритме 1*, данный режим реализуется с помощью таймера. При этом сигнал периодического срабатывания таймера связывается с разработанной функцией проигрывания сценария. Предварительно мы загружаем все пакеты из файла сценария в байтовый массив *K* размера *scrSize* (из заголовка scr-файла). Воспроизведение сценария реализует *Алгоритм 2*.

1. Инициализируем следующие переменные:
   gotFrame = false, // флаг нового неповторяющегося кадра.
   packetsCnt = 0, // счетчик считанных байт из блока данных scr-файла.
   frameCnt = 0, // счетчик проигранных повторных пакетов (кадров).
   curSize = 0, // число считанных байт из блока данных пакета.
   CHUNK_HEADER = sizeof(id) + sizeof(size). // размер заголовка «чанка», в байтах.
2. Запустим таймер проигрывателя с помощью функции *start()*.
3. Цикл проигрывания с интервалом *queryInterval*
   3.1. Если *packetsCnt == scrSize*, то:
   Выйдем из цикла проигрывания с помощью функции *stop()*.
   3.2. Если *gotFrame ≠ true*, то:
   Считаем *blockSize* и *numRepeats* текущего пакета из массива *K*.
   curSize = 0.
   Цикл по «чанкам» в пакете, пока *curSize ≠ blockSize*.
   Считаем *id* и *size* текущего чанка.
   Считаем из поля *data* текущего «чанка» значение параметра визуализации согласно его *id*.
   Обновим в системе визуализации значение считанного параметра.
   curSize = curSize + CHUNK_HEADER + size.
   Конец цикла.
   packetsCnt = packetsCnt + PACKET_HEADER + blockSize.
   Если *numRepeats* ≠ 0, то *gotFrame = true;*
   В противном случае:
   frameCnt = frameCnt + 1.
   Если *frameCnt == numRepeats*, то *gotFrame = false, frameCnt = 0*.
   3.3. Визуализируем сцену с обновленными параметрами.
   Конец цикла.

Алгоритм 2. Воспроизведение сценария визуализации
Algorithm 2. Playback of the visualization scenario

2.2 Метод преобразования сценария в 4K-стереоролик

Для создания 4K-стереоролика необходимы эффективные медиаконтейнер и алгоритм кодирования видео, поддерживающие работу с большими объемами видеоданных (около 16 млн. пикселов на кадр). В данном исследовании предлагается использовать контейнер *MP4*, являющийся частью международного стандарта MPEG-4 (MPEG-4 Part 14, ISO/IEC 14496-14:2003) сжатия цифрового аудио и видео. В таблице 1 приведено сравнение MP4 с распространенными контейнерами AVI (Microsoft), MOV (Apple) и MKV (Open Source). Из

таблицы видно, что MP4 превосходит по функционалу контейнеры AVI и MKV, в частности за счет поддержки B-кадров (они позволяют значительно сжимать размер кодируемой видеопоследовательности). Другим важным преимуществом является поддержка покадрового редактирования видео (технология Edit-in-place), что позволяет изменять отдельные участки видео (например, разрезать или склеивать) без повторного сжатия всей видеопоследовательности. Контейнер MOV близок к MP4 по функциональности, однако ориентирован в первую очередь на использование в операционных системах компании Apple, в то время как MP4 является отраслевым стандартом и имеет более широкую поддержку. Также немаловажной отличительной чертой контейнера MP4 является поддержка онлайн-трансляции.

Таблица 1. Сравнение видео контейнеров
Table 1. Comparing of video containers

Контейнер	**Поддержка кодека H.264**	**Edit-in-place**	**Поддержка B-кадров**
MP4	mp4 - официальный стандарт контейнера для кодека H.264	да	да
AVI	затруднена в связи с ограниченной поддержкой B-кадров	нет	отсутствует в исходном формате, реализуется «хакерским» путем
MOV	да	да	да
MKV	да	нет	да

Контейнер MP4 поддерживает ряд современных видеокодеков, из которых для решения рассматриваемой задачи был выбран кодек *H.264*. Данный кодек также является частью международного стандарта MPEG-4 (MPEG-4 Part 10) и, хотя по эффективности сжатия он уступает таким кодекам, как H.265/HEVC и AV1, является самым распространенным, так как уже прошел многолетний путь внедрения в отрасль. В течение этого периода стандарт постоянно модернизировался, получил аппаратное декодирование в большинстве плееров и в итоге хорошо зарекомендовал себя на практике. Особенностью видеокодека H.264 является работа с макроблоками (группами пикселов изображения) размерами от 16x16 до 4x4 [11], что накладывает соответствующие требования к ширине и высоте кодируемого видео. В рассматриваемой задаче 4K-стереопара имеет размеры, кратные 16 (7680x2160 пикселов).

В данной работе запись стереопар в стереоролик реализуется с помощью набора открытых библиотек *FFmpeg* (Fast Forward MPEG) [12]. Из всего набора для решения рассматриваемой задачи используются следующие библиотеки: *libavcodec* (кодеры и декодеры видео и аудио), *libavformat* (мультиплексоры и демультиплексоры медиаконтейнеров), *libswscale* (функции масштабирования изображений и преобразования цветовых пространств и форматов пикселов) и *libavutil* (генераторы случайных чисел, математические и мультимедиа утилиты и др.).

Кодек H.264 имеет большое количество настроек, которые позволяют управлять качеством и скоростью кодирования видеопоследовательности. К ним относятся битрейт, количество кадров в каждой группе последовательных изображений потока (GOP), параметр сложности оценки движения, коэффициент компрессии видеопотока, профиль видеокодека и другие. В данной работе эти параметры задаются в структуре *AVCodecContext* из библиотеки *FFmpeg*.

Чтобы начать процесс кодирования видео с помощью *FFmpeg*, необходимо выполнить ряд предварительных действий: задать формат выходного видео *(AVOutputFormat)*, создать контекст ввода-вывода для записи видео (*AVFormatContext)*, создать поток записи видео (*AVStream*), связать поток с кодировщиком H.264 (*AVCodec*), а также инициализировать структуры *AVFrame*, *SwsContext*, *AVPicture* для обработки видеокадров и структуру *AVPacket* для добавления кадров в контейнер MP4. Более подробное описание реализации этих этапов можно найти в [12].

Процесс записи 4K-стереоролика включает в себя модифицированное проигрывание файла сценария (см. *Алгоритм 2*), в котором используется функция стереовизуализации виртуальной сцены в формате side-by-side [6]. Исходными данными, как и в *Алгоритме 2*, является массив *K* пакетов кадра, загруженный из файла сценария. Преобразование сценария в стереоролик реализует *Алгоритм 3*.

1. Создадим внеэкранный буфер *F* кадра размера $(2w_s)$ *x* h_s пикселов, где w_s и h_s - ширина и высота 4K-кадра в моно режиме.
2. Создадим буфер *I* (типа *QImage* из библиотеки Qt) размера $(2w_s)$ *x* h_s пикселов для хранения стереопары в оперативной памяти.
3. Зададим прямоугольник $P_0P_1P_2P_3$, вписанный по ширине в главное окно визуализации размера w_f x h_f:
   Вычислим высоту h_i прямоугольника $P_0P_1P_2P_3$ в системе координат нормализованного объема видимости (Normalized Device Coordinate System, NDCS):
   $h_i = 2a_s\,a_f$, где $a_s = 2w_s / h_s$ – аспект 4K-стереопары, $a_f = w_f / h_f$ – аспект главного окна визуализации.
   Запишем координаты вершин прямоугольника $P_0P_1P_2P_3$ (в системе NDCS координат нормализованного объема видимости):
   $P_0 = (-1, 0.5h_i)$, $P_1 = (-1, -0.5h_i)$, $P_2 = (1, 0.5h_i)$, $P_3 = (1, -0.5h_i)$.
4. Откроем файл для записи видео с помощью функции *url_fopen* из FFmpeg.
5. Цикл по *i* от 1 до *numFrames*
   Считаем *i*-ый пакет кадра из массива *K* и обновим в системе визуализации значения считанных параметров (см. пп. 3.1, 3.2 из *Алгоритма* 2).
   Выполним синтез текстуры *S* с изображением стереопары:
   Активируем внеэкранный буфер *F* кадра.
   Визуализируем виртуальную сцену в стереорежиме side-by-side.
   Деактивируем внеэкранный буфер *F* кадра.
   Добавим стереопару в MP4-контейнер:
   Выгрузим из видеопамяти текстуру *S* в буфер *I* стереопары.
   Конвертируем с помощью функции *sws_scale* из FFmpeg RGB-стереопару (буфер *I*) в YUV-изображение (структура *AVFrame)*, где Y-яркость, а U и V - цветоразностные компоненты.
   Закодируем YUV-изображение в байтовый массив данных H.264 с помощью функции *avcodec_encode_video* из FFmpeg.
   Добавим массив данных H.264 в структуру отдельного сжатого кадра *AVPacket* (контейнер MP4).
   Отобразим стереопару в главном окне визуализации:
   Установим область вывода с помощью функции *glViewport*(0, 0, w_f, h_f).
   Визуализируем прямоугольник $P_0P_1P_2P_3$ с наложенной текстурой *S*.
   Конец цикла.
6. Закроем файл для записи видео с помощью функции *url_fclose*.

Алгоритм 3. Преобразование сценария в стереоролик
Algorithm 3. Transforming the scenario to a stereoclip

По окончании работы *Алгоритма 3* формируется MP4-файл с результирующим 4K-стереороликом в базовом стереоформате side-by-side, который может быть воспроизведен на поляризационной стереоустановке с помощью программы-стереоплеера, например, Stereoscopic Player, а также преобразован (в этом же плеере) в ряд других популярных видов стерео (анаглифическое, чересстрочное и др.).

3. Результаты

Предложенные в данной работе технология и методы отложенного синтеза 4K-стереороликов были реализованы в программном комплексе визуализации результатов моделирования неустойчивого вытеснения нефти из пористых сред [13]. С помощью данного комплекса было выполнено исследование стадии развития неустойчивости вытеснения

нефти водой, в частности, изменения формы изоповерхности насыщенности вытесняющей жидкости. Исходными данными являлась последовательность из 65 трехмерных массивов значений насыщенности вытесняющей жидкости, полученных в результате пошагового моделирования процесса неустойчивого вытеснения нефти на расчетной сетке размера 100^3 ячеек. В процессе исследования для каждого из 65 шагов моделирования выполнялись построение и визуализация трехмерной полигональной модели изоповерхности. Исследование включало в себя динамическое изменение ориентации и масштаба модели изоповерхности, а также ее перестроение для различных постоянных значений насыщенности. Визуализация модели изоповерхности проводилась при разрешении Ultra HD 4K (3840x2160) на персональном компьютере с процессором Intel Core i7 950 3.06 ГГц, RAM 12Гб, видеокартой NVIDIA GeForce GTX 1080 Ti (VRAM 11 Гб, 3584 ядер). Средняя частота визуализации составила около 100 кадров в секунду.

Был выбран участок исследования длительностью около 2-х минут, для которого был записан сценарий визуализации с интервалом опроса равным 10 мс. Временные затраты на захват динамических параметров визуализации составили крайне малые значения (менее 1 мс на кадр) и не оказали заметного влияния на частоту визуализации. Размер полученного scr-файла сценария составил около 370 Кб. На основе созданного файла сценария был выполнен синтез 4K-стереоролика, демонстрирующего изменение формы изоповерхности вытесняющей жидкости. На рис. 5 изображен процесс синтеза 4K-стереоролика в системе визуализации. На рис. 6 изображено проигрывание полученного в результате 4K-стереоролика в Stereoscopic Player в режиме анаглифического стерео.

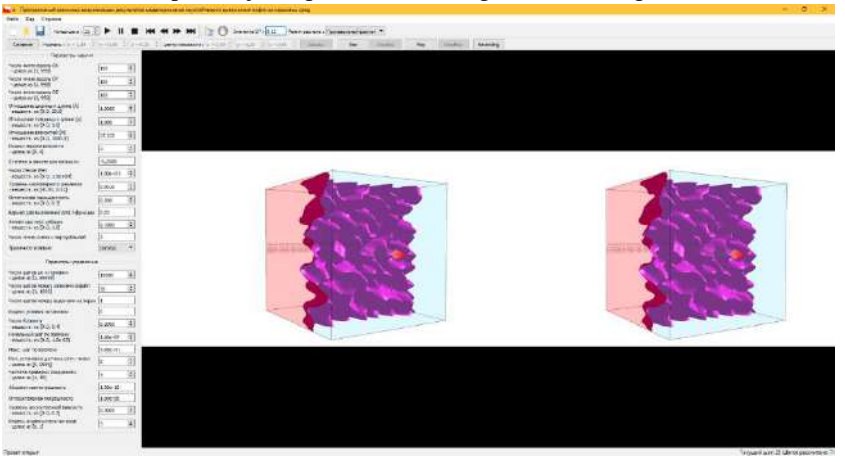


Рис. 5. Синтез 4K-стереоролика в системе визуализации
Fig. 5. Synthesis of 4K-stereoclip in a visualization system

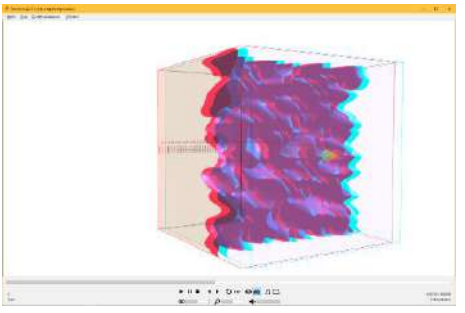


Рис. 6. Проигрывание синтезированного 4K-стереоролика в Stereoscopic Player
Fig. 6. Playing of synthesized 4K-stereoclip in Stereoscopic Player

4. Заключение

В статье рассмотрена задача записи стереовизуализации сложной динамической виртуальной сцены в 4K-стереоролик. Использование внешних программ видеозахвата является проблематичным, т.к. при визуализации каждого кадра заметная часть времени тратится на считывание и кодирование стереопары, что приводит к появлению рывков в стереоролике. В этой связи целесообразно выполнять синтез стереороликов непосредственно в системе визуализации (in situ) на основе захвата значений динамических параметров визуализации.

В данной работе предложена технология отложенного синтеза стереороликов, которая позволяет записывать 4K-стереоролики, не нарушая масштаб реального времени визуализации. Технология включает в себя этап захвата всех неповторяющихся подряд наборов динамических параметров процесса визуализации (сценария) и этап преобразования сценария в стереоролик, который может выполняться уже в офлайн режиме.

В статье предложены методы реализации этих этапов на примере задачи стереовизуализации изоповерхности насыщенности вытесняющей жидкости. Реализация включает в себя разработанный оригинальный формат «scr» файла сценария, основанный на «чанковых» структурах данных, которые дают возможность эффективно расширять и изменять формат, сохраняя обратную совместимость.

В работе описаны созданные эффективные алгоритмы построения и воспроизведения сценариев в системе визуализации, в которых повторяющиеся подряд пакеты значений параметров визуализации просто подсчитываются, а не записываются. Этап преобразования файла сценария в 4K-стереоролик реализуется с помощью технологии внеэкранного рендеринга виртуальной сцены и набора открытых библиотек FFmpeg обработки цифровых видеозаписей. В работе приведено сравнение распространенных медиаконтейнеров, в результате которого по ряду преимуществ выбирается контейнер MP4 и стандарт H.264 сжатия видео, а также описан алгоритм преобразования сценария в стереоролик, в котором реализован визуальный контроль процесса синтеза стереоролика.

Предложенное в статье решение было реализовано в системе визуализации результатов моделирования неустойчивого вытеснения нефти из пористых сред. Был синтезирован 4K-стереоролик, демонстрирующий изменение формы изоповерхности насыщенности вытесняющей жидкости на стадии развития процесса неустойчивого вытеснения нефти. Полученные результаты подтвердили адекватность предложенного решения поставленной задаче и его применимость для виртуальных лабораторий, систем виртуального окружения и научной визуализации, и др.

Список литературы / References

[1]. Puzyrkov D.V., Podryga V.O., Polyakov S.V. Parallel processing and visualization for results of molecular simulation problems. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 2, 2016, pp. 221-242. DOI: 10.15514/ISPRAS-2016-28(2)-15.

[2]. В.И. Гонахчян. Алгоритм удаления невидимых поверхностей на основе программных проверок видимости. Труды ИСП РАН, том 30, вып. 2, 2018 г., стр. 81-98. DOI: 10.15514/ISPRAS-2018-30(2)-5 / Gonakhchyan V.I. Occlusion culling algorithm based on software visibility checks. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue. 2, 2018, pp. 81-98 (in Russian).

[3]. V.F. Nikitin, L.I. Stamov, E.I. Skryleva, V.V. Tyurenkova, M.V. Mikhailyuk. Computer visualization of fluid displacement instability in porous medium. In Proc. of the 68th International Astronautical Congress (IAC 2017), 2017, pp. 1-5.

[4]. Mikhaylyuk M., Timokhin P. Effective GPU-based section visualization in isosurface of saturation of displacing liquid in a porous medium. In Proc. of the 2018 International Conference on Engineering Technologies and Computer Science (EnT 2018), 2018, pp. 57-60.

[5]. И.В. Козлов, Е.И. Скрылева. Математическое моделирование и обработка эксперимента по вытеснению нефти водой из неокомских песчаников. Вестник кибернетики, 2016, № 2, стр. 138-

145 / Kozlov I.V., Skryleva E.I. Mathematical modeling and data processing of water-oil displacement in neocomian sandstone. Proceedings in Cybernetics, 2016, issue 2, pp. 138-145 (in Russian).

[6]. М.В. Михайлюк, А.В. Мальцев, П.Ю. Тимохин. Методы стереовизуализации результатов моделирования неустойчивого вытеснения нефти из пористых сред. Труды НИИСИ РАН, том 8, № 2, 2018, стр. 125-129 / Mikhaylyuk M.V., Maltsev A.V., Timokhin P.Yu. The methods for 3D stereo visualization of data obtained in simulation of unstable oil displacement from porous media. Trudy NIISI RAN/Proc. of SRISA RAS, vol. 8, issue 2, 2018, pp. 125-129 (in Russian).

[7]. Stereoscopic Player. Available at: http://www.3dtv.at/, accessed 25.06.2019.

[8]. Open Broadcaster Software. Available at: https://obsproject.com/, accessed 25.06.2019.

[9]. NVIDIA VIDEO CODEC SDK. Available at: https://developer.nvidia.com/nvidia-video-codec-sdk, accessed 25.06.2019.

[10]. А.Г. Кушниренко, А.В. Мальцев, М.В. Михайлюк, А.А. Прилипко, П.Ю. Тимохин, М.А. Торгашев. Сжатие разделенных видео потоков в задачах дистанционного обучения. Известия Академии инженерных наук им. А.М. Прохорова, 2015, № 2, стр. 3-10 / Kushnirenko A.G., Maltsev A.V., Mikhaylyuk M.V., Prilipko A.A., Timokhin P.Yu., Torgashev M.A. The compression of separated video streams for distance education tasks. News Academy of Engineering Sciences A.M. Prokhorov, 2015, issue 2, pp. 3-10 (in Russian).

[11]. Гук И. Особенности сжатия видеоданных по рекомендации H.264. Компоненты и технологии, 2006, № 2, стр. 1-10 / Guk I. Features of video data compression under H. 264/MPEG 4 part 10 recommendation. Components & Technologies, 2006, issue 2, pp. 1-10 (in Russian).

[12]. FFmpeg. A complete, cross-platform solution to record, convert and stream audio and video. Available at: https://ffmpeg.org/, accessed 25.06.2019.

[13]. Программный комплекс визуализации результатов моделирования неустойчивого вытеснения нефти из пористых сред (ПО «Визуализатор неустойчивого вытеснения»). РФ, № 2019614787, дата заявки 30.04.2019, дата опубликования 13.05.2019. Доступно по ссылке: https://www1.fips.ru/registers-web/action?acName=clickRegister&regName=EVM, дата обращения: 25.06.2019 / Program complex for visualization of simulation results of unstable oil displacement («Vizualizator of unstable oil displacement»). Russian Federation, № 2019614787, request 30.04.2019, publication 13.05.2019 (in Russian). Available at: https://www1.fips.ru/registers-web/action?acName=clickRegister&regName=EVM, accessed 25.06.2019.

Информация об авторах / Information about authors

Петр Юрьевич ТИМОХИН – старший научный сотрудник НИИСИ РАН. Сфера научных интересов: компьютерная графика, визуализация.

Petr Yurievich TIMOKHIN – Senior Researcher of SRISA RAS. Research interests: computer graphics, visualization.

Михаил Васильевич МИХАЙЛЮК – доктор физико-математических наук, профессор, главный научный сотрудник НИИСИ РАН. Сфера научных интересов: компьютерная графика, визуализация, системы виртуального окружения.

Mikhail Vasilievich MIKHAYLYUK – Doctor of Physico-Mathematical Sciences, Professor, Chief Researcher of SRISA RAS. Research interests: computer graphics, visualization, virtual environment systems.

Евгений Михайлович ВОЖЕГОВ – ведущий программист НИИСИ РАН. Сфера научных интересов: компьютерная графика.

Evgeniy Mikhaylovich VOZHEGOV – Leading programmer of SRISA RAS. Research interests: computer graphics.

Клим Денисович ПАНТЕЛЕЙ – ведущий программист НИИСИ РАН. Сфера научных интересов: численное моделирование.

Klim Denisovich PANTELEY – Leading programmer of SRISA RAS. Research interests: numerical simulation.

DOI: 10.15514/ISPRAS-2019-31(4)-5

Применение подхода Fuzzy-DEMATEL при анализе проблем мобильных приложений

М. Панди, ORCID: 0000-0002-4339-6565 <mamta.pandey07@gmail.com>
Р. Литория, ORCID: 0000-0002-7285-422X <litoriya.ratnesh@gmail.com>
П. Панди, ORCID: 0000-0001-5384-6606 <pandeyprat@yahoo.com>
Инженерно-технологический университет Джайпи,
Индия, 473226, Гуна, Рагхогарх, шоссе А-В

Аннотация. Растущая популярность смартфонов привела к появлению большого количества мобильных приложений. Мобильные приложения динамичны по своей природе, поэтому классические подходы к разработке для них не подходят. Индивидуальные потребности пользователей, новые технологии, необходимость снижать энергопотребление и многие другие факторы побуждают разработчиков поставлять на рынок все новые и новые приложения. Однако из-за отсутствия формальных и специализированных для данной области практик разработки с мобильными приложениями связано множество различных проблем. Наличие таких проблем отрицательно сказывается на качестве приложений и на их восприятии конечными пользователями. В этой статье рассматриваются пятнадцать различных проблем, связанных с мобильными приложениями. Мы применили метод fuzzy-DEMATEL для анализа таких критически важных факторов и выделили среди этих факторов группы в соответствии с причинно-следственными связями. Сначала группа экспертов оценивает непосредственные связи между существенными проблемами мобильных приложений. Результаты оценок представляются в виде треугольных нечётких чисел (triangular fuzzy numbers,TFN). На втором шаге в TFN преобразуются лингвистические термины. На третьем шаге при помощи методов DEMATEL выполняется причинно-следственная классификация проблем. Проблемы из группы причин идентифицируются как критически важные в области разработки мобильных приложений. Результаты исследования сравниваются с другими вариантами DEMATEL, такими как G-DEMATEL и E-DEMATEL. Результаты сравнения показывает, что fuzzy-DEMATEL является наиболее подходящим методом анализа взаимосвязей между различными проблемами мобильных приложений. Выводы, содержащиеся в этой работе, могут способствовать эффективной идентификации значимых проблем, на которых должны быть сфокусированы усилия специалистов и менеджеров проектов в индустрии мобильных приложений.

Ключевые слова: мобильные приложения; многокритериальное принятие решений; нечеткий подход DEMATEL

Для цитирования: Панди М., Литория Р., Панди П. Применение подхода Fuzzy-DEMATEL при анализе проблем мобильных приложений. Труды ИСП РАН, том 31, вып. 4, 2019 г., стр. 73-96. DOI: 10.15514/ISPRAS-2019-31(4)-5

Application of Fuzzy DEMATEL approach in analyzing mobile application issues

Mamta Pandey, ORCID: 0000-0002-4339-6565 <mamta.pandey07@gmail.com>
Ratnesh Litoriya, ORCID: 0000-0002-7285-422X <litoriya.ratnesh@gmail.com>
Prateek Pandey, ORCID: 0000-0001-5384-6606 <pandeyprat@yahoo.com>
Jaypee University of Engineering and Technology
A-B Road, Raghogarh, Guna (M.P.), 473226, India

Abstract. In the current scenario, the popularity of smartphones has led to the emergence of an ample collection of mobile applications (apps). Mobile apps are dynamic in nature; therefore, classical software development approaches are not suitable. Individual needs of the customer, new technology, battery consumption, and many more issues force app developers regularly introduce new apps to the market. But due to the unavailability of any formal and customized practices of app development, various issues occur in mobile apps. These issues may adversely affect the application and user acceptance of the end product. In this paper, fifteen issues in mobile apps have been identified. Then we applied Fuzzy-DEMATEL (Decision Making Trial and Evaluation Laboratory) method to analyze the critical mobile issues (CMIs) and divide these issues into cause and effect groups. Firstly, multiple experts evaluate the direct relations of influential issues in mobile apps. The evaluation results are presented in triangular fuzzy numbers (TFN). Secondly, convert the linguistic terms into TFN. Thirdly, based on DEMATEL, the cause-effect classifications of issues are obtained. Finally, the issues in the cause category are identified as CMIs in mobile apps. The outcome of the research is compared with the other variants of DEMATEL like G-DEMATEL and E-DEMATEL and the comparative results suggest that fuzzy-DEMATEL is the most fitting method to analyze the interrelationship of different issues in mobile apps development. The outcome of this work definitely assists the mobile apps development industry to successful identification of the serious issues where professionals and project managers could really focus on.

Keywords: Mobile App; App Issues; Multi-Criteria Decision Making; Fuzzy-DEMATEL

For citation: Pandey M., Litoriya R., Pandey P. Application of Fuzzy DEMATEL approach in analyzing Mobile application issues. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 4, 2019. pp. 73-96 (in Russian). DOI: 10.15514/ISPRAS-2019-31(4)-5

1. Введение

Под «приложением» будем далее понимать прикладное программное обеспечение (ПО), предназначенное для выполнения на портативных устройствах, таких как смартфоны, планшеты и т.п. [1]. Существуют различные бесплатные и платные каналы (Google Play Store, iTunes, Blackberry и др.) для скачивания приложений. Изначально области применения мобильных приложений были ограничены такими сферами, как предоставление информации и повышение продуктивности, например, калькулятор, календарь, электронная почта, контакты. Развитие технологий и растущие запросы пользователей привели к появлению новых категорий приложений, включая такие категории как образование, развлечения, мобильные игры и др.; в настоящее время насчитывается более 33 категорий мобильных приложений [2].

При наличии подключения к интернету приложения предоставляют информацию и контент подобно веб-сайтам, однако некоторые приложения предоставляют сервисы и в отсутствие подключения в интернету, что является одним из важнейших преимуществ мобильных приложений [3]. Ожидается, что, доход рынка мобильных приложений, благодаря их растущей популярности, к 2020 г. достигнет $188.2 миллиардов долларов [4]. Количество пользователей приложений и число скачиваний также увеличилось за последние годы, но при этом недостаточное внимание уделяется количеству удаляемых и некачественных приложений [1]. Наиболее частая причина удаления приложений – неудовлетворенность пользователей. Среди прочих причин можно выделить дефицит свободного места на устройстве и эпизодический характер потребностей в некоторых приложениях.

Мобильные приложения представляют вновь зарождающееся направление в области разработки ПО; однако, если для классического ПО существуют развитые технологии, то область мобильных приложений является относительно новой. По этой причине подходы, применяемые в сфере разработки классического ПО, могут оказаться неподходящими для создания мобильных приложений. Отсутствие формализованных подходов к разработке усугубляют ситуацию и порождает ряд проблем. Эти проблемы отражены в сообщениях пользователей соответствующих платформ. В ряде недавних работ некоторыми исследователями рассмотрены проблемы, отмеченные в отзывах пользователей, которые содержат ценную информацию, например, жалобы, связанные с функциональностью, вопросы защиты персональных данных, запросы на добавление дополнительных возможностей.

Но многие из этих работ, посвященных анализу приложений, не ориентированы на вопросы разработки ПО [5-7]. С другой стороны, выявление причинно-следственных связей между факторами, влияющими на рейтинг приложений, может дать неплохое представление об этой проблеме менеджерам проектов и помочь им в принятии обоснованных решений в условиях неопределенности. В данной статье для выявления причинно-следственных связей используется нечеткий метод DEMATEL (fuzzy-DEMATEL). Этот подход основан на экспертных заключениях.

Из-за ограниченных возможностей экспертов выражать свои суждения или решения количественным образом был применен лингвистический анализ экспертных оценок. С целью преодоления неоднозначности, свойственной лингвистическому анализу, был выбран метод fuzzy-DEMATEL для группового принятия решений. Он позволяет классифицировать факторы, влияющие на качество приложений, в причинно-следственные группы, чтобы помочь менеджерам проектов улучшить качество принимаемых решений. Подходы, основанные на fuzzy-DEMATEL, успешно применялись для решения сложных проблем группового принятия решений, включая стратегическое планирование, эволюцию рынка электронного обучения, исследования и разработки [8].

Согласно литературным источникам, ключевой проблемой является то, что связи между различными факторами в области разработки мобильных приложений недостаточно изучены. В данной работе метод DEMATEL и некоторые его варианты, такие как нечеткий (fuzzy-DEMATEL), «серый» (G-DEMATEL) и основанный на свидетельствах (E-DEMATEL), используются для оценок взаимосвязей между различными проблемами мобильных приложений. Данное исследование будет полезно различным организациям, занимающимся разработкой мобильных приложений. Результаты, полученные при помощи fuzzy-DEMATEL, превосходят результаты, полученные при помощи альтернативных методов. Метод fuzzy-DEMATEL включает восемь шагов. Сначала происходит сбор данных в лингвистической форме, после чего вычисляется матрица непосредственных связей. Результаты вычисляются в форме TFN (Triangular Fuzzy Number), и затем нечеткие значения конвертируются в точные при помощи дефаззификации. Затем при помощи DEMATEL производится классификация причинно-следственных проблем. В данной статье проблемы, относящиеся к категории причин, идентифицируются как критически важные (critical mobile issues, CMI).

Дальнейшее содержание статьи организовано следующим образом. В разд. 2 приводится обзор литературы по мобильным приложениям и связанным с ними вопросам. В разд. 3 описывается применяемая методология. Метод проведения экспериментального исследования описывается в разд. 4. В разд. 5 представлен процесс исследований. Результатам и следствиям посвящен разд. 6. В разд. 7 обсуждаются факторы, влияющие на достоверность результатов исследования. Раздел 8 содержит заключение и обсуждение направлений дальнейших исследований.

2. Обзор публикаций

Этот раздел, посвященный обзору литературы, разделен на два подраздела. В первом подразделе обсуждаются сценарии разработки мобильных приложений в индустрии ПО, позволяющие получить общее представление об этой области. Во втором подразделе рассмотрены работы, посвященные различным аспектам создания мобильных приложений. На основе обзора литературы, а также обсуждений с экспертами выявлены пятнадцать проблем, связанных с мобильными приложениями.

2.1 Программная инженерия и проблемы мобильных приложений

Процесс разработки мобильных приложений во многих отношениях подобен процессу разработки классического ПО, весьма изощренному и включающему различные макро- и микроитерации. Так или иначе, это процесс разработки ПО, и основные этапы те же самые – формулировка требований, проектирование, программирование, тестирование. Тем не менее, классические методики создания ПО не могут быть перенесены в сферу мобильных приложений без существенных изменений [9]. Между классическим ПО и мобильными приложениями имеется множество различий. Некоторые из наиболее важных различий рассмотрены в [10]:

a. мобильные приложения существенно компактнее классического ПО;
b. мобильные приложения могут быть значительно сложнее, в основном, из-за того, что в них активно используются библиотеки сторонних производителей;
c. в мобильных приложениях практически не используется наследование;
d. разработчики часто не поддерживают связь между файлом манифеста Android и исходным кодом;
e. руководства по разработке приложений зачастую игнорируются.

Важные изменения, произошедшие за последнее десятилетие, привели к появлению новых проблем, связанных с разнородностью устройств, растущей квалификацией пользователей, а также с такими общими вопросами, как безопасность, производительность, надежность и ограничения по памяти. Тем не менее, можно видеть, что к мобильным приложениям предъявляется ряд специфических требований, не характерных для классического ПО, таких как возможность взаимодействия с другими приложениями, сенсорное управление, разнообразие приложений, различия платформ, безопасность, специфические пользовательские интерфейсы, сложность тестирования, ограничения по энергопотреблению [11].

В этом разделе обсуждаются исследовательские работы, посвященные проблемам мобильных приложений.

2.2 Проблемы мобильных приложений

Разработка мобильных приложений по существу является частью дисциплины разработки ПО в целом. В контексте создания мобильных приложений на стадии проектирования требований особое значение придается прояснению формулировке требований [12]. Другие фазы проектирования требований практически остаются без изменений для мобильных приложений. Об этом можно судить, например, по работе Крюкова и Демичева [13], в которой представлен сравнительный обзор различных типов децентрализованных хранилищ данных.

Большая часть проблем, возникающих у пользователей, связана с интерфейсом приложений. Поэтому одним из важнейших вопросов при создании мобильного ПО является разработка пользовательского интерфейса. С разработкой интерфейса связан целый ряд проблем, обусловленных такими особенностями мобильных приложений, как сложности ввода данных, малый размер экрана и его низкое разрешение [14].

В существующей литературе из области разработки ПО не уделяется достаточного внимания вопросам качества мобильных приложений, методикам и особенностям разработки, а также специфическим формальным подходам. Например, тестирование мобильных приложений отличается от тестирования обычного ПО.

Однако разработчики применяют общие методы тестирования, не всегда приемлемые для мобильных приложений. В частности, об этом свидетельствуют следующие работы. Армена-Кано и др. [15] обращаются к энергозависимому онлайн-планированию рабочих мест с конфликтом ресурсов. Предложена и представлена оптимизационная модель для распределения ресурсов с концентрацией работы. Черных и др. представили исследование [16], посвященное роли неопределенности в ресурсах облачных вычислений и предоставлении услуг. В нем также говорится о снижении рисков потери информации, отказа в доступе, прерываний в соединениях и утечки информации.

В данной статье представлена структурированная методология анализа, основанная на групповой работе и коллективном принятии решений. Эта методология обеспечивает новое понимание проблемы за счет выявления связей между различными факторами, что позволяет лучше понять причинно-следственные отношения между этими факторами. В качестве технологий анализа мы применяем fuzzy-DEMATEL, а также другие варианты DEMATEL, такие как G-DEMATEL и E-DEMATEL.

DEMATEL является испытанной методологией выявления связей между различными проблемами или факторами в определенном контексте. Этот метод применялся в различных отраслях экономики, таких как управление, проектирование в области механики, химии, разработка программного обеспечения [17]. До настоящего времени проводилось недостаточное число исследований по применению fuzzy-DEMATEL в области разработки мобильных приложений и связанных с этим проблем. Примеры применения методов DEMATEL в области разработки ПО представлены в табл. 1.

Табл. 1. Применение fuzzy-DEMATEL и DEMATEL в области программного обеспечения
Table 1. Application of Fuzzy DEMATEL and DEMATEL in the software field

No.	Авторы	Факторы/Область	Год
1.	Goel S., Nagpal R., Mehrotra D. [18]	Параметры удобства использования мобильных приложений. Взгляд изнутри	2017
2.	Han W.M., Hsu C.H., Yeh C.Y. [19]	Использование DEMATEL для анализа качественных характеристик мобильных приложений	2015
3.	Sugiyanto S., Rochimah S. [20] Roy Bijoyeta, Misra S.K. et al. [21]	Интеграция DEMATEL и ANP методы для вычисления веса характеристики программного обеспечения качества на основе модели ISO 9126. Комплексный подход DEMATEL и AHP для оценки персонала	2013 2012
4.	Wu W.W., Lan L.W., Lee Y.T. [22]	Изучение решающих факторов, влияющих на принятие организации SaaS: социологическое исследование	2011

Гойел, Нагпал и Мехротра [18] применили метод DEMATEL для исследования параметров удобства использования мобильных приложений и выявили взаимосвязи между этими параметрами.

Хан и другие в [19] провели исследование применимости метода DEMATEL для анализа характеристик качества мобильных приложений Основываясь на изучении литературы, они выделили восемь таких характеристик. Они применили вектора значимости и относительные векторы для разбиения характеристик качества на две группы, аналогичные группам причин и следствий. Результаты работы полезны для выявления характеристик качества, оказывающих наибольшее влияние или наиболее зависимых.

Суджиянто и Рошимах [20] интегрировали методы DEMATEL и ANP (Analytic Network Processes) для вычисления весов характеристик качества на основе модели ISO-9126. Результаты оказались полезными для уточнения весов тех подхарактеристик по ISO-9126, которые представляют уровни значимости характеристик и подхарактеристик.

Рой Биджайета, Мисра С.К. и др. [21] рассмотрели различные критерии, такие как опыт, технические навыки и т.п., которые могут потребоваться в различных проектах по разработке ПО, а затем для оценки причинно-следственных связей между критериями и выбором наилучшего критерия применили методологии DEMATEL и AHP.

By B.B., Лан Л.В. и Ли Й.Т. [22] использовали DEMATEL для анализа различных проблем в SAAS (Software As A Service). Результаты их исследований могут стать для поставщиков такого ПО ключом к разработке более эффективных маркетинговых стратегий с целью продвижения бизнеса SaaS.

Наш анализ показывает, что разработка мобильных приложений является относительно новой областью по сравнению с другими формами ПО, такими как классическое ПО или веб-приложения. Качество приложения определяется количеством имеющихся в нем дефектов. Чем меньше дефектов, тем выше качество приложения. Выше мы рассмотрели применение методологии fuzzy-DEMATEL в различных областях и показали, что этот метод хорошо подходит для выявления взаимосвязей между различными проблемами, связанными с мобильными приложениями.

3. Методология

Анализ дефектов мобильных приложений является сложной проблемой, поскольку между этими дефектами существует множество взаимосвязей. Поэтому для анализа дефектов мобильных приложений необходима изощренная многокритериальная методика принятия решений (multi-criteria decision-making, MCDM), позволяющая учитывать конфликтующие цели и компромиссы. Среди различных методик MCDM для проведения исследований наиболее широко применяются интерпретационное структурное моделирование (interpretive structural modeling, ISM) и метод анализа иерархий (analytical hierarchy process, AHP).

Методика DEMATEL превосходит другие многокритериальные методики принятия решений, такие как ISM и AHP, поскольку она позволяет оценить общую степень влияния различных факторов или проблем, выделить причинно-следственные группы и установить причинно-следственные связи [23, 24]. Применение нечеткости в DEMATEL позволяет использовать неточную информацию, которая типична для человеческих суждений. Нечеткий метод DEMATEL использовался в различных областях, таких как управление, информационные технологии, производство [25-27]. В этой статье fuzzy-DEMATEL применяется к анализу проблем разработки мобильных приложений. Мы представим методологию оценки факторов, определяющих успех или неуспех в разработке мобильного приложения. Методология включает следующие четыре шага.

- **Сбор данных** (**Acquisition, A**). Для выявления областей, в которых возможно совершенствование, необходимо собрать данные, касающиеся пользовательского опыта, чтобы можно было применить различные количественные и качественные операции для уточнения деталей.
- **Идентификация (Identification, ID)**. Информация, собранная на шаге A, важна для выявления потенциальных проблем, препятствующих созданию успешного мобильного приложения. Исходя из характера полученной информации, проводится количественный и качественный анализ данных. Возможно также преобразование качественных данных в количественные и обратно.
- **Анализ взаимосвязей (Relationship Analysis, RA).** Число проблем, выделенных на шаге ID, может варьироваться в диапазоне от нескольких единиц до очень больших значений. Мы полагаем, что ни одна из проблем не существует сама по себе, вне связи с другими.

Иначе говоря, каждая проблема может влиять на другие или зависеть от других проблем. Таким образом, важно проанализировать взаимосвязи между проблемами.

- **Интерпретация (Interpretation, I)**. На этом шаге выполняется интерпретация результатов анализа, проведенного на шаге RA.

4. Эксперименты

Для проведения экспериментов, описываемых в этом разделе, применялась методология, описанная в разд. 3.

4.1 Сбор данных и обработка текстовой информации

Мы собрали отзывы о мобильных приложениях от различных пользовательских сообществ, соответствующих различным приложениям. Всего было собрано около 115000 пользовательских отзывов по 31-й категории мобильных приложений, включая такие, как социальные сети, фотография, персонализация. После стадии отбора примерно 40% от общего числа отзывов было признано информативными. Таким образом, для дальнейшего анализа было доступно примерно 45000 обзоров. Затем мы провели обработку текста по некоторым отобранным ключевым словам, используя инструмент прослушивания социальных сетей (social media listening tool) Radian 6, чтобы выявить различные типы пользовательских претензий для разных категорий мобильных приложений. После этого мы кластеризовали близкие по смыслу претензии, чтобы выделить существенные проблемы (дефекты). Ниже приведены детализация примененного похода, а также использованные источники информации и инструменты обработки текста.

i. Сбор отзывов: отзывы были собраны из различных магазинов приложений (Google Play Store, BlackBerry App Store и т.п.).
ii. Фильтрация не английских слов и сленга: были удалены такие слова как “Kool”, “nhi” и т.п.
iii. Обработка текстов: для обработки текстов обзоров были применены приложения POS tagger, word2vec. Например, после фильтрации и обработки были выявлены часто встречающиеся слова *battery, slow, login, heat* и т.д.

4.2 Идентификация проблем

После кластеризации мы заметили, что каждое такое слово можно отнести к одному из 15 типов проблем, перечисленных в табл. 2. Например, слово *request* (запрос) предполагает отсутствие некоторой возможности (запрос дополнительной возможности); аналогично, слово *drain* (разрядка) относится к проблеме энергопотребления. Эти типы проблем согласуются с существующими исследованиями в области разработки мобильных приложений [28, 29].

4.3 Анализ связей между проблемами

Проблемы обычно не возникают изолированно друг от друга; наличие одной проблемы оказывает влияние на другие. Практически невозможно создать приложение, не имеющее ошибок или дефектов. Чтобы создать успешное приложение, необходим некий компромисс, определяемый в зависимости от типа приложения и желаемых целей. Поэтому необходимо понимать взаимосвязи между различными типами проблем.

Метод fuzzy-DEMATEL представляет собой основанный на экспертных заключениях подход к выявлению проблем и других факторов, влияющих на качество систем [8]. Для анализа взаимных влияний различных факторов мы применяем восьмиэтапный подход DEMATEL. Самый первый шаг, рассмотренный ранее в подразделе 4.2, предназначен для идентификации проблем; однако выявленные нами проблемы, как оказалось, соответствуют тем, которые уже

были выявлены ранее в различных исследованиях и представлены в табл. 2 вместе со ссылками на соответствующие источники. Мы применили метод fuzzy-DEMATEL к анализу проблем, связанных с мобильными приложениями, и сравнили результат с другими вариантами – G-DEMATEL и E-DEMATEL. Fuzzy-DEMATEL, G-DEMATEL, и E-DEMATEL описаны в подразделах 4.4, 4.5 и 4.6 соответственно. Применение fuzzy-DEMATEL к выявленным проблемам продемонстрировано в подразделе 5.1.

Табл. 2 Перечень проблем мобильных приложений
Table 2. List of mobile app issues

Проблема	**Описание**	**Ссылка**
Дополнительные расходы (I1)	Пользователи жалуются на скрытые расходы, требуемые для получения доступа ко всем функциям приложения	[30]
Функциональная жалоба (I_2)	Возникновение неожиданного поведения или сбоя приложения	[31]
Жалобы на содержание (I3)	Конкретный контент непригляден или отсутствует	[32]
Сбои (I4)	Приложение часто аварийно завершается	[33-34]
Запрос на удаление функции (I5)	Одна или несколько конкретных функций разрушают приложение	[28]
Запрос на добавление функции (I6)	В приложении необходимо добавить функции	[35]
Проблема совместимости (I7)	У приложения имеются проблемы на конкретном устройстве или версии ОС	[36]
Проблемы при установке (I8)	Сбой происходит в процессе установки	[28]
Проблемы подключения к сети (I9)	У приложения имеются проблемы подключения к сети, например, возникает отставание сети (network lag)	[37-38]
Большой расход ресурсов (I10)	Приложение потребляет слишком электропитания или памяти	[39-42]
Время отклика (I11)	Приложение медленно реагирует на ввод или отстает в целом	[43-45]
Трафик (I12)	Приложение потребляет больше сетевого трафика, чем ожидает пользователь	[46]
Проблема обновлений (I13)	Пользователи считают, что обновления приводят к появлению новых проблем	[47-48]
Пользовательский интерфейс (I14)	Пользователи жалуется на дизайн, элементы управления или визуализацию	[49-50]
Жалоба на безопасность (I15)	Приложение угрожает безопасности собственности пользователя	[51]

4.4 Fuzzy-DEMATEL

Человеческие суждения обычно предвзяты и неточны из-за неполноты информации. Нечеткая теория (fuzzy theory) помогает разрешать затруднения, связанные с неопределенностью, такие как неполнота информации и использование человеческих суждений [52-53]. Нечеткая теория позволяет получать значимые результаты на основе небольших объемов данных. Процесс принятия решений может комбинироваться с применением нечеткой теории, и этот подход используется в ряде областей, таких как управление, выбор производственно-сбытовой цепи [54], индустрия информационных технологий [55]. Применение нечеткой логики для обработки экспертных суждений по крайней мере частично решает проблему неопределенности.

В реальной жизни эксперты часто строят свои заключения, опираясь на имеющийся у них опыт и существующую практику. Экспертные оценки обычно формулируются с использованием лингвистически неоднозначной терминологии [56]. Хороший способ совместить различные взгляды, предположения, мотивы и идеи различных экспертов заключается в том, чтобы преобразовать эти оценки из лингвистической формы в форму нечетких чисел. Таким образом, для исследования сложных дилемм принятия решений в реальном мире в условиях неопределенного окружения может оказаться полезным сочетание нечеткой логики и метода DEMATEL. Этот подход применялся к решению различных проблем исследований и разработок на базе группового принятия решений [8].

Как упоминалось выше, в области создания мобильных приложений было выявлено пятнадцать проблем. Чтобы выявить среди этих проблем группы причин и следствий, был применен метод fuzzy-DEMATEL, являющийся расширением обычного метода DEMATEL.

4.5 Grey DEMATEL (G-DEMATEL)

Grey DEMATEL представляет собой комбинацию теории серых систем (grey system theory) и DEMATEL. Денг предложил математическую теорию серых систем, основанную на понятии серых чисел [57]. Человеческим суждениям свойственна неоднозначность, и серые системы позволяют работать с этими неоднозначностями. Проблемы, связанные с неопределенностью, неточностью и неоднозначностью, могут решаться при помощи серых систем, в особенности в ситуациях многокритериального принятия решений [58]. Серые числа могут быть трансформированы в точные при помощи модифицированного метода преобразования нечетких данных CFCS (Converting Fuzzy data into Crisp Scores).

Серое число обозначается как $\otimes X$. Если $\overline{X}$ – верхняя граница серого числа $\otimes X$, а $\underline{X}$ – его нижняя граница, то

$$\otimes X = [\underline{X}, \overline{X}]. \tag{1}$$

Пусть $\otimes X_{ij}^k$ – это серое значение оценки эксперта k, который определяет влияние фактора i на фактор j. Пусть $\underline{X_{ij}^k}$ и $\overline{X_{ij}^k}$ – нижнее и верхнее значения серого числа $\otimes X_{ij}^k$ соответственно.

Для получения обычного количественного значения (crisp value) требуются следующие действия:

1) нормализация

$$\overline{X_{ij}^k} = \frac{\overline{X_{ij}^k} - min_j \overline{X_{ij}^k}}{\Delta_{min}^{max}}, \tag{2}$$

$$\underline{X_{ij}^k} = \frac{\underline{X_{ij}^k} - min_j \underline{X_{ij}^k}}{\Delta_{min}^{max}},$$

2) вычисление совокупного нормализованного обычного значения

$$Y_{ij}^k = \frac{\underline{X_{ij}^k}\left(1 - \underline{X_{ij}^k}\right) + \overline{X_{ij}^k} \times \overline{X_{ij}^k}}{1 - \underline{X_{ij}^k} + \overline{X_{ij}^k}} \tag{4}$$

3) вычисление окончательного обычного значения

$$Z_{ij}^k = min_j \underline{X_{ij}^k} + Y_{ij\Delta_{min}^{max}}^k \tag{5}$$

4.6 Evidental DEMATEL (E-DEMATEL)

E-DEMATEL это комбинация теории D-чисел и DEMATEL. Теория D-чисел является обобщением теории свидетельств (evidence theory), или теории Демпстера-Шафера

(Dempster–Shafer theory, DST), и этот подход широко применяется для оценки лингвистических данных [59].

Эта теория способствует обработке неточной и недостоверной информации путем присваивания вероятности не отдельным объектам, а подмножествам из более чем одного элемента с последующим применением методов теории вероятностей. Наиболее сложным понятием теории DST являются правила комбинирования, при помощи которых может быть объединена информация из нескольких источников. Этот метод широко применяется во многих областях, таких как объединение информации (information fusion) [60], анализ рисков [61], выбор поставщиков [62], многокритериальное принятие решений [63], задача классификации [64], меры неопределенности [65] и управление конфликтами [66].

Ниже приведено краткое описание этой теории.

Пусть Ω – непустое конечное множество элементов $\{\theta1, \theta2, \theta3 \dots \theta n\}$. Пусть 2^{Ω} – множество всех подмножеств Ω:

$$2^{\Omega} = \{\emptyset, \{\theta1\}, \{\theta2\}, \dots \{\theta n\}, \{\theta1, \theta2\}, \dots \{\theta1, \theta2, \dots, \theta n\}, \dots, \Omega\}.$$

Полное и пустое множество обозначены здесь через Ω и $\emptyset$ соответственно. В DST используется понятие *присваивания базовых вероятностей* (basic probability assignment, BPA) – отображение $m: 2^{\Omega} \to [0,1]$, удовлетворяющее следующим условиям:

$$\sum_{A \subseteq \Omega} m(A) = 1, \tag{6}$$

$$m(\emptyset) = 0. \tag{7}$$

Условием фокальности элемента A является $m(A) > 0$, и набор всех фокальных элементов называется доказательной базой (Body Of Evidence, BOF). Когда доступно несколько независимых BOE, для получения объединенного доказательства используется правило комбинирования Демпстера:

$$m(A) = \frac{\sum_{B,C \subseteq \Omega, B \cap C = A} m_1(B) m_2(C)}{1 - K}, \tag{8}$$

где $K = \sum_{B \cap C = \emptyset} m_1(B) m_2(C)$ – мера конфликта. Если $K < 1$, то правило комбинации значимо, иначе оно бессмысленно.

5. Процесс исследований

Общий ход нашей работы разделен на восемь шагов, которые показаны на рис. 1 и поясняются далее.

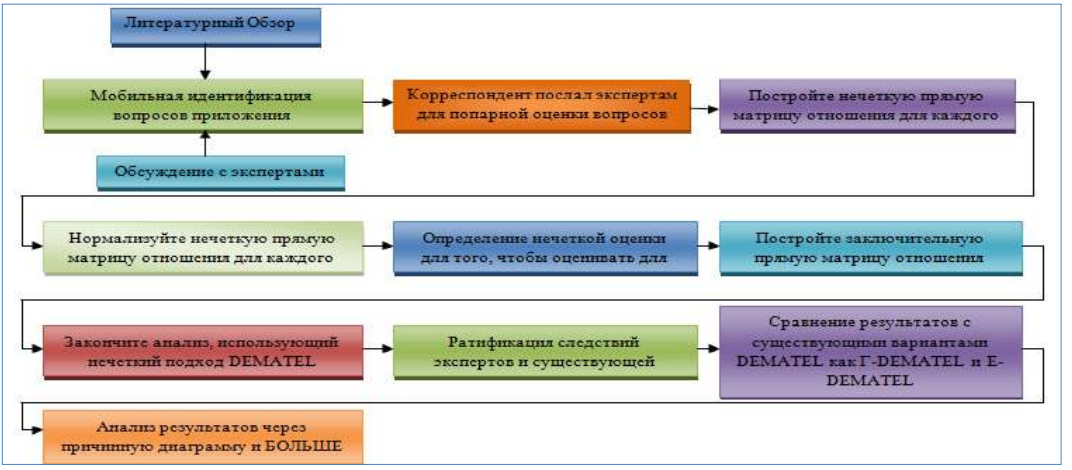


Рис. 1. Предлагаемая схема нашего научного исследования
Fig. 1. Proposed framework of our research study

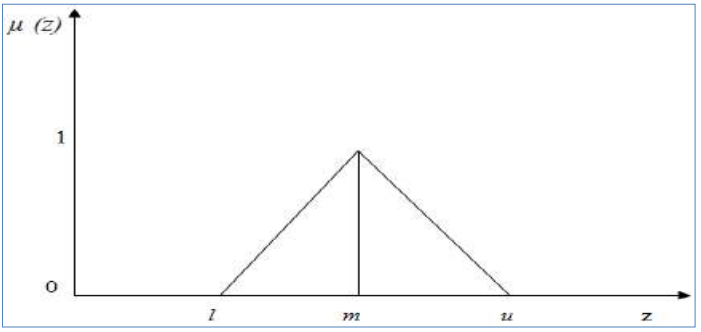


Рис 2. Треугольное нечеткое число $\tilde{z}$
Fig. 2. A triangular fuzzy number $\tilde{z}$

Табл. 3. Представление лингвистического термина и лингвистических значений
Table 3. Representation of linguistic term and linguistic values

Лингвистические термины	**Лингвистические значения**
Very High ifluence (VH),	(0.75, 1, 1)
High ifluence (H)	(0.5, 0.75, 1)
Low ifluence (L)	(0.25,0.5,0.75)
Very Low ifluence (VL)	(0, 0.25, 0.5)
No ifluence (No)	(0, 0, 0.25)

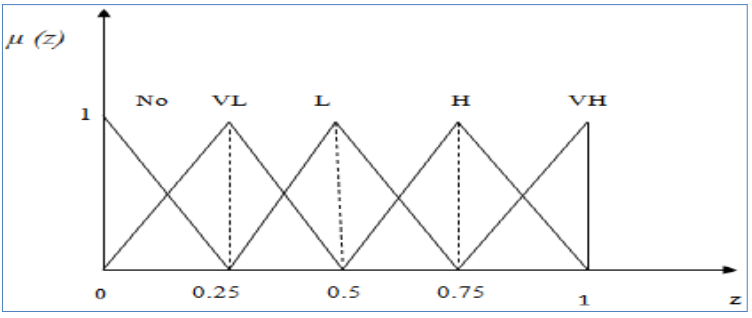


Рис. 3. Представление треугольных нечетких чисел для лингвистических переменных
Fig. 3. Representation of triangular fuzzy numbers in for linguistic variables

Шаг 1: Формирование группы компетентных лиц, принимающих решения, которые имеют опыт решения проблем мобильных приложений.

Группа состоит из 10 экспертов, включающих консультанта в проекте по разработке программного обеспечения, менеджера проекта, старшего разработчика программного обеспечения и т.д. Все имеют техническое образование и большой опыт. В настоящее время все эксперты работают над проектами по разработке мобильных приложений. Группе экспертов предоставляется список вопросов, ответы на которые используются в данном исследовании.

Шаг 2: Построение критериев оценки и разработка нечеткой лингвистической шкалы.

На этом этапе мы определяем различные критерии и степени относительной значимости каждой проблемы, а также представляем их в лингвистических классификационных терминах: *very high, high, low, very low* и *no influence*. Ответы экспертов, преобразованные в нечеткие числа с использованием размытой шкалы, представлены в табл. 3. Использовались треугольные нечеткие числа; треугольное нечеткое число $\tilde{z}$ определяется следующим

образом: $\tilde{z} = (l, m, u)$, где l, m и u действительные числа и $l \leq m \leq u$. Функция принадлежности $\mu_{\tilde{z}}$ определяется так:

$$\mu_{\tilde{z}}(x) = \begin{cases} \dfrac{x-l}{m-l} \text{при } l \leq x \leq m \\ \dfrac{u-x}{u-m} \text{ при } m \leq x \leq u \\ 0 \text{ во всех остальных случаях} \end{cases}$$

Графическое представление треугольного нечеткого числа приведено на рис. 2. Определение лингвистических терминов и лингвистических значений дано в табл. 3. Связи, представленные в форме треугольных нечетких чисел, показаны на рис. 3.

Шаг 3: Составление суждений экспертов, принимающих решения.

Мера связи между различными проблемами сравнивалась в терминах лингвистических значений. Сформируем нечеткие матрицы $\widetilde{z_1}, \widetilde{z_2}, \widetilde{z_3}, \ldots, \widetilde{z_p}$. Треугольные нечеткие числа были сгенерированы в соответствии с суждениями экспертов, участвовавших в принятии решений. Начальную прямую матрицу будем называть нечеткой матрицей $\widetilde{z^k}$.

$$\widetilde{z^k} = \begin{bmatrix} 0 & \widetilde{z_{12}^{(k)}} & \ldots & \widetilde{z_{1n}^{(k)}} \\ \widetilde{z_{21}^{(k)}} & 0 & \ldots & \widetilde{z_{2n}^{(k)}} \\ \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots \\ \widetilde{z_{n1}^{(k)}} & \ldots & \ldots & 0 \end{bmatrix},$$

где $k = 1, 2, 3, \ldots, p$; $\widetilde{z_{ij}^{(k)}} = \left(l_{ij}^{(k)}, m_{ij}^{(k)}, u_{ij}^{(k)}\right)$.

Без ограничения общности $\widetilde{z_{ii}^{(k)}}$ $(i = 1, 2 \ldots n)$ будет рассматриваться как треугольное нечеткое число $\tilde{z} = (0, 0, 0)$, когда это необходимо.

Шаг 4. Анализ нормализованной нечеткой матрицы прямых связей.

Пусть

$$r^k = max_{i=1}^{n}\left(\sum_{j=1}^{n} u_{ij}^k\right). \quad (10)$$

Для преобразования шкалы критериев в шкалу сопоставимых значений использовано линейное преобразование, и нормализованная нечеткая матрица прямых связей, полученная из экспертных оценок, представляется следующим образом:

$$\widetilde{x^k} = \begin{bmatrix} \widetilde{x_{11}^{(k)}} & \widetilde{x_{12}^{(k)}} & \ldots & \widetilde{x_{1n}^{(k)}} \\ \widetilde{x_{21}^{(k)}} & \widetilde{x_{22}^{(k)}} & \ldots & \widetilde{x_{2n}^{(k)}} \\ \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots \\ \widetilde{x_{n1}^{(k)}} & \widetilde{x_{n2}^{(k)}} & \ldots & \widetilde{x_{nn}^{(k)}} \end{bmatrix}, \quad (11)$$

где $k = 1, 2, 3 \ldots p$;

$$\widetilde{x_{ij}^{(k)}} = \frac{\widetilde{z_{ij}^{(k)}}}{r^k} = \left(\frac{l_{ij}^{(k)}}{r^k}, \frac{m_{ij}^{(k)}}{r^k}, \frac{u_{ij}^{(k)}}{r^k}\right). \quad (12)$$

Подобно тому, как это принято в обычном методе DEMATEL, мы считаем, что имеется по крайней мере одно значение i такое, что $\sum_{j=1}^{n} u_{ij}^k < \sum_{j=1}^{n} r^k$.

$\tilde{X}$ обозначает среднюю точку зрения всех лиц, участвовавших в принятии решений:

$$\tilde{X} = \frac{\widetilde{x^1} + \widetilde{x^2} + \cdots + \widetilde{x^p}}{p} \tag{13}$$

$$\tilde{X} = \begin{bmatrix} \widetilde{X_{11}} & \widetilde{X_{12}} & \dots & \widetilde{X_{1n}} \\ \widetilde{X_{21}} & \widetilde{X_{22}} & \dots & \widetilde{X_{2n}} \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ \widetilde{X_{n1}} & \widetilde{X_{n2}} \dots & \dots & \widetilde{X_{nn}} \end{bmatrix}, \tag{14}$$

где $\widetilde{X_{ij}} = \frac{\sum_{k=1}^{p} \widetilde{x_{ij}^{(k)}}}{p}$.

Шаг 5: Для вычисления общей нечеткой матрицы связей $\tilde{T}$ была введена и проанализирована структурная модель. Известно, что $lim_{w\to\infty} X^w = \Theta$, где Θ – нулевая матрица. Кроме того, известно, что $lim_{w\to\infty}\left(\mathrm{I} + \tilde{X} + \widetilde{X^2} \dots + \widetilde{X^w}\right) = X\left(\mathrm{I} - \tilde{X}\right)^{-1}$. Оба эти соотношения доказаны в [8]. Последняя матрица и является общей нечеткой матрицей связей $\tilde{T}$.

Шаг 6: Вычисляется общая матрица связей $\tilde{T}$. Также вычисляются вектора значимости и относительного положения.

Шаг 7: На этом шаге все нечеткие числа переводятся в точные значения. Для этого используется следующий вариант метода CFCS. Пусть имеются треугольные нечеткие числа $\widetilde{N_k} = (l_k, m_k, u_k);\ k = 1,2, \dots, n.$ Пусть $L = min(l_k), R = max(u_k), \Delta = R - L.$ Тогда обычное значение $\widetilde{N_k}$ вычисляется по следующей формуле:

$$\tilde{N}_k^{def} = L + \Delta \times \frac{(m_k - L)(\Delta + u_k - m_k)^2(R - l_k) + (u_k - L)^2(\Delta + m_k - l_k)^2}{(\Delta + m_k - l_k)(\Delta + u_k - m_k)^2(R - l_k) + (u_k - L)(\Delta + m_k - l_k)^2(\Delta + u_k - m_k)}. \tag{15}$$

Шаг 8: Наконец, по результатам вычислений шага 7 выводятся диаграммы причин и следствий.

5.1 Применение fuzzy-DEMATEL к анализу проблем мобильных приложений

Теперь применим шаги описанной процедуры к анализу взаимосвязей между различными проблемами мобильных приложений.

Шаг 1: Выбор группы экспертов в области разработке мобильных приложений, включая разработчиков, менеджеров проектов и т. д.

Шаг 2: Построение критериев оценки и разработка нечеткой лингвистической шкалы.

На этом этапе разная степень влияния одного фактора на другую проблему представлена в форме пяти лингвистических терминов: *very high*, *high*, *low*, *very low* и *no influence*; соответствующие треугольные нечеткие числа показаны в табл. 3 и на рис. 3. Оценка по мнению первого эксперта в форме языковых терминов показана в таблице 4.

Шаг 3: Получение экспертных оценок от членов группы.

Измерение взаимосвязей между различными проблемами $I_i || i = 1, 2, \dots, 15$, число лиц, участвующих в принятии решений – 10. Итого , получено 10 нечетких метрик $\widetilde{z^k}$ и соответствующих матриц нечетких треугольных чисел.

Шаг 4: На этом шаге строится нормализованная нечеткая матрица прямых связей. Каждая нечеткая матрица прямых связей $\widetilde{x^k}$ вычисляется с использованием нечетких треугольных чисел $\widetilde{z_{ij}^{(k)}}$ из матрицы $\widetilde{z^k}$ на основе соотношений (10) и (11). Суммарная нормализованная нечеткая матрица прямых связей $\tilde{X}$ рассчитывается на основе соотношений (12) и (13).

Шаг 5: Вычисляется общая нечеткая матрица связей $\tilde{T}$.

Шаг 6: На этом шаге вычисляются сумма строк и сумма столбцов общей матрицы связей $\tilde{T}$. Сумма строк обозначается через $\widetilde{D}_\iota$, а сумма столбцов через $\widetilde{R}_\iota$. В табл. 5 показаны результаты для $(\widetilde{D}_\iota + \widetilde{R}_\iota)$ и $(\widetilde{D}_\iota - \widetilde{R}_\iota)$.

Шаг 7: Теперь, в соответствии с формулой (15) выполняется процесс дефаззификации для векторов значимости и относительного положения. Преобразование дефаззификации показано в табл. 6, а связи между различными проблемами показаны ниже в виде диаграммы на рис. 4.

Табл. 4. Прямое влияние каждой проблемы на другие проблемы: оценка первого эксперта в форме лингвистических терминов
Table 4. Direct influence of each issue on the other issues: The assessment of the first expert opinion by linguistic term

Mobile app issues	**I1**	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**
I1	NO	NO	L	NO	VH	VL	L
I2	NO	NO	VL	H	VL	NO	VL
I3	NO	L	NO	H	VL	NO	VL
I4	NO	L	L	NO	NO	H	H
I5	VH	H	L	L	NO	H	H
I6	VH	L	H	L	NO	NO	H
I7	L	L	NO	L	L	L	NO
I8	NO	L	NO	VL	NO	NO	NO
I9	NO	L	NO	L	NO	NO	L
I10	H	NO	H	NO	VH	L	L
I11	NO	H	H	VH	H	H	L
I12	VL	VL	NO	H	H	L	H
I13	NO	VL	VL	NO	NO	NO	L
I14	NO	NO	L	NO	L	NO	VL
I15	H	NO	NO	NO	VL	H	VL

Mobile app issues	**I_8**	**I_9**	**I_{10}**	**I_{11}**	**I_{12}**	**I_{13}**	**I_{14}**	**I_{15}**
I1	NO	NO	NO	NO	NO	NO	NO	NO
I2	VH	L	NO	L	L	NO	NO	NO
I3	VH	L	NO	L	L	NO	NO	NO
I4	VL	NO	VL	VL	L	L	VL	NO
I5	NO	NO	VH	L	VL	VL	L	L
I6	NO	NO	VH	VH	L	L	NO	L
I7	NO	L	L	L	NO	VH	L	L
I8	NO	H	VH	NO	L	H	NO	NO
I9	L	NO	VH	VH	VH	VH	NO	NO
I10	VL	NO	NO	NO	H	VH	L	NO
I11	VL	VH	VH	NO	VH	VH	H	L
I12	L	VH	VH	VH	NO	NO	NO	L
I13	VL	NO	VH	L	NO	NO	NO	NO
I14	NO	NO	H	H	L	NO	NO	NO
I15	NO	NO	VH	VH	H	NO	VH	NO

Табл. 5. Суммарные значения $\widetilde{D}_i$, $\widetilde{R}_i$, вектор значимости и вектор относительного положения
Table 5. The total amounts of $\widetilde{D}_i$, $\widetilde{R}_i$, prominence and relative vectors

Mobile app issues	$\widetilde{D}_i$	$\widetilde{R}_i$	**Prominence Vector** $(\widetilde{D}_i + \widetilde{R}_i)$	**Relative Vector** $(\widetilde{D}_i - \widetilde{R}_i)$ **(positive side)**	**Relative Vector** $-(\widetilde{D}_i - \widetilde{R}_i)$ **(negative side)**
I_1	(.527,.764,1.59)	(.839,.462,.421)	(.366,1.22,2.011)	(.312,.302,1.169)	(.312,.302,1.169)
I_2	(.287,.456,1.739)	(.301,.6,.935)	(.588,1.056,2.674)	(-.023,.144,.804)	(.023,.144,-.804)
I_3	(.438,.71,1.274)	(.285,1.07,1.163)	(.723,1.78, 2.473)	(.153,-.36,.111)	(-.153,.36,-.111)
I_4	(.768,1.476,.27)	(.423,.305,.743)	(1.191,1.78,.97)	(.345,1.171,.516)	(.345,1.171,.516)
I_5	(.508,.766,.15)	(.417,.541,.456)	(.925,1.307,.607)	(.091,.225,-.305)	(-.091,.225,.305)
I_6	(.516,.865,2.13)	(.381,.403,.322)	(.897,1.268,2.45)	(.135,.462,1.808)	(.135,.462,1.808)
I_7	(.447,.256,.319)	(.418,.265,.322)	(.865,.521,.641)	(.029,.009,.003)	(-.029,.009,.003)
I_8	(.675,.543,1.25)	(.454,.438,.459)	(1.129,.981,1.709)	(.221,.105,.791)	(-.221,.105,.791)
I_9	(.813,.735,.359)	(1.203,.684,.92)	(2.016,1.419,1.279)	(-.39,.051,-.561)	(.39,-.051,.561)
I_{10}	(.412,.761,.352)	(.91,.748,1.466)	(1.322,1.509,1.818)	(.498,.013,1.114)	(.498,.013,1.114)
I_{11}	(.612,.413,.521)	(.268,.435,.431)	(.88,.848,.952)	(.344,-.022,.09)	(-.344,.022,-.09)
I_{12}	(.517,.652,1.29)	(.401,.499,.546)	(.918,1.151,1.836)	(.116,.153,.744)	(-.116,.153,.744)
I_{13}	(.378,.376,2.58)	(.463,1.093,1.44)	(.841,1.46,4.022)	(.085,.726,1.138)	(.085,.726,1.138)
I_{14}	(.329,.673,.557)	(1.087,.827,1.23)	(1.156,1.5,1.787)	(.416,1.5,1.787)	(-.416,1.5,1.787)
I_{15}	(.218,.564,1.32)	(1.344,2.158,3.52)	(.562,2.722,4.84)	(-.126,1.594,2.2)	(.126,1.594,2.2)

Табл. 6. Результат дефаззификации векторов значимости и относительного положения
Table 6. The defuzzification of prominence and relative vector

Mobile app issues	Prominence Vector $(\widetilde{D}_i + \widetilde{R}_i)^{def}$	Relative Vector $(\widetilde{D}_i - \widetilde{R}_i)^{def}$
I_1	2.12	-0.5
I_2	3.10	2.98
I_3	2.81	-0.7
I_4	1.61	-1.2
I_5	1.68	-1.0
I_6	2.81	2.75
I_7	2.64	-0.97
I_8	1.91	-0.68
I_9	2.40	2.71
I_{10}	2.19	2.29
I_{11}	1.35	-0.62
I_{12}	1.41	-0.5
I_{13}	2.10	2.2
I_{14}	2.23	2.6
I_{15}	1.69	2.19

Табл. 7(a). Рейтинги причинно-следственных проблем
Table 7(a). Rank factors of cause and effect issues

Категория	**Проблема**	**Ранг** $(\widetilde{D_i} - \widetilde{R_i})^{def}$
Причина	I_2 Возникновение неожиданного поведения или сбоя приложения	2.98
	I_6 В приложении необходимо добавить функции	2.75
	I_9 У приложения имеются проблемы подключения к сети, например, возникает отставание сети	2.71
	I_{14} Пользователи жалуются на дизайн, элементы управления или визуализацию	2.6
	I_{10} Приложение потребляет слишком электропитания или памяти	2.29
	I_{13} Пользователи считают, что обновления приводят к появлению новых проблем	2.2
	I_{15} Приложение угрожает безопасности собственности пользователя	2.19
Следствие	I_1 Пользователи жалуются на скрытые расходы, требуемые для получения доступа ко всем функциям приложения	-0.5
	I_{12} Приложение потребляет больше сетевого трафика, чем ожидает пользователь	-0.5
	I_{11} Приложение медленно реагирует на ввод или отстает в целом	-0.62
	I_8 Сбой происходит в процессе установки	-0.68
	I_3 Конкретный контент непригляден или отсутствует	-0.7
	I_7 У приложения имеются проблемы на конкретном устройстве или версии ОС	-0.97
	I_5 Одна или несколько конкретных функций разрушают приложение	-1.0
	I_4 Приложение часто аварийно завершается	-1.2

Табл. 7(b). Причинно-следственная классификация и ранжирование проблем с использованием fuzzy-DEMATEL, G-DEMATEL и E-DEMATEL
Table 7(b). Cause-effect classification and ranking of issues using fuzzy-DEMATEL, G-DEMATEL and E-DEMATEL

Категория	**Проблемы с мобильным приложением и рейтинг**		
	fuzzy-DEMATEL	*G-DEMATEL*	*E-DEMATEL*
Причина	I_2	I_2	I_2
	I_6	I_6	I_6
	I_9	I_9	I_9
	I_{14}	I_{14}	I_{14}
	I_{10}	I_{10}	I_{10}
	I_{13}	I_{13}	I_{13}
	I_{15}	I_{15}	I_{15}
Следствие	I_1	I_1	I_7
	I_{12}	I_{12}	I_8
	I_{11}	I_{11}	I_1
	I_8	I_8	I_{11}
	I_3	I_3	I_3
	I_7	I_7	I_{12}
	I_5	I_5	I_5
	I_4	I_4	I_4

5.2 Оценка

Оценка результатов была выполнена на основе абсолютного среднего отклонения (mean absolute error, MAE); это обеспечивает количественную оценку сходства важности проблем на основе векторов значимости и относительного положения. MAE позволяет учитывать неопределенность и нечеткость данных и вычисляется следующим образом:

$$MAE = \frac{1}{N}\sum_{i=1}^{N}\left|V_y^i - V_n^i\right|, \quad (16)$$

где N – число влияющих проблем, V_y^i обозначает значение положительной стороны проблемы i, V_n^i – значение отрицательной стороны проблемы i. Значения MAE для проблем, вычисленное с положительной и отрицательной стороны, представлено в табл. 8.

Табл. 8. MAE важности проблемы, вычисленная с положительной и отрицательной стороны
Table 8. The MAE of issue importance calculated from positive side and negative side

	G-DEMATEL	E-DEMATEL	fuzzy-DEMATEL
MAE	0.28	0.33	0.14

Меньшие значения MAE указывают на то, что важность проблем, вычисленная с положительной и отрицательной стороны, лучше соответствуют реальности. По этим оценкам fuzzy-DEMATEL оказывается более разумным подходом, чем G-DEMATEL и E-DEMATEL с точки зрения выявления связей между проблемами мобильных приложений, являющимися причинами и следствиями.

Одним словом, по сравнению с большинством существующих методов fuzzy-DEMATEL в большей степени обладает возможностью способствовать улучшению качества мобильных приложений. В сравнении с G-DEMATEL и E-DEMATEL, fuzzy-DEMATEL лучше справляется с проблемами субъективности оценок в лингвистической форме и более приспособлен для выявления CMI в мобильных приложениях, так как этот процесс по своей сути связан с обработкой лингвистических оценок.

5.3 Сравнительный анализ нашего метода и других вариантов метода DEMATEL

В обзоре существующей литературы (разд. 2) показано, что существует целый ряд методов для анализа проблем мобильных приложений. Тем не менее, лишь немногие из них принимают во внимание связи между проблемами мобильных приложений и позволяют формулировать критерии для оптимального выбора проблем, которые необходимо решать. В сравнении с другими существующими методами, fuzzy-DEMATEL позволяет идентифицировать проблемы мобильных приложений и получать общее связи ними. На основе fuzzy-DEMATEL можно существенно упростить процесс оптимизации мобильного приложения, рассматривая его как процесс оптимизации отдельных проблемных ситуаций.

Табл. 9. Сравнение G-DEMATEL, E-DEMATEL и нечеткого DEMATEL (F-DEMATEL) при проведении лингвистических оценок
Table 9. Comparison of G-DEMATEL, E-DEMATEL and fuzzy DEMATEL in the linguistic assessment

	G-DEMATEL	**E-DEMATEL**	**fuzzy--DEMATEL**
Нечеткость лингвистической шкалы	×	✓	✓
Субъективность экспертной оценки	✓	✓	×
«×» означает, что метод не предусматривает возможности решать проблему, «✓» означает, что метод может решить проблему			

Был также проведен сравнительный анализ методов G-DEMATEL, E-DEMATEL и fuzzy-DEMATEL, результаты которого представлены в табл. 9.

- **Fuzzy-DEMATEL в сравнении с G-DEMATEL:** Нечеткие системы идеально подходят для описания лингвистических явлений, в то время как в «серой» системе возможны неверное понимание или неправильная интерпретация.
- **Нечеткий DEMATEL в сравнении с E-DEMATEL:** Хотя оба метода позволяют работать в условиях субъективности экспертных оценок, теория свидетельств не очень хорошо подходит для оценок в лингвистической форме из-за наличия гипотезы о том, что все элементы области суждений должны быть взаимно исключающими.

Таким образом, в сравнении с G-DEMATEL и E-DEMATEL, fuzzy-DEMATEL является более подходящим методом для идентификации проблем мобильных приложений, поскольку этот процесс существенно основан на оценках, представленных в лингвистической форме.

Классификация проблем, являющихся причинами и следствиями, в соответствии с оценками по методам fuzzy-DEMATEL, G-DEMATEL и E-DEMATEL, представлена в табл. 7 (a) и (b). Из этих таблиц можно видеть, что CMI, идентифицированные при помощи fuzzy-DEMATEL, G-DEMATEL и E-DEMATEL одни и те же и включают I_2, I_6, I_9, I_{14}, I_{10}, I_{13} и I_{15}.

Табл. 10. Попарное сравнение рейтинга сходства G-DEMATEL, E-DEMATEL и F-DEMATEL с использованием коэффициента корреляции Спирмена
Table 10. Pair-wise comparison of the importance of G-DEMATEL, E-DEMATEL, and F-DEMATEL using the Spearman correlation coefficient

	G-DEMATEL	**E-DEMATEL**	**fuzzy--DEMATEL**
G-DEMATEL	1	0.7808	0.8083
E-DEMATEL	0.7608	1	0.7435
fuzzy--DEMATEL	0.8964	0.8709	1

Кроме того, для того чтобы точно отразить соответствие векторов значимости и относительного положения, полученных при помощи этих методов, в табл. 10 представлены коэффициенты корреляции Спирмена (Spearman correlation coefficient), вычисленные для каждой пары методов. Таблица показывает, что рейтинг сходства для fuzzy-DEMATEL выше, чем для двух остальных методов, поскольку более высокое значение коэффициента корреляции Спирмена означает более высокий рейтинг сходства.

6. Результаты и следствия

Результаты этого анализа показывают наличие сложных взаимозависимостей между различными проблемами и сложность мобильных приложений. Эти результаты также показывают, что различные виды проблем могут быть охарактеризованы как причины и следствия на основе исследования их влияния и чувствительности к влиянию других проблем. Разработчик мобильного приложения может использовать эти результаты, чтобы решить, какие проблемы требуют безотлагательного внимания, а какие могут и подождать. Результаты анализа могут быть критически важны для повышения качества мобильных приложений и уровня их популярности у пользователей.

Хотя эта статья не дает прямых рекомендаций по разработке мобильных приложений, разработчики могут использовать представленные результаты для совершенствования процесса разработки. Например, можно провести анализ корреляции между типами проблем и рейтингом приложений, чтобы составить представление об относительной значимости отдельных проблем.

На рис. 4 (a) показаны различные типы проблем и их влияние на рейтинг приложений. Недостаток функциональности (I_2) оказывается наиболее значимой проблемой с влиянием 2.98, в то время как аварийное завершение является наиболее значимой проблемой с

влиянием -1.2. Более высокий ранг проблемы показывает ее влияние, низкий ранг отражает высокое влияние проблемы на качество приложений.

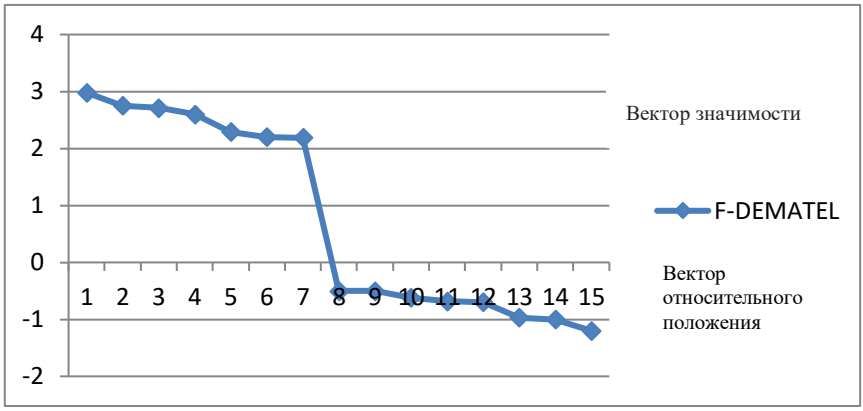
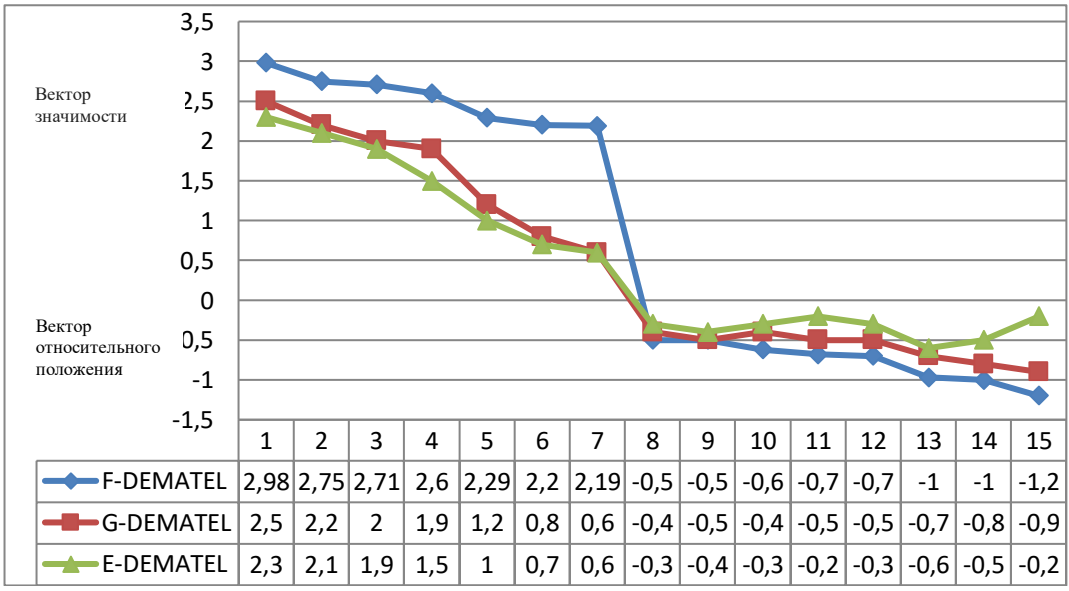


Рис. 4(a). Диаграмма причинно-следственного порядка проблем мобильных приложений при использовании fuzzy-DEMATEL
Fig. 4(a). Cause-effect order diagram of mobile app issues using fuzzy-DEMATEL

Сравнительный анализ методов G-DEMATEL, E-DEMATEL и fuzzy-DEMATEL представлен на рис. 4(b). Наилучшим из этих трех подходов оказался fuzzy-DEMATEL. Положительные значения представляют вектор значимости, а отрицательные – вектор относительного положения.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
F-DEMATEL	2,98	2,75	2,71	2,6	2,29	2,2	2,19	-0,5	-0,5	-0,6	-0,7	-0,7	-1	-1	-1,2
G-DEMATEL	2,5	2,2	2	1,9	1,2	0,8	0,6	-0,4	-0,5	-0,4	-0,5	-0,5	-0,7	-0,8	-0,9
E-DEMATEL	2,3	2,1	1,9	1,5	1	0,7	0,6	-0,3	-0,4	-0,3	-0,2	-0,3	-0,6	-0,5	-0,2

Рис. 4 (b). Сравнительная диаграмма причинно-следственного порядка проблем мобильных приложений
Figure 4 (b). Comparative cause-effect order diagram of mobile app issues

Имеется ряд предшествующих исследований различных проблем, связанных с мобильными приложениями. Для успешной разработки мобильных приложений необходимо проанализировать значимость различных проблем мобильных приложений и взаимосвязи между ними. Взаимосвязи между проблемами мобильных приложений могут быть

критичными. Ранее решалась задача количественной оценки влияния различных проблем мобильных приложений [67]. Однако задача причинно-следственных зависимостей между различными проблемами мобильных приложений не была исследована. Цель данной работы – восполнить указанный пробел и представить соответствующую диаграмму. На диаграмме причинно-следственных отношений показаны важные взаимосвязи, что может помочь менеджерам проектов в принятии решений, касающихся разработки мобильных приложений, дать более наглядное представление процесса разработки и обеспечить максимально быстрый выпуск новых приложений на рынок.

Например, из диаграммы видно, что жалобы на недостатки функциональности (I_2), имеющие максимальное значение $\left(\widetilde{\boldsymbol{D}_i} + \widetilde{\boldsymbol{R}_i}\right)^{def}$, являются наиболее значимым причинным фактором для разработки мобильных приложений. С другой стороны, $\left(\widetilde{\boldsymbol{D}_i} - \widetilde{\boldsymbol{R}_i}\right)^{def}$ для проблемы совместимости (I_7) имеет максимальное по абсолютной величине отрицательное значение, а значит, также является наиболее значимым фактором в группе следствий. Положительное значение показателя значимости указывает, что соответствующая проблема принадлежит к группе причин, а отрицательные значения указывают, что проблема принадлежит к группе следствий.

7. Угрозы валидности

Данное исследование имеет ряд ограничений. В частности, здесь рассмотрено только 15 проблем; в дальнейшем к ним могут быть добавлены и другие. Применение методов многокритериального принятия решений, таких как ANP и нечеткий ANP в сочетании с результатами данной работы может оказать более подходящим подходом для анализа проблем, связанных с мобильными приложениями.

8. Выводы и направления будущей работы

Разработка мобильных приложений была выделена как отдельная область разработки программного обеспечения в целом. Мобильные приложения в настоящее время являются насущной потребностью бизнеса, но из-за недостаточной развитости формальных научных методов разработки этого класса приложений, возникает множество проблем. В отличие от предшественников – веб-приложений и приложений для настольных компьютеров – процесс разработки мобильных приложений зачастую носит «любительский» характер и находится в фазе эволюции.

Данное исследование является попыткой выделить важнейшие проблемы, связанные с мобильными приложениями. Предложенный подход может быть включен в фазу планирования при разработке любого мобильного приложения, чтобы при принятии стратегических решений можно было сверяться с задачами обеспечения качества приложения. Настоящее исследование может помочь компаниям, занимающимся разработкой ПО, в выделении ключевых проблем, связанных с мобильными приложениями.

В этой работе было рассмотрено пятнадцать проблем, связанных с мобильными приложениями, а именно: *дополнительные затраты (I_1), недостатки функциональности (I_2), жалобы на контент (I_3), аварийное завершение (I_4), удаление ранее имевшихся возможностей (I_5), запросы на новые возможности (I_6), проблемы совместимости (I_7), проблемы установки (I_8), сетевое соединение (I_9), ресурсоемкость (I_{10}), время отклика (I_{11}), избыточный трафик (I_{12}), обновления (I_{13}), пользовательский интерфейс (I_{14}) и безопасность (I_{15}).* После выявления указанных проблем для разделения их на группу причин и группу следствий был применен метод fuzzy-DEMATEL. *Недостатки функциональности, запросы на новые возможности, сетевое соединение, пользовательский интерфейс, ресурсоемкость, обновления и безопасность* вошли в группу причин. Остальные восемь проблем, включая *дополнительные затраты, жалобы на контент, аварийное завершение, запросы на новые*

возможности, проблемы совместимости, проблемы установки, время отклика и *избыточность трафика* входят в группу следствий.

В сравнении с аналогичными подходами, такими как G-DEMATEL В E-DEMATEL, fuzzy-DEMATEL, дает лучшие результаты. Можно видеть, что хотя результаты были выведены из векторов значимости и относительного положения, эти два числовых значения имеют очевидные отличия. В качестве критерия для оценки результатов использовалась средняя абсолютная погрешность.

Результаты показывают, что «недостатки функциональности» являются наиболее важной проблемой мобильных приложений, в то время как «проблемы совместимости», являются в значительной мере следствием других проблем.

Поскольку исследования и полученные результаты основаны на коллективном опыте экспертов в области разработки ПО, увеличение объема этого опыта поможет снять эффект предвзятости. Дальнейшие исследования могут быть направлены на идентификацию большего числа проблем, связанных с мобильными приложениями, и изучение их влияния на уже выявленные причинно-следственные связи.

References

[1]. Inukollu V.N., Keshamoni D.D., Kang T., and Inukollu M. Factors Influencing Quality of Mobile Apps: Role of Mobile App Development Life Cycle. International Journal of Software Engineering and Applications, vol. 5, no. 5, 2014, pp. 15-34.

[2]. Pandey M., Litoriya R., Pandey P. Perception-Based Classification of Mobile Apps: A Critical Review. In Smart Computational Strategies: Theoretical and Practical Aspects. Springer, Singapore, 2019, pp. 121-133.

[3]. Jason Summerfield. Mobile Website vs. Mobile App: Which is Best for Your Organization? Available at: https://www.hswsolutions.com/services/mobile-web-development/mobile-website-vs-apps/.

[4]. Ashishdeep A., Bhatia J., Varma K. A software engineering model for mobile app development. International Journal of Computer Science and Communication, vol. 7, no. 1, 2016, pp. 150-153.

[5]. Pandey M., Litoriya R., Pandey P. Mobile applications in context of big data: A survey. In Proc. of the Symposium on Colossal Data Analysis and Networking (CDAN), 2016.

[6]. McIlroy S., Ali N., Hassan A.E. Fresh apps: an empirical study of frequently-updated mobile apps in the Google play store. Empirical Software Engineering, vol. 21, issue 3, 2016, pp.1346-1370.

[7]. Pandey, M., Litoriya, R., Pandey, P. Mobile APP development based on agility function. Ingénierie des Systèmes d'Information, vol. 23, no. 6, 2018, pp. 19-44.

[8]. Lin C.J., Wu W.W. A causal analytical method for group decision-making: Under fuzzy environment. Expert Systems with Applications, vol. 34, issue 1, 2007, pp. 205-213.

[9]. Litoriya, R., Kothari, A. (2013). Cost Estimation of web projects in context with Agile paradigm: Improvements and validation. International Journal of Software Engineering (A Publication of Software Engineering Competence Center - Egypt), 6(2), 91-114.

[10]. Minelli R. and Lanza M. Software analytics for mobile applications - Insights & Lessons Learned. In Proc. of the 17th European Conference on Software Maintenance and Reengineering, 2013, pp. 144-153.

[11]. Stepanova E., Kirikova M. Continuous requirements engineering for mobile application development. In Joint Proceedings of Workshops, Doctoral Symposium, Research Method Track, and Poster Track co-located with the 23rd International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2017), 2017.

[12]. Seyff N., Graf F. User-driven requirements engineering for mobile social software. In Software Engineering (Workshops), 2010, pp. 503-512.

[13]. Kryukov A.P., Demichev A.P. Decentralized Data Storages: Technologies of Construction. Programming and Computer Software, vol. 44, no. 5, 2018, pp. 303-315.

[14]. Kaufman J. Principles of Mobile App Design. Apteligent White Paper, 2016, 20 p. Available at: https://www.apteligent.com/wp-content/uploads/2016/07/PRINCIPLES-MOBILE-APP-DESIGN-WP.pdf.

[15]. Armenta-Cano F.A., Tchernykh A., Cortés-Mendoza J.M., Yahyapour R., Drozdov A.Y., Bouvry P., Nesmachnow S. Min_c: Heterogeneous concentration policy for energy-aware scheduling of jobs with resource contention. Programming and Computer Software, vol. 43, no. 3, 2017, pp. 204-215.

[16]. Tchernykh A., Schwiegelsohn U., Talbi E.G., Babenk M. Towards understanding uncertainty in cloud computing with risks of confidentiality, integrity, and availability. Journal of Computational Science. Available online 22 November 2016, DOI: 10.1016/j.jocs.2016.11.011.
[17]. Mavi R.K., Standing C. Critical success factors of sustainable project management in construction: A fuzzy DEMATEL-ANP approach. Journal of Cleaner Production, vol. 194, no. 1, 2018, pp. 751-765.
[18]. Goel S., Nagpal R., Mehrotra D. Mobile applications usability parameters: Taking an insight view. In Proc. of the International conference on Information and Communication Technology for Sustainable Development, 2018, pp. 35-43.
[19]. Han W.M., Hsu C.H., Yeh C.Y. Using DEMATEL to analyze the quality characteristics of mobile applications. In Proc. of the International Conference on Future Information Engineering and Manufacturing Science, 2014, pp. 131-134.
[20]. Sugiyanto S., Rochimah S. Integration of DEMATEL and ANP methods for calculate the weight of characteristics software quality based model ISO 9126. In Proc. of the International Conference on Information Technology and Electrical Engineering, 2013, pp. 143-148.
[21]. Bijoyeta Roy, S.K. Misra, Preeti Gupta, Akanksha Goswami. An Integrated DEMATEL and AHP approach for personnel estimation. International Journal of Computer Science and Information Technology & Security, vol. 2, no. 6, 2012, pp. 1206-1212.
[22]. Wu W.W., Lan L.W., and Lee Y.T. Exploring decisive factors affecting an organization's SaaS adoption: A case study. International Journal of Information Management, vol. 31, issue 6, 2011, pp. 556-563.
[23]. Venkatesh V.G., Zhang A., Luthra S., Dubey R., Subramanian N., Mangla S. Barriers to coastal shipping development: An Indian perspective. Transportation Research. Part D: Transport and Environment, vol. 52, part A, 2017, pp.362-378.
[24]. Pandey P., Litoriya R., Tiwari A. A framework for fuzzy modelling in agricultural diagnostics. Journal Européen des Systèmes Automatisés, vol. 51, no. 4-6, 2018, pp. 203-223.
[25]. Wu Y.C., Lin C.W. National port competitiveness: Implications for India. Management Decision, vol. 46, no. 10, 2008, pp. 1482-1507.
[26]. Han Y., Deng Y. An enhanced fuzzy evidential DEMATEL method with its application to identify critical success factors. Soft Computing, vol. 22, no. 15, 2018, pp. 5073–5090.
[27]. Bhatia M.S., Srivastava R.K. Analysis of external barriers to remanufacturing using grey-DEMATEL approach: An Indian perspective. Resources, Conservation & Recycling, vol. 136, 2018, pp. 79-87.
[28]. Zhang L., Huang X.Y., Jiang J., Hu Y.K. CSLabel: An Approach for Labelling Mobile App Reviews. Journal of Computer Science and Technology, vol. 32, no. 6, 2017, pp. 1076–1089.
[29]. Maalej W., Kurtanovic Z., Nabil H., Stanik C. On the automatic classification of app reviews. Requirement Engineering, vol. 21, no. 3, 2015, pp. 311-331.
[30]. Gui J., McIlroy S., Nagappan M., and Halfond W.G. Truth in advertising: The hidden cost of mobile ads for software developers. In Proc. of the 37th International Conference on Software Engineering, 2015, pp. 100-110.
[31]. Xu X., Dutta K., and Datta A. Functionality-based mobile app recommendation by identifying aspects from user reviews. In Proc. of the 35th International Conference on Information Systems, 2014, pp. 1-10.
[32]. Khalid H. et al. What Do Mobile App Users Complain About? A Study on Free iOS Apps. IEEE software, vol. 32, no. 3, 2015, 70-77.
[33]. Tan S.H. et al. Repairing Crashes in Android Apps. In Proc. of the 40th International Conference on Software Engineering, 2018, pp. 187-198.
[34]. Rajput G.S., Litoriya R. Corad Agile Method for Agile Software Cost Estimation. Open Access Library Journal, vol. 1, no.3, 2014, pp. 1-13.
[35]. Vu P.M. et al. Mining user opinions in mobile app reviews: A keyword-based approach. In Proc. of the International Conference on Automated Software Engineering, 2015, pp. 749-759
[36]. Zhang T. et al. Compatibility testing service for mobile applications. In Proc. of the IEEE Symposium on Service-Oriented System Engineering, 2015, pp. 179-186.
[37]. Bonne B. et al. Insecure Network, Unknown Connection: Understanding Wi-Fi Privacy Assumptions of Mobile Device Users. Information, vol. 8, no. 3, 2017, pp. 1-20.
[38]. Vu P.M. Mining user opinions in mobile app reviews: A keyword-based approach. In Proc. of the International Conference on Automated Software Engineering, 2015, pp. 749-759.
[39]. Wilke C. et al. Energy consumption and efficiency in mobile applications: A user feedback study. In Proc. of the IEEE International Conference on Green Computing and Communications and IEEE International Conference on Internet of Things, and IEEE International Conference on Cyber, Physical and Social Computing, 2013, pp. 134-141.

[40]. Datta S.K., Bonnet C., and Nikaein N. Android Power Management: Current and Future Trends. In Proc. of the First IEEE Workshop on Enabling Technologies for Smartphone and Internet of Things, 2012, pp. 48-53.
[41]. Pandey M., Litoriya R., Pandey P. Novel Approach for Mobile Based App Development Incorporating MAAF. Wireless Personal Communications, vol. 107, issue 4, 2019, pp. 1687–1708.
[42]. Ferreira D., Dey A.K. and Kostakos V. Understanding Human-Smartphone Concerns: A Study of Battery Life. Lecture Notes in Computer Science, vol. 6696, 2011, pp. 19–33.
[43]. Willocx M., Vossaert J. and Naessens V. Comparing performance parameters of mobile app development strategies. In Proc. of the International Conference on Mobile Software Engineering and Systems, 2016, pp 38-47.
[44]. Litoriya R., Sharma N., Kothari A. Incorporating Cost driver substitution to improve the effort using Agile COCOMO II. In Proc. of the CSI Sixth International Conference on Software Engineering, 2012, pp. 1-7.
[45]. Falaki H., Lymberopoulos D, and Mahajan R. A First Look at Traffic on Smartphones. In Proc. of the 10th ACM SIGCOMM conference on Internet measurement, 2010, pp. 281-287.
[46]. Comino S., Manenti F.M., and Mariuzzo F. (2015) Updates Management in Mobile Applications. iTunes vs Google Play. Available at SSRN: https://ssrn.com/abstract=2664463 or http://dx.doi.org/10.2139/ssrn.2664463.
[47]. Hassan S., Shang W., and Hassan A.E. An empirical study of emergency updates for top android mobile apps. Empirical Software Engineering, vol. 22, no. 1, 2017, pp. 505-546.
[48]. Ranjan A., Litoriya R. Relational Algebra Interpreter in Context of Query Languages. International Journal of Computer Theory and Engineering, vol. 3, no. 1, 2011, pp. 9-15.
[49]. Andreou A.S. et al. Key issues for the design and development of mobile commerce services and applications. International Journal of Mobile Communications, vol. 3, no. 3, 2005, pp. 303–323.
[50]. Perez B.M., Diez I.D., and Coronado M.L. Privacy and security in mobile health apps: A review and recommendations. Journal of Medical Systems, vol. 39, no. 1, 2017, pp. 1-8.
[51]. Pandey P., Litoriya R. An activity vigilance system for elderly based on fuzzy probability transformations. Journal of Intelligent and Fuzzy Systems, vol. 36, no. 3, 2019, pp. 2481-2494.
[52]. Armand A., Allahviranloo T., Gouyandeh Z. Some Fundamental Results on Fuzzy Calculus. Iranian Journal of Fuzzy Systems, vol. 15, no. 3, 2018, pp. 27-46.
[53]. Pandey P., Kumar S., Shrivastav S. A fuzzy decision making approach for analogy detection in new product forecasting. Journal of Intelligent & Fuzzy Systems, vol. 28, no. 5, 2015, pp. 2047-2057.
[54]. Bhadauriya S., Sharma V., Litoriya R. Empirical Analysis of Ethical Issues in the Era of Future Information Technology. In Proc. of the 2nd International Conference on Software Technology and Engineering, vol. 2, 2010, pp. 31-35.
[55]. Tseng M.L., Wu K.J., Nguyen T.H. Information technology in supply chain management: a case study. Procedia - Social and Behavioral Sciences, vol. 25, 2011, pp. 257-272.
[56]. Deng J.L. Control problems of grey systems. Systems and Control Letters, vol. 1, no. 5, 1982, pp. 288–294.
[57]. Tseng M.L. A causal and effect decision making model of service quality expectation using grey-fuzzy DEMATEL approach. Expert Systems with Applications, vol. 36, no. 4, 2009, pp. 7738-7748.
[58]. Pandey P., Kumar S., Shrivastav S. Forecasting using Fuzzy Time Series for Diffusion of Innovation: Case of Tata Nano Car in India. National Academy Science Letters, vol. 36, no. 3, 2013, pp. 299-309.
[59]. Pandey P., Kumar S., Shrivastav S. A fuzzy decision making approach for analogy detection in new product forecasting. Journal of Intelligent and Fuzzy Systems, vol. 28, no. 5, 2015, pp. 2047-2057.
[60]. Wang F. et al. A semantics-based approach to multi-source heterogeneous information fusion in the internet of things. Soft Computing, vol. 21, no. 8, 2017, pp. 2005–2013.
[61]. Zheng X. and Deng Y. Dependence assessment in human reliability analysis based on evidence credibility decay model and IOWA operator. Annals of Nuclear Energy, vol. 112, 2018, pp. 673-684.
[62]. Liu T., Deng Y., Chan F. (2018) Evidential supplier selection based on DEMATEL and game theory. International Journal of Fuzzy Systems, vol. 20, no. 4, 2018, pp. 1321-1333.
[63]. Han Y., Deng Y. A hybrid intelligent model for assessment of critical success factors in high-risk emergency system. Journal of Ambient Intelligence and Humanized Computing, vol. 9, issue 6, 2018, pp. 1933–1953.
[64]. Liu Z. et al. Combination of classifiers with optimal weight based on evidential reasoning. IEEE Transactions on Fuzzy Systems, vol. 26, issue 3, 2018, pp. 1217-1230.
[65]. Song Y. et al. Combination of interval-valued belief structures based on intuitionist fuzzy set. IEEE Transactions on Fuzzy Systems, vol. 26, issue 3, 2018, pp. 61–70.

[66]. Wang J., Wu J., Wang J., Zhang H., Chen X. Multi-criteria decision-making methods based on the hausdorff distance of hesitant fuzzy linguistic numbers. Soft Computing, vol. 20, no. 4, 2016, pp. 1621–1633

[67]. Pandey M, Litoriya R., and Pandey P. An ISM approach for modeling the issues and factors of mobile app development. International Journal of Software Engineering and Knowledge Engineering, vol. 28, no. 7, 2018, pp. 937-953.

Информация об авторах / Information about authors

Мамта ПАНДИ – научный сотрудник, факультет компьютерных наук и инженерии. Научные интересы: мобильные приложения, разработка программного обеспечения.

Mamta PANDEY, Research Scholar, Department of Computer Science and Engineering. Her research interests include mobile applications, software engineering.

Д-р Ратнеш ЛИТОРИЯ, доцент, Ph.D, факультет компьютерных наук и инженерии. Его область исследований и преподавания включает разработку программного обеспечения, управление проектами, компьютерную графику, алгоритмы и мобильные сети.

Dr. Ratnesh LITORIYA, Assistant Professor, Ph.D, Department of Computer Science and Engineering. His research and teaching proficiency area focus on software engineering, project management, computer graphics, algorithms, and mobile adhoc networks.

Пратик ПАНДИ, доцент, PhD, факультет компьютерных наук и инженерии. Доктор Панди работает в областях управления бизнес-процессами, разработки программного обеспечения и прогнозирования.

Prateek PANDEY, Assistant Professor, PhD, Department of Computer Science and Engineering. Dr. Pandey has worked in the area of Business Process Management, Software Engineering, and Forecasting.

DOI: 10.15514/ISPRAS-2019-31(4)-6

Проектирование интерфейсов классов графовой модели нейронной сети

[1] *Ю.Л. Карпов, ORCID: 0000-0001-6844-5530 <y.l.karpov@yandex.ru>*
[2] *И.А. Волкова, ORCID: 0000-0002-9869-7154 <irina.a.volkova@gmail.com>*
[2] *А.А. Вылиток, ORCID: 0000-0001-8643-873X <alexey.vylitok@gmail.com>*
[3,2] *Л.Е. Карпов, ORCID: 0000-0001-6400-8325 <mak@ispras.ru>*
[4,5] *Ю.Г. Сметанин, ORCID: 0000-0001-8828-0091 <ysmetanin@rambler.ru>*
[1] *ООО Люксофт Профешнл,*
123060, Россия, г. Москва, 1-й Волоколамский проезд, 10
[2] *Московский государственный университет имени М.В. Ломоносова,*
119991, Россия, Москва, Ленинские горы, д. 1.
[3] *Институт системного программирования им. В.П. Иванникова РАН,*
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.
[4] *Федеральный исследовательский центр «Информатика и управление» РАН*
119333 Москва, ул. Вавилова, д. 44, кор. 2.
[5] *Московский физико-технический институт (технический университет)*
141701 Московская область, г. Долгопрудный, Институтский пер., 9

Аннотация. Описывается подход к тестированию искусственных нейронных сетей, реализованный в программе на языке Си++ в виде набора структур данных и алгоритмов их обработки. В качестве структур данных используются классы языка Си++, реализующие работу с такими объектами, как вершина графа, ребро, ориентированный и неориентированный граф, остовное дерево, цикл. Приводятся интерфейсы важнейших перегруженных операций над используемыми объектами и тестирующих процедур. Дан пример реализации одной из тестирующих процедур, использующий перегруженные операции над используемыми объектами.

Ключевые слова: искусственная нейронная сеть; эвристика; тестирование нейронной сети; теория графов; граф; вершина; ребро; остовное дерево; цикл в графе; отрицательный цикл; устойчивость нейронной сети.

Для цитирования: Карпов Ю.Л., Волкова И.А., Вылиток А.А., Карпов Л.Е., Сметанин Ю.Г. Проектирование интерфейсов классов графовой модели нейронной сети. Труды ИСП РАН, том 31, вып. 4, 2019 г., стр. 97-112. DOI: 10.15514/ISPRAS-2019-31(4)-6

Благодарности. Авторы выражают благодарность Российскому фонду фундаментальных исследований, который поддерживает проекты № 18-07-0697-а, № 18-07-01211-а, № 19-07-00321-а, № 19-07-00493-а. Авторы также благодарны доценту кафедры системного программирования факультета ВМК МГУ им. М. В. Ломоносова Виктору Васильевичу Малышко, оказавшему помощь в проверке и уточнении диаграммы классов графовой модели.

Designing classes' interfaces for neural network graph model

[1] *Yu.L. Karpov, ORCID: 0000-0001-6844-5530 <y.l.karpov@yandex.ru>*
[2] *I.A. Volkova, ORCID: 0000-0002-9869-7154 <irina.a.volkova@gmail.com>*
[2] *A.A. Vylitok, ORCID: 0000-0001-8643-873X <alexey.vylitok@gmail.com>*
[3,2] *L.E. Karpov, ORCID: 0000-0001-6400-8325 <mak@ispras.ru>*
[4,5] *Yu.G. Smetanin, ORCID: 0000-0001-8828-0091 <ysmetanin@rambler.ru>*
[1] *Luxoft Professional LLC,*
10, 1-st Volokolamsky proezd, Moscow, 123060, Russia
[2] *Lomonosov Moscow State University,*
GSP-1, Leninskie Gory, Moscow, 119991, Russia.
[3] *Ivannikov Institute for System Programming of RAS,*
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
[4] *Federal Research Center "Informatics and Control" of RAS,*
44, korp. 2, Vavilova st., Moscow, 119333, Russia
[5] *Moscow Institute of Physics and Technology,*
9, Institutskii per., Dolgoprudnyi, Moscow region, 141700, Russia

Abstract. An approach to testing artificial neural networks is described. The model of neural network is based on graph theory, and operations that are used in theoretical works devoted to graphs, trees, paths, cycles, and circuits. The methods are implemented in a C++ program in the form of a set of data structures and algorithms for their processing. C++ classes are used as data structures for implementing the processing of such objects as a graph vertex, edge, oriented and undirected graph, spanning tree, circuit. Lists of standard methods (constructors and destructors, different assigning operations) are given for all classes. Additional operations are represented in details, and among them – adding one graph to another graph, adding an edge to a graph, removing edges and vertices from graph, normalizing graph, and some more. Many different searching operations are offered. Variants of graph sorting operations are also included into graph model, some of them are similar to array sorting algorithms, and some are more specific. Above these low-level operations several more complex operations are considered as graph model components. These operations include building spanning tree for arbitrary graph, building cograph for spanning tree of a graph, discovering circuits in a graph, evaluating of circuit sign, and so on. Examples of the interfaces of the most important overloaded operations on the objects used are given. An example is given of implementation of one of testing procedures where overloaded operations of graph model objects are used.

Keywords: artificial neuron network; heuristic; neuron network testing; graph theory; graph; vertex; edge; spanning tree; circuit in graph; negative circuit.

For citation: Karpov Yu.L., Volkova I.A., Vylitok A.A., Karpov L.E., Smetanin Yu.G. Designing classes' interfaces for neuron network graph model. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 4, 2019. pp. 97-112 (in Russian). DOI: 10.15514/ISPRAS-2019-31(4)-6

Acknowledgments. The authors are grateful to the Russian Foundation for Basic Research, which supports projects No. 18-07-0697-a, No. 18-07-01211-a, No. 19-07-00321-a, No. 19-07-00493-a. The authors are also grateful to the associate professor Viktor Vasilyevich Malyshko of the system programming department of the CMC faculty of Lomonosov Moscow State University, who assisted in the verification and refinement of the class diagram of the graph model.

1. Введение

Одним из важнейших мероприятий при использовании искусственных нейронных сетей для решения задач из самых разных прикладных областей, будь то задачи классификации, оптимизации, машинного перевода, обработки и анализа изображений в режиме реального времени [1–3], обработки сигналов [4] или ещё какие-нибудь, являются работы по тестированию самих нейронных сетей. Процесс тестирования нейронных сетей существенно отличается от процесса тестирования программного обеспечения, что не в последнюю очередь связано с трудностями отслеживания движения данных в нейронных

сетях. Авторы уже сформировали и описали в общем виде предлагаемый ими эвристический подход к тестированию искусственных нейронных сетей (см. [5]). Однако понятно, что верность эвристического подхода не может быть подтверждена одними только теоретическими рассуждениями. Её надо демонстрировать на реальных примерах.

Прежде, чем переходить к работе с искусственными сетями, используемыми при решении практических задач, необходимо разработать программную модель, способную выявлять некоторые важные свойства таких сетей. В настоящее время используются самые разнообразные структуры искусственных нейронных сетей [6-9], из чего следует, что включение классов объектов в нейросетевые модели может производиться на основе учёта очень разных видов признаков. К наиболее содержательным основаниям выбора тех или иных систем классов можно отнести такие различия в структурах сетей:

- по наличию или отсутствию в сети обратных связей;
- по рекуррентному или нерекуррентному режиму работы;
- по области значений параметров (дискретные или аналоговые);
- по времени (дискретное и непрерывное);
- по режиму обновления параметров (с синхронизацией и без неё);
- по обучению с учителем и без учителя (есть также смешанные стратегии);
- по детерминированному или вероятностному режиму работы;
- наконец, просто по размеру – от малых сетей до глубоких, содержащих сотни слоёв нейронов.

Естественно в качестве модели, способной описать практически наблюдаемое многообразие нейронных сетей, выбирать модель, основанную на понятии графа, что даст возможность пользоваться методами теории графов [10-12].

2. Искусственные нейронные сети и графы

Подход к построению модели тестирования на основе представления искусственной нейронной сети в виде графа в данном случае обладает дополнительным преимуществом. Он позволяет отвлечься от излишней детализации описания вычислительных узлов сети и сосредоточиться на исследовании структурных свойств и особенностей конкретных структур, встречающихся в большинстве практически используемых нейронных сетей.

Упомянутые особенности имеют критическое влияние на многие свойства, демонстрируемые нейронными сетями при решении задач с их использованием. Структуры одних (вполне успешно применяемых на практике) сетей радикально отличаются от структуры других (не менее успешных).

Такие факторы, как наличие или отсутствие промежуточных слоёв, их количество, могут серьёзно повлиять на способность строящейся нейронной сети решать те или иные классы задач, а иногда могут накладывать определённые ограничения на количественные параметры допустимых к решению задач. Столь же влиятельной является структура этих слоёв – наличие в них внутрислойных связей, интенсивность передач информации от слоя к слою, наличие обратных связей, как положительных, так и отрицательных.

Все вместе эти и другие важные структурные свойства сети могут влиять и на саму возможность получения решения конкретной задачи, и на устойчивость этого решения, то есть сохранение основных свойств решения при относительно малых искажениях входных данных.

Для всех искусственных нейронных сетей характерно наличие входного и выходного слоя нейронов, которые либо непосредственно взаимодействуют друг с другом, либо осуществляют это взаимодействие через промежуточные слои, количество которых определяется поставленной задачей. Наибольший интерес для исследования структурных

особенностей нейронных сетей представляют как раз промежуточные слои, что определяется и большим их количеством, и сложностью межнейронных связей.

Именно поэтому авторы в первую очередь решили начать строить свою модель для промежуточных слоёв искусственных нейронных сетей, и в прежде всего для сетей, в которых промежуточные слои образуют не простейшие линии передачи информации от входного слоя к выходному, что характерно для свёрточных сетей, а сложные переплетающиеся структуры с обратными связями, в которых практически невозможно отследить пути распространения данных.

3. Автоматизация тестирования через предварительное создание средств работы с графами

Тестирование искусственной нейронной сети представляет собой сложный эмпирический процесс подбора или подтверждения правильности подбора многочисленных параметров. Такие важные тестируемые параметры есть у всех нейронных сетей, но у сетей с разной структурой промежуточных слоёв они разные. Например, при тестировании такой нейронной сети, как сеть Хопфилда [13], в которой все слои являются в некотором смысле промежуточными, являясь одновременно и входными, выходными, требуется исследовать начальные значения коэффициентов матрицы весов, векторы порогов срабатывания нейронов, метрики, позволяющие сравнивать выходы сети с обучающими выходными образцами, вид сигмоидальной функции, используемой при формировании выходных векторов на основе матрицы весов и значений входных векторов.

При тестировании машин Больцмана [14] необходимо обращать внимание на следующие параметры:

- начальное значение искусственной температуры (если выбранное значение оказывается неверным, алгоритм может отдавать предпочтение неверным решениям);
- подмножество обучающих векторов, которое будет использовано при настройке сети на решение поставленной задачи;
- метод изменения весов и целевых функций, влияющий на скорость сходимости процесса (количество шагов, которое выполняется сетью) и качество решения (близость найденного локального минимума целевой функции к её глобальному минимуму);
- способ определения необходимости коррекции веса исследуемого синапса, то есть разработки компаратора весов.

Пытаясь разобраться с другими видами нейронных сетей, кроме уже упомянутых, можно встретить такие подлежащие тестированию и настройке параметры, как количество промежуточных слоёв и алгоритмы уменьшения размерности (вычисления максимального, минимального, среднего, медианного значений), а также множество других параметров, каждый из которых характерен для конкретных архитектур нейронной сети. Задача, которую решают при выборе нужной архитектуры, связана с получением конкретных рекомендаций по выбору комбинации свойств сети, способной содействовать поискам решения прикладной проблемы, стоящей перед пользователями нейронной сети. Разработка методов автоматизированного тестирования сетей позволит вплотную приблизиться к пониманию методики выработки подобных рекомендаций.

3.1 Графовая модель искусственной нейронной сети, графы ориентированные и неориентированные

Учитывая огромное разнообразие видов нейронных сетей, наиболее приемлемым выбором модели, которую можно было бы использовать при тестировании нейронных сетей, можно считать модель, основанную на теории графов, которая могла бы оперировать такими объектами:

- вершина графа;
- ребро графа;
- набор вершин графа;
- набор рёбер графа;
- граф, как совокупность вершин и рёбер;
- остовное дерево графа;
- цикл (простой замкнутый путь) в графе: все рёбра и вершины в цикле различны;
- матрица смежности вершин графа;
- матрица инциденции;
- матрица весов.

Фактическая объектно-ориентированная декомпозиция прикладной области уже была проделана усилиями многих математиков всего мира, и для программной реализации было естественно выбрать объектно-ориентированный язык программирования. В настоящее время объектно-ориентированный подход к программированию стал довлеющей концепцией при построении моделей реального мира, а также абстрактных построений, примером которых являются объекты, встречающиеся при решении задач теории графов. Существует огромное множество объектно-ориентированных языков программирования, из которых авторами был выбран язык Си++. Причин для такого выбора было множество – от объективных, связанных с наличием эффективных систем программирования и возможностью работать на разном оборудовании и в разном операционном окружении, до субъективных, возникших в связи личными преференциями авторов.

При оценке методов реализации графовой модели нейронной сети их эффективность рассматривалась авторами с учётом того, что эта модель должна использоваться для построения инструментальных программных средств. На практике при работе с графами часто возникают задачи с разными уровнями вычислительной сложности – от задач, разрешимых за линейное время, до NP-полных задач [12]. Это приводит к постоянной борьбе за снижение требований к вычислительным ресурсам, которые используются нейронными сетями. Проведение подготовительных мероприятий (анализ архитектуры сети, определение свойств отдельных компонентов сети, имеющихся в ней циклов, их весов и знаков) тоже приводят к необходимости построения вычислительно сложных, часто переборных алгоритмов. Однако тот факт, что подготовка и улучшение структуры сети проводятся на модельных данных на предварительном этапе, то есть ещё до начала решения основной задачи, позволяет менее требовательно относиться к сложности используемых алгоритмов, то есть в некоторых случаях выбор между сложностью алгоритма при исполнении и временем его реализации делался в последнего.

Дополнительным аргументом в пользу Си++ послужила та естественность, с которой в этом языке допускается использование концепции абстрактных типов данных.

3.2 Структура данных – классы и их взаимодействие (агрегация и наследование)

При выборе структур данных, которые должны использоваться в разрабатываемой графовой модели нейронной сети, за основу можно принимать одно из трёх эквивалентных представлений графа:

- перечисление рёбер и вершин графа;
- задание матрицы смежности вершин графа;
- задание матрицы инциденции.

Практические соображения заставили остановиться на первом способе, обладающем преимуществами наглядности и позволяющем одновременно с той информацией, которая

присутствует в матрицах смежности (количество рёбер, соединяющих некоторые вершины) и инциденции (признаки инциденции вершин рёбрам), задавать также веса рёбер, причём со знаками весов (рис. 1). По перечню рёбер и вершин легко восстанавливаются и матрица смежности, и матрица инциденции.

Выбранное представление исходной информации и представление об используемых в модели объектах использовались для формирования классов объектов (рис. 2). Для работы с достаточно простыми объектами или их свойствами были введены синонимы стандартных (встроенных) типов. Это помогло избежать неприятных ситуаций, когда при программировании некоторые объекты (или свойства объектов) модели могли случайно оказаться перепутанными со вспомогательными переменными, имеющими те же стандартные типы.

```
6  11 11
1  2  2.1
1  6  8.4
11 1  2
2  5  3.6
5  2  -3.2
2  2  8.4
```

Рис. 1. Пример задания рёбер и вершин графа
Fig. 1. Graph edges and vertices list example

В частности, для работы с номерами и именами вершин и рёбер был введён синоним `Node` стандартного беззнакового интегрального типа `size_t`, с той же целью для работы с весами рёбер вместо встроенного плавающего типа `double` было отдано предпочтение его синониму `Weight`.

```
class Vertex;           typedef vector <Vertex> Vertices;
class Edge;             typedef vector <Edge>   Edges;
class Graph;
class SpanT;
typedef size_t Node;    typedef vector <Node>   vNodes;
                        typedef set    <Node>   sNodes;
```

Рис. 2. Классы, использовавшиеся в графовой модели нейронной сети
Fig. 2. Classes used in graph model of neuron network

Объектам, соответствующим вершинам и рёбрам моделируемых графов, были сопоставлены классы `Vertex` и `Edge`, атрибутика которых выбрана предельно простой:

- вершина – это поле имени и три поля, предназначенных для хранения количества входящих, выходящих и полного числа входящих и выходящих рёбер;
- ребро – это две инцидентные вершины и вес, приписанный ребру.

Коллекции вершин и рёбер было принято решение реализовать, применяя один из библиотечных контейнеров-последовательностей, так как в определённых ситуациях, например, при построении циклов на графах, отношение следования становится весьма важным. В конечном итоге было решено использовать библиотечные контейнеры `vector`, которые могут эффективно использоваться при сортировке рёбер и вершин графов, при поиске циклов в графах, при модификациях графов.

Несмотря на стремление максимально пользоваться возможностями языка реализации и стандартной библиотеки, для объектов некоторых невстроенных типов потребовалось

определить новые классы, построить агрегации классов, а также, хотя и очень простую, но необходимую в данном случае наследственную иерархию (рис. 3).

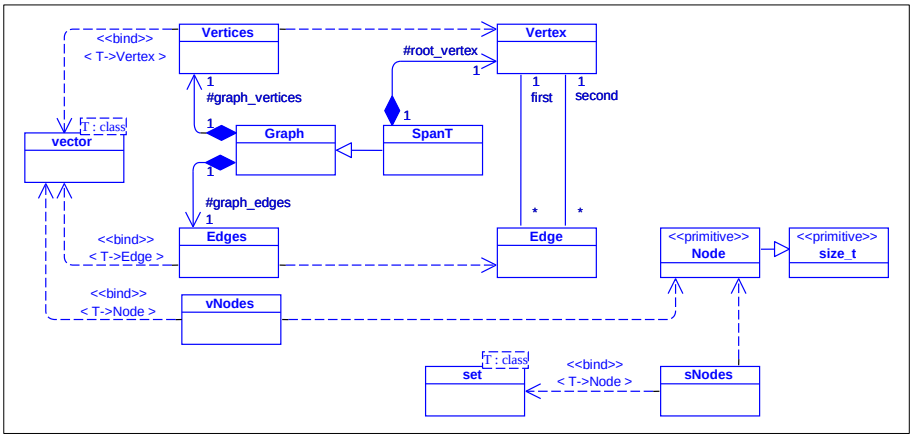


Рис. 3. Взаимоотношения классов графовой модели нейронной сети
Fig. 3. Neuron network graph model classes' relations

```
enum Graph_direction { Directed, NonDirected };
class    Vertex { Node name, degree_from, degree_twrd,
                            degree_both; Weight weight;
 public: Vertex ( Node vN = 0, Node vDF = 0, Node  vDT = 0,
                               Node vDB = 0, Weight vW = 0.0);
                                                  /* ... */
};
class    Edge   { Node name;   Vertex vertex_from;
                               Vertex vertex_twrd;
                  Node degree; Weight weight;
 public: Edge   ( Node eN, Vertex eF, Vertex eT,
                  Node degree = 0, Weight eW = 0.0); /* ... */
};
class   Graph { protected:  Vertices * Graph_Vertices;
                               Edges * Graph_Edges;
                    Graph_direction   Graph_Orientation;
public: Graph (const Graph_direction D = Directed);
        Graph (const Graph & G);
        Graph (      Graph &&G);
        Graph (const Graph_direction d = Directed);
        Graph (const Edges & E);
        virtual    ~ Graph   ();                    /* ... */
};
class SpanT: public Graph { protected: Vertex  Root_Vertex;
public: SpanT ():    Graph (Directed) {};
        SpanT (const SpanT & S);
        SpanT (      SpanT &&S);
        SpanT (const Edges & E);
        SpanT (const Edges & E, const Node nF, const Node nT);
        SpanT (const Edges & E, const Vertex & V);   /* ... */
};
```

Рис. 4. Интерфейсы конструкторов и деструктора основных классов графовой модели нейронной сети
Fig. 4. Constructors and destructor interfaces of neuron network graph model main classes

В этой иерархии класс `Graph` описания графов общего вида (как ориентированных, так и неориентированных) является базовым классом, производным от которого был организован класс `SpanT`, предназначенный для работы с графами специального вида – остовными деревьями. Построенная иерархия действительно очень проста, хотя в ней есть виртуальные функции, при этом многие методы, необходимые для работы с остовными деревьями, наследуются и используются в производном классе без изменения. Связана такая простота с тем, что практически вся нагрузка ложится на классы, агрегированные в графы и остовные деревья, то есть на коллекции рёбер и вершин.

3.3 Конструкторы и деструкторы

Конструкторы и деструкторы классов графовой модели имеют строго индивидуальный характер (рис. 4).

В самых простых классах (`Vertex` и `Edge`) вводятся только по одному специальному конструктору, а деструкторов вообще нет. В большинстве ситуаций эти классы могут обходиться теми возможностями, которые предоставляются автоматически генерируемыми специальными методами классов. Библиотечные коллекции (`Vertices` и `Edges`) имеют много разных конструкторов, все они предоставляются библиотекой языка Си++, не требуя никаких дополнительных построений.

Нетривиальный набор конструкторов есть у базового (`Graph`) и производного (`SpanT`) классов наследственной иерархии. Кроме обычного набора конструкторов копирования, конструктора умолчания и совмещённого с ним конструктора преобразования, определяющего ориентированные или неориентированные графы, в базовом классе присутствует ещё один конструктор преобразования, позволяющий создать объект по перечню рёбер. Такого конструктора оказывается достаточным для реализации содержательной работы по преобразованию произвольного списка рёбер в граф, с полностью определённой структурой рёбер и вершин. Аналогичный набор конструкторов имеется и в классе `SpanT` остовных деревьев. Единственным отличием этого класса от его базового класса является присутствие в производном классе делегирующего конструктора и конструкторов, позволяющих определить не только рёбра и вершины дерева, но также его корневую вершину.

Содержательный виртуальный деструктор достаточно иметь только у базового класса, поскольку этот класс неплоский и уничтожение соответствующих объектов должно предваряться освобождением памяти, ранее занятой составными частями объекта.

3.4 Операции над вершинами, рёбрами и графами

В теории графов вопрос об операциях над графами проработан в достаточной степени. Важность самого понятия графа и сложность работы с такими объектами давно привели к выработке большого количества разнообразных операций над графами и их составными частями. Некоторые операции довольно просты, другие сложнее, для некоторых в языке математики введены специальные знаки операций ($\cup$, $\cap$, x), другие выражаются словами («*построим матрицу смежности ...*»).

Прямого соответствия операций над графами операциям языка Си++ нет, но можно попытаться перегрузить некоторые операции так, чтобы некоторая семантическая близость между теоретическими и языковыми операциями всё же оставалась.

Введение некоторых операций определялось самим языком. Например, совершенно очевидно, что в каждом классе должна присутствовать операция присваивания, позволяющая создать ещё одну копию исследуемого объекта. Также очевидно, в классах, не являющихся плоскими, должна определяться и операция присваивания по правой ссылке (перенос значения).

Другие операции оказалось необходимым наполнять некоторым изначально неочевидным смыслом. Например, в классе `Edge` описания рёбер графа была определена унарная операция инверсии последовательности рёбер ('`-`'), а также бинарные операции '`%`' и '`%=`', выполняющие такую сортировку последовательности рёбер, что в отсортированной последовательности два ребра следуют друг за другом, если одно из них является входящим в некоторую вершину, а второе – выходящим из той же вершины (рис. 5). В отсортированную последовательность при этом включаются только те рёбра исходной последовательности, которые могут быть выстроены в указанном порядке, начиная с заданной вторым операндом вершины. Эти операции оказались очень удобными при работе с циклами в графах, так как для циклов указанные операции не приводят к исключению рёбер из отсортированных последовательностей.

```
class Edge  {                                                  /* ... */
friend       Edges&operator+=(      Edges&eL, const Edges&eR);
friend const Edges operator- (const Edges& E);
friend const Edges operator% (      Edges& E, const Node   n);
friend       Edges&operator%=(      Edges& E, const Node   n);
                                                               /* ... */
};
```

Рис. 5. Интерфейсы операций класса описания рёбер графа
Fig. 5. Interfaces of class Edge operators

```
class Graph {                                                  /* ... */
public:
virtual       Graph&   operator-=(const Node     & n);
virtual       Graph&   operator-=(const Edge     & e);
virtual       Graph&   operator-=(const Edges    & E);
virtual       Graph&   operator-=(const Vertex   & v);
virtual       Graph&   operator-=(const Vertices & V);
virtual       Graph&   operator+=(const Edge     & e);
virtual       Graph&   operator+=(const Edges    & E);
virtual       Graph&   operator+=(const Graph    & G);
virtual       Graph&   operator= (const Graph    & G);
virtual       Graph&   operator= (      Graph   && G);
virtual       Graph    operator% (const Node       n) const;
virtual       Graph&   operator%=(const Node       n);
virtual const Graph    operator+ () const;
        const Graph    operator~ ();
virtual       Graph&   operator>>(Graph    & G) const;
virtual       Edges&   operator>>(Edges    & E) const;
virtual       Vertices&operator>>(Vertices & V) const;
virtual const Graph    operator- () const;
        const Graph&   operator--();
        const Graph    operator--(int i);
 friend const Graph operator+(const Edges& E,const Graph& G);
 friend const Graph operator+(const Graph& G,const Edges& E);
 friend const Graph operator+(const Graph&gL,const Graph&gR);
                                                               /* ... */
};
```

Рис. 6. Интерфейсы операций класса описания графов
Fig. 6. Interfaces of class Graph operators

Аналогичные операции '`-`' (реверса), '`%`' и '`%=`' (специальной сортировки) удобно ввести и в класс `Graph` описания графов, так как в каждом графе имеется собственная

последовательность рёбер (рис. 6). Естественно, что эти операции наследуются классом *SpanT* описания остовных деревьев (рис. 7), над которыми также приходится строить циклы (добавлением некоторых рёбер) и, следовательно, проводить их сортировку.

```
class SpanT: public Graph {                                  /* ... */
public:        SpanT&   operator= (const SpanT  & S);
               SpanT&   operator= (      SpanT && S);
               SpanT&   operator= (const Graph  & S) override;
               SpanT&   operator= (      Graph && S) override;
        const SpanT&    operator--();
        const SpanT     operator--(int i);                   /* ... */
};
```

Рис. 7. Интерфейсы операций класса описания остовных деревьев
Fig. 7. Interfaces of class SpanT operators

В состав операций над объектами этих типов также было признано удобным вставить унарную операцию *'+'*, которая нормализует граф, то есть, беря за основу последовательность рёбер графа, составляет соответствующую ей последовательность вершин.

Оказалось, что это не единственный вид операции нормализации графа, которая должна применяться к графам при моделировании нейронных сетей. Другая унарная операция нормализации *'~'* при применении к графу оставляет в нём только вершины промежуточных слоёв, которые имеют и входящие, и исходящие рёбра, попутно удаляя из графа рёбра с нулевыми весами.

3.5 Сортировка и поиск, реверс графа

Функции сортировки и поиска играют важную роль при обработке любых коллекций данных. Важны они для реализации графовых моделей, которые так или иначе связаны с коллекциями объектов, предназначенных для моделирования вершин, рёбер, графов, деревьев.

```
const Node   Find   (const Vertices & V, const Node   & n);
const Node   Find   (const Vertices & V, const Vertex& v);
const Weight Find   (const Edges    & E, const Edge   & e);
const Weight Find   (const Edges    & E, const Edge   & e,
                     const Node_position nP);
const Node   Find   (const Edges    & E, const Node     n,
                     const Node_position nP);
const Node   Find   (const Edges    & E, const Node     F,
                                         const Node    vT);
const Node   Find   (const Graph    & G, const Node    nF,
                                         const Node    nT);
const Node   Find   (const Graph    & G, const Node     n,
                     const Node_position nP);
const Node   Find   (const Graph    & G, const Node    sE,
                     const Node nF,      const Node    nT);
const bool   Find   (const Graph    & G,       Edge    & e,
                     const Node nF,      const Node    nT);
const Weight Find   (const Graph    & G, const Edge   & e,
                     const Node_position nP);
const bool   Check  (const Graph    & G, const Edge   & e);
```

Рис. 8. Интерфейсы функций поиска и проверки вхождения и реверсирования
Fig. 8. Interfaces of Find and Check functions

Обычно таких функций требуется много, так как и для сортировки, и для поиска данных в коллекциях часто требуется реализовать множество алгоритмов и их вариантов.

При реализации графовой модели нейронной сети понадобилось более 10 вариантов для функций поиска и семантически примыкающей к ним функции проверки вхождения (рис. 8). Несколько вариантов оказалось необходимо реализовать и для сортировки (рис. 9), так как разные объекты должны сортироваться на основе совершенно разных критериев, а иногда и алгоритмов. В частности, если рёбра и вершины обладают некоторыми численными характеристиками (*степенями*), позволяющими, использовав наличие у них свойства быть операндами операций отношения, легко сортировать их по возрастанию или убыванию некоторых численных величин, то, например, для такого объекта, как цикл в графе, сортировка имеет совершенно другой смысл.

```
Edges Sort (Edges    & G, const Node n);
Graph Sort (Graph    & G, const Node n);
Node  Sort (Vertices & V, const Node_position Np);
void  Sort (Edges    & E, const Node_position Np);
Node  Sort (Graph    & G,
            const Node_position Vertex_degree_direction,
            const Node position  Edges weight direction);
```

Рис. 9. Интерфейсы функций сортировки графов, рёбер и вершин
Fig. 9. Interfaces of Sort functions

Сортировка и поиск – это операции, часто встречающиеся в самых разных программных системах, разрабатываемых для самых разных прикладных областей. Они используются не только при работе с графами, но и с другими структурами данных. Однако при работе с графами необходимо выполнять и более специфические операции, в частности, в разрабатываемой модели в состав операций классов была включена функция реверса и реализованная на её основе операция *'–'* (унарного минуса) (рис. 10). С помощью этой операции и лежащей в её основе функции можно выполнить перестановку элементов (рёбер) в контейнере типа *Edges* или в графе.

```
class Edge  {                                                  /* ... */
public:       Edges & Revert     (      Edges & E);
friend  const Edges   operator - (const Edges & E);  /* ... */
};
class Graph {                                                  /* ... */
public:       Edges & Revert     (      Edges & E);
virtual const Graph   operator - () const;
```

Рис. 10. Интерфейсы функций реверса перечня рёбер и графа
Fig. 10. Interfaces of graph and edges set Revert functions

3.6 Сложные тестирующие операции – поиск циклов, подсчёт весов и знаков циклов, вставка вспомогательных вершин и рёбер

Несмотря на исключительную полезность уже описанных структур и операций, суть графовой модели составляют другие её компоненты. Разработка модели производилась с целью исследования циклов, которые возникают в графах, соответствующих нейронным сетям, в которых между различными слоями нейронов имеются не только прямые, но и обратные связи. И те, и другие связи обладают весами, то есть численными характеристиками, от которых зависит степень взаимного влияния нейронов друг на друга. Структуры связей нейронов в разных нейронных сетях могут быть разными. Если обратных связей нет, сети называются сетями прямого распространения. Для таких сетей соответствующей графовой моделью будет дерево. Такие структуры встречаются

достаточно часто (например, в свёрточных сетях), но авторов больше интересовали сети с обратными связями, имеющие более сложные структуры, а среди таких сетей те из них, которые имеют отрицательные веса. Обратные связи могут приводить к образованию циклов, эти циклы могут иметь разные свойства, для изучения которых и строилась графовая модель. В работе [15] предлагается вычислять знак цикла, выявленного в графе нейронной сети, и доказывается, что наличие в графе циклов, имеющих отрицательные знаки, может приводить к возникновению ложных и неустойчивых решений. Знак цикла вычисляется как знак произведения весов входящих в цикл рёбер.

Авторы строили графовую модель и программную систему на её основе для проведения экспериментальной проверки и иллюстрации работы предложенного ими алгоритма модификации циклов, позволяющего избавиться от отрицательных циклов в структуре нейронной сети [16]. Доказанная в цитированной работе теорема о знаках циклов утверждает, что, если все циклы, образующиеся при присоединении к остовному дереву графа любого одного ребра из кографа этого остовного дерева, положительны, то все циклы данного графа будут также иметь положительный знак.

Для демонстрации разработанного авторами алгоритма были написаны несколько тестирующих процедур, реализующих отдельные шаги алгоритма (рис. 11).

```
Weight Build_Spanning_Tree
    (const Graph & G, const Node Root_node, SpanT & S);
void Build_CoGraph (const Graph & G, SpanT & S, Graph & CoG);
Node Count_sign
    (const Graph & G, const Edge & e, vNodes * C []);
Node Discover_Span_Circuits
    (const Graph & G, SpanT & S, vNodes * C []);
Node Discover_All_Circuits (const Graph & G, vNodes * C []);
Node Evaluate_Circuits_Signs
    (const Graph & G, vNodes * C [], Signs & Ss);
Node Add_Vertex_and_Edges (Graph & G,
      Node From_Vertex_Name,    Node Twrd_Vertex_Name,
    Weight First_part_weight, Weight Second_part_weight);
```

Рис. 11. Интерфейсы функций, реализующих основные шаги алгоритма модификации циклов графа
Fig. 11. Interfaces of main functions of graph modification algorithm

Основные процедуры, реализующие шаги алгоритма модификации графа, в котором присутствуют отрицательные циклы, были следующими:

- процедуры `Build_Spanning_Tree()` и `Build_CoGraph()` построения остовного дерева графа и кографа этого дерева;
- процедура `Discover_Span_Circuits()` выявления цикла, возникающего при добавлении к остовному дереву графа одного из рёбер, входящих в соответствующий этому дереву кограф;
- процедура `Discover_All_Circuits()` выявления всех циклов в графе;
- процедура `Count_sign()` вычисления знака цикла, возникающего при добавлении к остовному дереву графа одного из рёбер, входящих в соответствующий этому дереву кограф;
- процедура `Evaluate_Circuits_Signs()` вычисления знаков циклов;
- процедура `Add_Vertex_and_Edges()` модификации цикла путём замены одного из рёбер парой новых рёбер, имеющих одну общую вершину, также добавляемую к графу.

Приведённый перечень процедур в совокупности с ранее описанными операциями над графами, остовными деревьями, рёбрами и вершинами, а также со вспомогательными

процедурами построения внешнего представления графов и циклов в них позволил полностью и наглядно проиллюстрировать работу предложенного алгоритма модификации графов, детали которой можно найти в уже цитировавшейся работе [16].

Все эти процедуры запрограммированы с активным применением представленных в предыдущих разделах операциях над графами и остовными деревьями, а также над входящими в них вершинами и рёбрами.

Включение в созданную графовую модель большого числа перегруженных операций позволило записывать алгоритмы сложных тестирующих процедур очень наглядно, что в свою очередь способствовало снижению числа потенциальных ошибок в этих процедурах и снижению времени отладки всего комплекса программ. На рис. 12 приведён пример реализации одной из таких процедур.

Перечень операций над графами, используемых в текущей версии графовой модели, является открытым. При изменении каких-либо требований к модели или при их расширении или возникновении новых сценариев тестирования сетевых структур, этот перечень может дополнен дополнительными операциями. Единственным ограничением здесь могут быть возможности самого языка программирования Си++, который не позволяет перегружать некоторые операции и не позволяет вводить новые знаки операций. Однако, наталкиваясь на подобные ограничения, модель не теряет своей гибкости, хотя до определённой степени в случае роста числа моделируемых операций над графами может начать терять в наглядности представления этих операций.

```
Node Discover_Span_Circuits (const Graph  & G, SpanT & S,
                                   vNodes * C [])
{ Node Nc = 0;
  Graph CoG (G.get_Orientation());
  Build_CoGraph (G, S, CoG);
  for (auto & edge: CoG.get_Edges())
      { S  += edge;
        Nc += Discover_All_Circuits(S, C + Nc);
        -- S;
      }
  S = + S;
  return Nc;
}
```

Рис. 12. Реализация процедуры `Discover_Span_Circuits()` с помощью перегруженных операций и других процедур графовой модели

Fig. 12. Implementation of `Discover_Span_Circuits()` procedure using overloaded operations and other procedures of graph model

4. Заключение

В работе описан подход к тестированию искусственных нейронных сетей, реализованный на языке Си++ в программном комплексе в виде набора классов и алгоритмов обработки объектов этих классов. Представлены интерфейсы важнейших операций над используемыми объектами и тестирующих процедур. Продемонстрированный авторами подход даёт возможность унифицировать процесс решения прикладных задач с помощью нейронных сетей. При появлении конкретной задачи, которую требуется решить, несложно выбрать нейросетевую модель, которая лучше всего подходит для этой задачи, определить архитектуру сети, а также организовать процедуры и сценарии тестирования. Ядром используемых методов являются операции над графами, описывающими структуры связей между нейронами. Эти операции могут быть достаточно сложными, однако они не сложнее, чем операции, выполняемые самой нейронной сетью. Если используемое аппаратное

обеспечение даёт возможность реализовать нейронную сеть, оно также может дать возможность и протестировать её с помощью предлагаемых алгоритмов. Приведённые эксперименты, некоторые из которых описаны в другой работе авторов (см. [16]) показывают, что предложенный подход позволяет в значительной степени устранить как использование чрезмерно сложных моделей для решения конкретных прикладных задач, так и риск получения неверных решений.

Список литературы / References

[1]. Ciresan D., Meier U., Masci J., and Schmidhuber J. Multi-column deep neural network for traffic sign classification. Neural Networks, vol. 12, 2012, pp. 333-338.
[2]. David Talbot. CES 2015: Nvidia Demos a Car Computer Trained with "Deep Learning". MIT Technology Review, January 6, 2015, available at: https://www.technologyreview.com/s/533936/ces-2015-nvidia-demos-a-car-computer-trained-with-deep-learning/.
[3]. Roth S. Shrinkage Fields for Effective Image Restoration. In Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2014, pp. 2774-2781.
[4]. Deng L. and Yu D. Deep Learning: Methods and Applications, Foundations and Trends in Signal Processing, vol. 7, no. 3–4, 2014, pp. 1–19.
[5]. Ю.Л. Карпов, Л.Е. Карпов, Ю.Г. Сметанин. Адаптация общих концепций тестирования программного обеспечения к нейронным сетям. Программирование, т. 44, № 5, 2018, стр. 43-56. DOI: 10.31857/S013234740001214-0 / Yu.L. Karpov, L.E. Karpov, Yu.G. Smetanin. Adaptation of General Concepts of Software Testing to Neural Networks. Programming and Computer Software, vol. 44, № 5, 2018, pp. 324-334. DOI: 10.1134/S0361768818050031.
[6]. Rosenblatt Frank. Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms. Spartan Books, Washington DC, 1961, 616 p.
[7]. Kohonen T. Self-Organization and Associative Memory. New York: Springer, 1984, 332 p.
[8]. Grossberg S. Nonlinear Neural Networks: Principles, Mechanisms, and Architectures. Neural Networks, vol. 1, issue 1, 1988, pp. 17-61.
[9]. Hebb D.O. The Organization of Behavior. Wiley, New York, 1948, 335 p.
[10]. Harary F. Graph theory, Addison Wesley, 1969, 273 p.
[11]. Ore O. Theory of graphs. American Mathematical Society, Providence, RI, 1962, 269 p.
[12]. Иорданский М.А. Конструктивная теория графов и её приложения. Из-во Кириллица, 2016, 172 стр. / Iordanski M.A. Constructive graph theory and its applications. Cyrillic, 2016, 172 p. (in Russian).
[13]. J.J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. Proceedings of the National Academy of Sciences of the USA, vol. 79 no. 8, 1982, pp. 2554–2558.
[14]. Hinton D.E. and Seinowski T. Optimal Perceptual Inference. In Proc. of the IEEE Conference on Computer Vision and Pattern Recognition, 1983, pp. 448–453.
[15]. Julio Aracena, Jacques Demongeot, and Eric Goles. Positive and Negative Circuits in Discrete Neural Networks. IEEE Transactions on Neural Networks, vol. 15, No. 1, Jan. 2004, pp. 77-83.
[16]. Ю.Л. Карпов, Л.Е. Карпов, Ю.Г. Сметанин. Устранение отрицательных циклов в некоторых структурах нейронных сетей с целью достижения стационарных решений. Программирование, т. 45, № 5, стр. 25-35, 2019. DOI: 10.1134/S0132347419050029 / Yu.L. Karpov, L.E. Karpov, Yu.G. Smetanin. Elimination of Negative Circuits in Certain Neural Network Structures to Achieve Stable Solutions. Programming and Computer Software, vol. 45, № 5, pp. 241-250, 2019, DOI: 10.1134/S0361768819050025.

Информация об авторе / Information about the author

Юрий Леонидович КАРПОВ – кандидат технических наук, начальник отдела. Научные интересы: программирование, технологии тестирования программного обеспечения.

Yuri Leonidovich KARPOV – Candidate of Technical Sciences, Head of Department. Research interests: computer programming, software testing and engineering.

Ирина Анатольевна ВОЛКОВА – кандидат физико-математических наук, доцент по кафедре алгоритмических языков ф-та ВМК. Научные интересы: алгоритмические языки, методы трансляции.

Irina Anatolievna VOLKOVA – Candidate of Physics and Mathematics, Associate Professor in the Department of Algorithmic Languages, Faculty of CMC. Research interests: algorithmic languages, compilation techniques.

Алексей Александрович ВЫЛИТОК – кандидат физико-математических наук, доцент по кафедре алгоритмических языков ф-та ВМК. Научные интересы: языки программирования, формальные грамматики, объектно-ориентированная декомпозиция.

Alexey Alexandrovich VYLITOK – Candidate of Physics and Mathematics, Associate Professor in the Department of Algorithmic Languages, Faculty of CMC. Research interests: programming languages, formal grammars, object-oriented decomposition.

Леонид Евгеньевич КАРПОВ – доктор технических наук, ведущий научный сотрудник ИСП РАН, доцент кафедры системного программирования ф-та ВМК. Научные интересы: системное программирование, методы компиляции, системы программирования.

Leonid Evgenievich KARPOV – Doctor of Technical Sciences, Leading Researcher at ISP RAS, Associate Professor of the System Programming Department of the VMK Faculty. Research interests: system programming, compilation techniques, programming systems.

Юрий Геннадьевич СМЕТАНИН – доктор физико-математических наук, главный научный сотрудник ВЦ РАН им А.А. Дородницына ФИЦ «Информатика и управление» РАН, старший научный сотрудник кафедры интеллектуальных систем ф-та ФУПМ МФТИ. Научные интересы: комбинаторика слов, символическая динамика, нейронные сети.

Yuri Gennadievich SMETANIN – Doctor of Physical and Mathematical Sciences, Chief Researcher of the Dorodnicyn Computing Centre of FRC «Informatics and Control» RAS, senior researcher at the Department of Intelligent Systems, Faculty of Physics and Technology of MIPT. Scientific interests: combinatorics of words, symbolic dynamics, neural networks.

DOI: 10.15514/ISPRAS-2019-31(4)-7

Регуляризация Байеса при подборе весовых коэффициентов в ансамблях предикторов

А.С. Нужный, ORCID: 0000-0003-3319-2523 <nuzhny@inbox.ru>
Институт проблем безопасного развития атомной энергетики РАН,
Россия, 115191, г. Москва, Большая Тульская ул., д. 52

Аннотация. В статье рассматривается задача обучения с учителем: требуется восстановить зависимость, отображающую векторное множество в скалярное по конечному набору примеров такого отображения – обучающей выборке. Данная задача относится к классу обратных задач, и, как и большинство обратных задач, является математически некорректной. Это выражается в том, что если строить решение методом наименьших квадратов по точкам обучающей выборки, то можно столкнуться с переобучением – ситуацией, когда модель хорошо описывает обучающее множество, но дает большую ошибку на тестовом. Нами применяется подход, когда решение ищется в виде ансамбля предиктивных моделей. Ансамбли строятся с использованием метода бэггинга. В качестве базовых обучаемых моделей в работе используются персептроны и деревья решений. Конечное решение получается путем взвешенного голосования предикторов. Весовые коэффициенты подбираются путем минимизации ошибки ансамбля на обучающей выборке. Для борьбы с переобучением при подборе весовых коэффициентов применяется байесовская регуляризация решения. Чтобы подобрать параметры регуляризации, в работе предложено использовать метод ортогонализованных базисных функций, который позволяет получить их оптимальные значения без использования ресурсоемких итерационных процедур.

Ключевые слова: обучение с учителем; бэггинг; некорректные задачи; Байесовская регуляризация обучения.

Для цитирования: Нужный А.С. Регуляризация Байеса при подборе весовых коэффициентов в ансамблях предикторов. Труды ИСП РАН, том 31, вып. 4, 2019 г., стр. 113-120. DOI: 10.15514/ISPRAS-2019-31(4)-7

Bayes regularization in the selection of weight coefficients in the predictor ensembles

A.S. Nuzhny, ORCID: 0000-0003-3319-2523 <nuzhny@inbox.ru>
Nuclear Safety Institute of the Russian Academy of Sciences,
52 Bolshaya Tulskaya st., Moscow 115191, Russia

Abstact. The supervised learning problem is discussed in the article: it is necessary to restore the dependence that maps a vector set into a scalar based on a finite set of examples of such a mapping - a training sample. This problem belongs to the class of inverse problems, and, like most inverse problems, is mathematically incorrect. This is expressed in the fact that if you construct the solution using the least squares method according to the points of the training sample, you may encounter retraining – a situation where the model describes the training set well, but gives a big error on the test one. We apply the approach when a solution is sought in the form of an ensemble of predictive models. Ensembles are built using the bagging method. Perceptrons and decision trees are considered as basic learning models. The final decision is obtained by weighted voting of predictors. Weights are selected by minimizing model errors in the training set. To avoid over-fitting in the selection of weights, Bayesian regularization of the solution is applied. In order to choose regularization parameters, it is

proposed to use the method of orthogonalized basic functions, which allows obtaining their optimal values without using expensive iterative procedures.

Ключевые слова: supervised learning; bagging; ill-passed problem; Bayesian regularization of learning

Для цитирования: Nuzhny A.S. Bayes regularization in the selection of weight coefficients in the predictor ensembles. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 4, 2019. pp. 113-120 (in Russian). DOI: 10.15514/ISPRAS-2019-31(4)-7

1. Введение

В работе рассматривается задача обучения с учителем, когда требуется восстановить зависимость, отображающую векторное множество X в скалярное Y по конечному набору примеров такого отображения – обучающей выборке: $D = \{y_i, \vec{x}_i\}_{i=1}^{L}$. Данная задача относится к классу обратных задач, и, как и большинство обратных задач, является математически некорректной. Это выражается в том, что если строить решение методом наименьших квадратов по точкам обучающей выборки, то можно столкнуться с переобучением – ситуацией, когда модель хорошо описывает обучающее множество, но дает большую ошибку на тестовом.

Для борьбы с переобучением существует несколько подходов: метод валидационных выборок [1], метод регуляризации обучения [2], построение решения как суперпозиции ансамблей независимых предикторов [3]. В последнем случае работает эффект взаимной компенсации ошибок отдельных независимых предикторов при голосовании. Сами модели, участвующие в голосовании, строятся независимо друг от друга по обучающей выборке D.

Очевидно, что точность аппроксимации при таком подходе будет ограничена сверху тем обстоятельством, что конечная обучающая выборка позволит построить только конечное число независимых предикторов. После этого определенного улучшения точности предсказания можно добиться, например, введя дополнительно весовые коэффициенты для предикторов и проводя взвешенное голосование. Однако задача подбора весовых коэффициентов сама по себе также является некорректной, требующей регуляризации.

Для решения этой проблемы в статье предлагается искать весовые коэффициенты a_n перед предикторами $\Psi_n(\vec{x})$ путем минимизации квадратичной ошибки обучения в сумме со стабилизирующим функционалом в гауссовой форме:

$$\sum_i^L \left(y_i - \sum_{n=1}^N a_n \Psi_n(\vec{x}) \right)^2 + \lambda \sum_{n=1}^N a_n^2 , \quad (1)$$

где N – число базисных предикторов, а λ – регуляризационный множитель. Его значение будет находиться по методу Байеса [2,4].

В общем виде Байесовский метод поиска регуляризационного множителя сводится к дорогостоящей итерационной процедуре [2,4]. Однако в случае, когда решение ищется в виде ряда по набору базисных функций, а стабилизирующий функционал берется в гауссовой форме, можно применить метод ортогонализованных базисных функций (ОБФ) [5], который дает аналитическое выражение для λ, что существенно упрощает вычисления.

Ниже будет кратко изложена идея бэггинга на примере базовых обучаемых моделей двух типов – многослойных персептронов и деревьев решений; дано описание метода ортогонализованных базисных функций; предложен комбинированный алгоритм, использующий оба этих подхода; приведены результаты апробации предложенного алгоритма.

2. Ансамбли предиктивных моделей

В [3] был рассмотрен подход к задаче обучения с учителем, получивший название бэггинг. В нем решение строилось как суперпозиция различных (независимых) предиктивных моделей.

При этом сами модели получались обучением одного математического алгоритма на разных подмножествах обучающих данных. Было показано, что сложный предиктор, полученный голосованием ансамбля предиктивных моделей, дает в вероятностном смысле более точное решение по сравнению с простым предиктором.

Для построения ансамбля независимых предикторов с помощью какой-либо обучаемой модели необходимо обеспечить вариабельность процедуры обучения, внести в нее некоторый случайный фактор, чтобы различные эпохи обучения приводили в общем случае к разным конечным предикторам. Одним из способов обеспечения такой вариабельности является бэггинг, когда модель учится не на всем множестве данных, а только на случайной подвыборке. В результате предикторы, построенные на разных подмножествах данных, будут различны между собой и смогут создавать функциональный базис.

Кроме того, разнообразие предикторов может быть достигнуто благодаря особенностям самих обучаемых моделей. Например, многослойные персептроны [6] обучаются путем корректировки своих весов в сторону уменьшения ошибки обучения от некоторых начальных значений. В результате обучение сходится к состоянию, соответствующему некоторому минимуму ошибки (не обязательно глобальному). Какой конкретно минимум будет получен в результате обучения, зависит, в частности, от начальных значений весов персептрона. Таким образом, стартуя от разных начальных значений, мы будем получать разные предикторы.

В качестве еще одного примера можно привести деревья принятия решений, которые формируют набор разделяющих правил на основании обучающей выборки. При выборе разделяющего правила в узле дерева обычно используется так называемый «жадный» принцип, когда из всех возможных вариантов остается тот, который на данном шаге дает наилучшее значение разделяющего критерия [7]. Однако если при выборе разделяющего правила в каждом конкретном узле рассматривать только некоторое подпространство случайно выбранных входных признаков, а не все их множество, то можно построить ансамбль различных между собой деревьев решений. Данная идея реализуется в алгоритме, известном как *случайный лес* [8].

После того, как ансамбль предикторов $\Psi_n(\vec{x})$ построен, конечное решение может быть получено путем их простого голосования:

$$h(\vec{x}) = \frac{1}{N}\sum_{n=1}^{N} \Psi_n(\vec{x})$$

или взвешенного голосования:

$$h(\vec{x}) = \sum_{n=1}^{N} a_n \Psi_n(\vec{x}). \quad (2)$$

В данной работе рассматривается модель взвешенного голосования. Веса полагаются адаптивными параметрами ансамбля и подбираются путем минимизации ошибки обучения. Взвешенное голосование часто позволяет достичь лучшего или сравнимого с простым голосованием результата меньшим числом модулей, что делает модель более интерпретируемой и вычислительно менее затратной.

3. Метод ортогонализованных базисных функций

В методе ОБФ поиск решения $h(\vec{x})$ ведется в виде ряда по набору базисных функций (2). Предполагается, что векторы значений базисных функций в точках обучающей выборки $\vec{\Psi}_n = \{\Psi_n(\vec{x}_1), \Psi_n(\vec{x}_2)\}, \dots, \Psi_n(\vec{x}_L)$ ортогональны:

$$\sum_{i=1}^{L} \Psi_{m_i} \Psi_{n_i} = \delta_{m,n} \quad (3)$$

Если это условие не выполнено, то мы всегда можем построить линейное преобразование, приводящее к ортогональному набору векторов, который можно трактовать как значения

некоторых новых базисных функций в точках обучающей выборки. После того, как будут найдены коэффициенты разложения по ортогонализованным функциям, коэффициенты разложения решения по исходным функциям получаются обратным линейным преобразованием.

Делается стандартный для метода байесовской регуляризации набор предположений. Во-первых, предполагается, что данные зашумлены гауссовым шумом. В этом случае вероятность генерации решением $h(\vec{x})$ обучающего примера $y_i, \vec{x}_i$ можно оценить выражением:

$$P(y_i|h) = \frac{1}{Z_X} exp\left(-\beta\left(y_i - h(\vec{x}_i)\right)^2\right),$$

а генерации всей выборки $D = \{y_i, \vec{x}_i\}_{i=1}^{L}$, соответственно,

$$P(D|h) = \frac{1}{Z_X} exp\left(-\beta \sum_{i=1}^{L}\left(y_i - h(\vec{x}_i)\right)^2\right),$$

где β – параметр модели.

Во-вторых, делается предположение об априорной вероятности выбора того или иного решения. Априорная вероятность записывается в гауссовой форме:

$$P(h|\alpha,\beta) = \frac{1}{Z_A} exp\left(-\alpha \sum_{n=1}^{N} a_n^2\right).$$

Здесь также α – параметр модели.

По формуле Байеса

$$P(h|D,\alpha,\beta) = \frac{P(D|h)P(h|\alpha,\beta)}{P(D|\alpha,\beta)} \quad (4)$$

вероятность решения $h(\vec{x})$ будет равна [4]:

$$P(h|D,\alpha,\beta) = \frac{1}{Z_M} e^{-M}\ M = \beta, \text{где } \sum_{i=1}^{L}(y_i - h_i)^2 + \alpha \sum_{n=1}^{N} a_n^2$$

В приведенных формулах Z_X, Z_A, Z_M – нормировочные коэффициенты, которые получаются из условий нормировки соответствующих вероятностей на единицу. Если положить $\lambda = \frac{\alpha}{\beta}$, то функционал M будет полностью эквивалентен выражению (1).

Решение (2) ищется как наиболее вероятное. Максимизация вероятности решения $P(h|D,\alpha,\beta)$ (или минимизация M) по коэффициентам разложения a_n в случае выполнения условия (3) приводит к следующему выражению для их значений [5]:

$$a_n = \frac{\beta}{\beta + \alpha} \sum_{i=1}^{L} y_i \Psi_{n_i} \quad (5)$$

Параметры α и β находятся, как наиболее правдоподобные, путем максимизации логарифма знаменателя в формуле Байеса (4) [2, 4]. Последний выражается через нормировочные коэффициенты при экспонентах:

$$lnP(D|H) = lnZ_M - lnZ_A - lnZ_X.$$

В результате, как показано в [5], получается следующее выражение для логарифма знаменателя:

$$lnP(D|H) = \frac{\beta^2 S}{\beta + \alpha} + \frac{N}{2} ln \frac{\alpha}{\beta + \alpha} - \beta \vec{y}^2 + \frac{L}{2} ln \frac{\beta}{\pi},$$

где

$$S = \sum_{n=1}^{N} \left(\vec{y}\vec{\Psi}_n\right)^2, \quad (6)$$

$\vec{y}$ – вектор значений искомой функции в точках обучающей выборки.

Максимизация данного выражения по α и β сводится к системе двух нелинейных уравнений, которая имеет единственное аналитическое решение для параметров модели. В результате α и β выражаются через число точек в обучающем множестве L, число базисных функций N, квадрат вектора значений функции в точках обучающего множества $\vec{y}$ и величину (6):

$$\alpha = \frac{1}{2}\frac{(L-N)}{\left(\frac{L}{N}S - y^2\right)}, \qquad \beta = \frac{1}{2}\frac{(L-N)}{(\vec{y}^2 - S)} \tag{7}$$

Подробные выкладки приведены в работе [5].

Вычислив регуляризационные параметры α и β, можно получить коэффициенты разложения по ортогонализованным функциям (5) и, применив к ним преобразование, обратное ортогонализующему, получить коэффициенты разложения по исходным функциям. Таким образом, метод ортогонализованных базисных функций приводит к единственному решению для коэффициентов a_n в выражении (2), соответствующему максимуму правдоподобия.

4. Алгоритм построения взвешенного ансамбля

В качестве базовых обучаемых моделей в работе рассматривались многослойные персептроны и деревья принятия решений. Архитектура персептронов подбиралась заведомо простой, чтобы переобучение отдельного персептрона на обучающей выборке было невозможно. Их разнообразие обеспечивалось тем, что в каждом случае обучение проводилось от разных начальных значений синоптических коэффициентов и на разных подвыборках обучающих данных.

В случае деревьев принятия решений, разнообразие предикторов обеспечивалось тем, что каждое дерево строилось на случайно выбранной подвыборке обучающего множества (бэггинг).

Алгоритм построения решения выглядит следующим образом:

1) на обучающем множестве строится серия различных между собой предикторов;
2) вычисляются векторы значений полученных базисных предикторов в точках обучающей выборки;
3) векторы значений проверяются на линейную независимость, если условие независимости не выполнено, то число предикторов уменьшается;
4) строится ортогонализующее линейное преобразование этих векторов;
5) вычисляются параметры α и β по формулам (7);
6) находятся коэффициенты разложения по ортогонализованным базисным функциям (5);
7) рассчитываются коэффициенты разложения решения по исходным предикторам, для чего к коэффициентам разложения по ортогонализованным функциям применяется линейное преобразование, обратное ортогонализующему.

В работе [4] рассматривалась задача аппроксимации функции по точкам. Решение искалось в виде ряда по набору базисных функций путем минимизации функционала (1). Для поиска регуляризационного множителя применялся байесовский подход к регуляризации решения. Предложенный в [4] итерационный алгоритм выбора параметров модели дает значения весовых коэффициентов a_n, близкие к коэффициентам, полученным алгоритмом, рассматриваемым в данной статье. Однако алгоритм, рассмотренный в [4], обладает большими вычислительными затратами.

Итерационный алгоритм использует процедуру, где на каждом шаге решается система линейных алгебраических уравнений (СЛАУ), число уравнений которой равно числу базисных функций N. Таким образом, количество операций, необходимое для одной итерации, пропорционально N^3, а сложность всего алгоритма $\sim N^3 T$, где T – число итераций. Для надежной сходимости алгоритму обычно требуется не менее 10 итераций.

Наиболее ресурсоемким этапом рассматриваемого в данной статье алгоритма является шаг 4 – переход к ортогональному представлению. Вычислительная стоимость такой процедуры аналогична стоимости решения СЛАУ $\sim N^3$ операций. Таким образом, вычислительная стоимость приведенного алгоритма – порядка N^3 операций, что кратно меньше стоимости итерационного алгоритма $\sim N^3 T$.

Приведенная в работе модель сравнивалась с решением, полученным простым голосованием предикторов. Тестирование проводилось на стандартном дата сете для задачи регрессии, содержащем информацию о ценах на жилые объекты в Бостоне и параметрах, характеризующих эти объекты [9]. Данные были взяты из библиотеки scikit-learn (http://scikit-learn.org). Всего в выборке содержится 506 точек. В экспериментах 400 точек были использованы для обучения, на оставшихся 106-и проводилось тестирование.

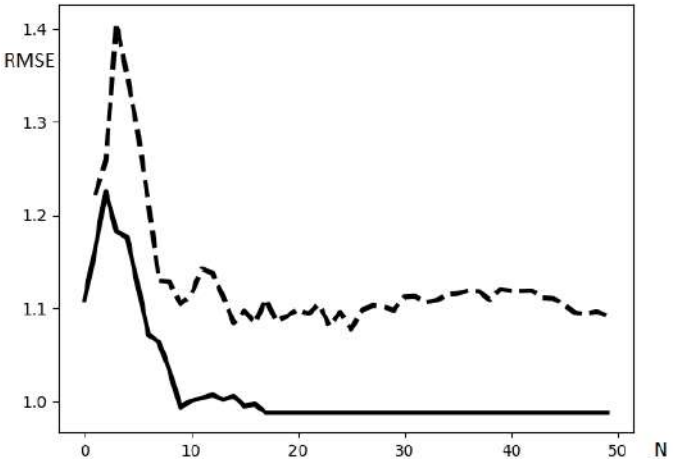


Рис.1. Графики зависимости значения RMSE от числа персептронов: пунктирная линия – решение ищется методом простого голосования, сплошная – решение представляется суммой модулей с весами, рассчитанными методом ОБФ

Fig.1. Plots of dependence of RMSE value on the number of perceptrons: dotted line – solution is determined by simple voting, solid line – solution is determined by the voting with weights, calculated by OBF method

В первом эксперименте в качестве базовой модели использовался многослойный персептрон.

На рис. 1 приведены графики зависимости квадратного корня из среднеквадратичных ошибок (RMSE) ансамблей предикторов на тестовом множестве от числа персептронов в них.

Пунктирная линия демонстрирует изменение RMSE модели, в которой решение выбирается простым усреднением модулей, сплошная линия – изменение RMSE модели, в которой веса перед модулями выбирались методом ортогонализованных базисных функций.

На рис. 2 приведены аналогичные графики для ансамблей, в которых в качестве базовой модели использовались деревья решений. Пунктирный график – зависимость RMSE от числа предикторов ансамбля, в котором решение получалось путем простого голосования, сплошной – та же зависимость для метода, в котором веса подбирались с помощью ОБФ.

В большинстве экспериментов ошибка взвешенного голосования оказывалась меньше, чем простого. Как правило, метод взвешенного голосования начинал выигрывать уже при малом количестве предикторов. На графиках видно, что в какой-то момент ошибка взвешенного голосования перестает меняться. Это связано с тем, что добавление очередного предиктора приводит к их линейной зависимости и последний добавленный модуль автоматически исключается алгоритмом. Таким образом, подбор коэффициентов методом ортогонализованных базисных функций позволяет не только добиваться более высокой точности ансамбля, но и ограничиваться при этом меньшим количеством модулей.

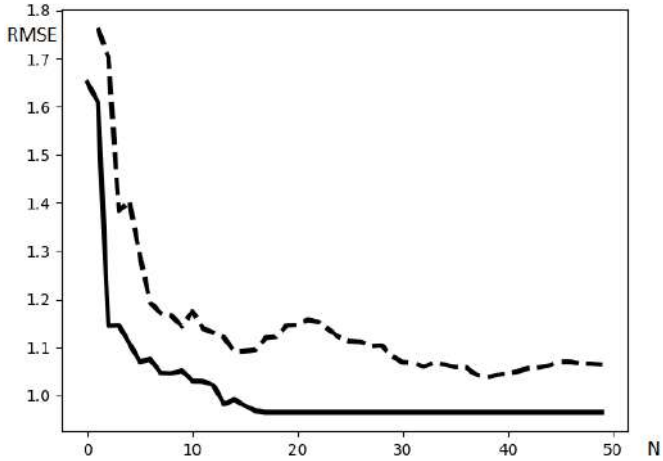


Рис.2. Графики зависимости значения RMSE от числа решающих деревьев: пунктирная линия – решение ищется методом простого голосования, сплошная – решение представляется суммой модулей с весами, рассчитанными методом ОБФ

Fig.2. Plots of dependence of RMSE value on the number of decision trees: dotted line – solution is determined by simple voting, solid line – solution is determined by the voting with weights, calculated by OBF method

5. Заключение

В работе рассматривалась задача подбора весовых коэффициентов в ансамбле голосующих предикторов. Задача решалась методом минимизации регуляризационного функционала, состоящего из суммы ошибки обучения и стабилизирующего функционала, взятого в гауссовой форме. Регуляризационный множитель подбирался по методу Байеса. Применение метода ортогонализованных базисных функций позволило уйти от итерационной процедуры подбора регуляризационного множителя. В этом подходе регуляризационный множитель имеет аналитическое выражение через исходные данные.

Предложенный алгоритм сравнивался с методом простого голосования предикторов. Проведенные численные эксперименты показали, что применение метода ортогонализованных базисных функций для подбора весовых коэффициентов, позволяет, во-первых, получать решение, приводящее, как правило, к меньшей ошибке на тестовой выборке, во-вторых, включающее меньшее число исходных модулей.

Список литературы / References

[1]. H.Zhu, R.Rohwer. No free lunch for cross-validation. Neural Computation, vol. 8, issue 7, 1996, pp. 1421-1426.
[2]. David J. C. MacKay. Bayesian Interpolation. Neural Computation, vol. 4, issue 3, 1992, pp. 415-447.
[3]. Breiman L. Bagging predictors. Machine Learning, vol. 24, no. 2, 1996, pp. 123–140.
[4]. А.С. Нужный, С.А. Шумский. Регуляризация Байеса в задаче аппроксимации функции многих переменных. Математическое моделирование, 2003, том 15, № 9, стр. 55-63 / A.S. Nuzhny, S.A. Shumsky. The Bayes regularization in the problem of function of many variables approximation. Matematical Modeling, vol. 15, no. 9, 2003, pp. 55–63 (in Russian).
[5]. A.S. Nuzhny. Bayesian regularization in the problem of point-by-point function approximation using orthogonalized basis. Mathematical Models and Computer Simulations, vol. 4, issue 2, 2012, pp. 203-209.
[6]. Simon Haykin. Neural Networks: A Comprehensive Foundation (2nd Edition). Prentice Hall, 1998, 842 p.
[7]. Breiman L., Friedman J.H., Stone C.J. , Olshen R.A. Classification and regression trees. Chapman and Hall/CRC, 1984, 368 p.
[8]. Breiman Leo. Random Forests. Machine Learning, vol. 45, no. 1, 2001, pp. 5—32.

[9]. Harrison D. and Rubinfeld D.L. Hedonic prices and the demand for clean air. Journal of Environmental Economics and Management, vol. 5, no. 1, 1978, pp. 81-102.

Информация об авторе / Information about the author

Антон Сергеевич НУЖНЫЙ – кандидат физико-математических наук, научный сотрудник ИБРАЭ РАН. Его научные интересы включают машинное обучение, распознавание образов, интеллектуальный анализ данных, информационный поиск, некорректные задачи.

Anton Sergeevich NUZHNY – Candidate of Physics and Mathematics, Research Fellow, IBRAE RAS. His research interests include machine learning, pattern recognition, data mining, information retrieval, and incorrect tasks.

DOI: 10.15514/ISPRAS-2019-31(4)-8

Local search metaheuristics for Capacitated Vehicle Routing Problem: a comparative study

S.M. Avdoshin, ORCID: 0000-0001-8473-8077 <savdoshin@hse.ru>
E.N. Beresneva, ORCID: 0000-0001-6710-2843 <eberesneva@hse.ru>
Department of Software Engineering,
National Research University Higher School of Economics,
20, Myasnitskaya st., Moscow, 101000 Russia

Abstract. This study is concerned with local search metaheuristics for solving Capacitated Vehicle Routing Problem (CVRP). In this problem the optimal design of routes for a fleet of vehicles with a limited capacity to serve a set of customers must be found. The problem is NP-hard, therefore heuristic algorithms which provide near-optimal polynomial-time solutions are still actual. This experimental analysis is a continue of previous research on construction heuristics for CVRP. It was investigated before that Clarke and Wright Savings (CWS) heuristic is the best among constructive algorithms except for a few instances with geometric type of clients' distribution where Nearest Neighbor (NN) heuristic is better. The aim of this work is to make a comparison of best-known local search metaheuristics by criteria of error rate and running time with CWS or NN as initial algorithms because there were not found any such state-of-the-art comparative study. An experimental comparison is made using 8 datasets from well-known library because it is interesting to analyze "effectiveness" of algorithms depending on type of input data. Overall, five different groups of Pareto optimal algorithms are defined and described.

Keywords: capacitated vehicle routing problem; local search metaheuristics; LKH-3; variable record-to-record travel; simulated annealing; guided local search; tabu search

For citation: Avdoshin S.M., Beresneva E.N. Local search metaheuristics for Capacitated Vehicle Routing Problem: a comparative study. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 4, 2019, pp. 121-138. DOI: 10.15514/ISPRAS-2019-31(4)-8

Эвристические методы конструирования маршрута для решения задачи маршрутизации с ограничением по грузоподъемности

С.М. Авдошин, ORCID: 0000-0001-8473-8077 <savdoshin@hse.ru>
Е.Н. Береснева, ORCID: 0000-0001-6710-2843 <eberesneva@hse.ru>
Департамент программной инженерии,
Национальный исследовательский университет «Высшая школа экономики», 101000,
Россия, г. Москва, ул. Мясницкая, д. 20.

Аннотация. Задача маршрутизации – одна из широко известных задач комбинаторной оптимизации. Она состоит в отыскании оптимального множества маршрутов для транспортных средств с целью однократного обслуживания определенного множества клиентов. В данной работе исследуется подвид задачи маршрутизации – задача маршрутизации с ограничением по грузоподъемности, в которой каждое транспортное средство имеет свою грузоподъемность. Задача является NP-трудной, поэтому вместо точных алгоритмов решения исследуются только эвристические алгоритмы, позволяющие получить приближенные решения за полиномиальное время. Данная работа является продолжением исследования, посвященного алгоритмам конструирования маршрута; оно позволяет получить первоначальные решения для данной задачи. Было выяснено, что эвристика Clarke and Wright Savings

(CWS) является одной из лучших, за исключением наборов данным с геометрическим расположением точек, для которых лучшим является алгоритм Nearest Neighbor (NN). Целью работы является сравнение локально-оптимальных метаэвристических алгоритмов решения задачи маршрутизации с ограничением по грузоподъемности по двум критериям: точность и время решения, для получения начального решения используются алгоритмы CWS и NN. Выявлено пять Парето оптимальных групп алгоритмов для различных типов входных данных. Интересно, что алгоритм «Lin, Kernighan and Helsgaun heuristic» (LKH-3), входящий во все Парето оптимальные группы для смежной задачи (задача коммивояжера), в данном случае вошел только в четыре группы из пяти.

Ключевые слова: задача маршрутизации с ограничением по грузоподъемности; локально-оптимальные метаэвристики; LKH-3; алгоритм переменного перемещения: метод отжига; алгоритм управляемого поиска; алгоритм поиска с запретами

Для цитирования: Авдошин С.М., Береснева Е.Н. Локально-оптимальные метаэвристики для решения задачи маршрутизации с ограничением по грузоподъемности. Труды ИСП РАН, том 31, вып. 4, 2019 г., стр. 121-138. DOI: 10.15514/ISPRAS-2019-31(4)-8

1. Introduction

The Vehicle Routing Problem (VRP) is one of the most widely known challenges in a class of combinatorial optimization problems. VRP is directly related to Logistics transportation problem and it is meant to be a generalization of the Travelling Salesman Problem (TSP). In contrast to TSP, VRP produces solutions containing some (usually, more than one) looped cycles, which are started and finished at the same point called a “depot”. As in TSP, each customer must be visited only by one vehicle. The objective is to minimize the cost (time or distance) of all tours. Despite the fact that both problems belong to the class of NP-hard tasks, VRP has higher solving complexity than TSP for the identical types of input data.

This work is aimed at analysis of VRP subcase, which is called Capacitated Vehicle Routing Problem (CVRP), where the vehicles have a limited capacity. A new constraint is that the total sum of demands in a tour for any vehicle must not exceed its capacity. In the paper we will use CVRP abbreviation having in mind the mathematical formulation that was described in our previous work [1].

There are three types of algorithms that are used to solve any subcase of CVRP: exact algorithms, constructive heuristics and metaheuristics.

- *Exact algorithms.* These algorithms find an optimal solution but take a great time for solving large instances. State-of-the-art exact methods can provide optimal solution for some SCVRP instances with up to 100 nodes, but it takes 30-40 minutes at average [2]. Due to these restrictions, researchers all over the world concentrate on heuristic methods.
- *Constructive heuristics.* They build an approximate solution iteratively, but they do not include further improvement stage. These heuristics are usually used for generation of an initial solution for improvement algorithms. A lot of experiments show that classical heuristics work much faster than to exact methods. For example, an instance of 100-150 nodes can be solved up to a few (1-2) seconds [2].
- *Metaheuristics*. These algorithms take as input some approximate initial solution and try to iteratively improve it. According to [3], metaheuristics are divided into two groups: metaheuristics based on local search and metaheuristics based on population, or natural inspired ones. The first group look for new solutions by moving at each iteration from a current state to another state in its neighborhood, while the second group works on a basis of a population of solutions which may be combined together in the hope of generating better ones, like in nature. Certain limitations are inevitable in any research, hence in this work we concentrate only on the first group of metaheuristics, because one of the most perspective algorithms for TSP – LKH-3 – belongs to this group, and it is very interesting for us to compare it with its “closest” alternatives.

Capacitated vehicle routing problems CVRP form the core of logistics planning and are hence of great practical and theoretical interest. There is no doubt that actuality of research and development of heuristics algorithms for solving CVRP is on its top, because in a real word there can be up to one thousand clients in a delivery net, that is why it is especially important to explore heuristic algorithms that allow to quickly generate near optimal solution in a polynomial time.

There are a lot of articles related to CVRP local search metaheuristics, but no works were found which compare improvement heuristics using the same input data of different types and sizes. We will compare these algorithms under criteria of quality, or error rate, and running time. Under the error rate we mean the percentage of difference in the obtained value of the solution with the optimal (or best-known) solution for the problem.

The aim of this work is to make a comparison of best-known local search metaheuristics by criteria of solution quality and running time with CWS or NN as initial algorithms as there were not found any such state-of-the-art comparative study. In addition, it is important to define sets of Pareto optimal metaheuristics for different types of input data.

The paper is structured as follows. In the second part, a general local search approach is described. After that, in the third section, some notes on most popular local search metaheuristics are provided, including short description of chosen algorithms to be intercompared. The fourth part presents design of experiments on local search metaheuristics. The fifth and the sixth sections describe results of solution qualities and computing times of algorithms, respectively. The seventh part consists of definition of Pareto optimal metaheuristics and five such sets are presented. In the last part we summarize our findings and suggest areas for future work.

2. Local search approach

Local search algorithms take as input some approximate initial solution and try to iteratively upgrade it with local improvements. These changes can either improve a single route (intra-route optimization) or change more than one routes simultaneously in such a way that the overall solution is improved (inter-route optimization).

Intra-route and inter-route optimization strategies consist of different schemes, which are fully described in [5]. In this research we will use the most-known and simplest but still effective local improvement heuristics: 1-point and 2-point moves, 2-opt.

The set of all solutions that can be obtained by applying the local improvements on a solution s is called the neighborhood $N(s)$. Of course, the bigger neighborhood is, the more likely it contains a new solution that can improve current one. However, to have large neighborhoods means to have inevitably higher computational complexity since more solutions need to be generated and evaluated [6]. At the same time, local search methods must deal with the problem of being stuck in a local optimum. Thus, a lot of methods to escaping the local optimum are applied, they will be described in the next section.

So, basically, local search approach consists of the following main steps.

1) Taking as input some initial solution s.
2) Generation of a neighborhood$N(s)$.
3) Selection of the best solution s^* from $N(s)$ using some acceptance criteria.
4) Make a new s equal to s^*.
5) Checking for exceeding different limits. If stop criteria is satisfied then terminate, otherwise continue with the step 2.

3. CVRP local search metaheuristics

Local search metaheuristics are used to solve a wide range of combinatorial optimization problems. Among heuristic methods for solving TSP there is one, which is the best – it is local search

metaheuristic, proposed by Lin, Kernighan and Helsgaun [7]. Also, local search algorithms are key part of most known methods for solving most subcases of VRP [3] [8] [9] [10] etc.

The most well-known schemes for solving CVRP that include local search steps are different variants of tabu search, forms of deterministic and simulated annealing, variable neighborhood search, guided local search [11]. Recently a new adoption of LKH for TSP called LKH-3 was proposed by one of the original authors, Helsgaun [12] . Also, in a recent study it was stated that improved version of record-to-record travel heuristic analyzed [13] is "… a well-performing metaheuristic", which combines strategies of deterministic annealing, tabu search, variable neighborhood search and both intra-route and inter-route optimizations described later. It was proposed by Li and others [14].

Of course, there are a lot of other metaheuristics for finding CVRP solution, however it was decided to concentrate on a set of several reputed local search algorithms that were honorably mentioned in recent studies.

As it was stated earlier, for all metaheuristics an initial solution must be obtained. For most input problems Clarke and Wright Savings (CWS) heuristic [14] is used, which is the best among construction algorithms except for a few instances with geometric type of clients' distribution. For these especial input files Nearest Neighbor (NN) heuristic is applied instead of CWS.

Thus, in this study we will intercompare following local search metaheuristics for solving classical CVRP.

3.1 A set of optimization operators (OPT)

As it was stated above, in this research the most-known and simplest local improvement heuristics are used. They are 1-point move, 2-point move and 2-opt.

In a tour of N vertices 1-point move operator (or relocate heuristic) moves some vertex $v_i, i \in N$ after another vertex $v_j, j \in N, i \neq j$ at the same tour. Another 2-point move operator (or exchange heuristic) swaps locations of two different vertices $v_i, i \in N$ and $v_j, j \in N, i \neq j$. And the main idea of 2-opt heuristic is to remove two edges $(v_i, v_j), i, j \in N$ and $(v_x, v_y), x, y \in N$ from the solution and replace them with two new edges (v_i, v_x) and (v_j, v_y). It is important to note that all mentioned operators can be applied for each of intra-optimization and inter-optimization as it is shown in Fig. 1.

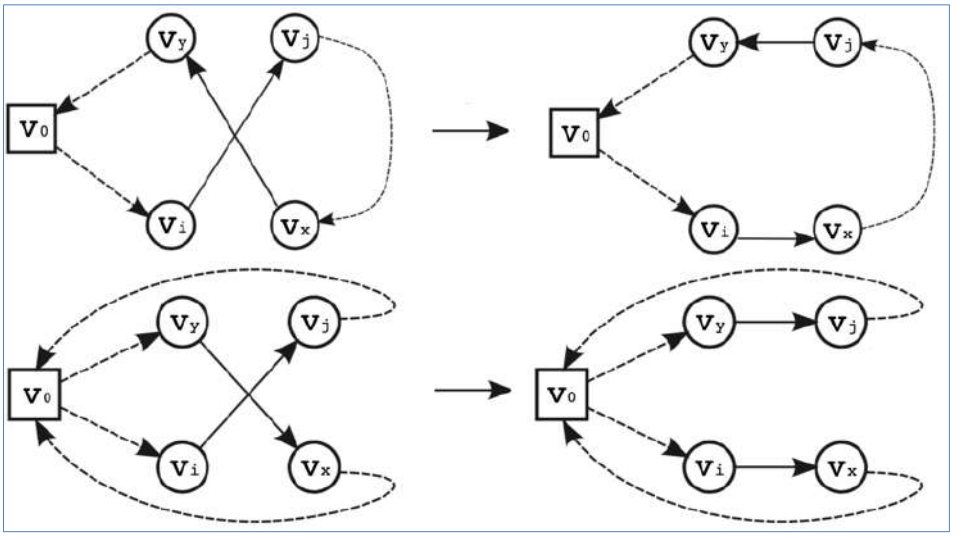


Fig. 1. Intra-route 2-opt (above) in compare to intre-route 2-opt (below).

3.2 Guided local search (GLS)

The guided local search metaheuristic is used to avoid local minima. It was initially proposed by [15] and later applied to CVRP by [16]. According to [17], this method is memory-based as it determines and penalizes "ineffective" edges by increasing its cost to a new $c^*(v_i, v_j) = c(v_i, v_j) + \lambda p(v_i, v_j) L$, where $p(v_i, v_j)$ counts the number of penalties of edge (v_i, v_j), L is a proxy for the average cost of an edge, computed as the costs of the starting solution divided by the number of customers, and λ controls the impact of penalties (authors suggest to always set $\lambda = 0.1$).

3.3 Tabu search (TS)

Tabu search originally was proposed by Glover and others [18]. The main idea of TS is as follows. If some solutions in $N(s)$ cannot be improved for several iterations or they violate the rules, they are made forbidden (or tabu) in order to prevent being in a stack of local minimum. Such forbidden solutions are put into a tabu-list, so this heuristic is also memory-based like GLS. The duration that a solution remains tabu is called its tabu-tenure and it can vary over different intervals of time.

We use recent TS algorithm that provides good results described in [19] as it was stated in [5].

3.4 Simulated annealing (SA)

This classical algorithm was developed in 1983 by [20]. It is based on an analogy from the annealing process of solids, where a solid is heated to a high temperature and gradually cooled in order for it to crystallize in a low energy configuration [21]. In this research we used adopted version of SA for CVRP by [22]. In SA some solution s^* from $N(s)$ at iteration i is chosen to be a new $s = \begin{cases} s^* \text{ with probability } P(s, s^*, i) \\ s \text{ with probability } 1 - P(s, s^*, i) \end{cases}$ accordingly to the probabilistic function $P(s, s^*, i) = \exp\left(-\frac{f(s) - f(s^*)}{Q_i}\right)$, where $f(s)$ is a length of solution s, Q_i is an element of an arbitrary decreasing, converging to zero, positive sequence, which specifies an analogue of the falling temperature in the crystal.

3.5 Lin-Kernighan-Helsgaun heuristic for CVRP (LKH-3)

LKH-3 is proposed by Helsgaun in 2017 [12]. The implementation of LKH-3 builds on the idea of transforming the problem into classical symmetric TSP. After that algorithm uses the principle of 2-opt algorithm and generalizes it. In this heuristic, the k-Opt, where $k = \overline{2..\sqrt{N}}$, is applied, so the switches of two or more edges are made in order to improve the tour. This method is adaptive, so decision about how many edges should be replaced is taken at each step [7] [23].

This algorithm was not developed by us as original source code of LKH-3 is free of charge for academic and non-commercial use and can be downloaded at [24].

3.6 Variable record-to-record travel heuristic (VRTR)

Li and others suggested a variable record-to-record travel heuristic, which is based on classical record-to-record travel algorithm (RTR). RTR combines approaches of deterministic annealing (which is a variant of simulated annealing heuristic) and tabu search. The main differences between VRTR and RTR are as follows. Firstly, VRTR considers 1-point, 2-point and 2-opt moves not only within individual routes as RTR does, but also between them. Secondly, "VRTR uses a variable-length neighbor list that should help focus the algorithm on promising moves and speed up the search procedure" [14].

4. Design of experiments

All algorithms from section III were implemented as sequential algorithms in C/C++, no multi-threading was explicitly utilized. They were executed on an Intel Core i5 clocked at 1.3GHz with 4 GB RAM running the macOS 10.14.3 operating system.

The computational testing of the solution methods for CVRP has been carried out by considering eight sets of test instances from the next well-known database [25]. Total number of instances in sets A, B, E, F, G, M, P, X is 211. All instances inside one set have its own characteristics and a way of generation: cluster-based / uniform / geometric distribution of clients, real-world / imitative cases etc. The integer Euclidean metric is used for all instances. The naming scheme and data format for each instance is described here [26]. Shortly, the first letter in names shows the name of used set, the figure after letter 'n' shows the number of nodes and the figure which stands after letter 'k' presents the number of vehicles.

Experiment starts with choice of a local search metaheuristic M from set {OPT, GLS, SA, TS, LKH-3, VRTR}. After one dataset D is selected from a list of all mentioned benchmark datasets, an instance file F from chosen dataset D is taken. Next, the following steps are repeated 51 times on instance F: chosen metaheuristic M is executed on a basis of initial solution obtained by CWS or NN (as it was explained in a previous chapter). During all iterations, except the first one, solution qualities $\varepsilon_{it}(M, F)$ and computing times $t_{it}(M, F)$ (in seconds) are calculated for algorithm M on test F. The first run is not taken into account in calculations because of specifity of C++ compiler. Solution quality (or percent above best-known, or gap) is calculated using the next formula [11]:

$$\frac{F(S^0) - F_{opt}(S)}{F_{opt}(S)} \cdot 100\%,$$

where $F(S^0)$ is a length of obtained solution and $F_{opt}(S)$ is a length of optimal solution or best-known one.

Also we calculate minimal value $\varepsilon_{\min}(M, F)$ among all figures $\varepsilon_{it}(M, F)$ and sample mean $\bar{X}_t(M, F) = \frac{1}{50}\sum_{it=1}^{50} t_{it}(M, F)$ among all figures $t_{it}(M, F)$. And finally, among all $\varepsilon_{\min}(M, F)$ from one dataset average sample mean $\bar{X}_\varepsilon(M, D) = \frac{1}{|D|}\sum \varepsilon_{\min}(M, F), \forall F \in D$ is calculated, which shows average gap for algorithm M on dataset D. And among all $\bar{X}_t(M, F)$ from one dataset average sample mean $\bar{X}_t(M, D) = \frac{1}{|D|}\sum \bar{X}_t(M, F), \forall F \in D$ is calculated, which shows average computing time of algorithm M on dataset D. $|D|$ is a number of input files in dataset D.

The plan of experiment on local search metaheuristics is described in fig. 2.

```
    Input: local search metaheuristics, datasets
1:  foreach local search metaheuristic M
2:      foreach dataset D from datasets
3:          foreach instance file F from D
4:              for it ∈ {0…50} // number of runs
5:                  init_sol = run CWS or NN on F
6:                  final_sol = run M on F with init_sol
7:                  if (it != 0) // if not the first run
8:                      calculate ε_it(M,F), t_it(M,F)
9:              calculate ε_min(M,F)
10:         calculate t_min(M,F)
11:         calculate t_max(M,F)
12:         calculate X̄_t(M,F)
13:         calculate s_t(M,F)
14:     calculate X̄_ε(M,D) // average gap on dataset
15:     calculate X̄_t(M,D) // average computing time
```

Fig. 2. Plan of experiment on constructive heuristics

Each metaheuristic is subsequently launched on all instances from every mentioned dataset, so no input file is missed.

5. Computational results on solution quality

Results about the best (= minimal) solution qualities $\varepsilon_{\min}(M,F)$ of metaheuristics available from experiments for set A are presented in fig. 3. The horizontal axis represents the name of instance data. The vertical axis shows the solution quality.

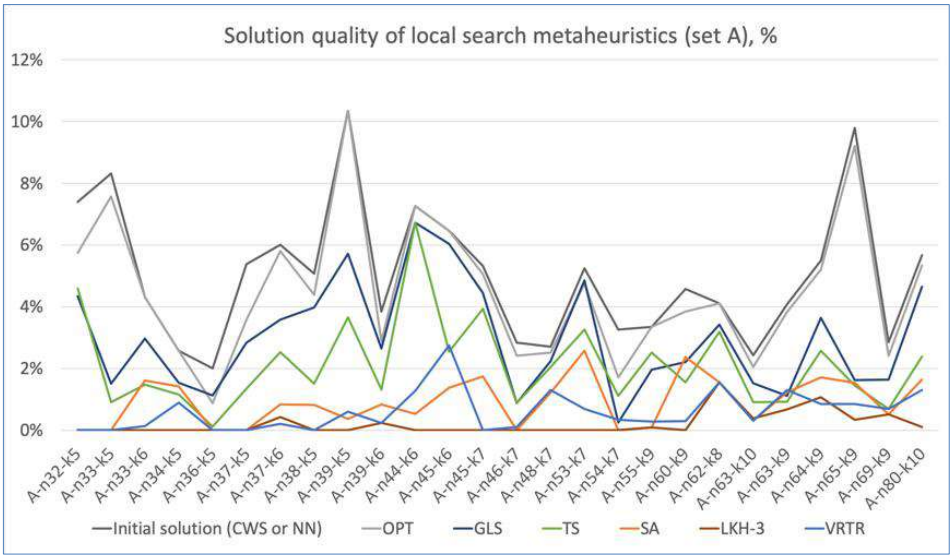


Fig. 3. Solution quality of local search metaheuristics, set A

Results for other sets cannot be presented here as detailed as for set A, because of its size, but they are aggregated in Table 1, where average gaps $\bar{X}_{\varepsilon}(M,D)$ for all metaheuristics and all datasets are given.

Table 1. Average gap $\bar{X}_{\varepsilon}(M,D)$ in the dataset, %.

Average gap $\bar{X}_{\varepsilon}(M,D)$ in the dataset		Local search metaheuristics						
		CWS or NN	OPT	GLS	TS	SA	LKH-3	VRTR
Set (its size)	A (26)	5,0%	4,5%	3,0%	2,1%	0,9%	0,2%	0,6%
	B (23)	4,4%	3,9%	3,1%	2,5%	1,0%	0,2%	0,6%
	E (11)	7,1%	5,3%	4,1%	3,6%	0,8%	0,4%	0,8%
	F (3)	4,4%	2,7%	3,3%	3,0%	1,8%	0,1%	1,9%
	G (20)	11%	10%	8,1%	9,7%	5,2%	2,1%	2,4%
	M (4)	4,7%	2,8%	2,6%	2,1%	0,5%	0,2%	0,5%
	P (24)	8,0%	6,6%	3,5%	4,5%	0,7%	0,5%	0,9%
	X(100)	5,9%	5,4%	4,9%	4,0%	4,1%	2,0%	1,7%

Fig. 4 is a visual representation of this table. These general figures can show an approximate overall effectiveness of algorithms by criterion of solution quality. Analysis of fig. 4 indicated a group of top-3 algorithms by criterion of solution quality: they are LKH-3, VRTR, SA. Let's take a closer look at their results.

It is clearly seen that in 7 out of 8 sets LKH-3 produces solutions with the least (= the best) solution qualities. Experiment results that are not shown here because of their large size reveal that LKH-3 produces not the best solutions in:

- 3 input files out of 26 for set A (≈12%);

- 1 input files out of 23 for set B (≈4%);
- 2 input files out of 11 for set E (≈18%);
- 1 input files out of 3 for set F (≈33%);
- 6 input files out of 20 for set G (30%);
- 0 input files out of 4 for set M (0%);
- 6 input files out of 24 for set P (≈25%);
- 61 input files out of 100 for set X (61%).

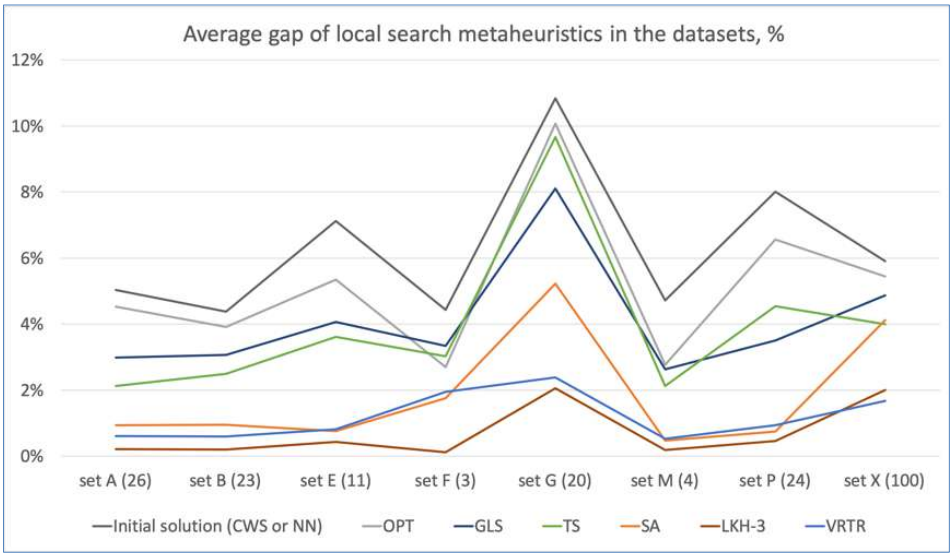


Fig. 4. Average gap of local search metaheuristics in the datasets (%)

On total, LKH-3 was not the best in quality criterion for 80 input files out of 211 (≈38%). Only 6 times SA was the best, while all other times VRTR was «the winner».

It is important to mention that LKH-3 tends to produce best solution for instances with no more than ≈100 clients in a delivery net regardless of type of distribution in the dataset. There are 86 instances with less than 102 clients, and solutions obtained using LKH-3 are the best in 86% (in 74 files).

In addition, it should be noted that LKH-3 is the best for input problems with cluster-based distribution of clients, when the number of clusters is a bit smaller than the amount of available vehicles (sets B and M). On the contrary, this algorithm is not the best for 61% of files from set X that consists of very different instances. Despite the fact that there are several files with cluster-based distribution, the amount of clusters is much less than the number of vehicles, and, as experiments showed, LKH-3 does not suits well for such cases.

VRTR nearly always takes «the second place in this race» except for set X. It can be noticed that VRTR works in a best way for instances with more than ≈320 clients in a delivery net for non-geometric distributions. There are 53 instances with more than 321 clients in set X, and solutions obtained using VRTR are the best in 89% (in 47 files). Nevertheless, if we take a range of ≈ [100; 320] clients, results show that either VRTR or LKH-3 are the best in nearly 50% of cases, so for this diapason both these algorithms can be admitted being equal.

For most of input files SA produces solutions which are nearly equal to other ones generated by VRTR. However, the situation is different for sets G and X, where SA is always worse than its closest “competitor”. So, we can come to the conclusion that with SA it is better to use non-geometric input data with no more than 100 clients in a delivery net. Nevertheless, only 6 times out of 211 SA is better than LKH-3 – this fact shows superiority of LKH-3 over SA.

Other three local search metaheuristics – TS, GLS and OPT (listed in increasing size of average gaps) – were nearly always worse than top-3 group.

TS was better than LKH-3 only 3 times out of 211, better than VRTR or SA in 6 input files out of 26 from set A, in 2 input files out of 23 from set B, in 1 input file out of 24 from set P. Also, TS is slightly more effective than SA for set X, however, the difference in solution qualities between LKH-3 and TS is significant in general.

Roughly speaking, GLS is usually worse than TS, except for sets G and P. Also, GLS is better than LKH-3 only in one file, thus, it cannot compete with the algorithms from top-3 group seriously.

And the last one and the least effective by criterion of solution quality metaheuristic is OPT. It nearly always produces solutions which are better than ones obtained by simple initial algorithm but worse than other local search metaheuristics.

6. Experimental results on computing time

Results about average running times $\bar{X}_t(\mathrm{M},\mathrm{F})$ of metaheuristics available from experiments for set A are presented in fig. 5. The horizontal axis represents the name of instance data. The vertical axis shows the running time in seconds.

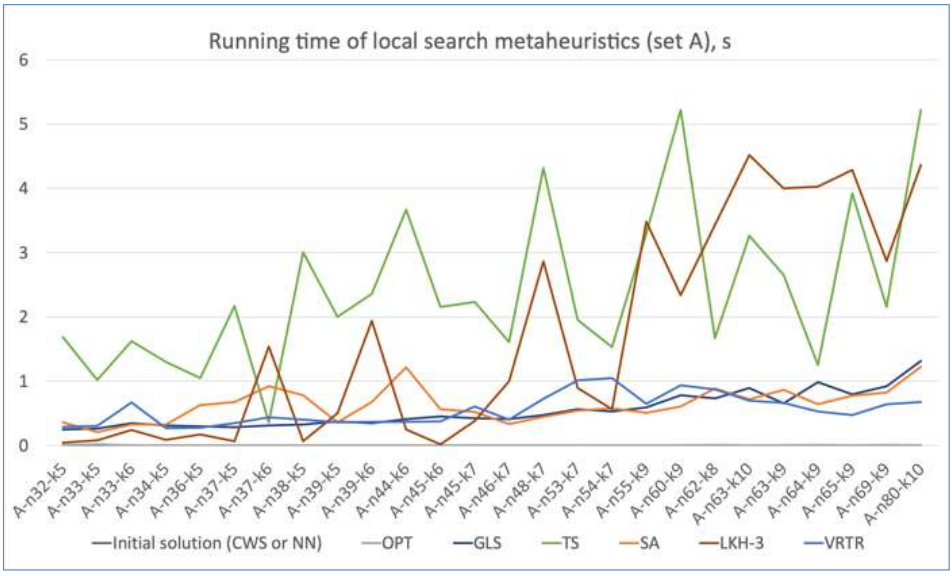


Fig. 5. Running time of local search metaheuristics, set A

Results for other sets cannot be presented here as detailed as for set A, because of its size, but they are aggregated in Table 2, where average computing times $\bar{X}_t(\mathrm{M},\mathrm{D})$ for all metaheuristics and all datasets are given.

Table 2. Average computing time $\bar{X}_t(M,D)$ in the dataset (in seconds)

Average gap $\bar{X}_\varepsilon(\mathrm{M},\mathrm{D})$ in the dataset		Local search metaheuristics						
		CWS or NN	OPT	GLS	TS	SA	LKH-3	VRTR
Set (its size)	A (26)	0,0003	0,004	0,540	2,411	0,635	1,692	0,554
	B (23)	0,0006	0,027	0,291	2,578	0,505	1,487	0,611
	E (11)	0,0008	0,016	0,311	2,172	0,857	1,930	0,851
	F (3)	0,0003	0,024	0,953	4,240	1,621	2,154	1,270
	G (20)	0,0468	0,569	7,979	16,34	15,78	8,563	9,748
	M (4)	0,0025	0,044	0,883	4,386	4,046	2,370	2,469
	P (24)	0,0008	0,023	0,367	2,216	0,566	2,166	0,554
	X (100)	0,041	0,62	7,088	65,15	56,74	37,41	9,998

Fig. 6 is a visual representation of this table. These general figures can show an approximate overall effectiveness of algorithms by criterion of running time.

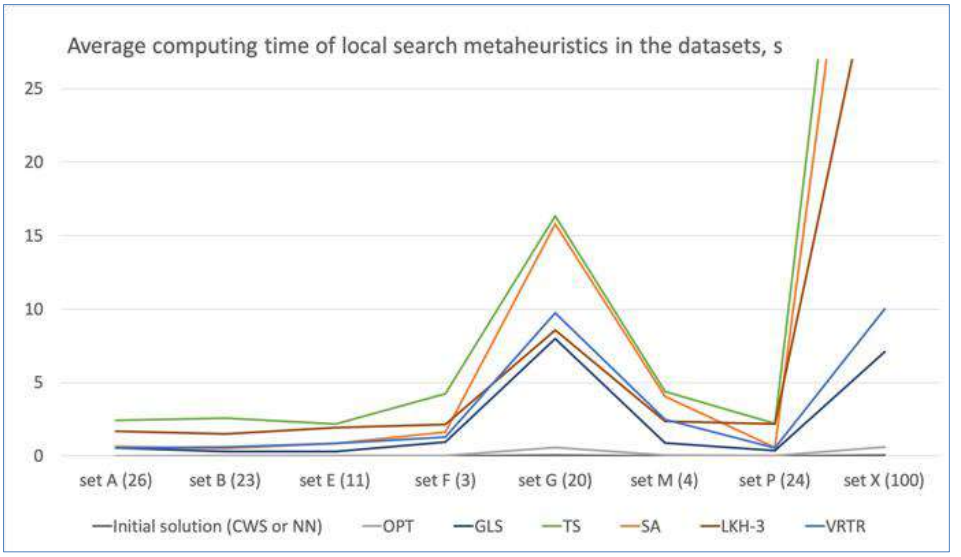


Fig. 6. Average computing time of local search metaheuristics in the datasets

It can be seen from Table 2 and fig. 4 that metaheuristics, which are listed in increasing size of average running times, are as follows: OPT, GLS, VRTR, SA, LKH-3, TS. Let us discuss their results in this order.

Analysis reveals that the «fastest» local search metaheuristic for all 211 instances is OPT as it was expected. Its running time is approximately 25 times bigger than computing time of initial algorithm, however, even for 1000 clients in a delivery net the maximum running time does not exceed 3 seconds.

Next algorithm is GLS, which is approximately 300 times slower than initial one. In sets B, E, F, P, M and X it nearly always (94%) ranks #2 just after OPT. Of course, there are cases when GLS works slightly slower than others, but the number of such situations is not very significant and the difference between obtained values does not exceed 1 second. However, in sets A and G GLS works with mixed results: computing times of GLS, SA, VRTR or LKH-3 (depending on dataset) are fluctuated and too close to each other, so it is impossible to find a leader.

Average computing time of VRTR steadily goes at the third plays, except for set G with geometric instances and several rare cases from other datasets. VRTR has smooth growth of speed, and no special aspects of its work are found apart from not very stable work with geometric-inspired instances.

Next one is SA. In comparison with VRTR, SA has bigger growth of speed and the plot of its running time is more fluctuated. In sets A, B, E, F and P this algorithm executes quicker than LKH-3 but slower than VRTR. However, in sets G, M and X it shows much worse effectivity, when the number of clients in a delivery net becomes more than 100. It means that SA is better to use for instances with up to one hundred delivery points.

LKH-3 is slower than other mentioned metaheuristics (except for TS) for all datasets apart from sets G, M and those instances from set X with 322 and more delivery points. The main unique feature of LKH-3 is its variability. Linear chart of running times of LKH-3 has a lot of drastic jumps and slumps. That is why this metaheuristic has not very positive computing time rate. Nevertheless, LKH-3 can work very quickly, especially when there are no more than 50-80 clients in a net.

Last one metaheuristic to be discussed concerning its computing time is TS. As LKH-3, linear chart of running times has a lot of drastic jumps and slumps. In average, this is the slowest algorithm in this group, however, in a third of cases it can compete with LKH-3 or SA but not very significant speeding up can be noticed.

7. Pareto optimal local search metaheuristics

The algorithm $m_0 \in \mathcal{M}$ is Pareto optimal if $(\forall m \in \mathcal{M})\Big((m \neq m_0) \Rightarrow \big(\bar{X}_\varepsilon(m,D) > \bar{X}_\varepsilon(m_0,D)\big) \vee \big(\bar{X}_t(m,D) > \bar{X}_t(m_0,D)\big)\Big)$. Thus, our aim is to find a sets of Pareto optimal algorithms for different types of input data.

Figures from 7 to 16 are plotted using values from Table 1 and Table 2. The horizontal axis represents average computing time $\bar{X}_t(\mathrm{M},\mathrm{D})$ in seconds. The vertical axis shows average solution quality $\bar{X}_\varepsilon(\mathrm{M},\mathrm{D})$.

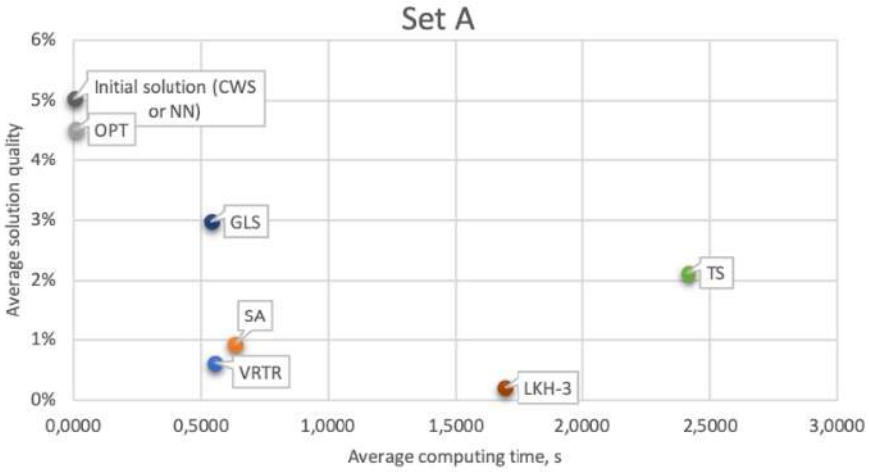


Fig. 7. Average solution quality vs. average running time, set A

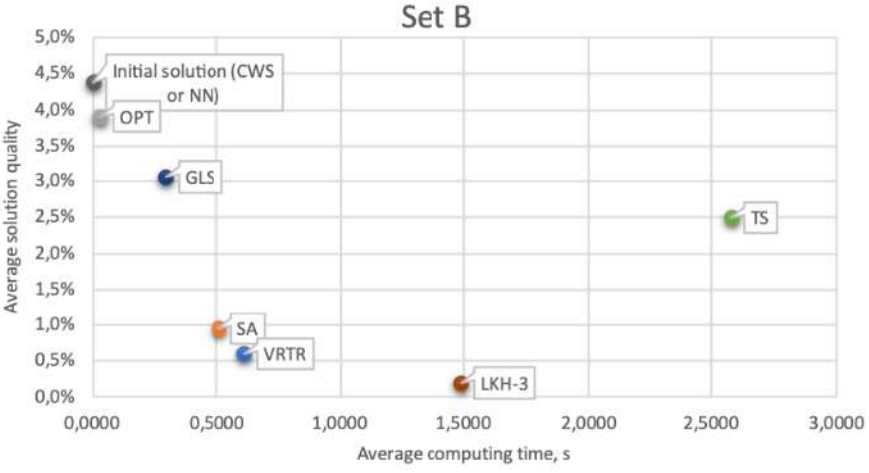


Fig. 8. Average solution quality vs. average running time, set B

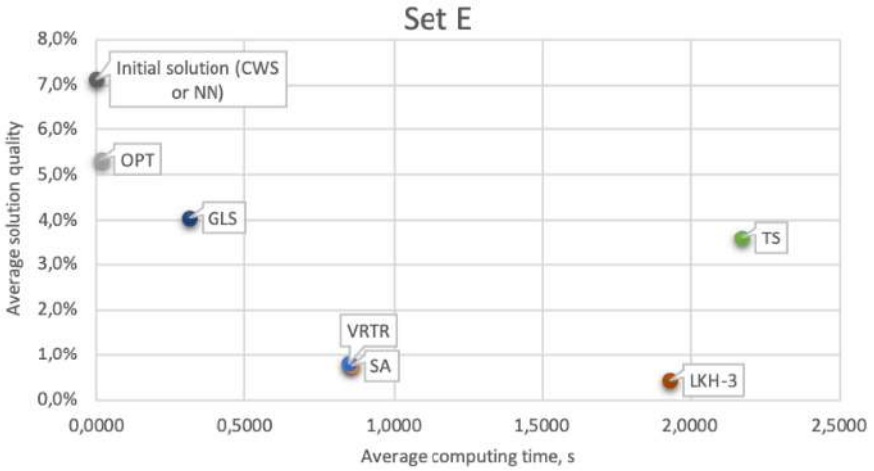


Fig. 9. Average solution quality vs. average running time, set E

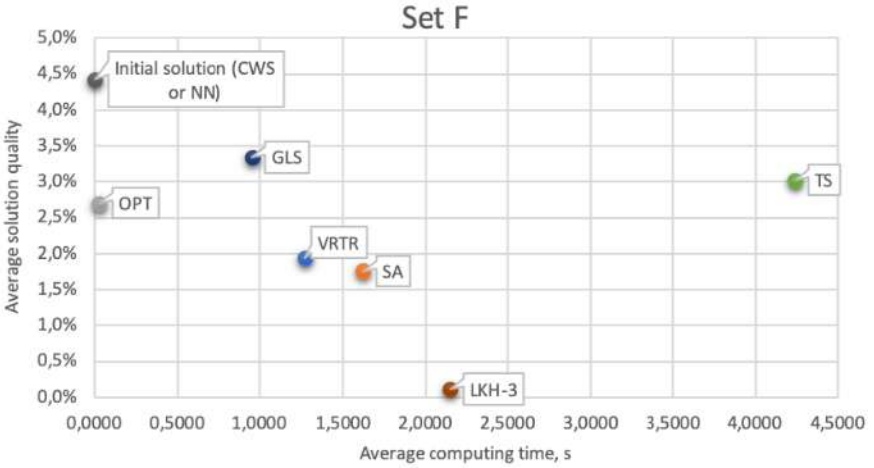


Fig. 10. Average solution quality vs. average running time, set F

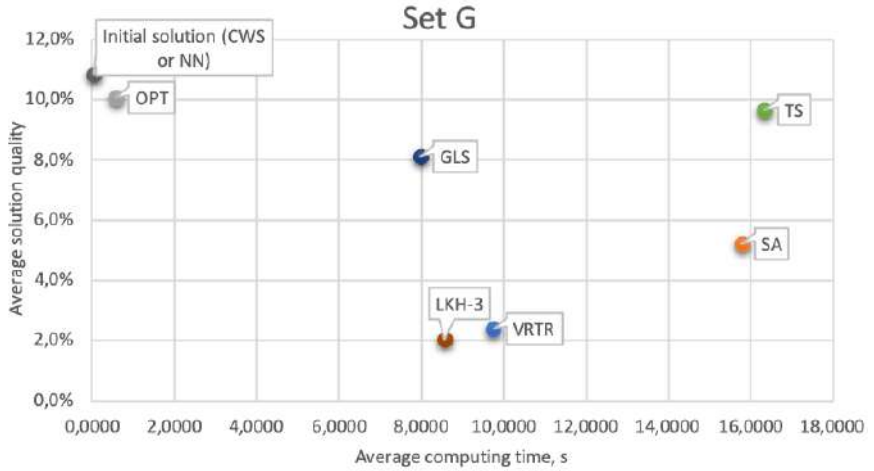


Fig. 11. Average solution quality vs. average running time, set G

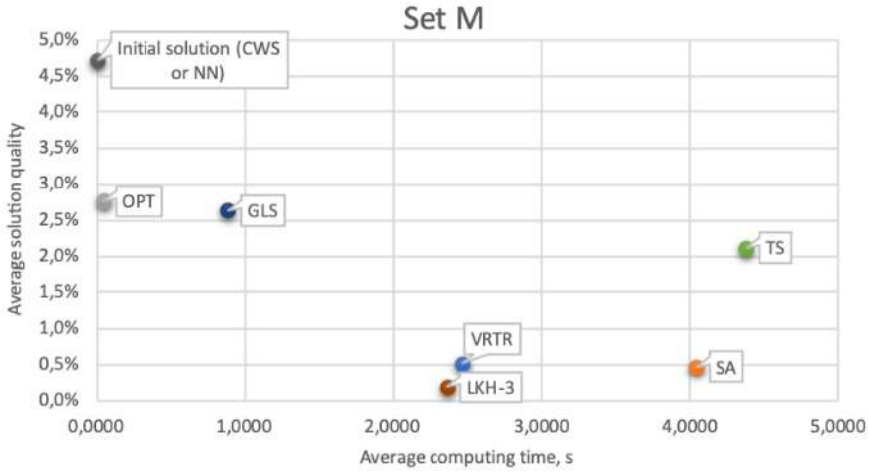


Fig. 12. Average solution quality vs. average running time, set M

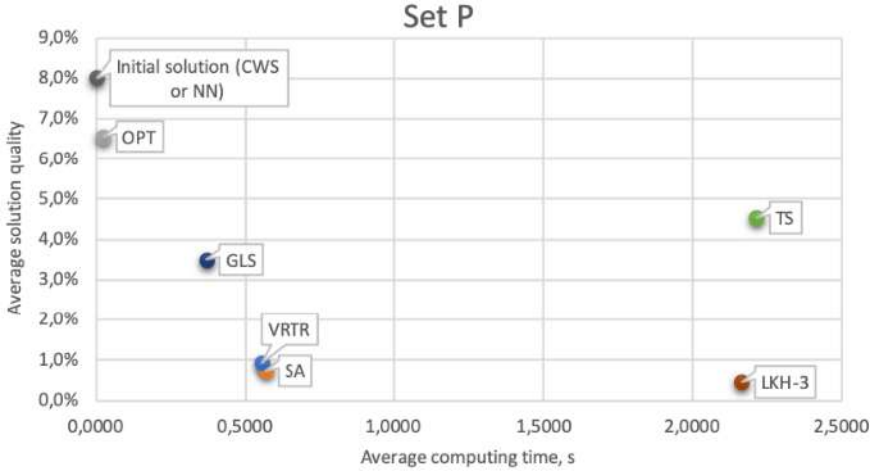


Fig. 13. Average solution quality vs. average running time, set P

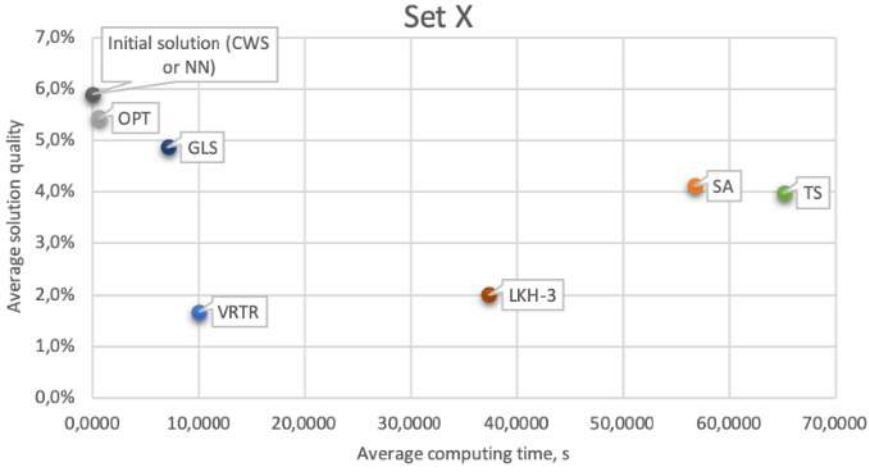


Fig. 14. Average solution quality vs. average running time, set X

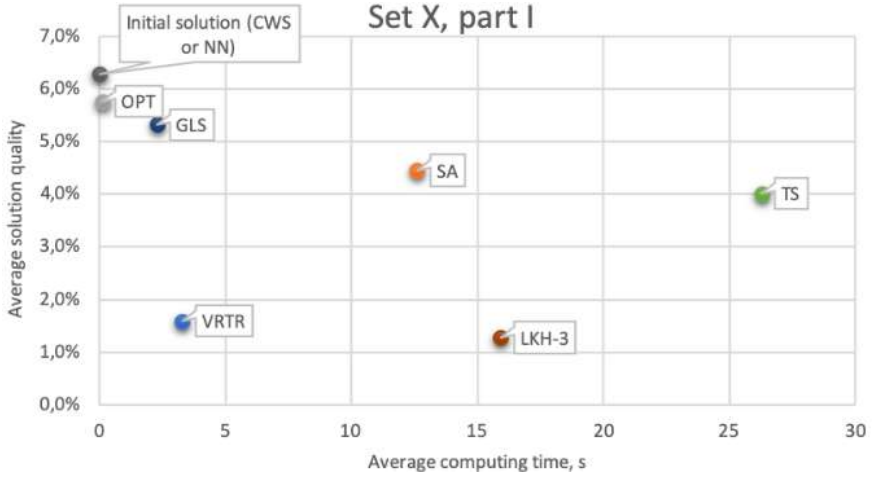


Fig. 15. Average solution quality vs. average running time, set X, part I

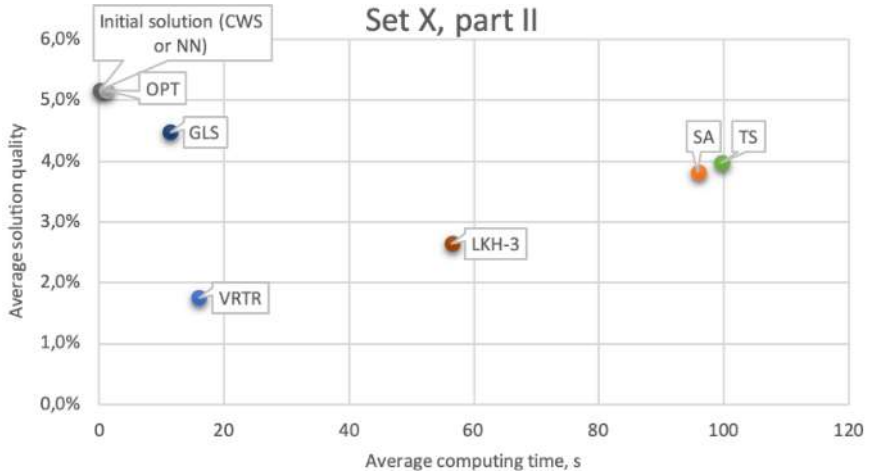


Fig. 16. Average solution quality vs. average running time, set X, part II

To sum up information presented in Figures from 7 to 16, Table 3 is formed.

Table 3. Involvement of algorithms in Pareto optimal groups for different datasets

Involvement of algorithms in Pareto optimal groups		Local search metaheuristics					
		OPT	GLS	TS	SA	LKH-3	VRTR
Set (its size)	A (26)	✓	✓		✓	✓	✓
	B (23)	✓	✓		✓	✓	✓
	E (11)	✓	✓		✓	✓	✓
	F (3)	✓			✓	✓	✓
	G (20)	✓	✓			✓	
	M (4)	✓	✓			✓	
	P (24)	✓	✓		✓	✓	✓
	X (100)	✓	✓				✓
	X, part I (47)	✓	✓			✓	✓
	X, part II (53)	✓	✓				✓

Table 3 shows involvement of local search metaheuristics in Pareto optimal groups for different datasets. Algorithms that belong to a group of Pareto optimal heuristics for some set are marked with a big tick. SA is marked by a small tick for set A as this metaheuristic can be in optimal by Pareto group as the difference between it and VRTR is too close, so it can be neglected. Also, it should be mentioned that two more rows are added – they are "set X, part I" which consists of instances with up to 322 clients and "set X, part II" which is vice versa has instances with 322 and more delivery points inside input files. This division is connected with the big size of set X and with the fact that was described in section V – VRTR works in a best way for instances with more than ≈320 clients in a delivery net for non-geometric distributions but LKH-3 produces good results for instances with no more than ≈100 clients regardless of type of distribution in the dataset.

The following conclusions are made on a base of results from Table 3.

1) Sets A, B, E and P can be aggregated together because a group of Pareto optimal algorithms is the same for all these sets. We will name this aggregation as *Group_1*. It represents two different types of inputs. The first one is with clients' coordinates and demands that are formed from a uniform distribution with some outlying cases. The second one is with nodes that are formed into clusters, and the number of clusters is equal or greater than number of available vehicles. Demands are also formed from a uniform distribution with some outlying cases. All input files from *Group_1* have 101 clients in a delivery net as maximum.
2) Set F forms a second group *Group_2* with only 3 instances obtained from real goods deliveries. Number of delivery points varies from 45 to 135.
3) *Group_3* is formed of sets G and M that also have the same Pareto optimal metaheuristics. Number of delivery points varies from 100 to 483. Set G has instances with locations in a form of concentric squares, pointed stars and rays, while set M consists of only 4 input files with locations that are grouped into clusters, and the number of clusters is equal or smaller by 1 than number of available vehicles.
4) *Group_4* is formed of the first part of set X. There are 47 instances in it, and the number of instances is up to 322 clients. *Group_4* is a mix of input data types: it has different combinations of demand distribution (unitary demands, small/large values, small/large variance), depot positioning (central, eccentric, random) and customer positioning (practical cases, uniform distribution, cluster-based).
5) *Group_5* is formed of the second part of set X. There are 53 instances in it, and the number of instances is from 322 to 1001 clients. Other characteristics of this group are the same as *Group_4* has.

Above-mentioned conclusions are outlined in Table 4 with information about involvement of algorithms in Pareto optimal groups depending on types of input data. All algorithms are listed in increasing order of average computing times (from best to worst) and in decreasing order of average solution qualities (from worst to best).

8. Conclusion

Overall, the next recommendation should be given to the problem which has described variant of mathematical model of CVRP. In general, for all types of clients' distribution the best algorithm to be applied is Clarke and Wright Savings, however, in case of having input data in form of concentric rays (like in Fig. 6) it is better to use Nearest Neighbor algorithm. Also, a few instances were solved best of all by Clarke and Wright Savings 2 algorithm, so it is important to have this algorithm in mind, however the difference between it and CWS is not very significant (no more than 1%).

One more conclusion is that it is unreasonable to use Sweep heuristic as it is not able to construct a set of routes without exceeding the number of vehicles for more than 50% of input files.

Finally, for our research it means that for all instances, except those 8 from set G, CWS heuristic will be used as initial algorithm for metaheuristic, otherwise – we will apply NN.

Table 4. Involvement of algorithms in Pareto optimal groups depending on types of input data

	Number of delivery points	**Distribution of delivery points**	**Distribution of demands**	**Pareto optimal algorithms**
Group_1	Up to 101	1. Uniform (with some outlying cases). 2. Cluster-based, the number of clusters is equal or greater than number of available vehicles.	Uniform (with some outlying cases)	OPT GLS SA VRTR LKH-3
Group_2	Up to 135	Real-world	Real-world	OPT SA VRTR LKH-3
Group_3	From 100 to 483	1. Geometric (concentric squares, pointed stars and rays). 2. Cluster-based, the number of clusters is equal or smaller by 1 than number of available vehicles.	Constant or uniform	OPT GLS LKH-3
Group_4	From 100 to 322	Mixed	Mixed	OPT GLS VRTR LKH-3
Group_5	From 322 to 1001	Mixed	Mixed	OPT GLS VRTR

As it was expected, unfortunately, there is no one universal metaheuristic that takes the first places by both criteria of solution quality and running time. Overall, the next recommendations should be given to people who are interested in metaheuristics solving CVRP.

1) For uniform (with some outlying cases) or cluster-based distribution of clients' locations, where the number of clusters is equal or greater than number of available vehicles it is better to apply Set of optimization operators, Guided local search, Simulated annealing, Variable record-to-record travel algorithm or Lin-Kernighan-Helsgaun heuristic for CVRP depending on desired solution quality and available time for calculations. Here and elsewhere the algorithms are listed in increasing order of average computing times (from best to worst) and in decreasing order of average solution qualities (from worst to best).
2) For real-world instances it is better to use following local search metaheuristics: Set of optimization operators, Simulated annealing, Variable record-to-record travel algorithm or Lin-Kernighan-Helsgaun heuristic for CVRP.
3) For geometric (with concentric squares, pointed stars and rays) or cluster-based distribution of clients' locations, where the number of clusters is equal or smaller by 1 than number of available vehicles, it is better to apply Set of optimization operators, Guided local search or Lin-Kernighan-Helsgaun heuristic for CVRP.
4) Finally, for mixed up combinations of demand distribution, depot positioning and customer positioning there are two recommendations. If there up to approximately 350 clients in a delivery net, it is better to use Set of optimization operators, Guided local search algorithm, Variable record-to-record travel algorithm or Lin-Kernighan-Helsgaun heuristic for CVRP. Otherwise, if there are nearly 320 delivery point and more then LKH-3 stops being so effective, so it is better to use following local search metaheuristics: Set of optimization operators, Guided local search algorithm or Variable record-to-record travelling algorithm.

Also experiments revealed absolute inefficiency of Tabu search as it is not in any of Pareto optimal groups.

In future we are planning to extend our study by conducting experiments using:

1) Population-based or nature-inspired metaheuristics as there are a lot of productive algorithms, too.

2) Other input data sets, including the most recent one with thousands of nodes in each instance. It is important to check local search metaheuristics on problems with extra-large dimensions to analyze their effectiveness.

References

[1]. E. Beresneva and S. Avdoshin. Analysis of Mathematical Formulations of Capacitated Vehicle Routing Problem and Methods for their Solution. Trudy ISP RAN/Proc. ISP RAS, vol. 30, no. 3, 2018, pp. 233-250. DOI: 10.15514/ISPRAS-2018-30(3)-17.
[2]. K. Braekers, K. Ramaekers, and I. Nieuwenhuyse. The vehicle routing problem: State of the art classification and review. Computers and Industrial Engineering, vol. 99, 2016, pp. 300-313.
[3]. B. Golden, S. Raghavan, and E. Wasil. The vehicle routing problem: Latest advances and new challenges. New York: Springer, 2008, 591 p.
[4]. P. Toth and D. Vigo. Vehicle Routing Problems, Methods, and Applications. Philadelphia: SIAM, 2014, 481 p.
[5]. F. Arnold, M. Gendreau, and K. Sorensen. Efficiently solving very large-scale routing problems. Computers and Operations Research, vol. 107, 2019, pp. 32-42.
[6]. K. Helsgaun. An effective implementation of the Lin–Kernighan traveling salesman heuristic. European Journal of Operational Research, vol. 12, issue 1, 2000, pp. 106-130.
[7]. E. Zachariadis and C. Kiranoudis. An open vehicle routing problem metaheuristic for examining wide solution neighborhoods. Computers and Operations Research, vol. 37, no. 4, 2010, pp. 712-723.
[8]. E. Taillard, G. Laporte and M. Gendreau. Vehicle routeing with multiple use of vehicles. Journal of the Operational Research Society, vol. 47, no. 8, 1996, pp. 1065-1070.
[9]. S. Ropke and D. Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. Transportation Science, vol. 40, no. 4, 2006, p. 455–472.
[10]. P. Toth and D. Vigo. An overview of vehicle routing problems. In The Vehicle Routing Problem, SIAM, 2002, pp. 1-26.
[11]. K. Helsgaun. An Extension of the Lin-Kernighan-Helsgaun TSP Solver for Constrained Traveling Salesman and Vehicle Routing Problems. Technical Report, Roskilde University, 2017, 60 p.
[12]. P. Schittekat and K. Sorensen. Deconstructing record-to-record travel for the capacitated vehicle routing problem. Operational Research and Management Science Letters, vol. 1, no. 1, 2018, pp. 17-27.
[13]. F. Li, B. Golden, and E. Wasil. Very large-scale vehicle routing: new test problems, algorithms, and results. Computers and Operations Research, vol. 32, issue 5, 2005, p. 1165–1179.
[14]. G. Laporte and F. Demet. Classical Heuristics for the Capacitated VRP. In The Vehicle Routing Problem, SIAM, 2002, pp. 109-128.
[15]. C. Voudouris, E. Tsang, and A. Alsheddy. Guided local search. In Handbook of metaheuristics, Springer, 2010, pp. 321-361.
[16]. D. Mester and O. Braysy. Active-guided evolution strategies for large-scale capacitated vehicle routing problems. Computers and Operations Research, vol. 34, no. 10, 2007, pp. 2964–2975.
[17]. F. Arnold and K. Sorensen. Knowledge-guided local search for the Vehicle Routing Problem. Computers and Operations Research, vol. 105, 2019, pp. 32-46.
[18]. F. T. E. Glover. A User's Guide to Tabu Search. Operations Research, vol. 41, no. 1, 1993, pp. 1-28.
[19]. E. Zachariadis and C. Kiranoudis. A strategy for Reducing the Computational Complexity of Local Search-Based Methods for the Vehicle Routing Problem. Computers and Operations Research, vol. 37, 2010, pp. 2089–2105.
[20]. S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing. Science, vol. 220, no. 4598, 1983, pp. 671–680.
[21]. NEO. Simulated Annealing. [Online]. Available: http://neo.lcc.uma.es/vrp/solution-methods/metaheuristics/simulated-annealing/. Accessed 27.02.2019.
[22]. I. H. Osman. Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem. Annals of Operations Research, vol. 41, 1993, pp. 421-451.
[23]. S. Avdoshin and E. Beresneva. The Metric Travelling Salesman Problem: The Experiment on Pareto-optimal Algorithms. Trudy ISP RAN/Proc. ISP RAS, vol. 29, no. 4, pp. 123-138, 2017. DOI: 10.15514/ISPRAS-2017-29(4)-8.
[24]. K. Helsgaun. LKH-3. [Online]. Available: http://akira.ruc.dk/~keld/research/LKH-3/. Accessed 01.2019.
[25]. I. Xavier. CVRPLIB. [Online]. Available: http://vrp.atd-lab.inf.puc-rio.br/index.php/en/. Accessed 09 05 2018.

[26]. Heidelberg University. TSPLIB. [Online]. Available: https://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/. Accessed 09 05 2018.

Информация об авторах / Information about authors

Екатерина Николаевна БЕРЕСНЕВА – с 2017 года преподаватель департамента программной инженерии НИУ ВШЭ, с 2019 года – аспирант НИУ ВШЭ. Профессиональные интересы – дискретная математика, задача маршрутизации транспорта, задача коммивояжера.

Ekaterina Nikolaevna BERESNEVA – lecturer at the School of Software Engineering, Faculty of Computer Science, National Research University Higher School of Economics since 2017. Her research interests include discrete mathematics, the vehicle routing problem and the travelling salesman problem.

Сергей Михайлович АВДОШИН – профессор, руководитель департамента программной инженерии факультета компьютерных наук НИУ ВШЭ с 2005 года. Сфера научных интересов: разработка и анализ компьютерных алгоритмов, имитация и моделирование, параллельные и распределенные процессы, теневой интернет, технология блокчейн.

Sergey Mikchailovitch AVDOSHIN – Professor, Head of the School of Software Engineering at National Research University Higher School of Economics since 2005. Research interests are design and analysis of computer algorithms, simulation and modeling, parallel and distributed processing, deep Web, blockchain technology.

DOI: 10.15514/ISPRAS-2019-31(4)-9

Method for Building UML Activity Diagrams from Event Logs

N.S. Zubkova, ORCID: 0000-0002-0123-7689 <nszubkova@edu.hse.ru>
S.A. Shershakov, ORCID: 0000-0001-8173-5970 <sshershakov@hse.ru>
Laboratory of Process-Aware Information Systems (PAIS Lab),
National Research University Higher School of Economics,
20, Myasnitskaya st., Moscow 101000, Russia

Abstract. UML Activity Diagrams are widely used models for representing software processes. Models built from event logs, recorded by information systems, can provide valuable insights into real flows in processes and suggest ways of improving those systems. This paper proposes a novel method for mining UML Activity Diagrams from event logs. The method is based on a framework that consists of three nested stages involving a set of model transformations. The initial model is inferred from an event log using one of the existing mining algorithms. Then the model, if necessary, is transformed into an intermediate form and, finally, converted into the target UML Activity Diagram by the newly proposed algorithm. The transforming algorithms, except one used at the last stage, are parameters of the framework and can be adjusted based on needed or available models. The paper provides examples of the approach application on real life event logs.

Keywords: process mining; Petri nets; UML activity diagrams; process discovery

For citation: Zubkova N.S., Shershakov S.A. Method for Building UML Activity Diagrams from Event Logs. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 4, 2019. pp. 139-150. DOI: 10.15514/ISPRAS-2019-31(4)-9

Acknowledgements. This work is supported by the Basic Research Program of the National Research University Higher School of Economics.

Метод построения UML диаграмм деятельности по журналам событий

Н.С. Зубкова, ORCID: 0000-0002-0123-7689 <nszubkova@edu.hse.ru>
С.А. Шершаков, ORCID: 0000-0001-8173-5970 <sshershakov@hse.ru>
Лаборатория процессно-ориентированных информационных систем (ПОИС),
Национальный исследовательский университет «Высшая школа экономики»,
101000, Россия, Москва, ул. Мясницкая, д.20

Аннотация. UML диаграммы деятельности широко используются для представления процессов в программной инженерии. Модели, построенные по журналам событий, могут предоставить ценную информацию о реальные процессах в информационных системах, на основании которой эти процессы можно улучшить. Данная статья представляет новый метод майнинга UML диаграмм деятельности по журналам событий. Метод основам на параметрической схеме, которая состоит из трех вложенных ступеней, включающих в себя набор преобразований над моделями. Начальная модель извлекается из журнала событий один из существующих алгоритмов синтеза. Эта модель, если требуется, преобразуется в промежуточную форму, и затем конвертируется в целевую диаграмму деятельности с помощью предлагаемого алгоритма. Алгоритмы синтеза, помимо используемого на последней стадии, являются параметрами схемы и могут быть изменены, исходя из имеющихся или требуемых моделей. В статье представлены примеры применения подхода к журналам событий из реальной жизни.

Ключевые слова: анализ процессов; сети Петри; UML диаграммы деятельности; process discovery

Для цитирования: Зубкова Н.С., Шершаков С.А. Метод построения UML диаграмм деятельности по журналам событий. Труды ИСП РАН, том 31, вып. 4, 2019 г., стр. 139-150 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(4)-9

Благодарности. Работа выполнена при поддержке Программы фундаментальных исследований Национального исследовательского университета «Высшая школа экономики».

1. Introduction

Process mining techniques [1] aim at analyzing and improving real-life processes by taking information from event logs. Event logs are generally produced by process-aware information systems (PAIS) that support these processes. One particular problem of process mining is *process discovery*; its goal is to build a model of a process, based on the data in an event log. Such models can be expressed in different notations. For example, transition systems (TS) naturally represent sequences of events (*traces*) as they are recorded in event logs. However, if a process contains concurrent behavior, transition systems tend to be very complex and as large as the event log itself. This occurs due to the fact that similar patterns are not joined together and concurrency is expressed in the form of interleaving.

There are other types of models that allow to represent concurrency patterns, namely choice and parallelism, and that are widely used in the field of process mining. *Petri nets (PN), BPMN*, *Fuzzy maps* and *UML Activity Diagrams (AD)* are examples of such models. Unified Modeling Language (UML) [18] is a standard for defining, documenting and visualizing artifacts, especially in the software engineering domain. Particularly, UML Activity Diagrams are used, among other, to represent and analyze actual or expected behavior of software systems. AD is not the only UML class that allows to represent concurrency [9]. For instance, *UML State Machine Diagrams* have their own semantics to illustrate concurrency. However, they reflect different *states* of a system that are not explicitly represented in event logs. These states, therefore, have to be mined using different techniques, i.e. encoding states with trace prefixes. Given that event logs contain information representing *activities* performed by process participants and supporting systems, we regard Activity Diagrams as the de- sired class of target models in this paper.

In our work, we propose a framework for building UML Activity Diagrams from event logs, consisting of a number of steps. The frameworks *essential* part is the algorithm for converting Petri nets into UML ADs. Other intermediate models (namely, TS and PN) can be synthesized using different algorithms which are parameters of the framework. Here we consider the algorithm of regions [8] as a means to generate Petri nets which are consequently converted into target UML ADs. ADs are usually more compact than Petri nets and are more easily interpretable. Moreover, generated diagrams can be imported and used in different visual modeling and design tools used in the software engineering domain, i.e. Sparx Enterprise Architect, and be later included as part of bigger software models.

The contributions of this paper are as follows: (1) a framework for generating UML AD from event logs, (2) a novel method for UML AD synthesis from a Petri nets as intermediate models, (3) implementation of the proposed framework specified by a particular set of synthesis algorithms. The rest of the paper is organized as follows. Section II gives a brief overview of related work. Section III defines necessary concepts needed for the explanation of the proposed approach. The framework is described in Section IV and the PN-to-UML AD conversion algorithm is presented in Section V. Section VI contains models derived from real-life event logs. Finally, Section VII concludes the paper and outlines possible directions for future work.

2. Related work

There exist many approaches to construct Petri nets from event logs [1], [2], [19]. The algorithm of regions and its extensions are described, particularly, in [6], [8], [10]. The algorithm produces a Petri

net from a given TS that serves as an input of the algorithm. The behavior of the derived PN is guaranteed to be equivalent to the TS. Previously, Petri nets have also been used as intermediate models for constructing other types of target models, such as BPMN in [14].

The similarity between UML Activity Diagrams and Petri nets are studied in numerous works. Arlow et al. present UML specification in application to Unified Process including UML AD structural elements, and also mention that UML AD are based on the Petri Net techniques [5]. In [12] authors formalize AD semantics and compare them to semantics of Petri Nets. There are many works dedicated to the transformation of UML Activity Diagrams into Petri nets; the reverse transformation is studied scantily. In [13] the author describes an approach to translate UML AD into Petri nets. Agarwal [4] developed a method for transforming AD into Petri nets for verification purposes. The author considers a set of UML patterns and indicates corresponding Petri net instances.

3. Preliminaries

$B(X)$ is the set of all multisets over some set X. For a given set X, X^+ is the set of all non-empty finite sequences over X.

3.1 Trace, Event Log

Let Z be a set of activities. A *trace* is a finite sequence $\sigma = (a_1, a_2, \dots, a_i, \dots, a_n) \in Z^+$. By $\sigma(i) = a_i$ we denote i-th element. $L \in B(Z^+)$, such that $|L| > 0$, is an event log. Here, $|L|$ is the number of all traces.

3.2 Labeled Petri Net, Well-structured Labeled Petri Net

A *labeled Petri net* (*PN*) is a tuple $= \{P, T, F, l\}$, where P is a set of places, T is a set of transitions, $P \cap T = \emptyset$, $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation, l is the labeling function $l : T \to \Lambda$, and Λ is a set of labels. In process mining, labels of transition represent events.

Given $p \in P$, the set $p^\bullet = \{y | (p, y) \in F\}$ is the postset of p.

In this paper, we denote by Petri net a well-structured Petri net, i.e., a hierarchical Petri net that can be recursively divided into parts having single entry and exit points [15].

3.2 UML Activity Diagram

A *UML Activity Diagram* is a tuple $AD = \{N, E, NT\}$, where

- $\boldsymbol{NT}$ is a set of node types, $\boldsymbol{NT} = \{\boldsymbol{control}, \boldsymbol{object}, \boldsymbol{executable}\}$;
- $\boldsymbol{N}$ is a set of nodes. $\boldsymbol{n} \in \boldsymbol{N}: \boldsymbol{n} = (\boldsymbol{\lambda}, \boldsymbol{type}), \boldsymbol{\lambda} \in \boldsymbol{\Lambda}, \boldsymbol{type} \in \boldsymbol{NT}$;
- $\boldsymbol{E}$ is a set of edges. $\boldsymbol{e} \in \boldsymbol{E}: \boldsymbol{e} = (\boldsymbol{n1}, \boldsymbol{n2}), \boldsymbol{n1}, \boldsymbol{n2} \in \boldsymbol{N}$.

Similar definition was used in [11]. In this paper we mainly focus on the following elements of the UML AD (see fig. 1):

1) A is a set of activity nodes, $a \in A: a = (\lambda, executable)$;
2) F is a set of parallel nodes, $f \in F: f = (control)$;
3) D is a set of decision nodes, $d \in D: d = (control)$;
4) initial and final are initial and final nodes, both of type control.

UML decision nodes should be equipped with guards that indicate the conditions under which the decision is made. In this paper, we regard non-deterministic Petri nets as intermediate models[1]. The

[1] There exists an extension to Petri nets that adds guards to its semantics. However, most of the process mining algorithms consider Petri nets without guards. Here we follow the same approach.

proposed conversion algorithm does not assume the presence of guard information in the event log and uses only the input Petri net. Thus, the produced Activity Diagram is non-deterministic as well.

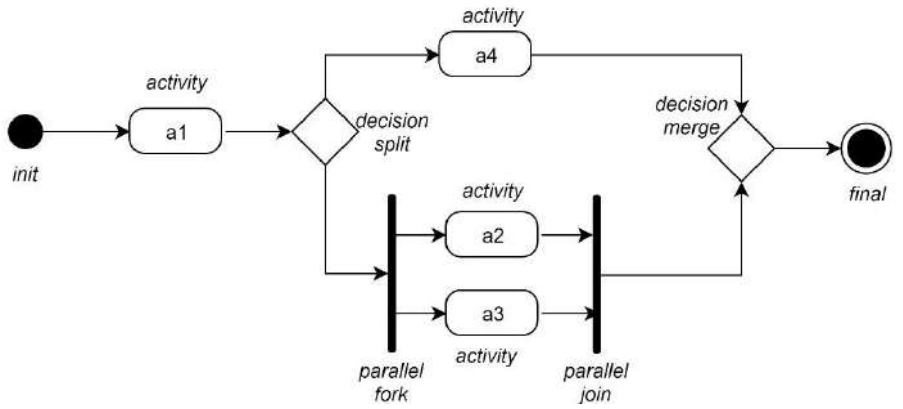


Fig. 1. An Activity Diagram example

4. Framework

The proposed framework is illustrated in fig. 2. The framework consists of a number of nested stages related to individual steps of the proposed method. At every step a transformation from one entity, event log or process model, into another is made. There exist numerous approaches to construct both Petri nets and transition systems. Models obtained from the same event log, but using different algorithms, represent the same process. However, they vary in details that are usually represented by quality metrics [7]. Depending on the task, specific combinations of quality metrics can be considered.

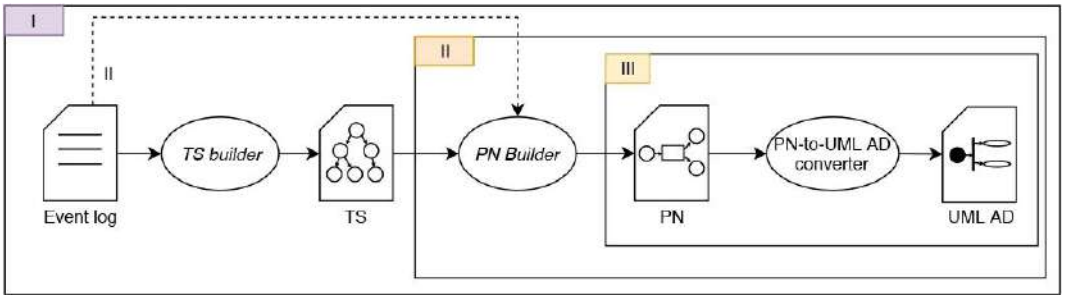


Fig. 2. Proposed framework: I = II + TS construction; II = III + PN synthesis; III = PN-to-UML AD conversion

The long path of the framework (I) includes first building a TS needed for the algorithm of regions. Here, various techniques for TS construction can be used, for instance, prefix tree synthesis [3], frequency based reduction [16], neural approach [17] etc. However, the TS synthesis can be bypassed and a Petri net can be generated directly from the event log (II). There are many algorithms for that, i.e., Inductive miner [15], α-algorithm [2], ILP-miner [20] and other. Finally, in III the generated Petri net is converted into a UML Activity Diagram.

4.1 Proposed implementation

In this paper we consider the full version of the framework with the following parameters.

1) Prefix tree builder, unlimited window, for TS construction.
2) The algorithm of regions for converting the TS into a PN.

3) The PN-to-UML AD converter described in the following section.

The following paragraph gives a brief overview of the algorithms used in steps 1 and 2.

Prefix tree builder [3] is an algorithm for TS synthesis. Event logs usually do not explicitly contain states that are needed for the construction of a transition system. A *state function* is introduced in order to infer such states. This function maps events in an event log onto states of a TS. Let E be a set of events in an event log, and S be a set of states in a transition system. For each event $e \in E$ the state function produces a state $s \in S$ regarding either pre- or posthistory of the state s. A prefix tree is a special type of transition systems, for which the state function consideres prehistory (prefix) of the states. Informally, a transition (s, e, s') appears if $prehistory_{s'} = prehistory_s + e$. If the prefix size is unlimited, the size of the generated TS can be equivalent to the size of the event log.

The algorithm of regions [8] used is based on finding equivalent behaviors in a given transition system. These behavioral fragments are grouped into so-called *regions*. Intuitively a region is equivalent to a place in a Petri net. Placing a token in such a place means allowing such a behavior to appear – via activating a post-transition. In UML Activity Diagrams transitions are translated into *activities*. Thus, considering a transitive dependence between an initial TS and an AD, one can ascertain a link between equivalent behavioral fragments in TS (regions) and corresponding nodes in AD.

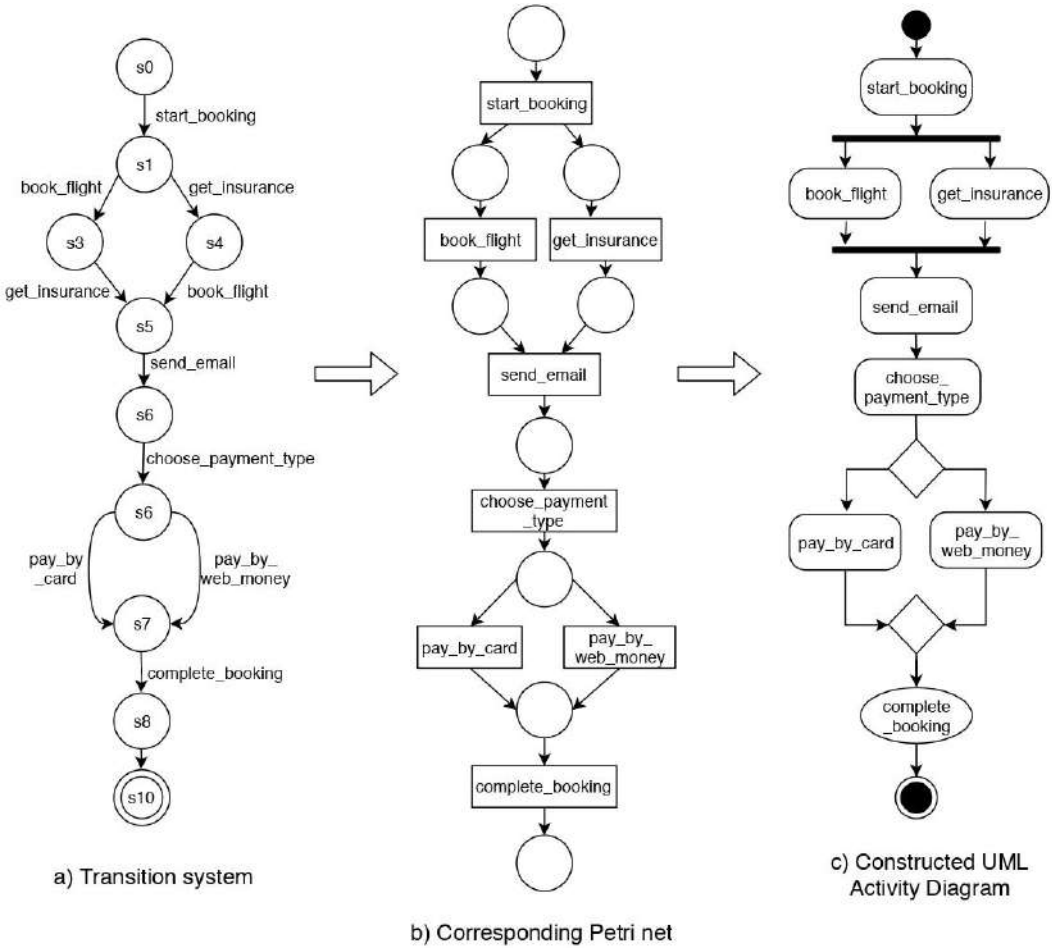


Fig.3. Example models

3. Petri Net to UML Activity Diagram conversion algorithm

The PN-to-UML AD conversion algorithm is based on the idea of converting places and transitions of a given Petri net into corresponding elements of the target UML Activity Diagram. UML AD specification notes that an activity diagram can only have a single entry point, whereas the inception of a process modeled by a Petri net can be determined by placing tokens in multiple places (an initial marking). Here, we consider all places without incoming edges as a potential starting place. Then a single starting point (initial node) in an Activity Diagram is constructed and connected to the following activities. Final places are also not explicitly indicated in Petri nets, however it is sensible to regard those without outgoing edges as such, corresponding final nodes are inserted in the AD.

While translating a Petri net into a UML activity diagram the algorithm considers special patterns, namely parallelisms and decisions. Such patterns can be translated into equivalent patterns in an Activity Diagram. A similar approach was used in [4], [13] for the reverse transformation.

In order to describe the proposed transformation we illustrate it on a running example (fig. 3).

We consider different types of AD nodes and describe the according transformations as follows.

5.1 Transformation functions

Let $\alpha: (T, l) \rightarrow (A, l)$ be a function transforming transitions of the Petri net into *activities* of the constructed UML AD, tagged by the same labels;

Let $\varphi: P \rightarrow D$ be a function transforming appropriate positions of the PN into decision nodes of the UML AD;

Let $\xi: T \rightarrow F$ and $\psi: P \rightarrow F$ be functions transforming PN transitions and sets of PN places into UML parallel nodes accordingly.

5.2 Building a UML Activity Diagram

UML Activity Diagram construction includes the following procedures.

5.2.1 Constructing activity nodes

The semantics of Petri nets suggests that transitions, which model events in Petri nets, correspond to activities in Activity Diagrams. So the first transformation step of the algorithm is turning transitions of a given Petri net into UML AD activities, i.e. for each transition $t \in T$ we create an activity $a = \alpha(t)$ in the AD.

5.2.2 Detecting parallel forks

We now need to connect nodes and identify more complex behaviors. In a Petri net a concurrent pattern occurs if a transition has multiple outgoing edges, allowing tokens to appear in *all* of the following places when the transition is fired (see Fig. 4). Considering a transition $t \in T$ of a Petri net, let T^* be a set of transitions reachable from t in one step. For each transition $t \in T$, if t has:

a) 0 outgoing edges, then activity $\boldsymbol{\alpha(t)}$ is connected to a final node;

b) 1 outgoing edge, then activity $\boldsymbol{\alpha(t)}$ is connected to $\boldsymbol{\alpha(t^*)}$, for each $\boldsymbol{t^* \in T^*}$;

c) > 1 outgoing edge, activity $\boldsymbol{\alpha(t)}$ is connected to a fork node $\boldsymbol{\xi(t)}$, and $\boldsymbol{\xi(t)}$ is then connected to $\boldsymbol{\alpha(t^*)}$, for each $\boldsymbol{t^* \in T^*}$.

5.2.3 Detecting parallel join

In order for the model to be more interpretable, for each parallel fork there should be a reciprocal parallel join. So for each fork, described in 2) we need to find the corresponding join. This is done according to the following steps.

a) For each maximum set of places $\boldsymbol{S} = \{\boldsymbol{p_1}, \ldots, \boldsymbol{p_n}\} \subseteq \boldsymbol{P}$ that have coinciding postsets ($\boldsymbol{p_1^\bullet} = \ldots = \boldsymbol{p_n^\bullet}$) and $\boldsymbol{n > 1}$, a $\boldsymbol{\psi(P)}$ join node is inserted in the $\boldsymbol{AD}$.

b) For each transition $\boldsymbol{t}$ immediately preceding each place from $\boldsymbol{S}$, the activity $\boldsymbol{\alpha(t)}$ is connected to $\boldsymbol{\psi(P)}$.

c) Join node $\boldsymbol{\psi(P)}$ is then connected to $\boldsymbol{\alpha(t')}$, for all $\boldsymbol{t'} \in \boldsymbol{T'}$, where $\boldsymbol{T'}$ is a set of transitions immediately following places $\{\boldsymbol{p_1}, \ldots, \boldsymbol{p_n}\}$.

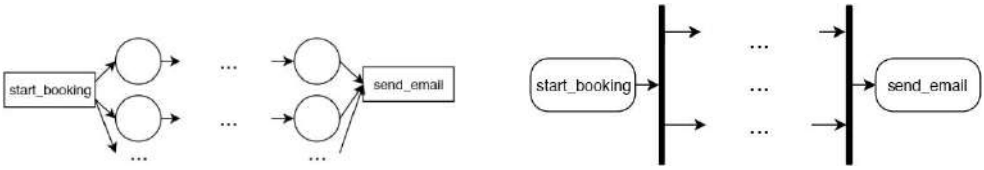


Fig. 4. Concurrency pattern in Petri Net and UML AD

5.2.4 Detecting decision splits and merges

A decision pattern in a Petri net occurs if a place has multiple outgoing edges allowing *only one* consecutive transition to fire (see fig. 5). So for each place $\in P$, that has more than one outgoing edge a decision node $\varphi(p)$ is inserted into the AD and is connected to $\alpha(\tilde{t})$, for all $\tilde{t} \in \tilde{T}$, $\tilde{T}$ are PN transitions connected to p (both before and after). Likewise, if the place p has multiple incoming edges, a reciprocal *merge* node $\varphi(p)$ is inserted into the AD.

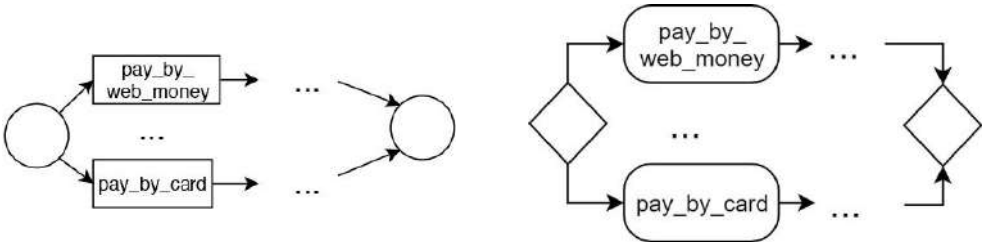


Fig. 5. Choice pattern in Petri Net and UML AD

Applying the steps 5.2.1 – 5.2.4 to an input Petri net, the target UML Activity Diagram is constructed.

6. Application

In this section, we provide examples of models obtained from real logs. Log1 and Log2 consist of 243 and 1132 traces respectively. For observability purposes, intermediate transition systems were reduced using a frequency reduction algorithm described in [16].

In fig. 6 models were generated with window size 1 and frequency reduction parameter 0.04. Log1 contains information about bank operations.

In fig. 7 models were generated on a log containing information about building permit applications from five Dutch municipalities. Transition system was built with unlimited window parameter and reduced with frequency reduction parameter of 0.15.

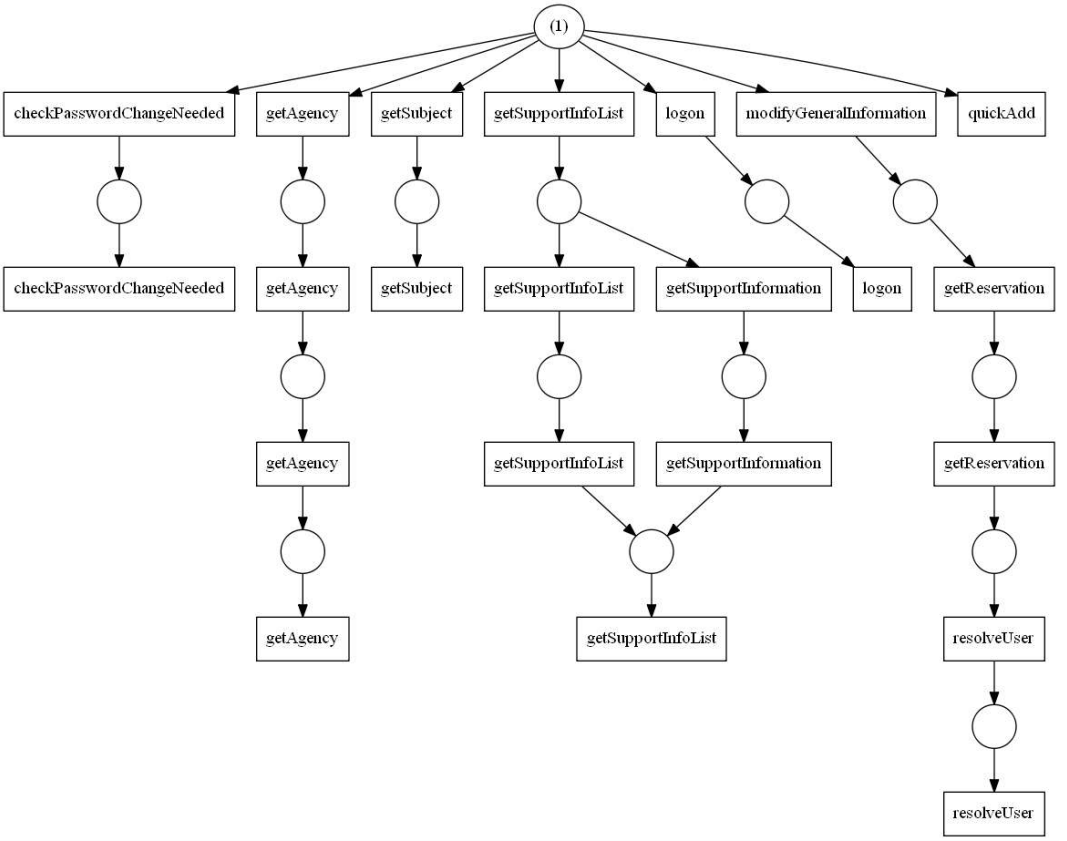


Fig. 6a. Log1: Petri net

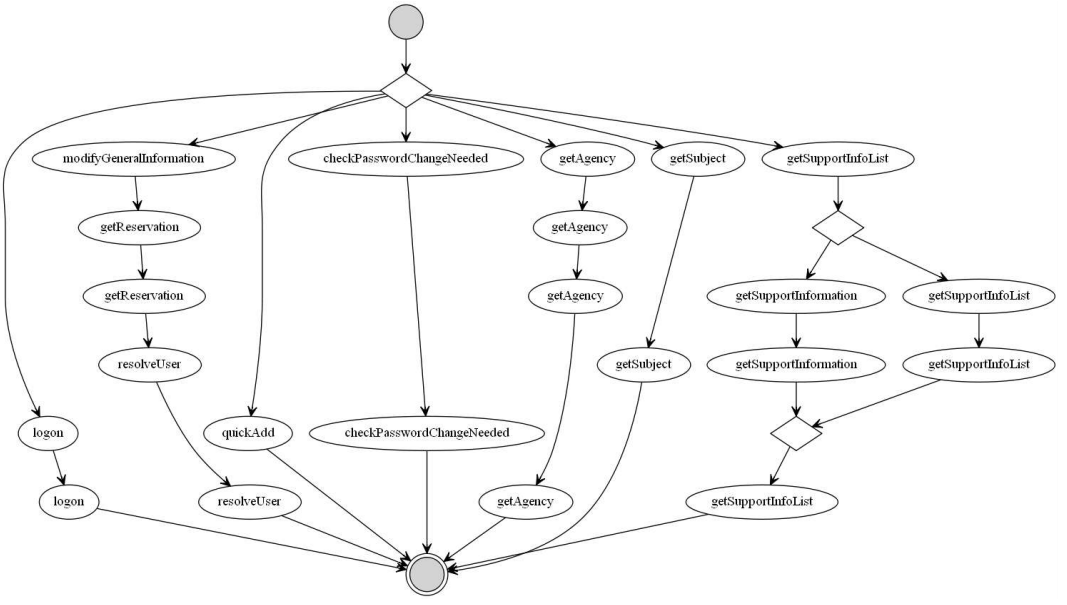


Fig. 6b. Log1: UML Activity Diagram

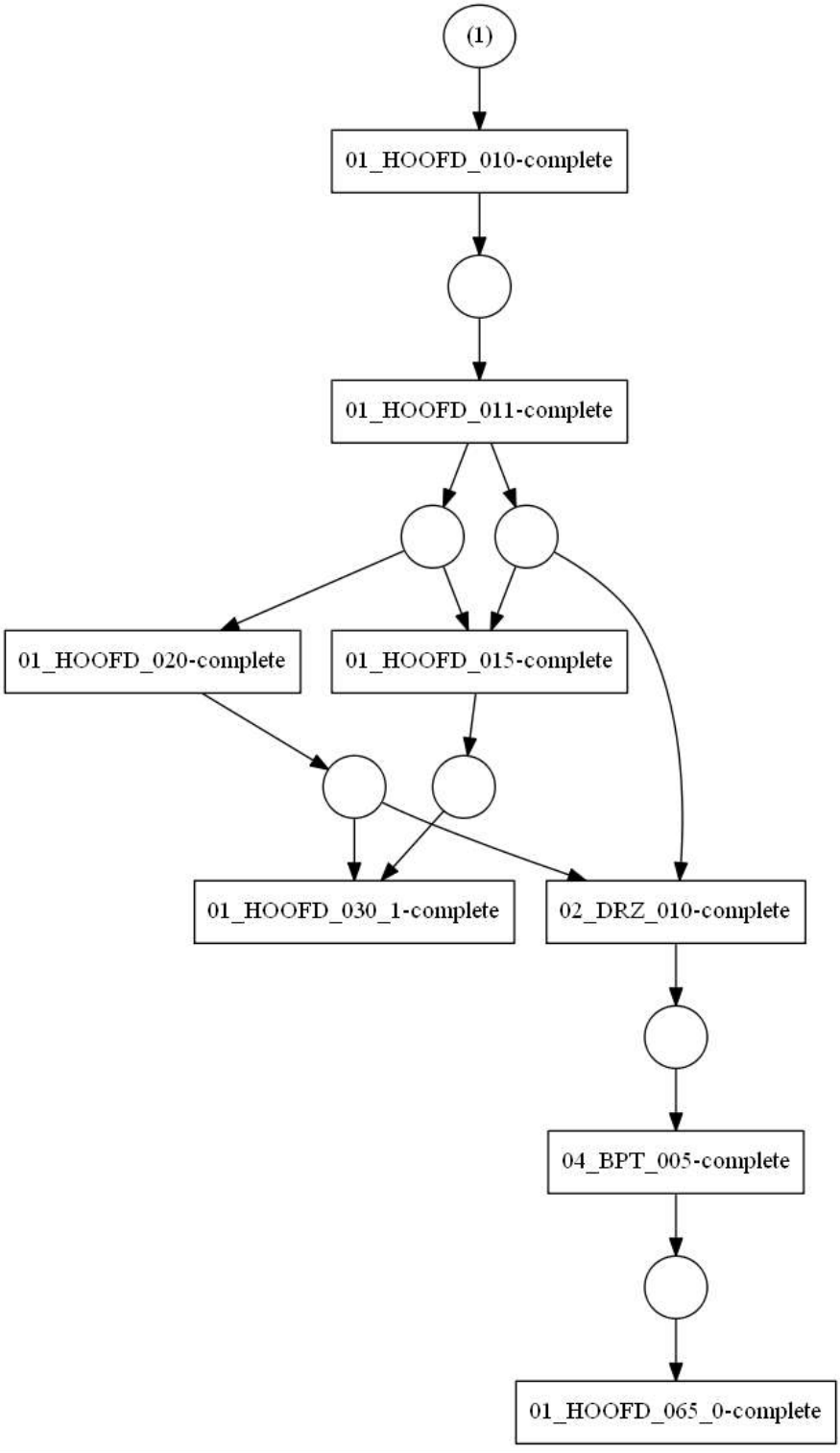


Fig. 7a. Log2: Petri net

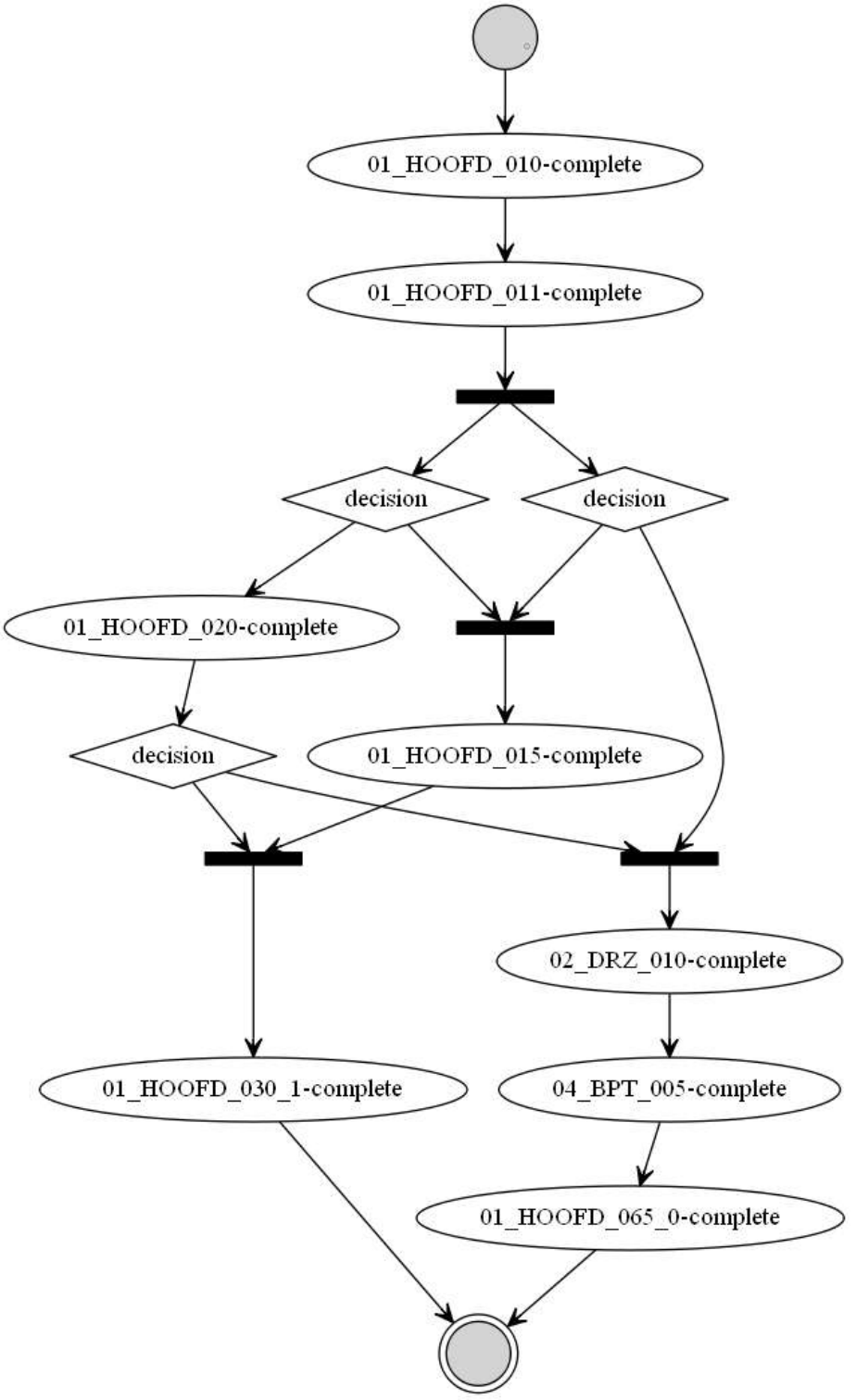


Fig. 7b. Log2: UML Activity Diagram

7. Conclusion

In this paper, we proposed a method based on a framework to build UML Activity Diagrams from event logs and introduced a novel algorithm for converting a well-structured Petri net into a UML Activity Diagram. The method is implemented as a part of the LDOPA[2] library. Future work includes studying the execution semantics of Petri nets with guards, mining dependencies and adding guards to Activity Diagrams. Moreover, the framework can be further investigated by implementing different TS and PN synthesis algorithms.

References

[1]. Van der Aalst W. Data science in action. In Process Mining, Springer, 2016, pp. 3-23.
[2]. Van der Aalst W.M.P., Van Dongen B.F. Discovering petri nets from event logs. Lecture Notes in Computer Science, vol. 7480, 2013, pp. 372-422.
[3]. Van der Aalst W., Rubin V., Verbeek H., van Dongen B., Kindler E., Günther C. Process mining: a two-step approach to balance between underfitting and overfitting. Software and Systems Modeling, vol. 9, no. 1, 2010, pp. 87-111.
[4]. Agarwal B. Transformation of UML activity diagrams into Petri nets for verification purposes. International Journal of Engineering and Computer Science, vol. 2, no. 3, 2013, pp. 798-805.
[5]. Arlow J., Neustadt I. UML 2 and the unified process: practical object-oriented analysis and design. Pearson Education, 2005, 624 p.
[6]. Badouel E., Bernardinello L., Darondeau P. Polynomial algorithms for the synthesis of bounded nets. Lecture Notes in Computer Science, vol. 915, 1995, pp. 364-378.
[7]. Buijs J.C.A.M., van Dongen B.F., van der Aalst W. M. P. On the role of fitness, precision, generalization and simplicity in process discovery. Lecture Notes in Computer Science, vol. 7565, 2012, pp. 305-322.
[8]. Carmona J., Cortadella J., Kishinevsky M. A region-based algorithm for discovering Petri nets from event logs. Lecture Notes in Computer Science, vol. 5240, 2008, pp. 358-373.
[9]. Concurrency in UML. Available at: https://www.omg.org/ocup-2/documents/concurrency_in_uml_version_2.6.pdf. Accessed: 2019-03-05.
[10]. Cortadella J. et al. Deriving Petri nets from finite transition systems. IEEE Transactions on Computers, vol. 47, no. 8, 1998, pp. 859-882.
[11]. Davydova K.V., Shershakov S.A. Mining hybrid UML models from event logs of SOA systems. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 4, 2017, pp. 155-174. DOI: 10.15514/ISPRAS-2017-29(4)-10.
[12]. Eshuis R., Wieringa R. A comparison of Petri net and activity diagram variants. In Proc. of the 2nd Int. Coll. on Petri Net Technologies for Modelling Communication Based Systems, 2001, pp. 93-104.
[13]. Fahland D. Translating uml2 activity diagrams to petri nets. Informatik-Berichte 226, Humboldt-Universitat zu Berlin, 2008.
[14]. Kalenkova A., van der Aalst W., Lomazova I., Rubin V. Process mining using BPMN: relating event logs and process models, Software and Systems Modeling, 2017, vol. 16, no. 4, pp. 1019-1048.
[15]. Leemans S.J.J., Fahland D., van der Aalst W.M.P. Discovering block-structured process models from event logs-a constructive approach. Lecture Notes in Computer Science, vol. 2472, 2013, pp. 311-329.
[16]. Shershakov S.A., Kalenkova A.A., Lomazova I.A. Transition systems reduction: balancing between precision and simplicity. Lecture Notes in Computer Science, vol. 10470, 2017, pp. 119-139.
[17]. Shunin T., Zubkova N., Shershakov S. Neural Approach to the Discovery Problem in Process Mining. Lecture Notes in Computer Science, vol. 11179, 2018, pp. 261-273.
[18]. UML specification. Available at: https://www.omg.org/spec/UML/About-UML/. Accessed: 2019-03-01.
[19]. Weijters A., van Der Aalst W., De Medeiros A.K.A. Process mining with the heuristics miner-algorithm. Technische Universiteit Eindhoven, Tech. Rep. WP, 2006, 34 p.
[20]. Van der Werf J. M. E. M., van Dongen B. F., Hurkens C. A., Serebrenik A. Process Discovery Using Integer Linear Programming. Lecture Notes in Computer Science, vol. 5062, 2008, pp. 368-387.

Информация об авторах / Information about the authors

Наталья Сергеевна ЗУБКОВА в настоящее время является студенткой бакалаврской программы «Программная инженерия» факультета компьютерных наук. Область ее научных

[2] Available at https://prj.xiart.ru/projects/ldopa

интересов включает анализ и моделирование процессов, интеллектуальный анализ данных и машинное обучение.

Natalia Sergeyevna ZUBKOVA is currently a student enrolled in the «Software Engineering» bachelor's program, faculty of Computer Science. Her research interests include process modelling and analysis, data mining and machine learning.

Сергей Андреевич ШЕРШАКОВ получил степень магистра в области программной инженерии в НИУ ВШЭ (Москва) в 2012 году. В настоящее время он является научным сотрудником научно-учебной лаборатории процессно-ориентированных информационных систем факультета компьютерных наук. В число научных интересов входят извлечение и анализ процессов (process mining), верификация программного обеспечения, архитектуры информационных систем и преподавание программной инженерии.

Sergey Andreevitch SHERSHAKOV received the MS degree in software engineering from HSE (Moscow, Russia) in 2012. He is currently a research fellow at PAIS Lab of the Faculty of Computer Science. His research interests include process mining, software verification, information systems architectures and teaching software engineering.

DOI: 10.15514/ISPRAS-2019-31(4)-10

Simulating Petri Nets with Inhibitor and Reset Arcs

P.A. Pertsukhov, ORCID: 0000-0003-1923-976X <papertsukhov@edu.hse.ru>
A.A. Mitsyuk, ORCID: 0000-0003-2352-3384 <amitsyuk@hse.ru>
PAIS Lab, Faculty of Computer Science,
National Research University Higher School of Economics,
3 Kochnovskiy Proezd, Moscow, 101000, Russia.

Abstract. Event logs of software systems are used to analyze their behaviour and inter-component interaction. Artificial event logs with desirable specifics are needed to test algorithms supporting this type of analysis. Recent methods allow to generate artificial event logs by simulating ordinary Petri nets. In this paper we present the algorithm generating event logs for Petri nets with inhibitor and reset arcs. Nets with inhibitor arcs are more expressive than ordinary Petri nets, and allow to conveniently model conditions in real-life software. Resets are common in real-life systems as well. This paper describes the net simulation algorithm, and shows how it can be applied for event log generation.

Keywords: Petri nets; inhibitor arcs; reset arcs; simulation; event logs

For citation: Pertsukhov P.A., Mitsyuk A.A. Simulating Petri Nets with Inhibitor and Reset Arcs. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 4, 2019. pp. 151-162. DOI: 10.15514/ISPRAS-2019-31(4)-10

Acknowledgments. This work is supported by the Basic Research Program at the National Research University Higher School of Economics.

Симуляция сетей Петри с ингибиторными дугами и дугами сброса

П.А. Перцухов, ORCID: 0000-0003-1923-976X <papertsukhov@edu.hse.ru>
А.А. Мицюк, ORCID: 0000-0003-2352-3384 <amitsyuk@hse.ru>
Лаборатория ПОИС, факультет компьютерных наук,
Национальный исследовательский университет «Высшая школа экономики»
101000, Россия, Москва, Кочновский проезд, 3

Аннотация. Журналы событий программных систем используются для анализа их поведения и взаимодействия между компонентами. Искусственные журналы событий с подходящими свойствами необходимы для тестирования алгоритмов, используемых для такого анализа. Современные методы позволяют генерировать искусственные журналы событий в результате симуляции обычных сетей Петри. В этой статье мы представляем алгоритм, генерирующий журналы событий для сетей Петри с ингибиторными дугами и дугами сброса. Сети с ингибиторными дугами более выразительны, по сравнению с классическими сетями Петри, и позволяют удобно моделировать условия в реальном программном обеспечении. Операции сброса также распространены в реальных системах. В этой статье описывается алгоритм симуляции сетей Петри с ингибиторными дугами и сбросами, а также показано, каким образом его можно применять для генерации журнала событий.

Ключевые слова: сети Петри; ингибиторные дуги; дуги сброса; симуляция; журналы событий

Для цитирования: Перцухов П.А., Мицюк А.А. Симуляция сетей Петри с ингибиторными дугами и дугами сброса. Труды ИСП РАН, том 31, вып. 4, 2019 г., стр. 151-162 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(4)-10

Благодарности. Работа выполнена при поддержке Программы фундаментальных исследований Национального исследовательского университета Высшая школа экономики.

1. Introduction

Recently, process analytics evolved into an advanced field of computer-based technology. Automated methods have been created to find bottlenecks and inefficiencies in process models of information systems.

One particular technology that helps to automate process analysis is process mining [1]. Experts in this technology employ algorithms and methods which use the records of a system behaviour, which are called «event logs» or «system logs». This information can be explored to discover a model of how the real process behaves [1].

An existing process model runs can be aligned to the records of an event log to check if the model conforms to the real system behaviour [2]. The field also provides an expert with method to improve/repair processes and process models.

The process simulation methods are also applied in the field of process analytics [3].

Recently, it has been stated that process mining and simulation form "a match made in heaven" [4]. In particular, process model simulation can be applied to look in the future of a process, and to test *what-if* alternative scenarios possible because of process change. Moreover, the development of process mining algorithms is impossible without sample models and event logs with a suitable characteristics [1]. Sample event logs can be generated using the process model simulation methods [5]–[8]. Process mining and simulation can also be matched in other way. The results of process discovery and conformance checking can be applied to improve simulation models.

Various modelling formalisms are employed in the field of process analytics [1], [3]. Among them, the language of Petri nets is one of the most well-established, well-researched, simple, and commonly-used modelling languages [9]. Lots of process discovery and analysis techniques are based on this language [1].

A strength of the Petri net language is that on top of simply defined P/T-nets many extensions have been built. These are high-level Petri nets: Coloured Petri nets [10], Nested Petri nets [11], Object nets [12] etc. Method to simulate Petri nets of various types have been proposed in literature. However, for many types of Petri nets still there are no simulation techniques/tools.

This paper presents an approach and a tool to simulate Petri nets with reset and inhibitor arcs. The addition of these arc types improves the net expressiveness significantly. Thus, these nets are used when the process cannot be (conveniently) modelled by P/T-nets.

This paper is organized as follows. Section 2 defines models and event logs. In Section 3 the main contribution is presented: algorithms to simulate Petri nets with inhibitor and reset arcs. These algorithms are implemented in the tool which is described in Section 4. Finally, Section 5 concludes the paper.

2. Petri Nets and Event Logs

In this section, we define process models and event logs. Let $\mathbb{N}$ denote the set of all non-negative integers, and $\mathbb{N}_+ = \mathbb{N}\backslash\{0\}$.

2.1 Ordinary Petri nets

Petri nets are directed bipartite graphs which allows for modelling and representation of processes in information systems [13]. More formally, an *ordinary Petri net* is a triple $N = (P, T, F)$, where P and T are two disjoint sets of places and transitions, and $F \subseteq (P \times T) \cup (T \times P)$ is a flow relation. As graphs, Petri nets have convenient visual representation. Fig. 1 shows an example model.

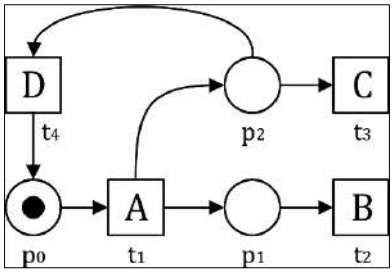


Fig. 1. An ordinary labelled Petri net

Places are shown by circles, transitions are shown by boxes, the flow relation is depicted using ordinary directed arcs. In fig. 1, there are three places (p_0, p_1, p_2) and four transitions (t_1, t_2, t_3, t_4). Transitions are labelled with activity names from the set $\mathcal{A} \cup \{\tau\}$. In the example, $\mathcal{A} = \{A, B, C, D\}$. Labels are placed inside the transition boxes. An Petri net can contain invisible *(silent)* process actions which are labelled with τ. Labels are assigned to transitions via a *labelling function* $\lambda: T \to \mathcal{A} \cup \{\tau\}$.

A state of an ordinary Petri net is called its *marking*. It is a function $M : P \to \mathbb{N}$ assigning natural numbers to places. In figures, a marking M can be designated by putting $M(p)$ black tokens into a place p of the net. By M_0 we denote the initial marking. For example, the initial marking of the net from Fig. 1 consists of a single token in the place p_0.

A transition represents an activity of a process. It is enabled in a current marking if in each of its input places (for $t \in T$ input places are $\bullet t = \{p \mid (p, t) \in F\}$) there enough tokens, that is $\forall p \in \bullet t$: $M(p) \geq 1$. An enable transition may fire that changes a marking of the net. It consumes tokens from the input places, and produces tokens to output places (for $t \in T$ input places are $t \bullet = \{p \mid (t, p) \in F\}$).

Consider a model from fig. 1: t_1 is the only enabled transition in the initial marking. It may fire, that corresponds to an occurrence of activity "A". Then, the transition consumes a single token from p_0 and produces tokens to p_1 and p_2. Fig. 2 illustrates this firing. The firing is local, each transition fires independently from other transitions.

2.2 Petri nets with Inhibitor and Reset Arcs

In this paper, we consider Petri nets with arcs of two additional types: reset and inhibitor arcs. These nets also contain places, transitions, and ordinary control flow arcs.

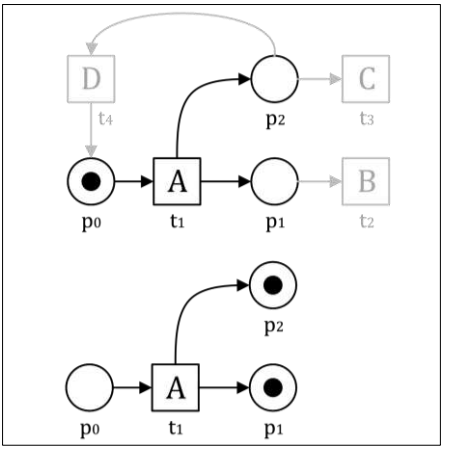


Fig. 2. Firing a Petri net

A labelled Petri net with weights, inhibitor and reset arcs (WIR Petri net) is a tuple $N_{WIR} = (P, T, F, W, R, I, \lambda)$, where

- (P, T, F) is an ordinary Petri net,
- $W \in F \to \mathbb{N}_+$ is an arc weight function,
- $R \subseteq P \times T$ is a function defining reset arcs,
- $I \subseteq P \times T$ is an inhibiting relation,
- and $\lambda: T \to A \cup \{\tau\}$ is a labelling function.

Fig. 3 shows an example WIR Petri net.

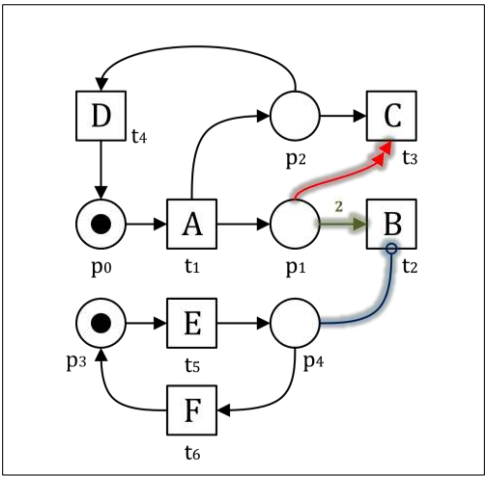


Fig. 3. A WIR Petri net

A *reset arc* removes all tokens from the place no matter of their number. These arcs also called *clear* arcs [14].

In Fig. 3, there is a reset arc from the place p_1 to the transition t_3 labelled with "C". Reset arcs are denoted with double arrows at the end. Note that the net contains a loop of two actions "A" (t_1) and "D" (t_4). The possible sequence of firings is $< t_1, t_4, t_1, t_4, t_1, t_4, t_3 >$. Before the last step, p_1 will contain 3 tokens all of which will be removed by the firing of t_3.

An *inhibitor arc* [15], [16] can be from a place to a transition. This transition cannot fire if there is a token in place connected with the transition using an inhibitor arc. $\circ t = \{p \mid (p, t) \in I\}$ denotes the set of inhibiting places for t. That is, an inhibiting place allows to prevent the transition firing. Transitions consume no tokens through inhibitor arcs.

Inhibitor arcs are shown with small circles instead of arrows at the end. In Fig. 3, there is a reset arc from the place p_4 to the transition t_2 labelled with "B". Thus, t_2 can not fire if there is a token in p_4 no matter how many tokens are in p_1. The whole part of the model that consists of places p_3 and p_4, transitions t_5 and t_6 is a switch with two possible states: open (there is a token in p_3) / closed (there is a token in p_4).

The firing of t_5 closes the switch and deprecates the firing of t_2. Thus, only t_3 is able to clear tokens from the place p_1 if there is a token in p_2. The firing of t_3 will end the process, because it will consume and remove all tokens from the upper part of the net.

The firing of t_6 opens the switch. Then, t_2 may consume tokens from p_1 independently of tokens in other places. Note that the arc from p_1 to t_2 has a weight of 2. Thus, each firing of t_2 consume exactly 2 tokens, and t_2 can not fire if there is only one token in p_1. Note that a particular WIR Petri net can contain zero number of reset and inhibitor arcs.

Marking of a WIR Petri net is defined in the same way as for an ordinary Petri net. But the firing rule is slightly different for WIR Petri nets. Each marking change is called *step*. In this paper, we

assume that each step consists of a single transition firing. The step from Fig. 2 is denoted by $M_0[t_1\rangle M'$, where $M'(p_1) = M'(p_2) = M'(p_3) = 1$ and $M'(p_0) = M'(p_4) = 0$.

2.3 Event Logs

In this paper, we apply process model simulation to generate event logs with records of the behavior. We define an *event log* as a finite multiset of traces $\in \mathcal{B}(\mathcal{A})$ [1]. A trace $\sigma \in \mathcal{A}^*$ is a finite sequence of events from the set $\mathcal{A}$. Note that transitions of a WIR Petri net are labelled with elements of $\mathcal{A}$. The only transitions which firing leaves no events are silent τ-transitions.

Technically, we record the event logs in XES (Extensible Event Stream) format[2] that will be considered in more detail in Section 4.

3. Petri Net Simulation Algorithm

This section describes the algorithm to simulate labelled WIR Petri nets.

The main idea of the algorithm is to iterate over all transitions and fire one of them at each iteration, recording corresponding events to the log. This procedure is performed in the main generating function called *generateTrace* (see Algorithm 1).

```
Input: transitions, initialMarking, finalMarking,
settings as {maxNumberOfSteps, maxIterations, isRemovingUnfinishedTraces}
Output: generated trace or NULL
1:    function generateTrace
2:    trace ← NULL;
3:    replayCompleted ← false;
4:    addTraceToLog ← false;
5:    iteration ← 0;
6:    repeat
7:       moveToInitialState();
8:       trace ← createTrace();
9:       stepNumber ← 0;
10:
11:         while stepNumber < maxNumberOfSteps
12:               and not replayCompleted do
13:
14:            transition ← chooseNextTransition();
15:            if transition = NULL then
16:               trace ← NULL;
17:               break;
18:            end if
19:
20:            fire(transition, trace);
21:            replayCompleted ← isCompleted(finalMarking);
22:            stepNumber ← stepNumber + 1;
23:         end while
24:
25:         iteration ← iteration + 1;
26:      until (iteration >= maxIterations or
27:         not isRemovingUnfinishedTraces or replayCompleted)
28:
29:      if not replayCompleted
30:            and isRemovingUnfinishedTraces then
31:         trace ← NULL;
32:      end if
```

[1] Here $\mathcal{B}(\mathcal{A}^*)$ denotes all multisets over $\mathcal{A}^*$, where $\mathcal{A}^*$ — all finite sequences with elements from $\mathcal{A}$.

[2] http://www.xes-standard.org/

```
33:      return trace;
34:   end function
```

Algorithm 1. One trace generation

This algorithm works as follows. We have *maxIterations* attempts to reach the final marking of the net. By default this number is 10. The function *moveToInitialState* initiates the trace generation by setting a marking of the to M_0. Then, we create an empty trace by calling *createTrace*.

At each step of the main loop, the algorithm chooses an enabled transition in the *chooseNextTransition*, and fire it (function *fire*). The function *fire* changes a marking of the net and writes an event to the trace. Then we call the function *isCompleted* to check if we reached the final marking and update *replayCompleted*. This loop iterates until we reach the final marking (*replayCompleted = true*) or exceed the specified limit of steps for one trace *maxNumberOfSteps*.

When we cannot find an enabled transition which is ready to fire, we clear the unfinished trace and begin a new attempt.

If no one of 10 attempts succeeded, we return NULL which will be recorded as an empty trace to the event log. Note that there is a setting of the prototype tool that removes all empty and unfinished traces.

The *chooseNextTransition* function selects an enabled transition using a specified rule. The most basic implementation of this function is shown in Algorithm 2. Here, the random transition among all enabled and noise transitions is selected.

```
Input: allTransitions, noiseTransitions
Output: selected transition or NULL
1: function chooseNextTransition(allTransitions, noiseTransitions)
2:    enabledTransitions ← findEnabledTransitions();
3:    return random transition among enabledTransitions
4:       and noiseTransitions or NULL;
5: end function
```

Algorithm 2. Looking for the next transition

This algorithm is based on the algorithm for ordinary Petri nets [17], and thus, is able to add noise to the event log. More complex rules to select the enabled transition can be applied. For example, priorities of preferences can be assigned to the transitions which affect the order of their firing. If there is no enabled transition, then NULL is returned.

To check if a transition t is enabled, we ensure that all input places connected with t with the help of ordinary arcs have enough tokens. Besides that, we check that places connected with t with the help of inhibitor arcs don't contain any tokens. Reset arcs don't affect if a transition is enabled or not. Algorithm 3 shows how this is done.

```
Input: allTransitions
Output: list of enabled transitions
1: function findEnabledTransitions(allTransitions)
2:    enabledTransitions ← ∅;
3:
4:    for transition in allTransitions do
5:       enabled ← true;
6:       for arc in transition.inputArcs do
7:          if arc.place.numberOfTokens < arc.weight then
8:             enabled ← false;
9:             break;
10:            end if
11:         end for
12:         if not enabled then
```

```
13:           continue;
14:        end if
15:        for arc in transition.inhibitorArcs do
16:           if arc.place.numberOfTokens > 0 then
17:              enabled ← false;
18:              break;
19:           end if
20:        end for
21:        if enabled then
22:           enabledTransitions.add(transition);
23:        end if
24:     end for
25:     return enabledTransitions;
26:  end function
```

Algorithm 3. Finding enabled transitions

Algorithm 4 shows the transition firing function. This function produces and consumes tokens, and then adds an event corresponding to this transition to the trace. The basic implementation is shown which considers only transition names. There are much more complicated implementations of this function for time-driven, resources, and priorities generation modes.

```
Input: transition, place
Output: traced events
1: function fire(transition, trace)
2:    for arc in transition.inputArcs do
3:       arc.place.consumeToken(arc.weight);
4:    end for
5:    for arc in transition.inputResetArcs do
6:       arc.place.consumeAllTokens();
7:       end for
8:
9:    log transition to trace or perform some noise event
10:
11:      for arc in transition.outputArcs do
12:         arc.place.produceTokens(arc.weight);
13:      end for
14:   end function
```

Algorithm 4. Firing function

4. Prototype Tool

The presented event log generation algorithm has been implemented as a prototype tool. It is written in Java and Kotlin programming languages. In this section, we consider the tool. The tool consists of two parts: *Generation Setup* unit and *Generation* unit.

4.1 Preparing for generation

In preparation part we receive settings from the GUI (see fig. 4) or read a JSON (see fig. 5) file. Settings from JSON are validated. Then we load the model from a PNML[3] file and prepare this model for generation. Inhibitor and reset arcs could be either specified in settings file, or loaded from the PNML file. The initial and final markings are loaded in the same manner.

[3] www.pnml.org/

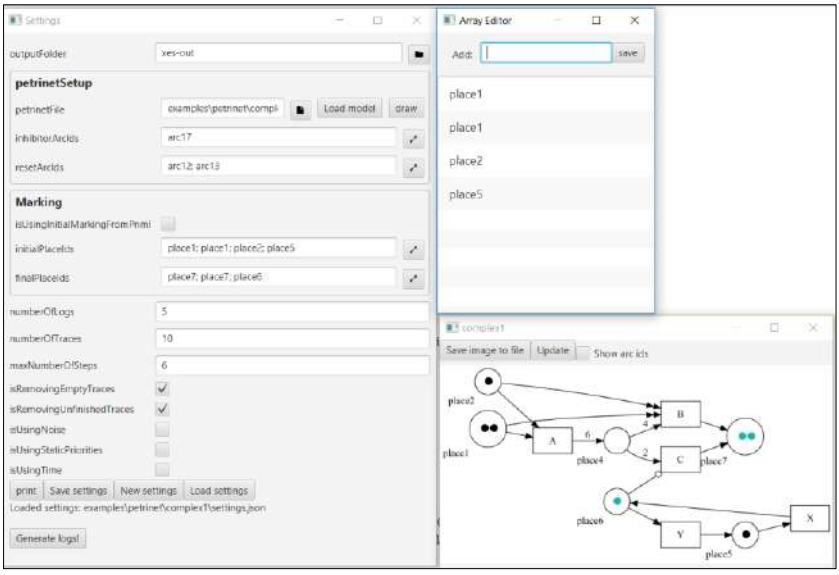


Fig. 4. Tool GUI


```
{
  "petrinetSetup" : {
    "petrinetFile" : "examples\\petrinet\\complex1\\complex1.pnml",
    "marking" : {"isUsingInitialMarkingFromPnml": false...},
    "inhibitorArcIds" : [ "arc17" ],
    "resetArcIds" : [ "arc12", "arc13" ]
  },
  "outputFolder" : "xes-out",
  "isRemovingEmptyTraces" : true,
  "isRemovingUnfinishedTraces" : true,
  "numberOfLogs" : 5,
  "numberOfTraces" : 10,
  "maxNumberOfSteps" : 6,
  "isUsingNoise" : false,
  "noiseDescription" : {
    "noiseLevel" : 5,
    "isSkippingTransitions" : true,
    "isUsingExternalTransitions" : true,
    "isUsingInternalTransitions" : true,
    "internalTransitionIds" : [ ],
    "existingNoiseEvents" : [ {
      "activity" : "NoiseEvent",
      "executionTimeSeconds" : 600,
      "maxTimeDeviationSeconds" : 120
    } ]
  },
  "isUsingStaticPriorities" : false,
  "staticPriorities" : {"maxPriority": 1...},
  "isUsingTime" : false,
  "timeDescription" : {"generationStart": "2019-04-07T22:27:06.991Z"...}
}
```


Fig. 5. Generation settings in JSON

After that we create an instance of special class *GenerationHelper*, which encapsulates the main code for choosing transitions, looking for enabled transitions, handling noise and artificial log events

if it is needed. There are different helpers for simple generation, generation with priorities, and generation with time.

Also, we convert each transition to a special loggable transition-related class which is used during generation. They contain methods of event recording to a trace. Such a class also consumes tokens from input places and produces tokens to output places. Methods to check if a transition is enabled are also here.

4.2 Preparing for generation

A singleton class is used to record the event log. We setup this logging class. It uses the OpenXES[4] library with the help of which we can write XES log files. The XES format is common for the field of process mining [1]. OpenXES library creates a separate file for each log. A fragment of the example output in XES file is shown in fig. 6. This example contains two traces with names "Trace 4" and "Trace 5". These names should be unique.

The first trace is of the three events: "B", "D", and "D". Note that XES is XML-based and is easy-to-read for a machine or a human.

Then we use a *Generator* class to launch the main generation method (see Algorithm 1), passing a *generationHelper* to this class.

```
<trace>
    <string key="concept:name" value="Trace 4"/>
    <event>
        <string key="concept:name" value="B"/>
    </event>
    <event>
        <string key="concept:name" value="D"/>
    </event>
    <event>
        <string key="concept:name" value="D"/>
    </event>
</trace>
<trace>
    <string key="concept:name" value="Trace 5"/>
    <event>
        <string key="concept:name" value="D"/>
    </event>
    <event>
        <string key="concept:name" value="B"/>
    </event>
    <event>
        <string key="concept:name" value="D"/>
    </event>
</trace>
```

Fig. 6. Fragment of XES file

4.3 Tool Usage

Let us test our tool on the model from the fig. 7.

The initial marking consists of four tokens (shown as black dots): one token lies in *place2*, one is in *place5*, and two tokens are in *place1*. Our goal is to reach the final marking (shown as green dots): two tokens in *place7*, and one token in *place6*.

Transition C is enabled only when place place6 is empty. C consumes 2 tokens from *place4* and produces a token to *place7*. Transition B removes all tokens in places *place1* and *place2*. When this transition fires, it consumes 4 tokens from *place4* and produces a token to *place7*. Thus, B can not

[4] http://www.xes-standard.org/openxes/start

fire before A. At each step either X or Y is enabled, so these transitions may produce a trace of any length. We set-up our tool to remove unfinished traces and limited max number of steps to 10. The result of the generation is shown in fig. 8.

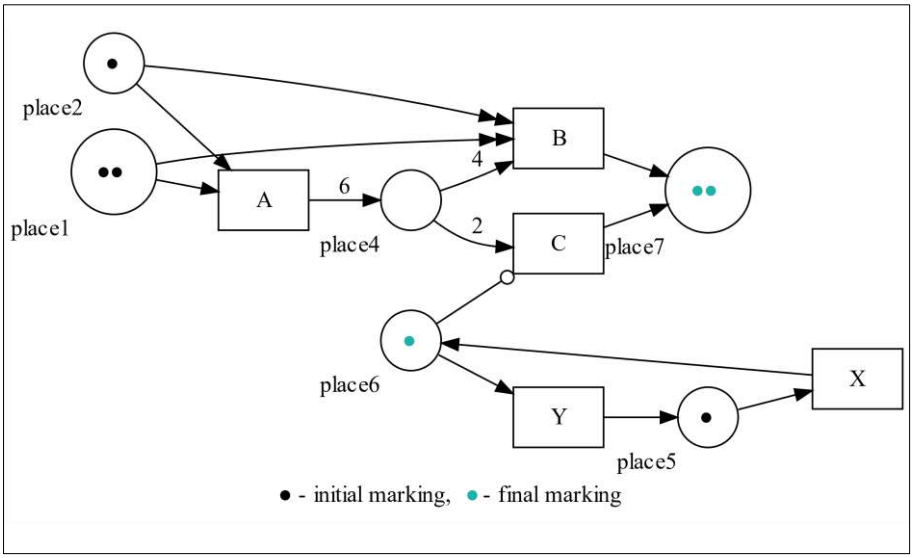


Fig. 7. Petri net used for log generation

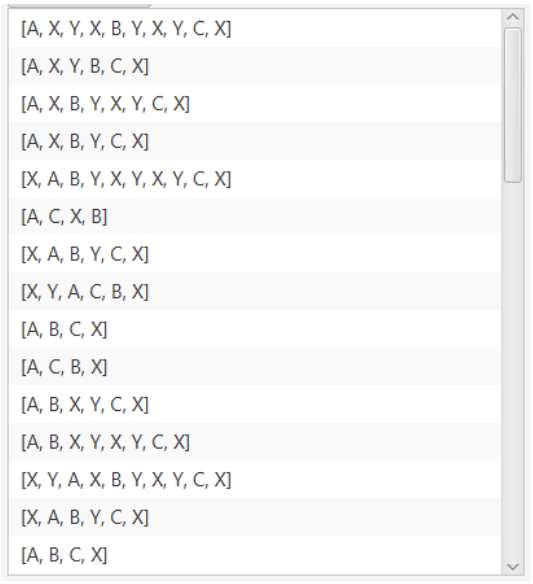


Fig. 8. Resulting traces

Let us look at some trace, for example: $<A, X, B, Y, C, X>$. First fired transition was A, and it produced 6 tokens to *place4*. Then X fired. It consumed a token from *place5* and produced one to *place6*. C cannot fire in this marking. Thus, B fires. 4 of 6 tokens was consumed from *place4*, and one token produced to *place7*. Then Y fired and «opened» C. C has been fired just after Y was fired, when a token has been removed from *place6*. B and C were fired once each, and produced 2 tokens in total to *place7*. *place1* and *place2* were cleared by B. The last event was X, which placed a token to *place6*. At the end of this run all tokens reside in the final marking.

5. Conclusion

In this paper, we have presented the algorithm to simulate a process model in the form of weighted labelled Petri net with inhibitor and reset arcs. This algorithm can be applied to generate event logs from the event log. Proposed algorithm continues the previous works on Petri net simulation with the purpose of generating artificial event logs. The prototype implementation is based on Gena tool[5].

We have plans for future work. Firstly, we plan to comprehensively evaluate the proposed algorithm on artificial and real-life process models. For now, we just tested it on sample models to check algorithm validity. Secondly, we also plan to improve the prototype implementation and make it stable and usable. Thirdly, Gena is able to simulate timed process models, models with resources, data, add noise to an event log [8], [17]. Recently, an extension for Gena to simulate the multi-agent system has been proposed [18]. We plan to merge these extensions and the algorithm presented in this paper. Then, it will be possible to simulate WIR Petri nets with time/resources, and data.

References

[1]. Wil M.P. van der Aalst. Process mining: Discovery, Conformance and Enhancement of Business Processes. Springer, 2011, 352 p.
[2]. J. Carmona, B.F. van Dongen, A. Solti, and M. Weidlich. Conformance Checking – Relating Processes and Models. Springer, 2018, 270 p.
[3]. M. Weske. Business Process Management: Concepts, Languages, Architectures. Springer, 2007, 408 p.
[4]. Wil M.P. van der Aalst. Process mining and simulation: a match made in heaven! In Proc. of the 50th Computer Simulation Conference, 2018, Article No. 4.
[5]. A. Burattin and A. Sperduti. PLG: A framework for the generation of business process models and their execution logs. Lecture Notes in Business Information Processing, vol. 66, 2010, pp. 214–219.
[6]. A. Burattin. PLG2: multiperspective process randomization with online and offline simulations. In Proc. of the BPM Demo Track 2016, CEUR Workshop Proceedings, vol. 1789, 2016.
[7]. T. Jouck and B. Depaire. Ptandloggenerator: A generator for artificial event data. In Proc. of the BPM Demo Track 2016, CEUR Workshop Proceedings, vol. 1789, 2016.
[8]. A.A. Mitsyuk, I.S. Shugurov, A.A. Kalenkova, and W.M.P. van der Aalst. Generating event logs for high-level process models. Simulation Modelling Practice and Theory, vol. 74, 2017, pp. 1–16.
[9]. W. Reisig, Understanding Petri Nets – Modeling Techniques, Analysis Methods, Case Studies. Springer, 2013, 260 p.
[10]. K. Jensen and L. M. Kristensen. Coloured Petri Nets – Modelling and Validation of Concurrent Systems. Springer, 2009, 384 p.
[11]. I.A. Lomazova. Nested Petri Nets: Multi-level and Recursive Systems. Fundamenta Informaticae, vol. 47, no. 3-4, 2001, pp. 283–293.
[12]. R. Valk. Object Petri nets: Using the nets-within-nets paradigm. Lecture Notes in Computer Science, vol. 3098, 2003, pp. 819–848.
[13]. W.M.P. van der Aalst and K.M. van Hee. Workflow Management: Models, Methods, and Systems. MIT Press, 2002, 384 p.
[14]. C. Lakos and S. Christensen. A general systematic approach to arc extensions for coloured Petri nets. Lecture Notes in Computer Science, vol. 815, 1994, pp. 338–357.
[15]. R. Janicki and M. Koutny. Semantics of inhibitor nets. Information and Computation, vol. 123, no. 1, 1995, pp. 1–16.
[16]. H. C. M. Kleijn and M. Koutny. Process semantics of p/t-nets with inhibitor arcs. Lecture Notes in Computer Science, vol. 1825, 2000, pp. 261–281.
[17]. S. Shugurov and A. A. Mitsyuk. Generation of a Set of Event Logs with Noise. In Proc. of the 8th Spring/Summer Young Researchers Colloquium on Software Engineering (SYRCoSE 2014), 2014, pp. 88–95.
[18]. Nesterov R.A., Mitsyuk A.A., Lomazova I.A. Simulating Behavior of Multi-Agent Systems with Acyclic Interactions of Agents. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 3, 2018, pp. 285-302. DOI: 10.15514/ISPRAS-2018-30(3)-20.

[5] https://pais.hse.ru/research/projects/gena

Информация об авторах / Information about authors

Павел Алексеевич ПЕРЦУХОВ – студент бакалавриата НИУ ВШЭ в Москве по специальности программная инженерия, поступил в 2016 году. Стажёр в лаборатории ПОИС. Основные исследовательские интересы: сети Петри, process mining, компиляторы, синтаксический анализ.

Pavel Alexeevitch PERTSUKHOV is a bachelor student at the HSE Moscow, on the program 'Software Engineering', starting from 2016, PAIS Lab intern. His research interests include Petri nets, process mining, compilation, syntax analysis.

Алексей Александрович МИЦЮК – научный сотрудник лаборатории процессно-ориентированных информационных систем факультета компьютерных наук НИУ ВШЭ в Москве. Алексей закончил Московский государственный институт электроники и математики в 2009 году. Работал в качестве разработчика программного обеспечения и системного инженера. В 2013 году Алексей стал сотрудником вновь созданной лаборатории ПОИС под совместным руководством проф. И.А.Ломазовой и проф. В.М.П. ван дер Аалста. В 2019 году Алексей получил степень кандидата компьютерных наук от НИУ ВШЭ. Основные исследовательские интересы: сети Петри, process mining, моделирование процессов, архитектура информационных систем и программного обеспечения.

Alexey Alexandrovitch MITSYUK is a research fellow at the Laboratory of Process-Aware Information Systems of the Computer Science Faculty, HSE Moscow. Alexey graduated from Moscow State Institute of Electronics and Mathematics in 2009. He worked as a software developer and system engineer in industry. In 2013 Alexey has joined newly created PAIS Lab under the co-supervision of prof, I.A. Lomazova and prof. W.M.P. van der Aalst. Alexey received his Ph.D. in Computer Science from Higher School of Economics in 2019. His research interests include Petri nets, process mining, process modeling, architecture of information systems and software.

DOI: 10.15514/ISPRAS-2019-31(4)-11

Computing Transition Priorities for Live Petri Nets

K.G. Serebrennikov, ORCID: 0000-0002-8420-9826 <cyrilsilver94@gmail.com>
National Research University Higher School of Economics,
20, Myasnitskaya st., Moscow, 101000, Russia

Abstract. In this paper, we propose an approach to implementation of the algorithm for computing transition priorities for live Petri nets. Priorities are a form of constraints which can be imposed to ensure liveness and boundedness of a Petri net model. These properties are highly desirable in analysis of different types of systems, ranging from business processes systems to embedded systems. The need for them is imposed by resource limitations of real-life systems. The algorithm for computing transition priorities considered in the study has exponential time complexity, since it is based on construction and traversal of the coverability graph. However, its performance may be sufficient for the majority of real-life cases. This paper covers the design considerations of the implementation, including the approach to handling the high time complexity of the algorithm and optimizations introduced in the original algorithm. We target parallelization as the main method of performance increase. While, for some steps of the algorithm the parallelization approach proves to be viable, for others its applicability is questioned. Analysis of different design decisions is provided in the text. On the basis of the actual implementation an application for computing priorities was developed. It can be used for further analysis of the algorithm applicability for real-life cases.

Keywords: formal methods; Petri nets; coverability graph; priority relation; cyclic behavior

For citation: Serebrennikov K.G. Computing transition priorities for life Petri nets. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 4, 2019. pp. 163-174. DOI: 10.15514/ISPRAS-2019-31(4)-11

Acknowledgments. This work is supported by the Basic Research Program at the National Research University Higher School of Economics.

Вычисление приоритетов срабатывания переходов для живых сетей Петри

К.Г. Серебренников, ORCID: 0000-0002-8420-9826 <cyrilsilver94@gmail.com>
Национальный исследовательский университет «Высшая школа экономики»,
101000, Россия, г. Москва, ул. Мясницкая, д. 20

Аннотация. В данной статье представлен подход к реализации алгоритма вычисления приоритетов срабатывания переходов для живых сетей Петри. Приоритеты являются одной из форм условий срабатывания и могут быть присвоены переходам для обеспечения живости и ограниченности сети Петри. Наличие этих свойств крайне желательно при анализе различных систем, начиная от бизнес-процессов и заканчивая встраиваемыми системами. Необходимость в них обусловлена ограниченностью ресурсов, характеризующей большинство систем из реальной практики. Рассматриваемый в данном исследовании алгоритм для вычисления приоритетов срабатывания переходов имеет экспоненциальную временную сложность, так как основан на процедурах построения и обхода графа покрытия. Однако, его производительность может быть достаточна для большинства практических целей. В данной работе затронуты различные аспекты проектирования реализации, включая подход к решению проблемы высокой сложности алгоритма и примененные к нему оптимизации. В качестве основного метода повышения производительности алгоритма были выбраны параллельные вычисления. Несмотря на то, что для одних шагов алгоритма данный подход продемонстрировал свою жизнеспособность, для других его эффективность оказалась не столь

однозначной. В работе представлен анализ различных решений. Также, на основе реализации алгоритма было разработано приложение для вычисления приоритетов срабатывания. Данное приложение может быть использовано для дальнейших исследований реальной применимости алгоритма.

Ключевые слова: формальные методы; сети Петри; граф покрытия; отношение приоритетов; цикличное поведение

Для цитирования: Серебренников К.Г. Вычисление приоритетов срабатывания переходов для живых сетей Петри. Труды ИСП РАН, том 31, вып. 4, 2019 г., стр. 163-174 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(4)-11

Благодарности. Работа выполнена при поддержке Программы фундаментальных исследований Национального исследовательского университета «Высшая школа экономики».

1. Introduction

Petri nets are widely applicable for modeling and analysis of various distributed systems ranging from business processes systems to biological systems. Regardless the nature of such systems their models always have some properties of liveness and boundedness. Properties of liveness include reiteration of all subprocesses and return to some initial state of the system. Properties of boundedness are those related to finiteness of the set of possible states.

In most of the cases, it is highly desirable for the system to have finite set of states, i.e. its model should be bounded. Let us consider, for example, a Petri net shown in fig. 1. This Petri net is a model of a simple producer/consumer system, where the left cycle represents a producer, the right cycle – a consumer, and the place p_3 between them is a buffer. This net is live, i.e. in every reachable marking each transition can eventually fire. The net is unbounded, since the number of tokens in p_3 can be arbitrary large. It means that the buffer overflow will eventually occur. Thus, it is desirable to transform the model into live and bounded preserving the original structure of the net.

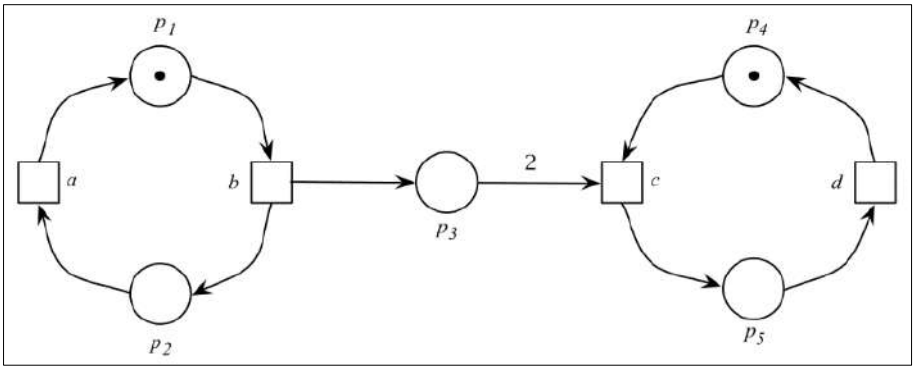


Fig. 1. Example of a marked Petri net. Model of a producer/consumer system

The problem of transformation of a given live and unbounded Petri net into live and bounded without modification of its original structure was considered in [1] and [2]. The authors focused on two approaches to control of Petri net behavior: through priority-based and through time-based constraints on transition firings. Algorithms for computing priorities and time intervals were proposed by them. This paper continues their study.

The proposed algorithms are based on construction of the spine tree, which is a subgraph of the reachability tree, containing exactly all feasible cyclic runs in a net. It represents the behavior that should be saved in a transformed Petri net to preserve the liveness property of the original net. The procedure of obtaining those cyclic runs each of which contains all transitions and is reachable from the initial marking was introduced in [3]. It is based on construction of the coverability graph that is finite by definition but can be extremely large. This fact affects negatively the overall time complexity of the algorithms.

Nevertheless, the performance of these algorithms may be optimal for the majority of real life cases. In this paper we target implementation considerations of the above mentioned algorithms. The study is focused on computation of transition priorities leaving apart computation of time intervals, since these two procedures have common foundation. The main contributions of this paper are the following.

1) An approach to implementation design of the algorithm for computing transition priorities based on construction steps optimization and adoption of parallelization.
2) A brief analysis of the actual implementation coded in Java programming language.
3) A Java application with GUI built upon the algorithm implementation that can be used for running experiments and researching the applicability of the algorithm in practical cases. The application supports the Petri Net Markup Language (.pnml) file format and can be used along with other tools for Petri net modeling and analysis.

The source code of the application along with build and run instructions can be found in the repository[1].

The structure of the paper is as follows. In Section 2 the main theoretical preliminaries are provided. Section 3 contains a brief description of the algorithm under consideration. In section 4 the approach to implementation design is described and results of implementation analysis are presented. Section 5 concludes the paper.

2. Preliminaries

For a more detailed introduction to the concepts presented in this section see, e.g., [4].

Let $\mathbb{N}$ denote the set of natural numbers (including 0). We define a marked Petri net as a tuple

$$(P, T, pre: T \times P \longrightarrow \mathbb{N}, post: T \times P \longrightarrow \mathbb{N}, m0: P \longrightarrow \mathbb{N})$$

where P and T are finite disjoint sets of places and transitions, respectively. $pre(t,p)$ is a number of tokens required to present on place p to enable transition t. The firing of t adds $post(t,p) - pre(t,p)$ tokens to p. Graphically, places are denoted by circles and transitions by squares. There is a directed arc from p to t if $pre(t,p) > 0$. The arc is annotated with $pre(t,p)$ if $pre(t,p) > 1$. Similarly, there is a directed arc from t to p if $post(t,p) > 0$. It is annotated with $post(t,p)$ if $post(t,p) > 1$. The pre-set of a transition t is the set of places p satisfying $pre(t,p) > 0$. The post-set of t is a set of places p satisfying $post(t,p) > 0$. A marking of a net is a mapping $m: P \longrightarrow \mathbb{N}$. The initial marking m_0 is represented by $m_0 0(p)$ tokens on place p.

An initial run is a sequence of transition firings, starting with the initial marking. Reachable markings are all those markings, which can be reached by the initial run. A cyclic run is a finite run starting and ending at the same marking.

A reachability graph of a Petri net is a labeled directed graph, in which vertices correspond to reachable markings of the net. A directed edge from vertex v to vertex v' is labeled with transition t, which is enabled by marking m represented by v and leads to marking m' represented by v'.

A Petri net is bounded if, for each place p, the number of tokens on p does not exceed some fixed bound $k \in \mathbb{N}$, i.e. for each reachable marking m the following is true: $m(p) \leq k$. Thus, a Petri net is bounded if and only if its reachability graph is finite.

A marking m' strictly covers a marking m if and only if for each place $p \in P, m'(p) \geq m(p)$ and $m' \neq m$.

In case of unbounded nets coverability graphs provide finite information about behavior. The construction of coverability graph is based on the notion of the generalized marking, which is formally a mapping: $P \longrightarrow \mathbb{N} \cup \{\omega\}$, where ω denotes an arbitrary number of tokens on a place. A coverability graph is defined constructively: it is constructed successively like the reachability graph starting from the initial marking. However, in case of the coverability graph, when a marking m'

[1] https://github.com/molassar/PN-transition-priority-computer

represented by a current leave v' in the reachability graph strictly covers a marking m represented by a vertex v, lying on the path from the root to v', then in the coverability graph the vertex v' gets a marking m_w, where $m_w(p) = \omega$ if $m'(p) > m(p)$, and $m_w(p) = m'(p)$ if $m'(p) = m(p)$. Fig. 2 shows a coverability graph of the example shown in fig. 1.

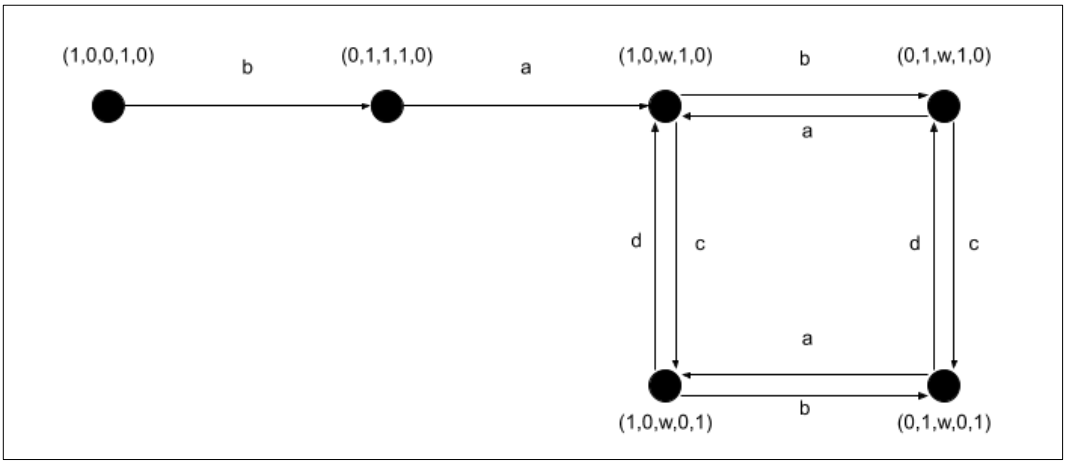


Fig. 2. A coverability graph of the Petri net of fig. 1

The coverability net of a Petri net $N = (P, T, pre, post, m_0)$ is a new Petri net $N' = (P', T', pre', post', m_0')$, which is constructed on the basis of a coverability graph (V, E, v_0) of the original net. The transitions of N' are mapped to the transitions of N by a labeling function λ': $T' \longrightarrow T$. The coverability net is formally defined as:

- $P' = V$,
- $T' = E$,
- $pre'((v', t, v''), v) = 1 \; if \; v = v'$,
- $pre'((v', t, v''), v) = 0 \; if \; v \neq v'$,
- $post'((v', t, v''), v) = 1 \; if \; v = v''$,
- $post'((v', t, v''), v) = 0 \; if \; v \neq v''$,
- $m_0'(v) = 1 \; if \; v = v_0$,
- $m_0'(v) = 0 \; if \; v \neq v_0$,
- $\lambda'(v, t, v') = t$.

The extended coverability net is the coverability net, that contains additional places to capture the token count change for ω-marked places of the original net. For each unbounded place p in the original net a place p is added to the extended coverability net. If transition t is in the pre-set of p in the original net N, then all transitions $t' \in T'$ with $\lambda'(t') = t$ are in the pre-set of the added place p. Arc weights are taken into account. The same holds for the post-sets of added places. The initial marking of added places coincides with the initial marking of these places in the original net. Fig. 3 demonstrates the extended coverability net constructed upon the coverability graph shown in fig. 2.

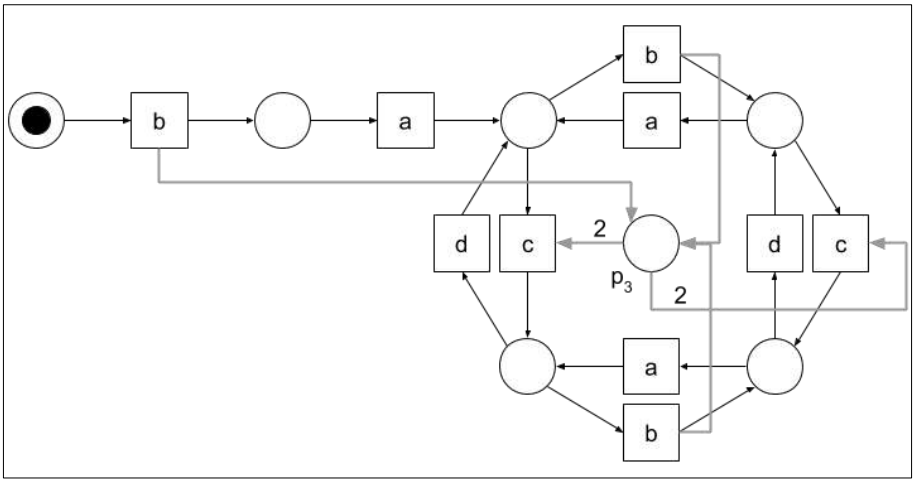


Fig. 3. The extended coverability net of the Petri net of Fig. 1

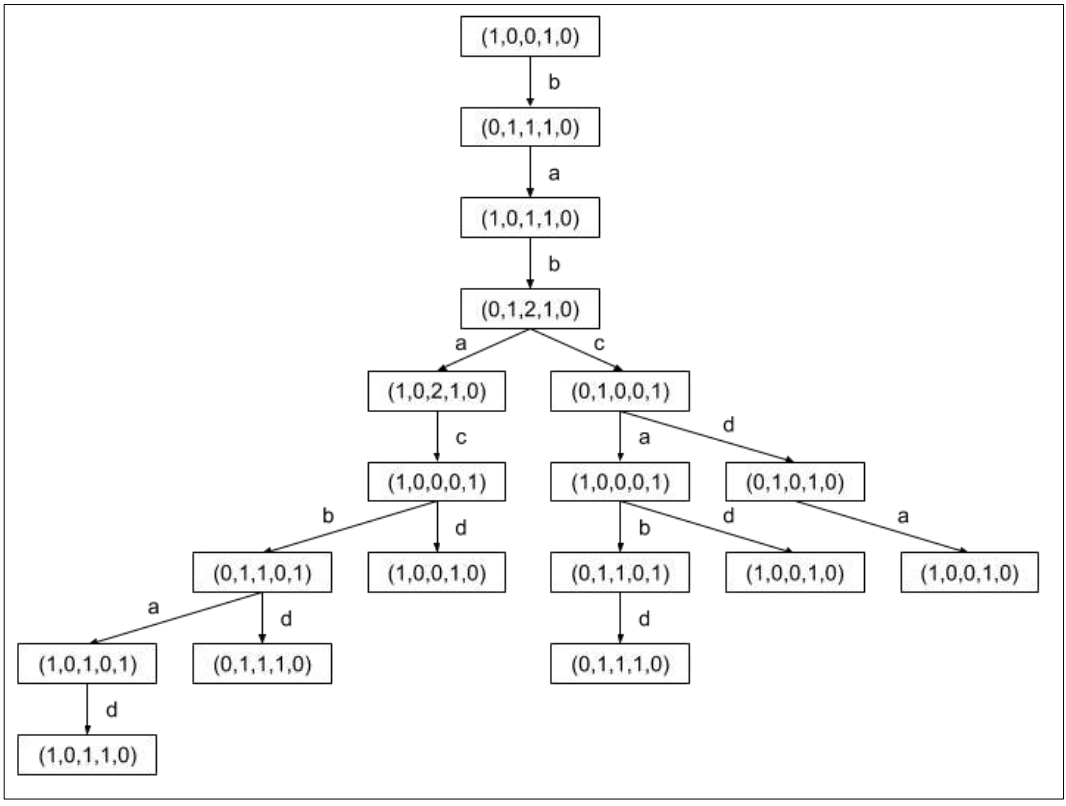


Fig. 4. The spine tree of the Petri net of Fig. 1

We define the set of all minimal feasible cyclic runs together with prefixes leading to the cycles in a Petri net N as

$$C(N) = \{\tau\sigma \mid \tau\sigma * \textit{ is an initial run in } N,$$
$$\tau \textit{ does not include } \sigma \textit{ and } \sigma \textit{ includes all transitions in } N\}$$

where σ is a minimal feasible cyclic run in N and τ is a finite initial run leading to σ. A spine tree is a subgraph of a reachability tree, that contains exactly all runs from $C(N)$. The spine tree contains

the behavior that should be saved in course of transformation in order to keep a Petri net live. For the Petri net in fig. 1 $C(N) = \{babacd, babcad, babcda, bababcbd, babcabd, bababcbad\}$. Thus, the spine tree of the net has the construction as shown in fig. 4.

A priority relation for a Petri net N is a partial order $(T, \ll)$, i.e., the relation is reflexive, antisymmetric and transitive. A priority relation $\ll$ can be specified by assigning a priority label $\pi(t) \in \mathbb{N}$ to each transition t. Thus, $t \ll t'$ if and only if $\pi(t) < \pi(t')$. A Petri net with priorities is a Petri net together with a priority relation. In a Petri net with priorities if Q is a set of all transitions enabled in a marking *m*, then only transitions with the highest priority may fire.

3. Algorithm overview

Let N be a live and unbounded Petri net. The task is to check, if it is possible to transform this net into live and bounded by adding priorities to its transitions. To accomplish the task we should find those transition priorities, which exclude runs leading to unboundedness.

It is possible to distinguish two major stages in the algorithm. The first is the search for cyclic runs in a Petri net. The presence of cyclic runs is a necessary condition for existence of transition priorities for a given Petri net. On the second stage the spine tree is built with the cyclic runs found on the previous stage forming its skeleton. The whole algorithm has the following sequence of actions.

1) Given a Petri net build its coverability graph;
2) Given the coverability graph constructed on the previous step build a coverability net;
3) Transform the coverability net into an extended coverability net;
4) Find behavioral cycles of the extended coverability net;
5) If the set of cyclic runs computed on the previous step is not empty build a spine tree in which the cyclic runs form the skeleton;
6) Given the spine tree build a spine-based coverability tree in which all leaves are colored in either red or green[2];
7) Traverse the spine-based coverability tree. For each non-red node *a* with its incoming edge labeled after transition t_1 if there exists a red sibling *b* with its incoming edge labeled after transition t_2 add (t_1, t_2) to the priority relation;
8) Assign priority labels to the transitions on the basis of the priority relation computed on the previous step.

The steps 1-4 form the first stage of the algorithm – search for cyclic runs. The procedure was introduced in [3]. The second stage – computation of a priority relation through spine tree construction – is represented by the steps 5-7. It was described in [1].

4. Implementation approach

4.1 Cyclic runs search

The problem of the search for cyclic runs in a Petri net can be reduced to the search for cyclic runs in its extended coverability net. The original algorithm for the cyclic runs search is comprised of four steps. We have reduced the number of steps to two by optimizing the construction phases of coverability and extended coverability nets.

The first step is the construction of a coverability graph. This step can not be avoided since the coverability graph is the foundation for the rest of the algorithm. Since the coverability graph can grow exponentially the overall time complexity of the algorithm is also exponential. We have chosen parallelization to target the problem. The implementation of coverability graph construction and traversal steps was designed to be easily parallelized.

[2] For the complete algorithm see [1].

Listing 1 demonstrates the pseudocode of the algorithm for building the coverability graph. Each node is processed in the following way: the set of transitions is filtered to find the transitions enabled by the marking corresponding to a node, then all the filtered transitions are fired to produce new generalized markings. Directed arcs are added from the vertex labeled by the marking of the node to the vertices labeled by the produced generalized markings. New nodes are generated from the obtained markings for further processing. Processing of a node does not depend on processing of other nodes so this task can be scheduled in parallel. The actual implementation of the pseudocode has the time complexity

$$O(|V| * |T| * d(CG) * |P|)$$

where $|V|$ is the number of vertices in a coverability graph, $|T|$ – the cardinality of the transition set of a Petri net, $d(CG)$ – depth of the coverability graph, i.e., the distance from the root to the most distant vertex, and $|P|$ – the cardinality of the set of places.

```
procedure buildCG(m0,T)
Input:  Initial marking m0 ∈ M,
          M - set of generalized markings;
        set of transitions T
Output:   Directed graph G = (V,E), |V| = |M|
G = initGraph()
root = Node(marking: m0, parentNode: null)
Q = makequeue(root)
while Q is not empty:
   node = dequeue(Q)
   m = marking(node)
   v = Vertex(label: label(m))
   incidentFrom = listIncidentFrom(G,v)
   if incidentFrom is empty:
      fireableT = filter(T, predicate: isFireableFrom(m))
      for each t ∈ fireableT:
         mn = fire(m,t)
         parentNode = node
         while parentNode is not null:
            mp = marking(parentNode)
            if isStrictlyCoveredBy(mn,mp):
               mn = generalize(mn,mp)
               break
            parentNode = parentNode(parentNode)
         u = Vertex(label: label(mn))
         e = Edge(label: label(t))
         addIncidentFrom(G,v,e,u)
         newNode = Node(marking: mn, parentNode: node)
         enqueue(Q,newNode)
```

Listing. 1. Pseudocode of the algorithm for building the coverability graph

We have made two implementations of the algorithm: single-threaded and parallel. In the parallel version, node processing tasks are submitted to a thread pool. Since the worker threads process the submitted tasks asynchronously, there is a need for some synchronization mechanism in order to prevent the function call in the master thread from returning before the graph is fully constructed. For the purposes of synchronization, we have used Phaser. It is a reusable synchronization barrier, which is provided by the standard Java library. The two main operations of Phaser are register and arrive. It is possible to block on Phaser, while the number of registered is not equal to the number of arrived. Phaser is incorporated into the implementation in the following way: the master thread submits the first task to the pool and blocks on Phaser, when a working thread generates a new task it is registered in Phaser, and after completion it arrives. Thus, the function call does not return until all the submitted tasks are processed, i.e. the graph is completely built. We used ForkJoinPool from

the fork/join Java framework as the implementation of the thread pool, since it provides the work-stealing mechanism. If a worker thread runs out of tasks in its pool, it can steal tasks from other threads that are busy. Thus, all the available processing powers are utilized and the performance increases.

We have also conducted a benchmarking of the two implementations with the use of JMH open source tool. Coverability graph construction for the Petri net in Fig. 1 was benchmarked. The *single shot time* mode was selected for the test. In this mode the time for a single operation is measured. Thus the "cold" performance of an algorithm is estimated, since no preliminary warm-up is conducted. This mode is most similar to the real-life usage scenario of the algorithm. The results are presented in Table 1.

Table 1. Benchmark results for implementations of the coverability graph construction algorithm

Benchmark mode	Single-Threaded Score	Parallel Score
Single shot time	5080.768 us/op	7632.496 us/op

The results in Table. 1 demonstrate that the single-threaded implementation performs slightly better for the Petri net of fig. 1. The probable reason is the low complexity of the net. The time complexity of processing a single node is

$$O(|T| * d(CG) * |P|)$$

and, hence, there are strong reasons to believe, that with increase in concurrency of the model and in number of transitions and places the parallel implementation will outperform the single-threaded one. Further experiments conducted with more complex nets proved this assumption.

The construction of the coverability net and the extended coverability net was optimized: we used coverability graph built on the previous stage and a separate data structure for capturing the number of tokens on unbounded places to find all feasible cyclic runs.

Initially, we have designed an implementation of the algorithm for the cyclic runs search, which also targeted parallelization. That implementation resembled closely the implementation of the coverability graph construction algorithm. Both of them were built upon separate nodes processing and since this procedure is completely independent for each node it can be easily parallelized. However, several tests proved this approach to be practically inapplicable. The reason is high memory space consumption. The number of enqueued nodes (logically they represent paths to be checked for cyclic behavior) increases exponentially rapidly exhausting memory capacity.

Thus, we have designed another implementation based on the backtracking principle. It is much more memory efficient, since only one node (path) is processed and stored in memory in every single moment of time. The problem with this approach is that it is difficult to be parallelized. And the developed implementation is also single-threaded. However, processing of each path in the actual implementation has the time complexity $O(|E|)$ where $|E|$ is the number of edges in the coverability graph. This means that in general case, the amount of work needed to be done to process a single path is insignificant for a modern process, and hence no major increase in performance should be observed with addition of parallelization. Nevertheless, the transformation of the proposed single-threaded implementation into a parallel one can be a subject of future research. The pseudocode of the cyclic runs search procedure based on backtracking is presented in Listing 2.

```
procedure backtrackingCyclicRunsSearch(G,m0,upm0,T)
Input:  Coverability graph G;
        initial marking m0 ∈ M,
           M - set of generalized markings;
        initial marking of unbounded places upm0;
        set of transitions T
Output: List of all cyclic runs L
L = initEmptyList()
root = Node(marking: m0, unboundedPlaces: upm0,
```

```
         transition: null, parentNode: null)
path = root
isBacktracked = false
while path is not null:
   if containsCyclicRun(path):
      cyclicRun = extractCyclicRun(path)
      if containsAllTransitions(cyclicRun,T):
         addRun(L,cyclicRun)
         path = parentNode(path)
         isBacktracked = true
         continue
   // no cyclic run was detected
   nodeM = marking(path)
   nodeUPM = unboundedPlaces(path)
   if not isBacktracked:
      incrementPhase(nodeM)
   v = Vertex(label: label(nodeM))
   incidentFrom = listIncidentFrom(G,v)
   isBacktracked = true
   for each (e,u) in incidentFrom:
      tForE = transitionForEdge(e)
      mForU = markingForVertex(u)
      // check if transition tForE from
      //     marking m to marking mForU is not
      //     in the sequence represented by path
      if isNotInSequence(path,tForE,mForU):
         // check if e was not followed already
         // in current phase of nodeM
         if wasNotFollowedInPhase(e,phase(nodeM)):
         // calculate marking of unbounded places
         upmn = fireForUnbounded(nodeUPM,tForE)
         if for every marking in upmn marking >= 0:
            path = Node(marking: mForU,
                        unboundedPlaces: upmn,
                        transition: tForE,
                        parentNode: path)
            markAsFollowedInPhase(e,phase(nodeM))
            isBacktracked = false
   // current path can not be extended; backtracking
   if isBackTracked:
      for each (e,u) in incidentFrom:
         if wasFollowedInPhase(e,phase(nodeM)):
            unmarkAsFollowedInPhase(e,phase(nodeM))
      decrementPhase(nodeM)
      path = parentNode(path)
```

Listing 2. Pseudocode of the backtracking-based procedure for cyclic runs search

A comment on the pseudocode in Listing 2 should be provided. Since it is possible to get into a vertex in multiple different ways, and every time each edge incident from it and not in path should be checked, we have introduced the notion of phase to take account of visited edges and prevent an infinite traversal.

4.2 Priorities computation

The rest of the algorithm is based on the construction of the spine tree of a Petri net and its traversal. The procedure of constructing the spine tree from the list of cyclic runs is quite straightforward and we omit its description here. Similarly, the algorithm for the construction of the spine-based coverability tree was described in details in [1] and we have mostly followed this description in the

process of implementation design. In the spine-based coverability tree the red leaves (ω-leaves) are those nodes which strictly cover some markings preceding them in their branches. To guarantee boundedness they should be cut off. This is achieved with priorities. Listing 3 demonstrates the pseudocode for computing priority relation on the basis of the spine-based coverability tree.

It was proved in [1] that if the relation $\ll$ constructed for the live and unbounded Petri net (N, m_0) is a partial order (i.e. antisymmetric), then $\ll$ is a priority relation for N, and the Petri net $(N, \ll, m_0)$ is live and bounded. However, it is possible for the algorithm in Listing 3 to produce relations with contradictory pairs, thus violating the antisymmetric property. In [2] it is suggested to remove such contradictory pairs from the relation. In this case the Petri net with the resulting priority relation should be checked for boundedness and liveness.

The concluding step of the algorithm implementation is computation of priority labels of transitions. For transitions which are not included in the priority relation the highest priority is assigned since this means that the order of their occurrence is not important. For the rest of transitions a topological sorting is used to order the transitions in the ascending priority and assign a label to each of them with respect to their position. This is possible because the priority relation can be represented as a directed acyclic graph. It should be noted that the obtained priorities can be stronger than it is required since topological sorting does not take into account relative independence of transitions in the priority relation.

```
procedure computePR(treeRoot)
Input:  Root of spine-based coverability tree treeRoot
Output: Priority relation PR = {(t1,t2) : t1 ∈ T ∧ t2 ∈ T},
          T - set of transitions
PR = initEmptyRelation()
Q = makequeue(treeRoot)
while Q is not empty:
   node = dequeue(Q)
   childNodes = children(node)
   redNodes = filter(childNodes, predicate: isRed())
   greenNodes = filter(childNodes, predicate: isGreen())
   yellowNodes = filter(childNodes, predicate: isYellow())
   enqueueAll(Q,yellowNodes)
   for each rn in redNodes:
      // get transition represented by incoming arc of node
      tr = transitionOfIncArc(rn)
      for each yn in yellowNodes:
         ty = transitionOfIncArc(yn)
         addToRelation(PR,(ty,tr))
      for each gn in greenNodes:
         tg = transitionOfIncArc(gn)
         addToRelation(PR,(tg,tr))
```

Listing 3. Pseudocode of the algorithm for priority relation computation

5. Conclusion

In this paper, we have presented our approach to implementation design of the algorithm for computing transition priorities for live Petri nets. In the worst case the performance of the algorithm may be not optimal for the task since it is based on construction and traversal of the coverability graph which can grow exponentially. However, it may prove optimal for the majority of real-life system models. We have proposed parallelization as the main method of handling the complexity. A number of experiments were conducted to estimate its effect on the performance of the implementation. While for some steps this approach proved to be viable, for others its applicability was questioned.

Priority constraints can be helpful in analysis of technical systems, since they ensure liveness and boundedness of such systems. For the purposes of computing priorities the application was developed. This application is based on the algorithm implementation presented in the paper and can be used for further studies on the problem.

However, it should be noted that the application inherits the weak points of the algorithm it is based on, i.e. the high time complexity. Hence, further experiments should be conducted to determine the limits of applicability of the application and the algorithm in particular.

References

[1]. Lomazova I.A., Popova-Zeugmann L. Controlling Petri Net Behavior using Priorities for Transitions. Fundamenta Informaticae, vol. 143, no. 1-2, 2016, pp. 101-112.

[2]. Lomazova I.A., Popova-Zeugmann L., Bartels A. Controlling Boundedness for Live Petri Nets. In Proc. of the 4th International Conference on Control, Decision and Information Technologies, 2017, pp. 236-241.

[3]. Desel J. On Cyclic Behaviour of Unbounded Petri Nets. Application of Concurrency to System Design (ACSD). In Proc. of the 13th International Conference on Application of Concurrency to System Design, 2013, pp. 110-119.

[4]. Reisig W. Understanding Petri Nets. Springer-Verlag Berlin Heidelberg, 2013, 230 p.

[5]. Cormen T., Leiserson C., Rivest R., Stein C. Introduction to Algorithms. The MIT Press, 2009, 1312 p.

[6]. Goetz B., Peierls T., Bloch J., Bowbeer J., Holmes D., Lea D. Java Concurrency in Practice. Addison-Wesley Professional, 2006, 432 p.

Информация об авторах / Information about authors

Кирилл Геннадьевич СЕРЕБРЕННИКОВ в 2019 г. получил степень магистра в области программной инженерии в НИУ ВШЭ, Москва. В число научных интересов входят распределенные системы, формальная верификация распределенных алгоритмов, параллельное программирование.

Kirill Gennadievitch SEREBRENNIKOV received his master degree in software engineering from HSE Moscow in 2019. His research interests include distributed systems, formal verification of distributed algorithms, concurrent programming.

DOI: 10.15514/ISPRAS-2019-31(4)-12

FSM abstraction based method for deriving test suites with guaranteed fault coverage against nondeterministic Finite State Machines with timed guards and timeouts

[1] *A.S. Tvardovskii, ORCID: 0000-0001-7705-7214 <tvardal@mail.ru>*
[2,3] *N.V. Yevtushenko, ORCID: 0000-0002-4006-1161 <evtushenko@ispras.ru>*
[1] *National Research Tomsk State University,*
36, Lenin Ave., Tomsk, Russia 634050
[2] *Ivannikov Institute for System Programming of the Russian Academy of Sciences,*
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia
[3] *National Research University Higher School of Economics,*
3, Kochnovskiy Proezd, Moscow, 101000, Russia

Abstract. Finite State Machine (FSM) based approaches are widely used for deriving tests with guaranteed fault coverage for discrete event systems and as the behavior of many nowadays information and control systems depends on time, classical FSMs are extended by clock variables. Moreover, optionality in the real system's specifications motivates the studying test derivation against models with the nondeterministic behavior. In this paper, we adapt classical FSM based test derivation methods for nondeterministic FSMs with timed guards and timeouts (TFSMs). We show that unlike classical FSM conformance relation, the check cannot be reduced to checking the correspondence between TFSMs transitions and this violates the main principle of FSM based test derivation methods. Respectively, a proposed approach and the appropriate fault model are based on the FSM abstraction of the given TFSM specification that is used to adequately describe the behavior of a TFSM. The fault domain contains TFSMs with the known upper boundary on the number of FSM abstraction states and allows to avoid explicit enumeration of implementations under test. We study properties of the FSM abstraction for a nondeterministic TFSM and justify that the use of an FSM abstraction allows to adapt classical FSM based test derivation methods when deriving tests with guaranteed fault coverage for TFSMs. A method is proposed for deriving a complete test suite for a complete possibly nondeterministic TFSM when an implementation under test is a deterministic complete TFSM.

Keywords: Finite State Machine; timeout; timed guard; nondeterministic Timed Finite State Machine; test derivation; fault coverage

For citation: Tvardovskii A.S., Yevtushenko N.V. FSM abstraction based method for deriving test suites with guaranteed fault coverage against nondeterministic Finite State Machines with timed guards and timeouts. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 4, 2019. pp. 175-188. DOI: 10.15514/ISPRAS-2019-31(4)-12

Acknowledgments. This work is partly supported by RFBR project № 19-07-00327/19.

Синтез тестов с гарантированной полнотой для недетерминированных автоматов с таймаутами и временными ограничениями на основе конечно автоматных абстракций

[1] *А.С. Твардовский, ORCID: 0000-0001-7705-7214 <tvardal@mail.ru>*
[2,3] *Н.В. Евтушенко, ORCID: 0000-0002-4006-1161 <evtushenko@ispras.ru>*
[1] *НИУ Томский Государственный университет,*
634050, Томск, пр. Ленина, 36
[2] *Институт системного программирования им. В.П. Иванникова РАН,*
109004, Россия, г. Москва, ул. А. Солженицына, д. 25
[3] *Национальный исследовательский университет «Высшая школа экономики»*
101000, Россия, Москва, Кочновский проезд, 3

Аннотация. Конечно-автоматные методы широко используются при синтезе проверяющих тестов с гарантированной полнотой для дискретных систем. Поскольку поведение современных информационных и управляющих систем часто зависит от времени, классическая модель конечного автомата расширяется введением временных переменных. Более того, опциональность в спецификациях реальных систем побуждает к исследованиям в области синтеза тестов для недетерминированных автоматов. В настоящей работе мы адаптируем классические конечно автоматные методы синтеза тестов к недетерминированным автоматам с временными ограничениями и таймаутами (временным автоматам). Показывается, что в отличие от классических конечных автоматов, проверка отношений конформности между временными автоматами не может быть сведена к проверке соответствия между переходами, что нарушает основной принцип конечно автоматных методов синтеза тестов. Соответственно, предложенный подход и модель неисправности основаны на конечно автоматной абстракции автомата-спецификации, которая используется для описания поведения временного автомата. Область неисправности содержит временные автоматы с известной верхней границей числа состояний конечно автоматных абстракций и позволяет избежать явного перечисления множества тестируемых реализаций. Мы исследуем свойства конечно автоматных абстракций недетерминированных временных автоматов и показываем, что использование такой абстракции позволяет адаптировать классические методы к синтезу тестов с гарантированной полнотой для временных автоматов. Предложенный метод синтеза тестов позволяет строить полные проверяющие тесты для полностью определённых возможно недетерминированных автоматов с таймаутами и временными ограничениями для тестирования реализаций, поведение которых описывается детерминированными временными автоматами.

Ключевые слова: конечный автомат; таймаут; временные ограничения; недетерминированные временные автоматы; синтез тестов с гарантированной полнотой

Для цитирования: Твардовский А.С., Евтушенко Н.В. Синтез тестов с гарантированной полнотой для недетерминированных автоматов с таймаутами и временными ограничениями на основе конечно автоматных абстракций. Труды ИСП РАН, том 31, вып. 4, 2019 г., стр. 175-188 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(4)-12

Благодарности. Работа частично поддержана проектом РФФИ № 19-07-00327/19.

1. Introduction

Finite State Machines (FSMs) are widely used for analysis and synthesis of discrete event systems [1]. In particular, FSM based approaches can be effectively used when deriving test sequences for determining whether a given implementation considered as a «black box» conforms to its specification. A number of methods exist for deriving complete test suites with respect to various fault models [see, for example, 2-5] without the explicit enumeration of possible FSMs under test. Well-known W-method [2] and many its derivatives have been developed including those for FSMs with the nondeterministic behavior [4, 6]. In many papers, researchers consider the case when the specification is a nondeterministic FSM, while an implementation FSM is deterministic and conforms to the specification if the implementation behavior is contained in that of the specification

[6, 7]. In other words, the specification nondeterminism occurs according to the optionality of the informal requirements' description and the behavior of a conforming implementation must not violate the specification.

Nowadays time aspects become very important when describing the behavior of digital and hybrid control systems, and, respectively, similar to automata [8] classical FSMs were extended with clock variables [5, 9-14]. When the behavior of a system under test is described by a Timed Finite State Machine (TFSM), classical FSM-based methods have to be modified and extensions of the W-based methods are considered in the context of systems with timed constraints [9], [14]. In [11], Merayo et al. consider timed possibly nondeterministic FSMs where time elapsed when an output has to be produced after an input has been applied to an FSM under test is limited. The model also takes into account input timeouts at states. However, the authors do not consider test derivation; yet establish a number of conformance relations. El-Fakih et al. [10] consider the test derivation and assessment for FSMs with timed guards; such an FSM has a single clock that is reset at every transition. In the paper by Zhigulin et al. [13], a method is proposed for deriving complete test suites for FSMs with timeouts. The authors consider a traditional fault domain assuming that the number of states of an implementation TFSM (Implementation Under Test) does not exceed that of the state reduced specification TFSM as well as the maximal finite timeout of the IUT does not exceed that of the specification. However, as we further show, two reduced TFSMs with timeouts can be equivalent but not isomorphic and this fact violates the main idea of W-based methods of checking the isomorphism or homomorphism between the specification and implementation under test. In [12], the authors show that the behavior of a deterministic TFSM can be adequately described by its FSM abstraction and this is a hint that a fault model can be derived based on such abstraction for which well elaborated FSM based methods for deriving tests with guaranteed fault coverage can be applied. Such a fault model is considered in [15] for deriving a complete test suite against deterministic TFSMs.

In this paper, we consider FSMs with timed guards, timeouts and output delays (TFSM) which generalize the TFSM model that has only timed guards or only input timeouts [12]. Moreover, in our case, a TFSM can be nondeterministic. Timed guards describe the system behavior depending on a time instance when an input is applied. If no input is applied until an (input) timeout expires then the system can spontaneously move to another state. An output delay describes a time for processing a given transition.

We propose a method for deriving a test suite with guaranteed fault coverage against a complete possibly nondeterministic specification FSM with timed guards, input timeouts and output delays with respect to the reduction relation assuming that an implementation TFSM under test is complete and deterministic. The fault model and a procedure for deriving a complete test suite are based on the FSM abstraction of a given TFSM specification since according to [12], the behavior of a TFSM can be adequately described by its corresponding (untimed) FSM abstraction.

The structure of the paper is as follows. Section 2 contains the preliminaries for classical and timed FSMs. It also contains the explanation how the behavior of a TFSM can be described using an appropriate FSM abstraction. In Section 3, a brief sketch of related work on test derivation methods for nondeterministic FSMs with respect to the reduction relation is presented while Section 4 contains such a review on test derivation against Timed FSMs. In Section 5, a method is proposed for deriving a complete test suite against a nondeterministic FSM with timed guards and timeouts by determining an appropriate fault model based on their FSM abstractions; the section also contains an example for a test derivation procedure. Section 6 concludes the paper.

2. Preliminaries

This section contains basic definitions of classical Finite State Machines as well as of Timed Finite State Machines as their extension. We also show how the behavior of a TFSM can be adequately described by the corresponding FSM and establish some useful properties of such FSM abstractions.

2.1 Finite State Machines

A Finite State Machine (FSM) [1] describes the behavior of a system that moves from state to state under input stimuli and produces predefined output responses. Formally, an initialized FSM is a 5-tuple $\mathsf{S} = (S, I, O, h_{\mathsf{S}}, s_0)$ where S is a finite non-empty set of states with the designated initial state s_0, I and O are input and output alphabets, and $h_{\mathsf{S}} \subseteq (S \times I \times O \times S)$ is the transition (behavior) relation. A transition (s, i, o, s') describes the situation when an input i is applied to S at the current state s. In this case, the FSM moves to state s' and produces the output (response) o. FSM S is *nondeterministic* [6] if for some pair $(s, i) \in S \times I$, there exist several pairs $(o, s') \in O \times S$ such that $(s, i, o, s') \in h_{\mathsf{S}}$; otherwise, the FSM is *deterministic*. FSM S is *complete* [6] if for each pair $(s, i) \in S \times I$ there exists $(o, s') \in O \times S$ such that $(s, i, o, s') \in h_{\mathsf{S}}$; otherwise, the FSM is *partial*. FSM S is *observable* if for every two transitions $(s, i, o, s_1), (s, i, o, s_2) \in h_{\mathsf{S}}$ it holds that $s_1 = s_2$. In the following, we consider complete observable possibly nondeterministic FSM specifications, while an implementation is a complete deterministic FSM.

A trace or an *Input/Output sequence* α/γ, written often as an I/O sequence, of the FSM S at state s is a sequence of consecutive input/output pairs starting at the state s. Given a trace α/γ, α is the *input projection* of the trace (input sequence) while γ is the corresponding *output projection* (output sequence), i.e., a possible output response of the FSM when the sequence α is applied at state s. Given a complete nondeterministic FSM, there can exist several output responses for an input sequence at a given state. A complete nondeterministic FSM is reduced if for every two different states, the sets of traces do not coincide. The unique reduced form exists for any complete nondeterministic FSM and can be derived similar to that for complete deterministic FSMs [16].

Given states s and p of complete FSMs S and P, state p is a *reduction* of s (written, $p \leq s$) if the set of I/O sequences of FSM P at state p is contained in the set of I/O sequences of FSM S at state s. FSM P is a *reduction* of FSM S if the reduction relation holds between the initial states of these machines.

2.2 Timed Finite State Machines

A *Timed FSM* (TFSM) is extended with a clock variable, timed guards, timeouts and output delays [12, 13]. The timed guards at a state have less time upper bounds than the timeout at the state and describe the behavior at a given state for inputs which arrive at different time instances. The clock variable accumulates time and is reset to zero when applying an input, producing an output and moving between states by timeout transitions. Correspondingly, an initialized TFSM is a 6-tuple $\mathsf{S} = (I, S, O, h_{\mathsf{S}}, \Delta_{\mathsf{S}}, s_0)$ where S is a finite non-empty set of states with the designated initial state s_0, I and O are input and output alphabets, $h_{\mathsf{S}} \subseteq S \times I \times O \times S \times \Pi \times Z$ is the *transition relation* and Δ_{S} is the timeout function. The set Π is a set of *input timed guards* and Z is the set of output delays which are non-negative integers. The *timeout function* is the function $\Delta_{\mathsf{S}}: S \to S \times (N \cup \{\infty\})$ where N is the set of positive integers: for each state this function specifies the maximum time for waiting for an input. If no input is applied until an (input) timeout expires then the system can spontaneously move to another state. By definition, for each state of TFSM exactly one timeout is specified. An input timed guard $g \in \Pi$ describes the time domain of clock variable when a transition can be executed and is given in the form of interval $<min, max>$ from $[0; T)$, where $< \in \{(, [\}, > \in \{),]\}$ and T is the input timeout at the current state. We also denote the largest finite boundary of timed guards and timeouts as B_{S}. The transition $(s, i, o, s', g, d) \in S \times I \times O \times S \times \Pi \times Z$ means that TFSM S being at state s accepts an input i applied at time $t \in g$ measured from the initial moment or from the moment when TFSM S has moved to the current state; the clock then is set to zero and S produces output o exactly after d time units and moves to state s'. Given state s of TFSM S such that $\Delta_{\mathsf{S}}(s) = (s', T)$, if no input is applied before the timeout T expires, the TFSM S moves to state s'. If $\Delta_{\mathsf{S}}(s) = (s', \infty)$ then $s' = s$, and this means that the TFSM can stay at state s indefinitely long waiting for an input.

Given TFSM S, S is a *complete* TFSM if the union of all input timed guards at any state s under every input i equals [0; T) when $\Delta_S(s) = (s', T)$. In this paper, we consider only complete TFSMs and the question about the interpretation of undefined transitions in partial machines and their augmentation is out of the scope of this paper [17].

TFSM S is a *deterministic* TFSM if for each two transitions $(s, i, o_1, s_1, g_1, d_1)$, $(s, i, o_2, s_2, g_2, d_2) \in h_S$, $s_1 \neq s_2$, $d_1 \neq d_2$ or $o_1 \neq o_2$, it holds that $g_1 \cap g_2 = \varnothing$, otherwise, TFSM S is *nondeterministic*. In this paper, we assume that the system specification is a complete observable, possibly nondeterministic TFSM while the behavior of an implementation under test (IUT) is described by a complete deterministic TFSM. In other words, the specification describes a set of possible permissible behaviors of an IUT and a conforming implementation must be one of them.

Example. Consider a TFSM S in Figure 1 with two states, one input and three outputs, where a is the initial state and $\Delta_S(a) = (b, 2)$, i.e., the timeout at state a is 2. For state b, $\Delta_S(b) = (b, \infty)$, and this loop is not shown in the figure. If input i is applied to the TFSM at state a at time instance 1 measured from the initial moment then S moves to state b producing output o_2 after one time unit. However, if no input is applied to the TFSM until time value reaches 2 then S moves to state b using a timeout transition. At state b, TFSM S can wait for an input infinitely long.

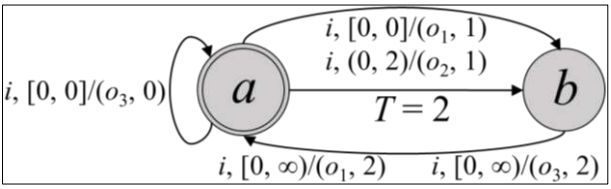


Fig. 1. Timed Finite State Machine S

A *timed input* is a pair (i, t) where $i \in I$ and t is a real; a timed input (i, t) means that input i is applied to the TFSM at time instance t measured from the initial moment or from the moment when TFSM S has produced the last output. A timed output is a pair (o, d) where $o \in O$ and d is the output delay measured from the moment when an input has been applied. In order to determine the output response of the TFSM at state s to a timed input (i, t), state s', which is reached by the TFSM by timeout transitions at time instance t, is calculated first [13]. State s' is a state where TFSM moves from state s via timeout transitions such that the maximum sum Σ of all timeouts starting from state s is less than t. At the second step, a transition (or several transitions for nondeterministic TFSM) (s', i, o, s'', g, d) such that $t - \Sigma \in g$ is considered. According to this transition, the machine produces the timed output (o, d) to a timed input (i, t) applied at state s and moves to the next state s''.

A sequence of timed inputs $\alpha = (i_1, t_1) \ldots (i_n, t_n)$ is a *timed input sequence*, a sequence of timed outputs $\gamma = (o_1, d_1) \ldots (o_n, d_n)$ is a *timed output sequence*. Given the initialized TFSM at state s_1 with the value of the clock variable equal to 0 at the initial moment and a timed input sequence $(i_1, t_1) \ldots (i_n, t_n)$, an input i_1 is applied when the value of the clock variable reaches $t_1' = t_1 - \Sigma_1$ where Σ_1 that is the maximum sum of timeouts for the sequence of timeout transitions starting from state s_1 is less than t_1, but becomes equal or bigger when adding the timeout at the current state s_1'; after applying the input at state s_1' the clock variable is set to 0 and the machine produces an output o_1 and moves to a prescribed state s_2 when clock value is equal to d_1. After producing the output o_1 the clock is reset and the machine is waiting for another input i_2 that is applied when the clock variable value equals $t_2' = t_2 - \Sigma_2$, etc. A sequence $\alpha/\gamma = (i_1, t_1)/(o_1, d_1) \ldots (i_n, t_n)/(o_n, d_n)$ of consecutive pairs of timed inputs and timed outputs starting at the state s is a *timed I/O sequence* or a *timed trace* of TFSM S at state s. Note that time of the first timed input in the sequence is counted from startup of the system at state s while time of all next inputs is counted from the time instance when a previous output has been produced. Similar to FSMs, α is an applied timed input sequence while γ is the

corresponding output response of the TFSM to sequence α of applied inputs. Given a state of a complete nondeterministic TFSM, there can exist several output responses to a timed input sequence.

Similar to FSMs, the set of all timed traces at the initial state specifies the behavior of an initialized TFSM.

Example. Consider TFSM S in fig. 1. If a timed input sequence (i. 2.5).(i, 0) is applied to S at state a then TFSM first moves to state b by the timeout transition when the clock variable value reaches 2. The clock is reset and output (o_1, 2) or (o_3, 2) is produced, the system moves back to state a and the clock is reset. When the next input (i, 0) is immediately applied, the TFSM moves either to state a with timed output (o_3, 0) or to state b with timed output (o_1, 1).

Given states s and p of complete TFSMs S and P, state p is a *reduction* of s (written, $p \leq s$) if the set of timed I/O sequences of TFSM P at state p is contained in the set of timed I/O sequences of TFSM S at state s. TFSM P is a *reduction* of TFSM S if the reduction relation holds between the initial states of the machines. For deterministic TFSMs S and P, the reductions relation is reduced to the equivalence relation.

2.3 FSM abstraction

The behavior of a TFSM can be adequately described using a classical FSM that is called the *FSM abstraction* of the TFSM and is derived similar to [12]; however, in [12], output delays are not considered.

Given a complete observable possibly nondeterministic TFSM $\mathtt{S} = (S, I, O, \lambda_\mathtt{S}, \Delta_\mathtt{S}, s_0)$, the largest finite boundary of timed guards and timeouts $B_\mathtt{S}$ and maximum output delay D, we derive the FSM abstraction of TFSM S as the FSM $\mathtt{A_S} = (S_A, I \cup \{\mathtt{I}\}, O_A, \lambda_{A\mathtt{S}}, s_0)$ where $S_A \subseteq \{(s, 0), (s, (0, 1)), \ldots, (s, (B_\mathtt{S} - 1, B_\mathtt{S})), (s, B_\mathtt{S}), (s, (B_\mathtt{S}, \infty)): s \in S\}$, $O_A = \{(o, 0), (o, 1), \ldots, (o, D): o \in O\} \cup \{\mathtt{I}\}$. The input (output) I is a special input (output) of the FSM abstraction. Given state (s, t_j), $t_j = 0, \ldots, B_\mathtt{S}$, of FSM $\mathtt{A_S}$ and input i, a transition $((s, t_j), i, (o, d), (s', 0))$ is a transition of the FSM abstraction $\mathtt{A_S}$ if and only if there exists a transition $(s, i, o, s', g_i, d) \in \lambda_\mathtt{S}$ such that $t_j \in g_i$. Given state (s, g_j), $g_j = (0, 1), \ldots, (B_\mathtt{S} - 1, B_\mathtt{S}), (B_\mathtt{S}, \infty)$, of FSM $\mathtt{A_S}$ and input i, a transition $((s, g_i), i, (o, d), (s', 0))$ is a transition of $\mathtt{A_S}$ if and only if there exists a transition $(s, i, o, s', g, d) \in \lambda_\mathtt{S}$ such that $g_i \subseteq g$. In other words, transitions under input $i \in I$ correspond to timed inputs (i, t) where t is 'hidden' as the second item of states of the FSM abstraction $\mathtt{A_S}$. Transitions under the special input I correspond to the clock change between non-integer and integer values, or to a timeout transition between states. Given state s such that $\Delta_\mathtt{S}(s) = (s', T)$, transitions $((s, n), \mathtt{I}, \mathtt{I}, (s, (n, n + 1)))$ and $((s, (n - 1, n)), \mathtt{I}, \mathtt{I}, (s, n))$ are in the transition relation $\lambda_{A\mathtt{S}}$ if and only if $n < T$. Transition $((s, (n - 1, n)), \mathtt{I}, \mathtt{I}, (s', 0)) \in \lambda_{A\mathtt{S}}$ if and only if $n = T < \infty$. In [12], it is shown that the FSM abstraction of complete and deterministic TFSM S is also complete and deterministic. In the same way, it can be shown that the FSM abstraction of a complete observable nondeterministic TFSM S is complete observable and nondeterministic.

Example. For a deterministic TFSM S in fig. 1, the corresponding FSM abstraction is shown in fig. 2. FSM abstraction $\mathtt{A_S}$ has states $(a, 0), (a, (0, 1)), (a, 1), (a, (1, 2)), (b, 0), (b, (0, \infty))$. Transitions $((a, 0), i, (o_1, 1), (b, 0))$ and $((a, 0), i, (o_3, 0), (a, 0))$ exist in FSM abstraction $\mathtt{A_S}$ since TFSM S has transitions $(a, i, o_1, b, [0, 0], 1)$ and $(a, i, o_3, a, [0, 0], 0)$. FSM abstraction $\mathtt{A_S}$ has transition $((a, (0, 1)), i, (o_2, 1), (b, 0))$ since TFSM S has transition $(a, i, o_2, b, (0, 2), 1)$. Transition $((a, 0), \mathtt{I}, \mathtt{I}, (a, (0, 1)))$ of $\mathtt{A_S}$ corresponds to clock change at state a from time instance 0 to the interval (0, 1).

A timed input sequence α of TFSM S can be transformed into a corresponding input sequence α_{FSM} of the FSM abstraction $\mathtt{A_S}$. In this case, each timed input (i, t) is replaced by sequence I.I … I.i of inputs of the FSM abstraction where the number of inputs I equals the number of clock transitions between a non-integer and integer values for the time duration t. At the same time the response of the FSM abstraction to sequence I.I.….I.i equals I.I.….I.(o, d), where the number of inputs

I is the same as for the timed input (*i*, *t*) and (*o*, *d*) is the response of the TFSM to timed input (*i*, *t*). Thus, the output sequence of the FSM abstraction γ_{FSM} can be transformed into corresponding timed output sequence γ by removing all outputs I. The following statement can be established.

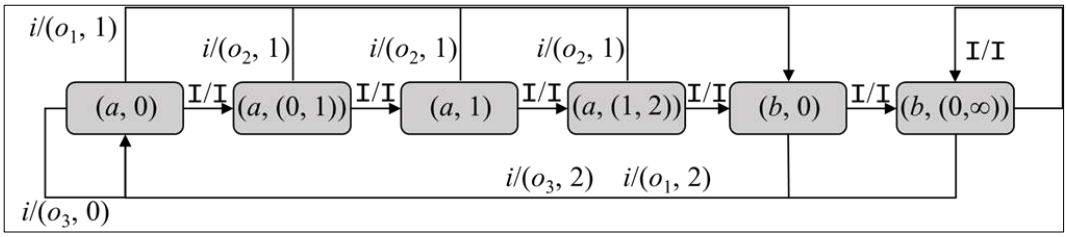


Fig. 2. FSM abstraction A_S of TFSM S in fig. 1

Proposition 1. A timed trace α/γ exists for TFSM S if and only if there exists a trace $\alpha_{FSM}/\gamma_{FSM}$ for the FSM abstraction A_S.

Proposition 2. There exists a timed trace α/γ at state *s* of a possibly nondeterministic TFSM S if and only if the FSM abstraction A_S has a trace $\alpha_{FSM}/\gamma_{FSM}$ at state (*s*, 0).

Indeed, all the transitions under input I are deterministic and correspond to the clock change between integer and non-integer value and equal to increasing of clock variable while transitions at state (*a*, *g*) of abstraction under another input *i* corresponds to transitions of TFSM at state *a* at time (or timed interval) *g*.

Example. Consider TFSM S in fig. 1 and its FSM abstraction in fig. 2. Timed trace (*i*, 2.5)/(o_1, 2).(*i*, 0)/(o_3, 0) of TFSM S corresponds to trace I/I.I/I.I/I.I/I.I/I.*i*/(o_1, 2).*i*/(o_3, 0) of FSM abstraction A_S, and vice versa.

According to Proposition 2, all the trace features of a TFSM are preserved for its FSM abstraction and thus, the set of reductions of a TFSM can be analyzed based on a set of reductions of a classical FSM. The following statement establishes necessary and sufficient conditions for two TFSM states to be in the reduction relation.

Proposition 3. State *s* of TFSM S is a reduction of state *p* of TFSM P if and only if state (*s*, 0) of the FSM abstraction A_S is a reduction of state (*p*, 0) of FSM A_P.

Thus, the conclusion about the reduction relation between two TFSMs can be drawn based on their FSM abstractions and there exist methods for checking the reduction relation between two FSM states or between two FSMs.

3. Fault models and test suites

FSM based testing can be preset and adaptive. We first consider the preset testing where *test cases* which are (timed) input sequences, are derived from the given TFSM specification to determine whether a given IUT, which is assumed to have the FSM behavior, conforms to the given specification.

In this paper, an implementation FSM P conforms to the specification if FSM P is a reduction of the specification FSM. In other words, an implementation FSM P conforms to the specification FSM if for each input sequence the output response of the FSM P is contained in the set of output responses of the specification FSM to this input sequence. In this case, the *fault model* $FM_m^{FSM} = <S, \leq, \Im_m>$ is considered where S is the specification that is a complete observable possibly nondeterministic FSM, $\leq$ is the reduction relation, $\Im_m$ is the fault domain which contains each deterministic complete FSM with at most *m* states over the same input alphabet as the specification. Here we notice that differently from the paper [18] where only deterministic FSMs are considered, the specification can be nondeterministic and the conformance relation is not the equivalence but the reduction relation.

Correspondingly, different transfer and separating sequences have to be used when deriving a test suite with guaranteed fault coverage.

A test suite is *complete* with respect to the $FM_m^{FSM} = <\mathsf{S}, \leq, \Im_m>$ if for each FSM $\mathsf{P} \in \Im_m$ that is not a reduction of S, the test suite has a sequence for which an output response of P is not in the set of output responses of S to this sequence.

A complete test suite with respect to FM_m^{FSM} can be derived using an appropriate modification of FSM based methods for nondeterministic FSMs [6] which are based on *deterministically-transfer* (*d-transfer*) and separating sequences. A state s is *deterministically reachable* (*d-reachable*) from the initial state of the FSM S if there exists an input sequence α such that for any output response β to α, the machine S moves from the initial state to state s when α is applied. In this case, α is a *d-transfer* sequence for state s. States s_1 and s_2 of an FSM S are *separable* if there exists an input sequence α such that the sets of output responses of the FSM at states s_1 and s_2 to α do not intersect; in this case, sequence α is called a *separating* sequence for states s_1 and s_2. If a sequence separates each pair of different states of the FSM S then this sequence is a separating sequence for FSM S. Once again we remind that differently from [18], not each input sequence is a *d*-transfer of the nondeterministic specification and separable states and separating sequences for the nondeterministic specification are defined in a different way.

If FSM S has a separating sequence γ and each state is *d*-reachable from the initial state, the procedure for deriving a complete test suite w.r.t. the fault model $FM_n^{FSM} = <\mathsf{S}, \leq, \Im_n>$ where n is the number of states of S, has the following steps:

1. A *d-cover* set of the FSM S is derived. This set contains a *d*-transfer sequence for each state of the FSM S.
2. Each sequence of the *d*-cover set is appended with the separating sequence γ of the FSM S and every input that also is appended with the separating sequence γ.

If an adaptive test suite is derived, an adaptive distinguishing sequence can be used instead of a separating sequence while *d*-transfer sequences can be replaced by adaptive transfer sequences (if they exist) [19]. Adaptive distinguishing (separating) and *d*-transfer sequences can be shorter then preset, and moreover, they exist more often.

An input sequence α is *adaptive* if the next input depends on the outputs of the FSM. Such an input sequence can be represented by an FSM called a *test case* [19]. At each state of a test case, either there are transitions for one input with all outputs or there are no transitions and in the latter case, a state is called *terminal*. Given a test case (TC) D for FSM S, an adaptive sequence specified by is applied in the following way. If input i_1 is defined at the initial state d_0 of D then first the input i_1 is applied to FSM S and TC D moves to the i_1o-successor d_1 of state d_0 if o is the output the response of S to the input i_1. The next input to apply is the input defined at state d_1, etc. The procedure terminates when a terminal state is reached.

A test case represents an *adaptive separating* sequence for states s_1 and s_2 of the FSM S if each input-output sequence from the initial to the terminal state of the test case can happen at most at one of states s_1 or s_2. In the former case, the state s_1 is identified, while in the latter case it will be state s_2. States s_1 and s_2 of the FSM S are *adaptively separable* if there is a test case that represents an adaptive separating sequence for states s_1 and s_2. In this case, the corresponding trace from the initial state to a terminal state of an adaptive separating test case allows to determine what was a state of the FSM S before the experiment.

If an adaptive sequence separates each pair of different states of the FSM S, then such a sequence is an *adaptive separating sequence* for the FSM S.

A test case can also represent an adaptive sequence from the initial state of the FSM S to the state s if each input-output sequence of the test case from the initial to a terminal state is ended at state s [19, 20]. In this case, state s is *adaptively reachable* from the initial state. The derivation of a complete adaptive test suite is almost the same as the preset: the only difference is that adaptive

distinguishing sequences are used instead of separating sequences and adaptive transfer sequences are used instead of *d*-transfer sequences.

If FSM S has no (adaptive) separating sequence or S has states which are not *d*-reachable from the initial state then a complete test suite cannot be derived using the above procedure. In this case, the so-called state counting reduction (SCR) method should be applied [6].

Below, we describe the main steps of the general SCR-method when deriving a complete preset test suite with respect to the fault model $FM_m^{FSM} = <\mathsf{S}, \leq, \Im_m>$.

1. Determine subset S_d of all *d*-reachable states and derive *d-cover* of the FSM S which contains a *d*-transfer sequence for each state of S_d.
2. Determine the set $R = \{R_1, R_2, \ldots, R_p\}$ of maximal subsets of pairwise separable states; for each subset $R_j \in R$, denote R_{jd} a subset of all *d*-reachable states of R_j. For each subset $R_j \in R$, derive a distinguishing set W_j that contains a separating sequence for each pair of different states of R_j.
3. For each state s_k of S_d, derive a set of input sequences N_k: an input sequence $\alpha \in N_k$ if for each *I/O* sequence α/β at state s_k, it holds that α/β traverses states of some $R_j \in R$ at least $m - |R_{jd}| + 1$ times and this does not hold for any proper prefix of α. Concatenate each prefix of sequence α with each sequence of the set W_j.
4. Concatenate each *d*-transfer sequence with each sequence of each set W_j that was used at Step 3 when terminating an input sequence of the set N_k, $k = 1, \ldots, p$.

Here we notice that in general case, complete test suites derived by SCR method are much longer than for the case when the specification FSM has a separating sequence and the derivation method is much more complex. To minimize our efforts for deriving a complete test suite with respect to the fault model $FM_m^{FSM} = <\mathsf{S}, \leq, \Im_m>$, the adaptive testing can be used instead of the preset [19].

It is known that a test suite can be usually shorter if the specification FSM has a sequence, which separates every two states [6]. In this case, set W_j contains only one separating sequences α and $R = \{S\}$. However, such a separating sequence does not always exist and thus, we are obliged to use a set of separating sequences for test derivation. Adaptive distinguishing (separating) sequences exist more often than the preset and are usually shorter, thus, adaptive distinguishing sequences can be preferable for test derivation. Anyway, using adaptive distinguishing sequences can increase the size of subsets of pairwise distinguishable states from *R*, and thus, shorten sets W_j and the sets N_k, and correspondingly, minimize a complete test suite.

In the next section, we consider an existing approach for adaptation classical FSM based test derivation methods for Timed FSM.

4. Related work on TFSM based testing

The problem of deriving a complete test suite against a nondeterministic FSM with timed guards with respect to the reduction relation has been considered in [20]. The proposed approach is based on the FSM abstraction of TFSM but that abstraction is a bit different from that considered in the «Preliminaries» section. In that case, one-to-one mapping between sets of states of TFSM and corresponding FSM abstraction has been established. The latter allows to inherit the above described steps for deriving a complete test with respect to the fault domain which contains each deterministic complete TFSMs with timed guards with at most *m* states over the same input alphabet as the specification TFSM S and the largest boundary B_{S} for input timed guards. However, in general case, this approach cannot be applied for FSMs with time guards and timeouts since the one-to-one mapping between transitions of two state reduced equivalent TFSMs with timeouts not always can be established.

In [15], it is shown that initialized reduced deterministic TFSM specification and TFSM implementation with timeouts can be equivalent yet not isomorphic; moreover, they can have

different number of states. The latter violates the main assumption of W-based methods about checking the correspondence between FSM transitions. As an example, consider TFSMs in fig. 3.

Each state in R and Q is reachable from the initial state and each two different states of each machine are not equivalent, i.e., both TFSMs are connected and state reduced. By direct inspection, one can assure that equivalent machines in Figure 3 have different number of states and thus, are not isomorphic.

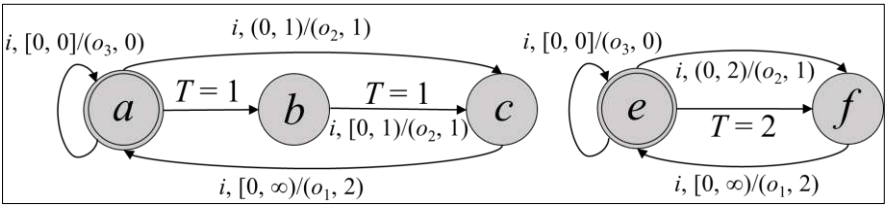


Fig. 3. Two state reduced deterministic complete TFSMs R and Q

On the other hand, according to Propositions 1-3, the necessary relationship holds between transitions of their FSM abstractions. For example, reduced forms of FSM abstractions of TFSMs R and Q (fig. 3) are isomorphic. FSM abstraction A_R and its reduced form is shown in fig. 4. Thus, in order to derive a complete test suite for deterministic TFSMs we considered the fault domain containing every TFSM P over the same input alphabet as S such that the reduced form of the FSM abstraction of P has at most $m > 1$ states [15]. A similar approach can be applied for the test derivation against nondeterministic FSMs with timeouts and timed guards; in the next section, corresponding fault model and test derivation method are proposed.

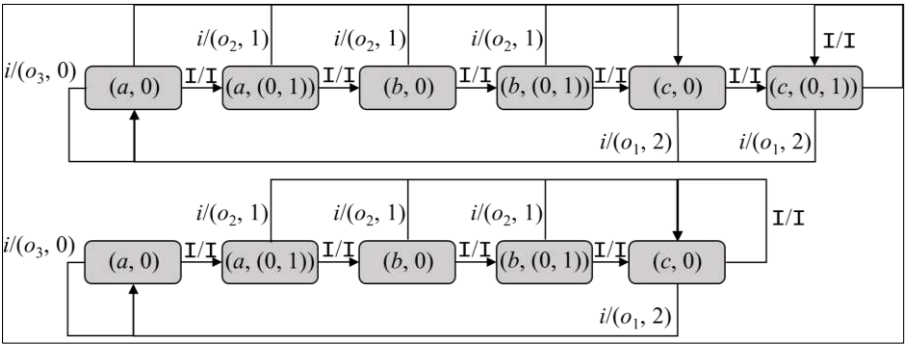


Fig. 4. The FSM abstraction A_R of TFSMs R (Figure 3) and its reduced forms

5. Test derivation method for nondeterministic FSM with timed guards and timeouts

In order to derive a test suite with guaranteed fault coverage against the nondeterministic specification TFSM, we propose a fault model based on the FSM abstraction of the TFSM and algorithm of applying the SCR-method to such abstraction.

Given a nondeterministic TFSM S with n states (fig. 1), two deterministic equivalent TFSM implementations R and Q (fig. 3) which are reductions of S can have different number of states. However, the reduced forms of their FSM abstractions are isomorphic and are reductions of FSM abstraction A_S. Another example in Figure 5 demonstrates that for nondeterministic TFSM Y there can exist a deterministic TFSM Y′ with the same number of states and the boundary B_S, such that the reduced form [16] of FSM abstraction $A_{Y'}$ has more states than that of A_Y.

Given the TFSM specification S, we consider the fault model $FM_m^{TFSM} = <S, \leq, \aleph_m>$, where S is the complete observable, possibly nondeterministic TFSM specification, $\leq$ is the reduction relation,

$\aleph_m$ is the fault domain that contains each deterministic complete TFSM P over the same input alphabet as the specification such that the reduced form of its FSM abstraction A_P has at most $m > 1$ states.

As it is demonstrated below it can well happen that some timed FSMs with less states than the specification TFSM are not included into the fault domain and vice versa, a number of timed FSMs which have more states than the specification TFSM are included into the fault domain.

Example. Consider TFSM specification S (fig. 1) with two states. Fault domain $\aleph_m$ of the fault model $FM_m^{TFSM} = <S, \leq, \aleph_m>$ contains TFSM R (fig. 3) with three states since the FSM abstraction A_R has not more states than the FSM abstraction A_S. At the same time, in Figure 5 the TFSM specification Y and its non-conforming implementation Y′ are shown such that both TFSMs have three states and the finite timed guards' boundary equal two. However, the fault domain $\aleph_m$ does not contain Y′ since the reduced form of its FSM abstraction has more states than A_Y. Thus, it can happen that nonconforming implementations with the same number of states as the specification TFSM can pass a complete test suite with respect to $<S, \leq, \aleph_m>$.

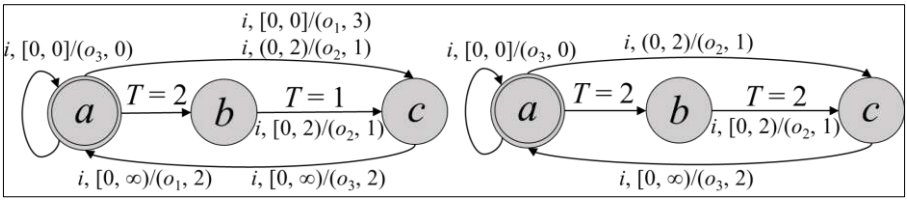


Fig. 5. TFSM Y and its non-conforming implementation Y′

Note that the FSM abstraction of TFSM S can have non-separable states, i.e., the FSM abstraction can have a pair of states for which a separating sequence does not exist when the specification TFSM S has a separating sequence, i.e., all states of the TFSM S are pairwise separable. For example, TFSM S in Figure 1 has a separating sequence (i, 1) while the corresponding FSM abstraction A_S has a pair of non-separable states (b, 0) and (b, (0, ∞)), for which the sets of input/output sequences coincide. In order to derive a complete test suite for such FSM, the SCR method can be used.

As mentioned above, similar to a deterministic FSM abstraction [15], a nondeterministic FSM abstraction can be minimized using the method from [16]. As an example, for FSM abstraction A_S (fig. 2) of TFSM S (Figure 1), equivalent states (b, 0) and (b, (0, ∞)) can be merged into one state. However, unlike deterministic machines, such optimization does not always allow to merge pairs of non-separable states of the FSM abstraction of the specification and thus, the SCR method is still used for test derivation.

Algorithm for deriving a complete test suite with respect to the fault model $FM_m^{TFSM} = <S, \leq, \aleph_m>$ where m is the number of states of the reduced form of the FSM abstraction of S

Input: The complete observable possibly nondeterministic specification TFSM S

Output: A complete test suite TS with respect to the fault model $FM_m^{TFSM} = <S, \leq, \aleph_m>$, where $\aleph_m$ contains every TFSM P over the same input alphabet as S such that the reduced form of the FSM abstraction of P has at most $m > 1$ states

Step 1. Derive the reduced form of the FSM abstraction A_S of TFSM S.

Step 2. Derive a test suite TS_A with respect to the fault model $FM_m^{FSM} = <A_S, \leq, \Im_m>$ using the SCR-method described above, where m is number of states of the FSM abstraction A_S.

Step 3. According to Proposition 1, transform sequences of the test suite TS_A into corresponding timed sequences over the TFSM S and obtain the test suite TS.

Proposition 4. The test suite TS returned by the above Algorithm is complete with respect to the fault model $FM_m^{TFSM} = <S, \leq, \aleph_m>$.

Proof. Let a test suite *TS* be returned by the above algorithm and TFSM P which is not a reduction of specification TFSM S is in the set $\aleph_m$. By definition of the fault domain $\aleph_m$, the reduced form of the FSM abstraction A_P has at most *m* states. Since P is not a reduction of S, the FSM A_P is not a reduction of A_S (Proposition 3). Thus, a test suite TS_A derived at Step 2 contains an input sequence α_{FSM}, which separates FSMs A_P and A_S. By Proposition 2, for sequence α_{FSM} of the FSM A_P there exists the corresponding timed input sequence α of the TFSM P that will demonstrate that P is not a reduction of the TFSM S. The latter guarantees that each non-conforming implementation P of the set $\aleph_m$ is detected by the test suite *TS*.

The fault domain in the above algorithm can be extended and for TFSM S with the reduced form of its FSM abstraction A_S which has *n* states, a complete test suite can be derived by SCR-method with respect to $\aleph_m$ when $m > n$. However, in this case length of a complete test suite significantly increases [21].

In the worst case, the length of a test suite derived by SCR-method exponentially depends on the number of states of FSM and this also holds for FSM with timed aspects. As experimental results show, in practice, length of adaptive distinguishing sequences is usually polynomial with respect to the number of FSM states when such a sequence exists [20, 7]. Respectively, similar results can be derived for a TFSM when proposed algorithm is used and the boundary on timed guards is not too big. Note that length of the test suite also significantly depends on timed aspects of the specification TFSM such as the upper bounds of timed guards and value of timeouts [20, 21].

We note again that the FSM abstraction of TFSM S can have non-separable states while all states of the TFSM are pairwise separable. However, we underline that the FSM abstraction inherits the *d*-reachability of states from the specification TFSM and the following proposition holds.

Proposition 5. States (*s*, 0), (*s*, (0, 1)), (*s*, 1), (*s*, (1, 2)) … of FSM abstraction A_S are *d*-reachable if and only if state *s* is *d*-reachable in TFSM S.

The statement is implied by Propositions 1-2 due to a deterministic transition under the special input I. Respectively, all states of FSM abstraction A_S are *d*-reachable if and only if all states of TFSM S is *d*-reachable.

Example. Consider TFSM S in Figure 1 and its FSM abstraction A_S in Figure 2. We derive a complete test suite with respect to the fault model $FM_6^{TFSM} = <S, \leq, \aleph_6>$. For state (*b*, 0) of A_S there exists a *d*-transfer sequence I.*i* and respectively, state *b* of TFSM S has a timed *d*-transfer sequence (*i*, 0,5). Other states of FSM abstraction are *d*-reachable from states (*a*, 0) and (*b*, 0) by a sequence of I inputs. Thus, all states of A_S are *d*-reachable from the initial state and for the FSM abstraction A_S, $S_d = \{(a, 0), (a, (0, 1)), (a, 1), (a, (1, 2)), (b, 0), (b, (0, \infty))\}$.

Given FSM A_S, we can also determine two maximal subsets of pairwise separable states $R_1 = \{(a, 0), (a, (0, 1)), (a, 1), (a, (1, 2)), (b, 0)\}$, $R_2 = \{(a, 0), (a, (0, 1)), (a, 1), (a, (1, 2)), (b, (0, \infty))\}$ and corresponding distinguishing sets $W_1 = W_2 = \{ i, I.i, I.I.i\}$. Note that $R_1 = R_{1d}$ and $R_2 = R_{2d}$ since all states of A_S are *d*-reachable.

Consider state (*b*, 0) and the set $N_{(b, 0)}$ of input sequences derived at Step 3 of the SCR-method when a test suite is derived with respect to the fault model $<S, \leq, \Im_6>$. Input/Output sequences with the input projection of the set $N_{(b, 0)}$ should traverse states of some R_j at least $2 = 6 - 5 + 1$ times while this does not hold for any proper prefix of the input sequence, and respectively, *i.i* is in the set $N_{(b, 0)}$ which traverses states (*a*, 0) and (*b*, 0) of R_1. Other sequences at state (*b*, 0) are *i*.I (traverses (*a*, 0), (*a*, (0, 1))), I.*i* (traverses (*b*, (0, ∞)), (*a*, 0)), I.I (traverses (*b*, (0, ∞)), (*b*, (0, ∞))) and thus, $N_{(b, 0)} = \{i.i, i.I, I.i, I.I, i, I\}$.

A fragment of the tree that is obtained when deriving a test suite, is shown in fig. 6. One of test sequences of TS_A is I.*i*.*i*.*i*.I.I.*i* and a corresponding timed input sequence of test *TS* is (*i*, 0,5).(*i*, 0).(*i*, 0).(*i*, 1) = γ where (*i*, 0,5) is a *d*-transfer sequence and (*i*, 1) is a separating sequence from W_1. Each sequence of the test suite is applied to TFSM implementation at the initial state. First input *i* of γ is applied when clock value is equal to 0,5; after applying the input the clock is set to 0 and the

machine produces corresponding output when clock value is equal to 1 when an implementation is conforming. After producing the output o_2 the clock is reset and the machine is waiting for the next input i from timed input (i, 0) that is immediately applied after resetting the clock. After applying this input the clock is reset again and the machine produces an output o_1 or o_3 when the clock value is equal to 2. After producing any of outputs by the TFSM the clock is reset and the machine is waiting for a next input, etc.

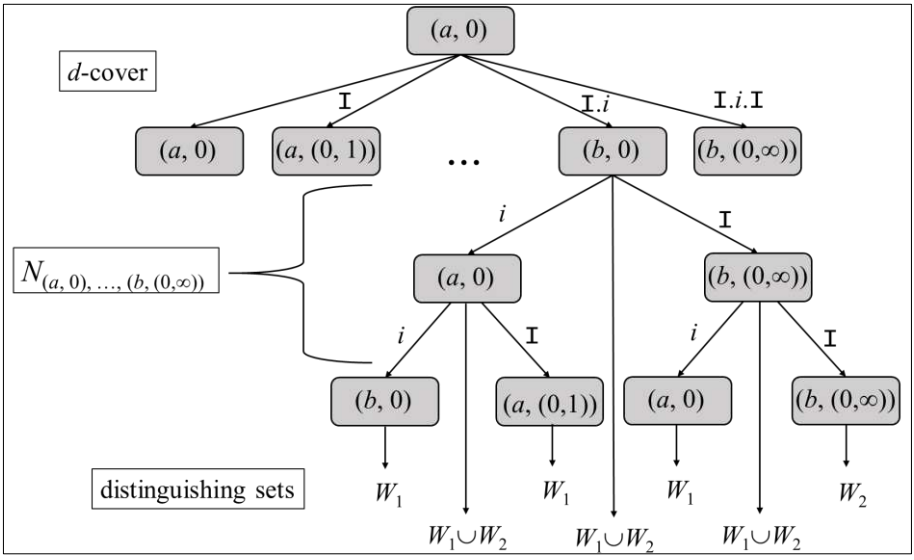


Fig. 6. A fragment of test suite TS_A for the FSM abstraction A_S

6. Conclusion

In this paper, we have proposed an approach for deriving complete test suites with respect to the reduction relation against nondeterministic Finite State Machines with timed guards and timeouts. Both, a proposed approach and a corresponding fault model are based on the FSM abstraction of machines with timed guards and timeouts and this allows inheriting the known FSM based SCR-method when deriving test suites with guaranteed fault coverage for nondeterministic TFSMs.

References / Список литературы

[1]. Gill A. Introduction to the Theory of Finite-State Machines, McGraw-Hill, 1962, 272 p. / Гилл А. Введение в теорию конечных автоматов. Наука, 1966, 272 стр.

[2]. Chow T.S. Test design modeled by finite-state machines. IEEE Transactions on Software Engineering, vol. 4, no. 3, 1978,pp. 178–187.

[3]. Dorofeeva R., El-Fakih K., Maag S., Cavalli A., Yevtushenko N. FSM-based conformance testing methods: A survey annotated with experimental evaluation. Information and Software Technology, vol. 52, issue 12, 2010, pp. 1286–1297.

[4]. Hierons R.M., Merayo M.G., Nunez M. Testing from a Stochastic Timed System with a Fault Model. Journal of Logic and Algebraic Programming, vol. 72, no. 8, 2009, pp. 98–115.

[5]. Krichen M. and Tripakis S. Conformance testing for real-time systems. Formal Methods in System Design, vol. 34, no. 3, 2009, pp. 238–304.

[6]. Petrenko A., Yevtushenko N. Conformance tests as checking experiments for partial nondeterministic FSM. Lecture Notes in Computer Science, vol. 3997, 2006, pp. 118–133.

[7]. Tvardovskii A.S., Yevtushenko N.V. Deriving adaptive distinguishing sequences for Finite State Machines. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 4, 2018, pp. 139–154 (in Russian). DOI: 10.15514/ISPRAS-2018-30(4)-9 / Твардовский А.С., Евтушенко Н.В. К синтезу адаптивных

различающих последовательностей для конечных автоматов. Труды ИСП РАН, том 30, вып. 4, 2018, стр. 139–154.
[8]. Alur R. and Dill D.L. A theory of timed automata. Theoretical Computer Science, vol. 126, issue 2, 1994, pp. 183–235.
[9]. Springintveld J., Vaandrager F., and D'Argenio P. Testing timed automata. Theoretical Computer Science, vol. 254, no. 1–2, 2001, pp. 225–257.
[10]. El-Fakih K., Yevtushenko N., and Fouchal H., Testing timed finite state machines with guaranteed fault coverage. In Proc. of the 21st IFIP WG 6.1 International Conference on Testing of Software and Communication Systems and 9th International FATES Workshop, 2009, pp. 66–80.
[11]. Merayo M.G., Nunez M., and Rodriguez I. Formal testing from timed finite state machines. Computer Networks, vol. 52, issue 2, 2008, pp. 432–460.
[12]. Bresolin D., El-Fakih K., Villa T., and Yevtushenko N. Deterministic timed finite state machines: Equivalence checking and expressive power. In Proc. of the 5th International Symposium on Games, Automata, Logics and Formal Verification (GandALF 2014), 2014, pp. 203–216.
[13]. Zhigulin M., Yevtushenko N., Maag S., Cavalli A. FSM-Based Test Derivation Strategies for Systems with Time-Outs. In Proc. of the 11th International Conference on Quality Software, 2011, pp. 141–150.
[14]. En-Nouaary A., Dssouli R., Khendek F. Timed Wp-Method: Testing Real-Time Systems. IEEE Transactions on Software Engineering, vol. 28, issue 11, 2002, pp. 1023–1038.
[15]. Tvardovskii A., El-Fakih K, Yevtushenko N. Deriving Tests with Guaranteed Fault Coverage for Finite State Machines with Timeouts. Lecture Notes in Computer Science, vol. 11146, 2018, pp. 149–154.
[16]. Starke P. Abstract Automata. North-Holland Publishing Company, 1972, 419 p.
[17]. Villa T., Kam T., Brayton R.K., Sandgiovanni-Vincentelli A. Synthesis of Finite State machines: Logic Optimization, Springer, 1997, 381 p.
[18]. Lee D., Yannakakis M. Principles and methods of testing finite state machines - a survey. Proceedings of the IEEE, vol. 84, issue 8, 1996, pp. 1090–1123.
[19]. Yevtushenko N., El-Fakih K., and Ermakov A., On-the-fly construction of adaptive checking sequences for testing deterministic implementations of nondeterministic specifications. Lecture Notes in Computer Science, vol. 9976, 2016, pp. 139–152.
[20]. Tvardovskii A.S., Gromov M.L., El-Fakih Khaled, Yevtushenko N.V. Testing Timed Nondeterministic Finite State Machines with the Guaranteed Fault Coverage. Automatic Control and Computer Sciences, vol. 51, № 7, 2017, pp. 724–730.
[21]. Tvardovskii A., Vinarskii E. Yevtushenko N. Experimental Evaluation of Timed Finite State Machine Based Test Derivation. In Proc. of the International Conference of Young Specialists on Micro/Nanotechnologies and Electron Devices, 2019, pp. 102–107.

Информация об авторах / Information about authors

Александр Твардовский получил степень магистра радиофизики в Томском государственном университете. С 2017 года обучается в аспирантуре. Исследовательские интересы включают теорию автоматов и тестирование программного обеспечения.

Aleksandr Tvardovskii received his Master's degree from Faculty of Radiophysics of Tomsk State University. He is a Ph.D. student since 2017. His research interests include automata theory and software testing.

Нина Евтушенко, доктор технических наук, профессор, главный научный сотрудник ИСП РАН, профессор НИУ ВШЭ. До 1991 года работала научным сотрудником в Сибирском физико-техническом институте. С 1991 г. работала в ТГУ профессором, зав. кафедрой, зав. лабораторией по компьютерным наукам. Её исследовательские интересы включают формальные методы, теорию автоматов, распределенные системы, протоколы и тестирование программного обеспечения.

Nina Yevtushenko, Doctor of Technical Sciences, professor, a chief researcher of ISP RAS, professor at HSE. She worked at the Siberian Scientific Institute of Physics and Technology as a researcher up to 1991. In 1991, she joined TSU as a professor and then worked as the chairhead and the head of Computer Science laboratory. Her research interests include formal methods, automata theory, distributed systems, protocol and software testing.

DOI: 10.15514/ISPRAS-2019-31(4)-13

Самотрансформация деревьев с ограниченной степенью вершин с целью минимизации или максимизации индекса Винера

И.Б. Бурдонов, ORCID: 0000-0001-9539-7853 <igor@ispras.ru>
Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. Рассматривается распределённая сеть, граф связи которой является неориентированным деревом. Предполагается, что сеть может сама менять свою топологию. Для этого предлагается предельно локальная атомарная трансформация – добавление ребра, соединяющего разные концы двух смежных ребер, и одновременное удаление одного из этих ребер. Такая трансформация выполняется по «команде» от вершины дерева, а именно, общей вершины двух смежных ребер. Показывается, что из любого дерева можно получить любое другое дерево с помощью только атомарных трансформаций. Если рассматриваются деревья с ограничением d ($d \geq 3$) на степени вершин, то трансформация не нарушает этого ограничения. В качестве примера цели такой трансформации рассматриваются задачи максимизации и минимизации индекса Винера дерева с ограниченной степенью вершин без изменения множества его вершин. Предлагаются соответствующие распределённые алгоритмы и доказываются линейные оценки их сложности.

Ключевые слова: распределенная сеть; самотрансформация графа; индекс Винера

Для цитирования: Бурдонов И.Б. Самотрансформация деревьев с ограниченной степенью вершин с целью минимизации или максимизации индекса Винера. Труды ИСП РАН, том 31, вып. 4, 2019 г., стр. 189-210. DOI: 10.15514/ISPRAS-2019-31(4)-13

Благодарности. Работа выполнена при поддержке Российского фонда фундаментальных исследований, проект 17-07-00682-а.

Self-transformation of trees with a bounded degree of vertices to minimize or maximize the Wiener index

I.B. Burdonov, ORCID: 0000-0001-9539-7853 <igor@ispras.ru>
Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. We consider a distributed network whose communication graph is a non-oriented tree. It is assumed that the network itself can change its topology. For this, an extremely local atomic transformation is proposed - the addition of an edge connecting the different ends of two adjacent edges, and the simultaneous removal of one of these edges. This transformation is performed by "command" from the vertex of the tree, namely, the common vertex of two adjacent edges. It is shown that any other tree can be obtained from any tree using only atomic transformations. If trees with degree bounded d (d>2) are considered, then the transformation does not violate this restriction. As an example of the goal of such a transformation, the tasks of maximizing and minimizing the Wiener index of a tree with a bounded degree of vertices without changing the set of its vertices are considered. Corresponding distributed algorithms are proposed and linear estimates of their complexity are proved.

Keywords: distributed network; transformation of graphs; Wiener index

For citation: Burdonov I.B. Self-transformation of trees with a bounded degree of vertices to minimize or maximize the Wiener index. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 4, 2019. pp. 189-210 (in Russian). DOI: 10.15514/ISPRAS-2019-31(4)-13

Acknowledgements. This work was supported by the Russian Foundation for Basic Research, project 17-07-00682-a.

1. Введение

В этой статье предлагаются два алгоритма самотрансформации неориентированного дерева, не нарушающие ограничения d ($d \geq 3$) на степени вершин, с целью минимизации или максимизации индекса Винера. Индекс Винера определяется как сумма расстояний между всеми парами вершин неориентированного графа. Он был предложен американским химиком Гарри Винером в 1947 г. [0], является наиболее старым из известных топологических индексов молекулярных графов, и используется во многих приложениях, особенно в математической и компьютерной химии. Мы далее предлагаем краткий обзор результатов в рассматриваемой области.

Максимальный индекс Винера, очевидно, имеет линейное дерево, содержащее две листовые вершины и путь между ними (или содержащее только одну изолированную вершину). Вид дерева с числом вершин n и ограничением d ($d \geq 3$) на степени вершин исследовался в работе [[2]], результаты которой (определение 2.1 и теорема 2.2., стр. 129) использованы в данной статье, и, специально для бинарных деревьев, в работе [[3]].

Самотрансформация дерева понимается как самотрансформация распределенной сети, топология которой и есть это дерево, динамически меняющееся в процесс трансформации. Это является частным случаем динамического графа, который формально определяется как последовательность «классических» (стационарных) графов, переход между которыми осуществляется теми или иными операциями. Изучением таких графов занимается зарождающаяся динамическая теория графов [[4], [5]], которая является теоретической основой для конструирования алгоритмов взаимодействия подвижных абонентов (вершин графа) и изучения самоорганизующихся сетей [[6]] различного физического и технического происхождения, в том числе, социальных сетей, нейронных сетей [[7]] и роевого интеллекта [[8], [9]].

К самоорганизующимся сетям относят Mesh (ячеистые), ad-hoc сети, беспроводные сенсорные сети (WSN) и др. [[6], [10], [11]]. Особенностью этих сетей является отсутствие разделения узлов на коммутаторы и хосты, а также отсутствие контроллеров, имеющих доступ ко всем или части коммутаторов и настраивающих, в частности, их таблицы маршрутизации. При исследовании таких сетей основное внимание уделяется вопросам маршрутизации, пропускной способности, помехоустойчивости, безопасности, распределения нагрузки и сетевых ресурсов, и т.п. С точки зрения самой сети изменение ее топологии является внешним фактором, который надо учитывать, но которым сама сеть не управляет или управляет лишь частично [[12], [13]]. Например, вершины графа понимаются как подвижные агенты в трехмерном пространстве, а наличие или отсутствие ребра $\{a, b\}$ определяется расстоянием в этом пространстве между агентами a и b. Когда агенты сближаются ближе некоторого порогового значения, ребро добавляется, а когда удаляются на большее расстояние, ребро удаляется. Задача заключается в корректировке информации, управляющей работой сети, в особенности маршрутизацией, при появлении или удалении вершин и ребер.

С другой стороны, в литературе много работ, посвященных как раз целенаправленной трансформации графа и, в частности, деревьев с целью оптимизации по тем или иным критериям. Одним из таких критериев как раз и является минимум или максимум индекса Винера. Такие трансформации носят глобальный или локальный характер. Примером глобальной трансформации может служить конструкция Мычельского (Mycielskian) [[14]],

когда к графу из n вершин и m ребер добавляется сразу $(n + 1)$ новых вершин и $(n + 2m)$ новых ребер.

В данной статье предлагается, напротив, предельно локальная трансформация, затрагивающая минимум вершин и ребер, максимально близких друг другу. Конкретно предлагается атомарная трансформация вида $a \to c \to b$, при которой конец c ребра $\{a, c\}$ «двигается» вдоль ребра $\{c, b\}$ к другому его концу b. Это сводится к одновременному удалению ребра $\{a, c\}$ и добавлению ребра $\{a, b\}$, а условием является наличие ребра $\{b, c\}$. Если граф является деревом, то при такой трансформации не появляются циклы, и сохраняется множество вершин.

Ранее специально для деревьев рассматривалась «элементарная» трансформация, которая также не добавляла циклов и не меняла множество вершин: добавление нового ребра $\{a, b\}$ и удаление в образовавшемся цикле $a = b_1, \ldots b_k = b, a$ одного из старых ребер $\{b_i, b_{i+1}\}$. В [[15]] показано, что любое дерево может быть получено из любого другого дерева с тем же множеством вершин с помощью конечного числа таких трансформаций (теорема 4.3, стр. 245). Эта трансформация, однако, не достаточно локальна, поскольку в ней участвуют вершины a и b и ребро $\{b_i, b_{i+1}\}$, которые могут сколь угодно далеко отстоять друг от друга. Тем не менее, эта трансформация сводится к двум цепочкам наших атомарных трансформаций $b_{i+1} \to b_i \to b_{i-1}$, $b_{i+1} \to b_{i-1} \to b_{i-2}$, …, $b_{i+1} \to b_2 \to a$ и $a \to b_{i+1} \to b_{i+2}$, $a \to b_{i+2} \to b_{i+3}$, …, $a \to b_{k-1} \to b$.

Трансформации наиболее близкие к нашей атомарной трансформации рассматривались в [[16], [17]]. В [[16]] трансформация, названная «the edge-growth transformation», применяется относительно (не висячего) ребра $\{a, b\}$ и заключается в замене всех ребер $\{b, c_1\}$, $\{b, c_2\}$, …, $\{b, c_k\}$, где $c_i \neq a$, на ребра $\{a, c_1\}$, $\{a, c_2\}$, …, $\{a, c_k\}$, после чего ребро $\{a, b\}$ становится висячим. Это эквивалентно множеству наших атомарных трансформаций вида $c_1 \to b \to a$, $c_2 \to b \to a$, …, $c_k \to b \to a$. Заметим, что эти атомарные трансформации могут выполняться одновременно, поскольку они «не мешают» друг другу: не удаляют ребро $\{a, b\}$, наличие которого является условием выполнения каждой из этих трансформаций. Трансформация из [[17]] названа «diameter-growing transformation relative to the pendent edge» и совпадает с нашей атомарной трансформацией $a \to c \to b$. Однако она применяется для специального случая, когда ребро $\{c, b\}$ висячее, а ребро $\{a, c\}$ лежит на самом длинном пути (longest path), что приводит к увеличению длины пути и, соответственно, индекса Винера.

Ключевым аспектом предлагаемых в данной статье алгоритмов является то, что это распределенные и параллельные алгоритмы, выполняемые совместно вершинами дерева. Дерево трансформирует само себя по «командам» от его вершин (точнее, от вычислительных единиц, соотносимых с вершинами). Как раз для этого нужна предельная локальность атомарной трансформации $a \to c \to b$: «команду» на ее выполнение подает вершина c, «знающая» только об инцидентных ей ребрах и указывающая два из них $\{a, c\}$ и $\{c, b\}$ как параметры «команды». Сообщения, которыми вершины могут обмениваться, пересылая их по ребрам графа, как это обычно и происходит в распределенной сети, нужны для координации совместных действий вершин с целью достижения цели. В данной статье такой целью является минимизация или максимизация индекса Винера.

Структура статьи следующая. После введения разд. 2 содержит описание используемой модели распределенной сети. В разд. 3 определяется атомарная трансформация и показывается ее достаточность для любых преобразований деревьев. В разд. 4 определяются основные понятия и доказываются основные утверждения, которые связаны с индексом Винера и на которые опираются предлагаемые алгоритмы. Сами алгоритмы описываются в разд. 5, там же доказываются их корректность и линейные оценки сложности. В заключении намечаются направления дальнейших исследований.

2. Модель

Рассматривается распределённая сеть с неориентированным графом связей $G = (V, E)$ без кратных рёбер и петель, где V – множество вершин, $E \subseteq \{ \{a, b\} \mid a \in V \,\&\, b \in V \}$ – множество рёбер. В таком графе путь длины k из вершины a в вершину b однозначно задаётся последовательностью вершин $a = a_1, \ldots, a_{k+1} = b$.

Граф предполагается *упорядоченным*: рёбрам, инцидентным каждой вершине $v \in V$, присвоены различные ненулевые номера, номер ребра 0 зарезервирован для служебных целей. Тем самым, каждое ребро $\{a, b\}$ имеет два номера: в вершине a и в вершине b, которые мы будем обозначать как $e(a, b)$ и $e(b, a)$, соответственно.

В каждой вершине графа находится вычислительная единица (автомат), которая может посылать сообщения по инцидентным вершине рёбрам и принимать сообщения по этим рёбрам, посланные с других их концов. Для краткости там, где это не приводит к двусмысленности, мы будем вместо «вычислительная единица, находящаяся в вершине», говорить просто «вершина». Память вершины будет рассматриваться как набор переменных. Переменные, значения которых сохраняются между приёмом сообщений, будем называть *постоянными* переменными; остальные переменные – *временные*. Постоянную переменную p в вершине x будем обозначать $p(x)$, а временную переменную q – просто q. Предполагается, что с самого начала в каждой вершине a инициализирована постоянная переменная $E(a) = \{ e(a, b) \mid \{a, b\} \in E \}$ – множество номеров $e(a, b)$ рёбер $\{a, b\}$, инцидентных вершине a. В дальнейшем при трансформации графа вершина a сама корректирует переменную $E(a)$.

Сообщение будет указываться его типом и набором параметров: $Tun(параметр_1, \ldots, параметр_k)$. Когда вершина a посылает сообщение по ребру $\{a, b\}$, она указывает номер $e(a, b)$ этого ребра. Когда вершина b принимает сообщение по ребру $\{a, b\}$, ей становится известным номер $e(b, a)$ этого ребра. Ребро $\{a, b\}$ с указанием направления пересылки сообщения из a в b будем обозначать $a \rightarrow b$.

Предполагается, что сообщения генерируются только вершинами (не рёбрами), и сообщения, передаваемые по ребру, не теряются и не обгонят друг друга.

Для оценки времени работы предлагаемых далее алгоритмов будем считать, что каждое сообщение по каждому ребру графа перемещается недетерминированное время, которое не больше 1 такта. Иными словами, мы будем оценивать длины пройденных сообщениями путей в «наихудшем» случае.

В данной статье рассматривается модель, в которой граф связей G является деревом с заданным ограничением d ($d \geq 3$) на степени его вершин: для каждой вершины v её степень не превышает d. Случай $d < 3$ тривиален, поскольку для каждого числа вершин n и ограничения d ($d < 3$) на степени вершин существует не более одного дерева с точностью до изоморфизма: это линейное дерево.

3. Трансформация дерева

Мы рассматриваем динамические деревья, которые трансформируется не случайно, а по «командам» от их вершин. Целью такой трансформации является достижение некоторого оптимального вида дерева. Мы будем рассматривать только такие трансформации, которые не меняют множество вершин дерева, оставляют его деревом и не превышают заданного ограничения d на степени вершин дерева.

Атомарной трансформацией назовём замену ребра $\{a, c\}$ на ребро $\{a, b\}$ при условии, что в дереве есть ребро $\{c, b\}$. Эту трансформацию будем обозначать $a \rightarrow c \rightarrow b$. Будем предполагать, что сообщения, передаваемые по изменяемому ребру в момент атомарной трансформации, не теряются, но если сообщение направлялось в вершину c, то получит его вершина b.

Утверждение 1. Атомарная трансформация не меняет множество вершин дерева и оставляет его деревом.

Доказательство. Условием атомарной трансформации $a \to c \to b$ является наличие в графе рёбер $\{a, c\}$ и $\{c, b\}$. Если бы после трансформации появился цикл, он проходил бы по добавляемому ребру $\{a, b\}$, т.е. имел бы вид $a, b, d_1, \ldots, d_k, a$. Но тогда до трансформации был бы цикл $a, c, b, d_1, \ldots, d_k, a$, чего быть не может. Следовательно, в результате трансформации циклы не появляются. Поскольку граф до трансформации был связным, в нём для любых двух вершин v и w существовал путь из v в w. Если этот путь проходил через удаляемое ребро $\{a, c\}$, т.е. имел вид $v, \ldots, a, c, \ldots, w$, то после трансформации будет существовать путь $v, \ldots, a, b, c, \ldots, w$, т.е. граф останется связным.□

Развилкой называется вершина степени 3 или больше. *Линейным деревом* называется дерево без развилок. Очевидно, дерево линейное тогда и только тогда, когда в нём ровно два листа (вершин степени 1), или оно содержит только одну изолированную вершину.

Утверждение 2. Любое дерево можно преобразовать в линейное дерево цепочкой атомарных трансформаций без нарушения ограничения d на степень вершин.

Доказательство. Пусть дерево не линейное. Выберем произвольный лист c_1 и будет двигаться от него в дереве по пути $c_1, c_2, \ldots, c_k$ до ближайшей развилки c_k. В дереве должны быть, по крайней мере, два ребра $\{a, c_k\}$ и $\{b, c_k\}$, где $a \ne c_{k-1}$ и $b \ne c_{k-1}$. Выполним цепочку атомарных трансформаций $a \to c_k \to c_{k-1}$, $a \to c_{k-1} \to c_{k-2}$, …, $a \to c_2 \to c_1$. По утверждению 1 каждая из трансформаций $a \to c_i \to c_{i-1}$ не меняет множество вершин дерева и оставляет его деревом. Такая трансформация уменьшает на 1 степень вершины c_i и увеличивает на 1 степень следующей вершины c_{i-1}, которая до этого была равна 2, если $i < 2$, или 1, если $i = 2$. Тем самым, каждая из этих трансформаций не нарушает ограничения d на степень вершин. Кроме того, каждая из этих трансформаций, кроме последней, не меняет число листьев дерева, а последняя трансформация $a \to c_2 \to c_1$ делает лист c_1 внутренней (не листовой) вершиной. Тем самым, эта цепочка атомарных трансформаций уменьшает число листьев дерева на 1. Будем выполнять эту процедуру выбора листа и выполнения цепочки атомарных трансформаций до тех пор, пока дерево не станет линейным при числе листьев 2.□

Утверждение 3. Любое дерево можно получить из линейного дерева с тем же множеством вершин цепочкой атомарных трансформаций без нарушения ограничения d на степень вершин.

Доказательство. Достаточно заметить, что атомарная трансформация обратима: после трансформации $T = a \to c \to b$, которая выполняется при наличии ребра $\{c, b\}$, можно сделать обратную трансформацию $T' = a \to b \to c$. Если в дереве до трансформации $a \to c \to b$ степени вершин не превышали d, то, очевидно, обратная трансформация не нарушает ограничения d на степени вершин. Соответственно, после выполнения цепочки атомарных трансформаций $T_1, T_2, \ldots T_{k-1}, T_k$ можно выполнить обратную цепочку $T_k', T_{k-1}', \ldots T_2', T_1'$. А тогда, поскольку любое дерево G можно трансформировать в линейное дерево по утверждению 2, то можно выполнить и обратную трансформацию линейного дерева в дерево G.□

Из утверждений 2 и 3 непосредственно следует, что для любых двух деревьев с ограничением d ($d \ge 3$) на степени вершин одно можно получить из другого с помощью цепочки атомарных трансформаций, причем по утверждению 1 в процессе трансформации ограничение d на степени вершин не будет нарушено.

Будем считать, что атомарная трансформация $a \to c \to b$ выполняется тогда, когда вершина c подаёт команду ***Изменить*** $(e(c, a), e(c, b), P_c)$, где P_c – дополнительные параметры, формируемые вершиной c и зависящие от того алгоритма, при выполнении которого делается эта трансформация. Выполнение этой команды заключается в следующем (рис. 1): 1) ребро $\{a, c\}$ меняет свою конечную вершину c на вершину b, 2) ребро $\{a, b\}$ получает в вершине a тот же номер, которое в этой вершине имело удаляемое ребро $\{a, c\}$, т.е. $e(a, c)$, а в вершине b – некоторый «свободный» номер $e(b, a)$, который не был номером какого-либо ребра, инцидентного вершине b в момент подачи команды ***Изменить***, 3) для того чтобы вершина b

«узнала» о появлении у неё нового ребра $\{a, b\}$, по ребру $a \to b$ автоматически посылается сообщение *Изменение*(P_c). Мы будем считать, что время атомарной трансформации, включая пересылку сообщения *Изменение*, не превышает 1 такта.

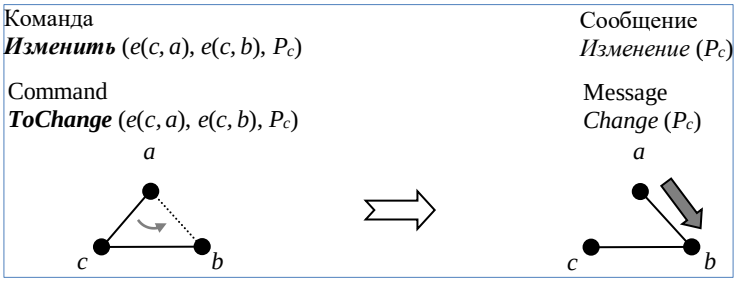


Рис. 1. Трансформация $a \to c \to b$: замена ребра $\{a, c\}$ на ребро $\{a, b\}$
Fig. 1. Transformation $a \to c \to b$: replacement of an edge $\{a, c\}$ by an edge $\{a, b\}$

После подачи команды ***Изменить*** ($e(c, a)$, $e(c, b)$, P_c) вершина c сама удаляет ребро $e(c, a)$ из множества $E(c) := E(c) \setminus \{e(c, a)\}$. Получив сообщение *Изменение*(P_c) по ребру $e(b, a)$, вершина b сама добавляет номер ребра во множество $E(b) := E(b) \cup \{e(b, a)\}$. Заметим, что если в момент подачи команды ***Изменить*** по ребру $a \to c$ пересылалось некоторое сообщение, то его получит не вершина c, а вершина b, причём раньше сообщения *Изменение*. Вершина b может распознать этот случай при получении сообщения по ребру с номером $e(b, a) \notin E(b)$.

4. Индекс Винера

Одной из важных числовых характеристик графа является *индекс Винера*, определяемый как сумма всех попарных расстояний между вершинами.

Максимальный индекс Винера, очевидно, имеет линейное дерево. Индекс Винера линейного дерева с n вершинами равен $(n - 1)n(n + 1)/6$ (последовательность A000292 [[18]]).

Без ограничения на степени вершин минимальный индекс Винера имеет граф-звезда: связный граф, все рёбра которого исходят из одной вершины. Индекс Винера звезды с n вершинами равен $(n - 1)^2$ (последовательность A000290 [[18]]). Но нас будут интересовать деревья, степень вершин которых ограничена сверху некоторым заданным числом $d \geq 3$. Вид дерева с минимальным индексом Винера на классе деревьев с n вершинами и ограничением d на степень вершин определён в [[2]]. Мы будем называть его *хорошим* деревом, термин взят из [[3]], где он применяется для бинарных ($d = 3$) хороших деревьев. Определим хорошее дерево своими словами.

Корневым деревом называется дерево, в котором выделена одна вершина, называемая *корнем*. *Высотой вершины* в корневом дереве называют расстояние от вершины до корня. *Высотой корневого дерева* называют максимум из высот его вершин. *Ветвью вершины* v в корневом дереве G называется подграф $G(v)$, порождённый множеством вершин, связанных с корнем путём, проходящим через v. Для ребра $\{a, b\}$ вершину a называют *отцом* вершины b, а вершину b – *сыном* вершины a, если путь из корня в вершину b проходит через вершину a. У каждой вершины, кроме корня, есть ровно один отец. ([19][1])

Если корневое дерево с корнем r упорядочено, то множество сынов каждой вершины v линейно упорядочено: $w_1, \ldots, w_k$. Будем говорить, что вершина w_i расположена *левее* вершины w_j, а вершина w_j расположена *правее* вершины w_i, если $i < j$. Эти линейные порядки сынов каждой вершины индуцируют линейный порядок вершин одной высоты h: вершина v расположена *левее* вершины v', если после общего префикса путей, ведущих из корня в эти

[1] В последнее время в англоязычной литературе употребляется асексуальная пара терминов *parent-child (родитель-ребенок)*.

вершины $r, v_1, \ldots, v_i, v_{i+1}\ldots, v_h = v$ и $r, v_1, \ldots, v_i, v'_{i+1}\ldots, v'_h = v'$, т.е. после вершины v_i, следующая вершина v_{i+1} на пути в вершину v расположена левее следующей вершины v'_{i+1} на пути в вершину v'. Соответственно, вершина v' расположена *правее* вершины v. Для любой вершины v на её ветви $G(v)$ любая вершина w имеет то же множество сынов, что и в дереве G. Поэтому, если дерево G упорядочено, то будем считать, что ветвь $G(v)$ также упорядочена: для каждой вершины на ветви $G(v)$ задан тот же линейный порядок её сынов, что в дереве G. Очевидно, что для любой вершины v на высоте h линейный порядок множества вершин ветви $G(v)$, находящихся в этой ветви на высоте k, является отрезком линейного порядка множества вершин дерева G на суммарной высоте $h + k$. Пусть задан путь из корня в лист v высотой h. Будем говорить, что вершина w на высоте $k \le h$ расположена *левее* (*правее*) этого пути, если она расположена левее (правее) вершины, лежащей на этом пути и имеющей ту же высоту k.

Корневое дерево высотой h с n вершинами и ограничением d ($d \ge 3$) на степени вершин будем называть *почти хорошим*, если дерево может быть так упорядочено, что 1) степень корня равна $min\{d - 1, n - 1\}$, 2) для $h \ge 3$ все вершины на высоте $1 .. h - 2$ имеют степень d, 3) для $h \ge 2$ на высоте $h - 1$ самая правая внутренняя вершина u имеет степень не больше d, вершины левее вершины u имеют степень d, а вершины правее вершины u имеют степень 1 (листья). Эту вершину u будем называть *разделяющей* вершиной в дереве. Пример почти хорошего дерева для $d = 3$, $h = 4$, $n = 26$ на рис. 2 справа.

Определение *хорошего* дерева отличается от определения почти хорошего дерева только первым условием: степень корня равна $min\{d, n - 1\}$. Пример хорошего дерева для $d = 3$, $h = 4$, $n = 27$ на рис. 2 слева.

Почти хорошее дерево высотой h будем называть *правильным почти хорошим деревом*, если 1) корень имеет степень 0 или $d - 1$, и 2) для $h \ge 2$ все вершины на высоте $h - 1$ имеют степень d. Очевидно, что все листья находятся на высоте h. Число вершин такого дерева обозначим $N(d, h) = 1 + (d - 1) + (d - 1)^2 + \ldots + (d - 1)^h = ((d - 1)^{h+1} - 1) / (d - 2)$. Пример правильного почти хорошего дерева для $d = 3$, $h = 3$, $n = 15$ на рис. 2 справа, если удалить «серые» вершины на высоте 4.

Хорошее дерево высотой h будем называть *правильным хорошим деревом*, если 1) корень имеет степень 0 или d, и 2) для $h \ge 2$ все вершины на высоте $h - 1$ имеют степень d. Очевидно, что все листья находятся на высоте h. Число вершин такого дерева обозначим $M(d, h)$. Очевидно, $M(d, 0) = 1$ и $M(d, h) = 1 + d + d(d - 1) + \ldots + d(d - 1)^{h-1} = 1 + dN(d, h - 1)$ для $h \ge 1$. Пример правильного хорошего дерева для $d = 3$, $h = 3$, $n = 22$ на рис. 2 слева, если удалить «серые» вершины на высоте 4.

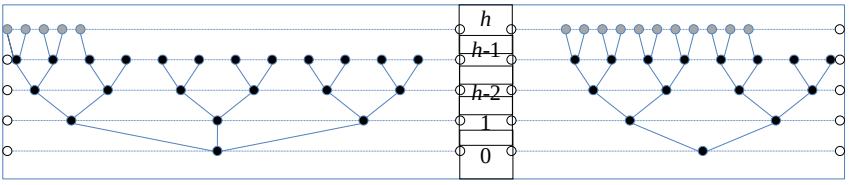


Рис. 2. Хорошее дерево (слева) и почти хорошее дерево (справа)
Fig. 2. Good tree (left) and almost good tree (right).

Утверждение 4. (Теорема 2.2. в [[2]]) Дерево с n вершинами и максимальной степенью вершин не больше d ($d \ge 3$) имеет минимальный индекс Винера тогда и только тогда, когда это хорошее дерево.

Утверждение 5. Для каждого $n \ge 1$ существует и единственное с точностью до изоморфизма, сохраняющего корень, (почти) хорошее дерево с числом вершин n.

Доказательство. Сначала докажем индукцией по n существование (почти) хорошего дерева с n вершинами. Для $n = 1$ утверждение очевидно. Пусть утверждение доказано для n и докажем его для $n + 1$. Пусть G (почти) хорошее дерево с n вершинами высотой h. Если это дерево

правильное, выберем самую левую вершину на высоте h, и соединим её ребром с новым листом. Очевидно, получится (почти) хорошее дерево с числом вершин $n + 1$ и высотой $h + 1$. Если дерево G не является правильным, то на высоте $h - 1$ можно выбрать самую левую вершину u степени меньше d. Соединим её ребром с новым листом. Очевидно, получится (почти) хорошее дерево с числом вершин $n + 1$ и высотой h.

Теперь покажем единственность с точностью до изоморфизма (почти) хорошего дерева с числом вершин n. Для вершины v обозначим через h_v высоту вершины v, а через i_v ее порядковый номер в линейном порядке вершин той же высоты. Изоморфизм определяется следующим соответствием вершин: вершины v и v' двух деревьев с n вершинами соответствуют друг другу, если $h_v = h_{v'}$ и $i_v = i_{v'}$. Если v корень, $h_v = 0$ и $i_v = 1$. Для завершения доказательства достаточно показать, что существование ребра $\{v, w\}$, где вершина v отец вершины w, однозначно определяется тем, является ли вершина v корнем хорошего дерева, и парами чисел (h_v, i_v) и (h_w, i_w). Действительно, если ребро $\{v, w\}$ существует, то $h_w = h_v + 1$. Из определения (почти) хорошего дерева следует, что все вершины на высоте h_v, расположенные левее вершины v, имеют $d - 1$ сынов, поэтому $i_w \geq (d - 1)(h_v - 1) + 1$. Если v корень хорошего дерева, то у вершины v не больше d сынов, поэтому $i_w \leq (d - 1)h_v + 1$. В противном случае у вершины v не больше $d - 1$ сынов, поэтому $i_w \leq (d - 1)h_v$. Если вершины v и w существуют, то эти соотношения пар чисел (h_v, i_v) и (h_w, i_w) однозначно определяют, существует ли ребро $\{v, w\}$, где вершина v отец вершины w.□

Если дерево G имеет высоту h, то его максимальное поддерево с тем же корнем, имеющее высоту $h - 1$, обозначим $G^\wedge$.

Утверждение 6. Пусть G (почти) хорошее дерево G с n вершинами высотой $h > 0$. Тогда 1) дерево $G^\wedge$ является правильным (почти) хорошим деревом высотой $h - 1$ 2); 2) число вершин дерева $G^\wedge$ равно $N(d, h - 1)$, если G почти хорошее дерево, и равно $M(d, h - 1)$, если G хорошее дерево; 3) число вершин дерева G на высоте h равно $n - N(d, h - 1)$, если G почти хорошее дерево, и равно $n - M(d, h - 1)$, если G хорошее дерево.

Доказательство. 1-я часть утверждения следует из определений (правильного) (почти) хорошего дерева. 2-я часть утверждения следует из 1-й части утверждения и определения величин $N(d, h - 1)$ и $M(d, h - 1)$ как числа вершин правильного почти хорошего дерева и правильного хорошего дерева высотой $h - 1$. 3-я часть утверждения следует из 1-й и 2-й частей утверждения и обозначения числа вершин дерева G через n.□

Утверждение 7. В (почти) хорошем дереве G ветвь $G(v)$ соседней с корнем вершины v является почти хорошим деревом.

Доказательство. Пусть h высота дерева G, n' число вершин ветви $G(v)$, а $h' \leq h - 1$ ее высота. Вершина v имеет в $G(v)$ степень на 1 меньше её степени в G, а степени остальных вершин $G(v)$ одинаковы в $G(v)$ и в G. Докажем для $G(v)$ выполнение условий в определении почти хорошего дерева.

- Условие 1. Если $h = 1$, вершина v имеет в G степень 1 и, следовательно, имеет в $G(v)$ степень 0, а $n' = 1$. Имеем: $0 = min\{d - 1, n' - 1\}$. Если $h = 2$, то все сыны вершины v листья, поэтому ее степень равна $n' - 1$. По условию 3 для G вершина v имеет в G степень от 1 до d и, следовательно, имеет в $G(v)$ степень от 0 до $d - 1$. Имеем: $n' - 1 = min\{d - 1, n' - 1\}$. Если $h \geq 3$, то по условию 2 для G вершина v имеет в G степень d и, следовательно, имеет в $G(v)$ степень $d - 1$, а $n' \geq d$. Имеем: $d - 1 = min\{d - 1, n' - 1\}$.
- Условие 2. Пусть $h' \geq 3$ и вершина w принадлежит $G(v)$ и находится в $G(v)$ на высоте от 1 до $h' - 2$. Тогда $h \geq 4$ и в G вершина w находится на высоте от 2 до $h' - 1 \leq h - 2$. По условию 2 для G вершина w в G имеет степень d, следовательно, в $G(v)$ она тоже имеет степень d.
- Условие 3. Пусть $h' \geq 2$ и вершина w принадлежит $G(v)$ и находится в $G(v)$ на высоте $h' - 1$. Тогда в G вершина w находится на высоте h'. Поскольку по условиям 2 и 3 для G

все листья в G находятся на высоте h или h - 1, в $G(v)$ все листья находятся на высоте h - 1 или h - 2. Отсюда $h' = h$ - 1 или $h' = h$ - 2.

Если $h' = h$ - 1, то $h \geq 3$ и по условию 3 для G в G на высоте h - 1 есть разделяющая вершина u. Линейный порядок $w_1, \ldots, w_m$ вершин ветви $G(v)$ на высоте h' - 1 в $G(v)$ является отрезком линейного порядка вершин G на высоте $h' = h$ - 1 в G. В $G(v)$ есть внутренняя вершина, лежащая в $G(v)$ на высоте h' - 1 и, следовательно, лежащая в G на высоте $h' = h$ - 1. Поэтому в G вершина u лежит не левее вершины w_1. Если u одна из вершин $w_1, \ldots, w_m$, выберем $u' = u$. Если u в G лежит правее w_m, выберем $u' = w_m$. В любом из этих двух случаев в $G(v)$ на высоте h' - 1 вершина u' является разделяющей. Если $h' = h$ - 2, то $h \geq 4$ и по условию 2 для G в G на высоте h - 2 все вершины имеют степень d. Следовательно, в $G(v)$ все вершины на высоте h - 3 = h' - 1 имеют степень d. В $G(v)$ самая правая вершина на высоте h' - 1 является разделяющей.□

Пусть (почти) хорошее корневое дерево G имеет n вершин. Упорядочим соседей корня $v_1, v_2, \ldots$ в порядке невозрастания числа вершин в их ветвях и обозначим эти числа:

- для почти хорошего дерева G: $N(d, n, 1) \geq \ldots \geq N(d, n, min\{d - 1, n - 1\})$,
- для хорошего дерева G: $M(d, n, 1) \geq \ldots \geq M(d, n, min\{d, n - 1\})$.

Следующее утверждение определяет эти числа.

Утверждение 8. Пусть $d \geq 3$ и G (почти) хорошее корневое дерево с n вершинами. Пусть $L(d, i) = N(d, i)$, если G почти хорошее дерево, и $L(d, i) = M(d, i)$, если G хорошее дерево. Пусть $n = L(d, h - 1) + m(d - 1) + r < L(d, h)$, где $0 \leq r < d - 1$. Пусть $m = p(d - 1)^{h-2} + q$, где $0 \leq q < (d - 1)^{h-2}$. Тогда для p самых левых соседей корня их ветви имеют по $N(d, h - 1)$ вершин, для следующего справа соседа корня его ветвь имеет $N(d, h - 2) + q(d - 1) + r$ вершин, а для остальных соседей корня их ветви имеют по $N(d, h - 2)$ вершин.

Доказательство. Пусть $n = L(d, h - 1)$. Тогда $p = q = r = 0$ и по утверждению 5 дерево G правильное (почти) хорошее дерево высотой h - 1. По утверждению 7 ветвь каждого соседа корня – это почти хорошее дерево. Поскольку в правильном дереве высотой h - 1 все листья находятся на высоте h - 1, в ветви соседа корня все листья находятся на высоте h - 2. Следовательно, ветвь соседа корня правильное почти хорошее дерево высотой h - 2. Поэтому ветви всех соседей корня имеют по $N(d, h - 2)$ вершин.

Пусть $L(d, h - 1) < n < N(d, h)$. Тогда для $n = L(d, h - 1) + m(d - 1) + r$, где $0 \leq r < d - 1$, дерево G имеет высоту h, а все листья находятся на высоте h - 1 или h. На высоте h - 1 имеется m вершин степени d, одна вершина имеет степень $r + 1$, а остальные вершины – листья.

Пусть $m = p(d - 1)^{h-2} + q$, где $0 \leq q < (d - 1)^{h-2}$. Правильное почти хорошее дерево высотой h - 2 на этой высоте имеет $(d - 1)^{h-2}$ листьев. Поэтому для p самых левых соседей корня дерева G ветвь каждого из этих соседей имеет листья на одной высоте h в дереве G, т.е. на высоте h - 1 в ветви соседа. Следовательно, это правильные почти хорошие деревья высотой h - 1, поэтому они имеют по $N(d, h - 1)$ вершин.

Для следующего справа соседа w корня его ветвь $G(w)$ – это почти хорошее дерево высотой h - 1. По утверждению 6 максимальное поддерево $G(w)^\wedge$ с тем же корнем и высотой h - 2 является правильным почти хорошим деревом высотой h - 2 и содержит $N(d, h - 2)$ вершин. Ветвь $G(w)$ содержит дополнительно $q(d - 1) + r$ листьев на высоте h - 1 и ведущих в них ребер. Поэтому общее число вершин в ветви $G(w)$ равно $N(d, h - 2) + q(d - 1) + r$.

Если $p + 1$ меньше степени корня, то для остальных соседей корня ветвь каждого из них имеет листья на одной высоте h - 1 в дереве G, т.е. на высоте h - 2 в ветви соседа. Следовательно, это правильные почти хорошие деревья высотой h - 2, поэтому они имеют по $N(d, h - 2)$ вершин.□

5. Алгоритмы

Линейное дерево имеет максимальный индекс Винера на классе деревьев с тем же числом вершин. По утверждению 2 любое дерево можно превратить в линейное дерево с помощью цепочки атомарных трансформаций, не нарушающих ограничения d ($d \geq 3$) на степени вершин.

По утверждению 4 хорошее дерево имеет минимальный индекс Винера на классе деревьев с тем же числом вершин и тем же ограничением на степени вершин. Утверждение 3 говорит о существовании обратной цепочки атомарных трансформаций линейного дерева, не нарушающих ограничение на степени вершин, в любое наперёд заданное дерево с тем же числом вершин и тем же ограничением на степени вершин. Тем самым, любое дерево можно превратить в хорошее дерево с помощью цепочки атомарных трансформаций, не нарушающих ограничение на степени вершин.

Эти утверждения, однако, не конструктивные: они говорят только о существовании нужной цепочки атомарных трансформаций. Нужны алгоритмы таких преобразований, по которым могли бы работать вычислительные единицы в вершинах распределённой сети.

Мы определим два алгоритма:

- *Алгоритм* $\mathcal{A}$ трансформации произвольного дерева в линейное дерево.
- *Алгоритм* $\mathcal{B}$ трансформации линейного дерева в хорошее дерево.

5.1 Алгоритм $\mathcal{A}$ – трансформация в линейное дерево

Пусть задано произвольное корневое дерево G. Если для вершины v ветвь ее сына w является линейным деревом $w = w_1, w_2, \ldots, w_k$, то путь $v, w_1, w_2, \ldots, w_k$ будем называть *линейкой*, ведущей из v. Если вершина v изолированная (и, следовательно, единственная в дереве), путь нулевой длины из вершины v тоже будем считать линейкой, ведущей из v. *Звездообразным* деревом называется корневое дерево, в котором только корень может иметь степень больше 2. Звездообразное дерево состоит из линеек, ведущих из корня.

Работу алгоритма $\mathcal{A}$ на дереве G с корнем *root* обозначим $\mathcal{A}(G, root)$. Для удобства описания алгоритма будем считать, что у корня дерева имеется фиктивное ребро с номером 0, ведущее к фиктивному отцу. Алгоритм $\mathcal{A}(G, root)$ начинает работать в корне с получения от его (фиктивного) отца сообщения *Старт*(). Алгоритм завершается посылкой из корня его (фиктивному) отцу сообщения *Линия*(n), где n – число вершин дерева.

Алгоритм выполняется рекурсивно, уровень рекурсии равен высоте вершины. На уровне рекурсии h алгоритм выполняется на каждой ветви $G(v)$, начиная с вершины v, имеющей в G высоту h. Алгоритм состоит из трёх этапов.

- Этап 1 (рис. 3). Вершина v получает сообщение *Старт* от своего отца, который фиктивен для $h = 0$ и принадлежит дереву для $h > 0$. Затем вершина v рассылает сообщение *Старт* всем своим сынам, после чего начинается этап 2.
- Этап 2 (рис. 4). Вершина v ожидает от своих сынов получения сообщений *Линия.* Когда сообщения *Линия* будут получены от всех сынов, начинается этап 3 алгоритма. В этот момент времени ветвь $G(v)$ является звездообразным деревом. Постоянная переменная $nson(v)$ равна числу ожидаемых сообщений *Линия*. Она инициализируется при получении вершиной v сообщения *Старт* числом сынов вершины v. При получении вершиной v сообщения *Линия* переменная $nson(v)$ уменьшается на 1.

  Этап 2 завершается, когда переменная $nson(v)$ достигает нуля. Также на этапе 2 вершина v подсчитывает число вершин ветви. Для этого используется постоянная переменная $sson(v)$. Она инициализируется единицей при получении вершиной v сообщения *Старт* () и увеличивается на число l вершин ветви $G(w)$ при получении от сына w сообщения *Линия* (l).

- Этап 3 (рис. 5). Пусть v^1_1, …, v^k_1 сыны вершины v. Для $i = 1 .. k$ имеется i-ая линейка $v = v^i_0, v^i_1, …, v^i_{x(i)}$ длиной $x(i)$. Вершина v запускает $k - 1$ параллельных цепочек атомарных трансформаций так, что для $i = 1 .. k - 1$ у первого ребра $i + 1$-ой линейки $\{v, v^{i+1}_1\}$ один его конец v^{i+1}_1 фиксируется, а другой конец двигается по i-ой линейке от вершины v до листа $v^i_{x(i)}$. Для этого вершина v выполняет сразу $k - 1$ атомарных трансформаций $v^{i+1}_1 \to v \to v^i_1$, подавая команду ***Изменить*** $(e(v, v^{i+1}_1), e(v, v^i_1), f(i))$, в последовательности $i = k - 1, ..., 1$. О параметре $f(i)$ мы скажем ниже.

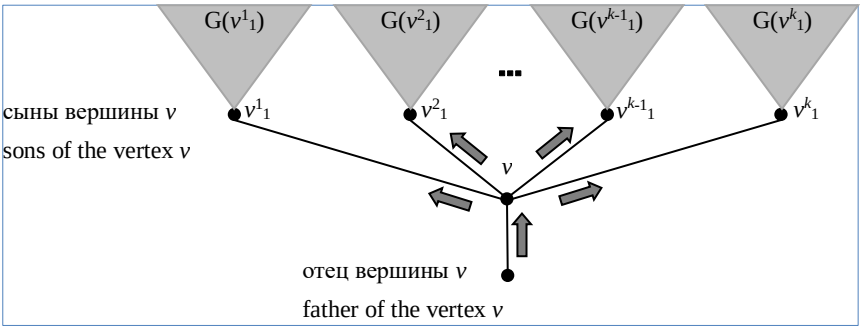


Рис. 3. Этап 1: Сообщения Старт и ветви дерева
Fig.3. Stage 1: Messages "Start" and tree branches

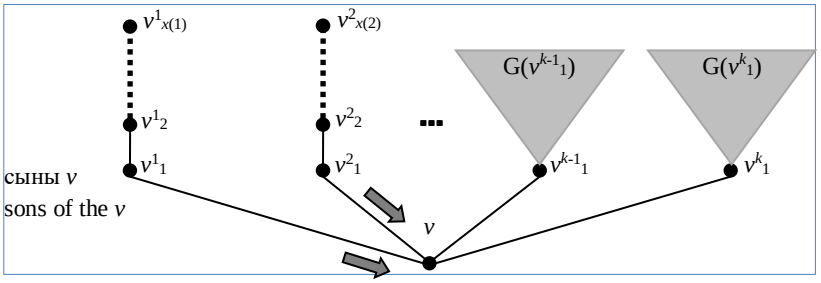


Рис. 4. Этап 2: Сообщения Линия и линейки
Fig. 4. Stage 2: Line and Messages "Line".

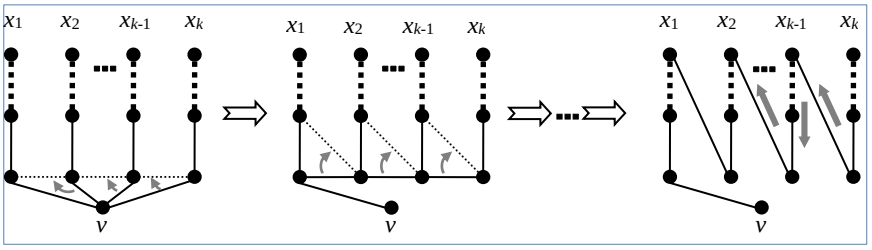


Рис. 5. Этап 3: Сообщения Финиш и трансформации
Fig. 5. Messages "Finish" and Transformations.

В результате для $i = 1 .. k - 1$ ребро $\{v, v^{i+1}_1\}$ заменится ребром $\{v^i_1, v^{i+1}_1\}$ и по этому ребру в вершину v^i_1 придёт сообщение *Изменение* $(f(i))$. Получив это сообщение, вершина v^i_1 выполнит следующую атомарную трансформацию $v^{i+1}_1 \to v^i_1 \to v^i_2$, и так далее. Для $i = 1 .. k - 1$ вдоль i-й линейки будет выполнена цепочка $v^{i+1}_1 \to v \to v^i_1$, $v^{i+1}_1 \to v^i_1 \to v^i_2$, $v^{i+1}_1 \to v^i_2 \to v^i_3$, …, $v^{i+1}_1 \to v^i_{x(i)-1} \to v^i_{x(i)}$ атомарных трансформаций. Отметим, что все эти цепочки атомарных трансформаций выполняются параллельно вдоль $(k - 1)$ линеек. Цепочка атомарных трансформаций вдоль i-й линейки

заканчивается в ее листе, и в результате произойдёт конкатенация i-й и (i + 1)-й линеек. Когда это случится для всех i = 1 .. k - 1 ветвь $G(v)$ станет линейным деревом, т.е. в графе G появится линейка, ведущая из отца вершины v через вершину v.

Для того, чтобы вершина v «узнала» о завершении конкатенации всех линеек, используются сообщение *Финиш* (), булевский параметр f и постоянная переменная $f(w)$ во всех вершинах w из ветви $G(v)$. Первое сообщение *Финиш* посылается, когда заканчивается конкатенация k-й и (k - 1)-й линеек. Для того чтобы отличить эту конкатенацию от всех остальных, используется параметр f команды ***Изменить*** и сообщения *Изменение*, который равен ***true*** для атомарных трансформаций при конкатенации k-й и (k - 1)-й линеек и равен ***false*** для атомарных трансформаций при остальных конкатенациях. Сообщение *Финиш* пересылается от сына к отцу, двигаясь по (k - 1)-й линейке через вершины $v^{k-1}{}_{x(k-1)}$, …, $v^{k-1}{}_1$ и далее, если к этому моменту времени уже завершена конкатенация (k - 1)-й и (k - 2)-й линеек, (k - 2)-й и (k - 3)-й линеек и т.д.

Если же конкатенации i-й и (i + 1)-й линеек ещё не завершена, сообщение *Финиш* может быть послано в вершину $v^i{}_j$ из вершины $v^{i+1}{}_1$, где $j < x(i)$, тогда, когда ребро $\{v^i{}_j, v^{i+1}{}_1\}$ появилось в результате некоторой атомарной трансформации и, следовательно, по ребру послано сообщение *Изменение*, но это сообщение ещё не дошло до вершины $v^i{}_j$. Когда вершина $v^i{}_j$ получит сообщение *Изменение*, она начнёт трансформацию $v^i{}_j \to v^{i+1}{}_1 \to v^i{}_{j+1}$, меняющую ребро $\{v^i{}_j, v^{i+1}{}_1\}$ на ребро $\{v^i{}_{j+1}, v^{i+1}{}_1\}$. По ребру $\{v^i{}_{j+1}, v^{i+1}{}_1\}$ будет послано сообщение *Изменение*, но оно придёт в вершину $v^i{}_{j+1}$ после сообщения *Финиш*, поскольку сообщения, посланные по ребру, не обгоняют друг друга. Тем самым, вершина $v^i{}_{j+1}$ получит сообщение *Финиш* по ребру, номера которого ещё нет в множестве $E(v^i{}_{j+1})$. В этом случае вершина $v^i{}_{j+1}$ ожидает последующего сообщения *Изменение* для того, чтобы начать следующую атомарную трансформацию, если $j + 1 < x(i)$, или послать сообщение *Финиш* дальше уже по i-й линейке, если $j + 1 = x(i)$. Иными словами, вершина $v^i{}_{j+1}$ должна «помнить», получила ли она сообщение *Финиш*. Для этого используется постоянная переменная $f(v^i{}_{j+1})$. Если вершина $v^i{}_{j+1}$ выполняет следующую атомарную трансформацию, то она использует параметр $f = f(v^i{}_{j+1})$ в команде ***Изменить*** для того, чтобы следующая вершина $v^i{}_{j+2}$, получив сообщение *Изменение* с этим параметром, тоже «узнала» о приходе сообщения *Финиш*. В целом параметр f = ***true*** в команде ***Изменить*** и сообщении *Изменение* показывает, что, когда сообщение пересылается из вершины b в вершину a, имеется линейка a, b, …

Когда все линейки склеются в одну и вершина v получит сообщение *Финиш*, она пошлёт своему отцу сообщение *Линия* ($sson(v)$), тем самым завершая работу алгоритма на ветви $G(v)$.

Определим алгоритм $\mathcal{A}$ (G, *root*) формально.

- Постоянные переменные вершины v в алгоритме $\mathcal{A}$ (G, *root*):
  - $E(v)$ – множество номеров рёбер, инцидентных вершине v;
  - $fat(v)$ – номер в вершине v ребра, ведущего из вершины v к её отцу;
  - $line(v)$ – признак (***Bool***) того, что вершина v послала сообщение *Линия*;
  - $nson(v)$ – число ожидаемых вершиной v сообщений *Линия* от ее сынов;
  - $sson(v)$ – сумма длин линеек от вершины v в полученных сообщениях *Линия*;
  - $f(v)$ – признак (***Bool***) получения *Финиш* раньше *Изменение*.
- Обозначения: $element(S)$ – функция выбора произвольного элемента из непустого множества S.
- Сообщения в алгоритме $\mathcal{A}$(G, *root*): *Старт*(); *Линия*(l), где l – длина линейки; *Изменение*(f), где f – признак (***Bool***) того, что сообщение передаётся в вершину v по ребру, с которого начинается линейка из вершины v; *Финиш*().
- Предусловие алгоритма $\mathcal{A}$ (G, *root*): В дереве G в каждой вершине v переменная $E(v)$ инициализирована корректно, т.е. множеством номеров рёбер, инцидентных вершине v.

Одна из вершин – корень *root*, в корень по фиктивному ребру с номером 0 приходит сообщение *Старт*().

- Постусловие алгоритма $\mathcal{A}$ (*G*, *root*): Дерево *G* линейное, множество вершин сохранено, вершина *root* – лист или единственная изолированная вершина. В каждой вершине *v* значение переменной *E*(*v*) корректно. Из корня *root* по фиктивному ребру с номером 0 послано сообщение *Линия*(*n*), где *n* число вершин дерева.

Алгоритм $\mathcal{A}$ (*G*, *root*). Вершина *v*.

```
1  while не конец алгоритма {
2  wait () { /* приём сообщения */
3     if приём Старт() по ребру i { /* этап 1 */
4        fat(v) := i; line(v) := false;  /* инициализация */
5        nson(v) := | E(v) \ { fat(v) } |; sson(v) := 1; f(v) := 0;
6        посылка Старт() по каждому ребру j ∈ E(v) \ { fat(v) };
7        if nson(v) = 0 { /* v – лист или изолированная вершина */
8           посылка Линия(sson(v)) по ребру fat(v); line(v) := true;
9           if fat(v) = 0 { конец алгоритма; } /*v – изолированная вершина*/
10    } }
11    if приём Линия(l) по ребру i { /* этап 2 */
12       nson(v) := nson(v) - 1; sson(v) := sson(v) + l; /* новая линейка
         из v */
13       if E(v) \ { fat(v) } = { i } { /* у вершины v один сын */
14          посылка Линия(sson(v)) по ребру fat(v); line(v) := true;
15          if fat(v) = 0 { конец алгоритма; }
16       }
17       else { /* v – развилка или корень степени 2 */
18          if nson(v) = 0 { /*начало этапа 3: G(v) звездообразное дерево */
19             j := element( E(v) \ { fat(v), i } ); /* выбор ребра */
20             /* трансформация a→v→b, где e(v, a) = i, e(v, b) = j */
21             команда Изменить(i, j, true);
22             E(v) = E(v) \ { i }; /* ребро удалено */
23             while | E(v) \ { fat(v) } | > 1 { /* цикл трансформаций */
24                m := j;
25                j := element( E(v) \ { fat(v), m } ); /* выбор ребра */
26                /* трансформация a→v→b, где e(v, a) = m, e(v, b) = j */
27                команда Изменить(m, j, false);
28                E(v) = E(v) \ { m }; /* ребро удалено */
29    } } } }
30    if приём Изменение(f) по ребру i { /* этап 3: трансформация a→c→v */
31       f(v) := f(v) ∨ f; /* f(v) = true, если имеется линейка v, a, … */
32       if | E(v) | = 2 { /*цепочка трансформаций по линейке не закончена*/
33          j := element(E(v) \ { fat(v) }; /* выбор следующего ребра */
34          /* трансформация a→v→b, где e(v, a) = i, e(v, b) = j */
35          команда Изменить(i, j, f(v)); f(v) := false;
36       }
37       else { /*| E(v) | = 1, цепочка трансформаций по линейке закончена*/
38          E(v) = E(v) ∪ { i }; /* ребро добавлено */
39          if f(v) { /* произошла конкатенация двух линеек */
40             посылка Финиш() по ребру fat(v); f(v) := false;
41    } } }
42    if приём Финиш() по ребру i { /* этап 3 */
43       if i ∉ E(v) { f(v) := true; } /*Финиш пришёл в v раньше Изменение*/
44       else {
```

```
45          if  line(v) { /* вершина v посылала Линия */
46             посылка Финиш() по ребру fat(v);
47          }
48          else { /* вершина v не посылала Линия */
49             посылка Линия(sson(v)) по ребру fat(v); line(v) := true;
50             if fat(v) = 0 { конец алгоритма; }
51 }} }  }  }
```

□

Утверждение 9. Алгоритм $\mathcal{A}$ на классе корневых деревьев с n вершинами не нарушает ограничения d ($d \geq 3$) на степень вершины и заканчивает работу с выполнением постусловия алгоритма за время $t(n) \leq 2n - 2$.

Доказательство. Пусть дерево G имеет n вершин. Доказательство будем вести индукцией по числу n. Напомним, что время вычислений в вершине не учитывается, и время выполнения алгоритма слагается из времени последовательных пересылок сообщений, включая сообщение *Изменение* при атомарной трансформации.

Для $n = 1$ дерево G состоит из одной изолированной вершины *root*. Время работы алгоритма без учёта пересылок сообщений *Старт* и *Линия* по фиктивному ребру с номером 0, очевидно, равно нулю: $t(1) = 0 = 2*1 - 2$. Постусловие алгоритма выполнено: дерево G линейное, *root* – изолированная вершина, $E(v) = \varnothing$, по фиктивному ребру с номером 0 послано сообщение *Линия*(1).

Пусть утверждение верно для любого числа вершин не больше n, где $n \geq 1$, и докажем его для числа вершин $n + 1$. Пусть у корня k сынов и числа вершин в ветвях сынов равны $x_1, \ldots, x_k$. Имеем $x_1 + \ldots + x_k = n$. Сначала покажем, что алгоритм работает время не более $2(n + 1) - 2 = 2n$.

Этап 1 рассылок сообщений *Старт* занимает время $t_1(n + 1) \leq 1$, поскольку сообщения рассылаются одновременно.

На этапе 2 алгоритм выполняется параллельно на ветвях всех сынов корня. Поэтому этап 2 занимает время $t_2(n + 1)$, слагаемое из максимума от времён выполнения алгоритма на ветвях сынов и времени пересылки от сына корню сообщения *Линия*. Пусть $t(x_1) \leq \ldots \leq t(x_k)$. Имеем $t_2(n + 1) = max\{ t(x_1), \ldots, t(x_k) \} + 1 \leq t(x_k) + 1$. Поскольку $x_k \leq n$, по предположению шага индукции $t(x_k) \leq 2x_k - 2$. Поэтому $t_2(n + 1) \leq 2x_k - 2 + 1 = 2x_k - 1$.

По предположению шага индукции каждую из ветвей сынов корня алгоритм трансформирует в линейное дерево, которое в дереве G будет линейкой. Этап 3 трансформации звездообразного дерева в линейное занимает время $t_3(n + 1)$, слагаемое из времени выполнения всех атомарных трансформаций и времени пересылки сообщения *Финиш* до корня. Атомарные трансформации вдоль разных линеек выполняются параллельно. Поэтому время выполнения трансформаций равно максимуму из времён выполнения трансформаций вдоль одной линейки, а это последнее время не превышает длины линейки. Сообщение *Финиш* проходит по всем линейкам, кроме последней k-й линейки, появившейся позже других. Тем самым, $t_3(n + 1) \leq max\{x_1, \ldots, x_{k-1}\} + (x_1 + \ldots + x_{k-1})$. Пусть $max\{x_1, \ldots, x_{k-1}\} = x_m$. Тогда $t_3(n + 1) \leq x_m + (n - x_k)$.

Суммарно имеем:

$t(n + 1) = t_1(n + 1) + t_2(n + 1) + t_3(n + 1) \leq\leq 1 + 2x_k - 1 + x_m + (n - x_k) = n + (x_m + x_k) \leq n + (x_1 + \ldots + x_k) = 2n$,

что и требовалось доказать.

Докажем выполнение постусловия работы алгоритма. По предположению шага индукции на этапе 2 каждая из ветвей сынов корня трансформируется в линейное дерево. После этого в результате трансформаций на этапе 3 звездообразное дерево трансформируется в линейное дерево. Переменная $sson(v)$ инициализируется единицей на этапе 1. По предположению шага индукции на этапе 2 от каждого своего сына w вершина v получит сообщение *Линия*(l), где l

– число вершин ветви $G(w)$. Сумма этих чисел, очевидно, на 1 меньше числа вершин ветви $G(v)$. На этапе 2 все эти величины суммируются в переменной $sson(v)$, которая в конце этапа 2 будет равна числу вершин ветви $G(v)$. Для корня *root* число $sson(root)$ равно числу вершин дерева, именно это значение будет в конце работы алгоритма указано как параметр сообщения *Линия*, которое корень пошлёт по фиктивному ребру с номером 0.

Этот алгоритм не нарушает ограничения $d \ge 3$ на степень вершин, поскольку 1) каждая атомарная трансформация $a \to c \to b$ происходит вдоль линейки …, *c, b,* … в сторону листа, т.е. вершина *b* имеет степень 2 или 1, а трансформация увеличивает её степень на 1, 2) на каждом уровне рекурсии не происходят никакие две трансформации $a \to c \to b$ и $a' \to c' \to b$ с общей конечной вершиной *b*. Более того, на этапе 3 на каждой ветви $G(v)$ первые трансформации в цепочках атомарных трансформаций по линейкам сразу уменьшают степень вершины *v* до 2, если $v \ne root$, или 1, если $v = root$. После этого степень каждой вершины ветви не больше 3 до конца трансформаций на этой ветви, когда она становится не больше 2. □

Оценка времени $2n - 2$ достигается на дереве, которое уже является линейным, а его корень – один из листьев. Для этого случая эта оценка не улучшаема, если алгоритм должен закончиться в корне. Это объясняется тем, что для того, чтобы выяснить, что дерево уже линейное, сообщение должно дойти из корня до другого листа и вернуться обратно, то есть пройти путь длиной $2n - 2$.

5.2 Алгоритм $\mathcal{B}$ – трансформация линейного дерева в хорошее дерево

Работу алгоритма $\mathcal{B}$ на линейном дереве *G* с корнем *root* обозначим $\mathcal{B}(G, root)$. Для своей работы алгоритму $\mathcal{B}(G, root)$ нужно «знать» число *n* вершин дерева и ограничение *d* ($d \ge 3$) на степени вершин. Выполнение алгоритма $\mathcal{B}(G, root)$ запускается сообщением *Начало* (*d, n*), которое приходит в корень *root* по фиктивному ребру с номером 0 в корне. Алгоритм завершается посылкой из корня по (фиктивному) ребру с номером 0 сообщения *Конец* ().

Алгоритм выполняется рекурсивно, уровень рекурсии равен высоте вершины в итоговом хорошем дереве, которое должно быть построено в конце работы алгоритма. Предполагается, что на уровне рекурсии *h* построена часть хорошего дерева на высотах от 0 до *h*. Вначале $h = 0$ и построенная часть хорошего дерева состоит только из корня. На уровне *h* алгоритм выполняется на каждой ветви $G(v)$, где вершина *v* имеет в *G* высоту *h*. Алгоритм состоит из двух этапов.

(*Почти*) *хорошим звездообразным* деревом с числом вершин *k* будем называть звездообразное дерево, у которого степень корня и числа вершин на ветвях соседей корня такие же, как у (почти) хорошего дерева с числом вершин *k*.

- Этап 1. В начале этого этапа на уровне рекурсии *h* ветвь $G(v)$, где вершина *v* имеет в *G* высоту *h*, является линейкой ведущей из *v*. Задача этого этапа – построить звездообразное дерево с корнем в вершине *v*. Это дерево должно быть хорошим звездообразным деревом, если $v = root$ ($h = 0$), или почти хорошим звездообразным деревом, если $v \ne root$ ($h > 0$). Напомним, что по утверждению 7 в (почти) хорошем дереве ветвь соседа корня является почти хорошим деревом, а по утверждению 5 (почти) хорошее дерево однозначно определяется с точностью до изоморфизма, сохраняющего корень, числом его вершин. Тем самым, звездообразное дерево, в которое нужно превратить линейку ветви $G(v)$, однозначно с точностью до изоморфизма, сохраняющего корень, определяется числом вершин в этой ветви и признаком $v = root$ или $v \ne root$, соответствующим хорошему и почти хорошему деревьям. Число вершин ветви $G(v)$ вычисляется по утверждению 8 и является параметром сообщения *Начало*, с получения которого начинается этап 1 на ветви $G(v)$.
- Этап 2. На этом этапе на уровне рекурсии *h* ветвь $G(v)$, где вершина *v* имеет в *G* высоту *h*, является хорошим (если $v = root$) или почти хорошим (если $v \ne root$) звездообразным

деревом, а из каждого сына w вершины v через ребро, отличное от ребра, ведущего к вершине v, ведет линейка. Вершина v посылает каждому своему сыну w сообщение *Начало* (d, l), где l – число вершин на ветви $G(w)$, инициируя работу алгоритма на следующем уровне рекурсии $h + 1$. Очевидно, это можно делать, не дожидаясь, когда этап 1 полностью завершён, то есть полностью построено звездообразное дерево с корнем в вершине v, а как только построена линейка нужной длины, ведущая из вершины w через ребро, отличное от ребра, ведущего к вершине v.

Вершина v ожидает от своих сынов сообщений *Конец* (). Постоянная переменная $nson(v)$ равна числу ожидаемых сообщений *Конец*. Она инициализируется при получении вершиной v сообщения *Начало* и равна числу сынов вершины v в звездообразном дереве, которое должно быть построено. Если l – число вершин звездообразного дерева, то $nson(v) = min\{d, l - 1\}$, если $v = root$, или $nson(v) = min\{d - 1, l - 1\}$, если $v \neq root$. При получении вершиной v сообщения *Конец* переменная $nson(v)$ уменьшается на 1. Этап 2 завершается, когда переменная $nson(v)$ достигает нуля. Вершина v посылает своему отцу сообщение *Конец* (). Если $v = root$, алгоритм заканчивается.

Оптимизация: заметим, что если число вершин ветви меньше 3, то ветвь уже является хорошим и почти хорошим деревом, и запуск алгоритма на ветви не требуется.

Рассмотрим подробнее построение звездообразного дерева на этапе 1. Мы будем использовать понятие *текущей вершины* и две операции: *перемещение* и *трансформация*. В начале этапа 1 на уровне рекурсии h на ветви $G(v)$, где вершина v имеет в G высоту h, текущей вершиной становится вершина v при получении ею сообщения *Начало*. Если $v = root$, строится хорошее звездообразное дерево, иначе строится почти хорошее звездообразное дерево.

Две операции: Параметр t означает число трансформаций, которые осталось сделать для построения линейки звездообразного дерева.

Перемещение $c \dashrightarrow b$. Предусловие: c текущая вершина, есть ребро $\{c, b\}$. Вершина c посылает в вершину b сообщение *Перемещение*(t). После получения вершиной b этого сообщения текущей вершиной становится вершина b.

Трансформация $a \rightarrow c \rightarrow b$. Предусловие: c текущая вершина, есть рёбра $\{a, c\}$ и $\{c, b\}$. Величина t уменьшается на 1: $t := t - 1$. Вершина c подаёт команду ***Изменить***$(e(c, a), e(c, b), t)$. После получения вершиной b сообщения *Изменение*(t) текущей вершиной становится вершина b.

Сначала опишем построение хорошего звездообразного дерева на ветви $G(v)$ как последовательность этих операций, см. рис. 6. На этом рисунке серыми стрелками показаны текущие вершины, белыми кружками – вершина v, а чёрными кружками – остальные вершины.

Пусть l число вершин на ветви $G(v)$. Вначале имеется линейка $v = v_1, v_2, \ldots, v_l$, текущая вершина v.

Обозначим:

$x = min\{d, l - 1\}$ – степень вершины v в хорошем звездообразном дереве;

$SM(d, l, 0) = 1$ – число вершин в дереве из одной вершины;

$SM(d, l, j) = 1 + M(d, l, 1) + M(d, l, 2) + \ldots + M(d, l, j)$, для $j = 1 .. x$, – суммарное число вершин в хорошем звездообразном дереве на первых j линейках плюс единица, соответствующая корню дерева;

$v^j_i = v_{SM(d, l, j - 1) + i}$, для $j = 1 .. x - 1$ и $i = 1 .. M(d, l, j)$, – i-я вершина j-й линейки;

$v^x_i = v_{SM(d, l, x) - i + 1}$, для $i = 1 .. M(d, l, x)$, – i-я вершина x-й линейки.

Строим линейки «справа налево», начиная от x-й и заканчивая 2-й.

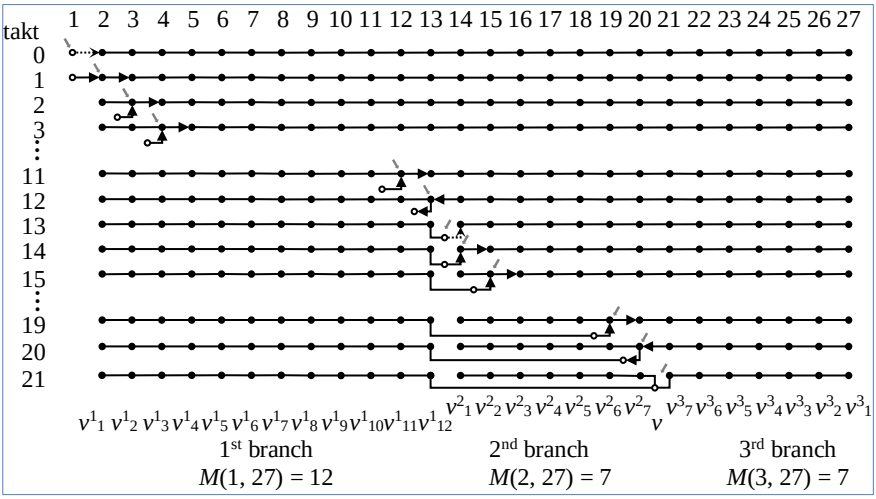


Рис. 6. Построение хорошего звездообразного дерева для n = 27 и d = 3
Fig. 6. Construction of a good starlike tree for n = 27 and d = 3

Построение j-й линейки хорошего звездообразного дерева для $j = x$ .. 2:

1. $t = M(d, l, j)$.
2. Одно перемещение $v \dashrightarrow v^j{}_1$.
3. Цепочка $M(d, l, j)$ - 1 трансформаций:
   $v \to v^j{}_1 \to v^j{}_2$, $v \to v^j{}_2 \to v^j{}_3$, …, $v \to v^j{}_{M(d, l, j) - 1} \to v^j{}_{M(d, l, j)}$;
   $t = M(d, l, j) - 1$, $t = M(d, l, j) - 2$, …, $t = 1$.
4. Одна завершающая $M(d, l, j)$-я трансформация: $v^{j+1}{}_1 \to v^j{}_{M(d, l, j)} \to v$.
5. Если $M(d, l, j) > 2$, запуск алгоритма построения почти хорошего звездообразного дерева на j-й ветви: вершина v посылает вершине $v^j{}_{M(d, l, j)}$, сообщение *Начало* $(d, M(d, l, j))$.
   После построения 2-й линейки также построена и 1-я линейка. Если $M(d, l, 1) > 2$, запуск алгоритма построения почти хорошего звездообразного дерева на 1-й линейке: вершина v посылает вершине $v^1{}_{M(d, l, 1)}$, сообщение *Начало* $(d, M(d, l, 1))$.

Построение почти хорошего звездообразного дерева с l вершинами отличается от алгоритма построения хорошего звездообразного дерева с l вершинами только тем, что число строящихся линеек x равно не $min\{d, l - 1\}$, а $min\{d - 1, l - 1\}$, и тем, что число вершин в j-й линейке равно не $M(d, l, j)$, а $N(d, l, j)$.

Определим алгоритм $\mathcal{B}(G, root)$ формально.

- Постоянные переменные вершины v в алгоритме $\mathcal{B}(G, root)$:
  - $E(v)$ – множество номеров рёбер, инцидентных вершине v;
  - $fat(v)$ – номер в вершине v ребра, ведущего из вершины v к её отцу;
  - $sson(v)$ – число вершин в ветви $G(v)$;
  - $d(v)$ – ограничение d на степени вершин;
  - $j(v)$ – номер строящейся линейки звездообразного дерева;
  - $first(v)$ – номер первого ребра строящейся линейки звездообразного дерева;
  - $nson(v)$ – число ожидаемых вершиной v сообщений *Конец*.
- Сообщения в алгоритме $\mathcal{B}(G, root)$: *Начало* (d, l), *Конец* $()$, *Перемещение* (t), *Изменение* (t), где d – ограничение на степень вершин, l – число вершин ветви, t число трансформаций, которые осталось сделать для построения линейки звездообразного дерева.

- Предусловие алгоритма $\mathcal{B}(G, root)$: Дерево G линейное с n вершинами и корнем *root* в одном из двух листьев. В каждой вершине v переменная $E(v)$ инициализирована корректно, т.е. множеством номеров рёбер, инцидентных вершине v. В корень по фиктивному ребру с номером 0 приходит сообщение *Начало* (d, n).
- Постусловие алгоритма $\mathcal{B}(G, root)$: Дерево G хорошее с корнем в вершине *root*. В каждой вершине v значение переменной $E(v)$ корректно. Из корня по фиктивному ребру с номером 0 послано сообщение *Конец* ().

Алгоритм $\mathcal{B}$ (*G*, *root*). Вершина *v*.

```
1  while не конец алгоритма {
2  wait () { /* приём сообщения */
3     if приём Начало(d, l) по ребру i {
4        d(v) := d; sson(v) := l; fat(v) := i; /* инициализация */
5        if fat(v) = 0 { nson(v) := min{d, sson(v) - 1}; }
6        else { nson(v) := min{d - 1, sson(v) - 1}; }
7        j(v) := nson(v); /* начинаем с последней линейки меньшей длины */
8        if sson(v) ≥ 3 { /* число вершин ≥ 3, нужно делать трансформацию
         */
9           if fat(v) = 0 { t := M(d, sson(v), j(v)); } else { t := N(d, s
            son(v), j(v)); }
10          first(v) := element( E(v) \ { i } ); /* выбор ребра */
11          посылка Перемещение(t) по ребру first(v);
12       }
13       else { /* число вершин ≤ 2, не нужно делать трансформацию */
14          посылка Конец() по ребру fat(v);
15          if fat(v) = 0 { конец алгоритма; } /* v = root */
16    }          }
17    if приём Перемещение(t) по ребру i {
18       e := element( E(v) \ { i } ); /* выбор ребра */
19       if t > 1 { /* первая трансформация цепочки трансформаций */
20          /* трансформация a→v→b, где e(v, a) = i, e(v, b) = e */
21          команда Изменить(i, e, t);
22          E(v) := E(v) \ { i }; /* ребро удалено */
23       }
24       else { /* завершающая трансформация */
25          /* трансформация a→v→b, где e(v, a) = e, e(v, b) = i */
26          команда Изменить(e, i, t);
27          E(v) := E(v) \ { e }; /* ребро удалено */
28    }          }
29    if приём Изменение(t) по ребру i {
30       if t > 2 { /* продолжаем цепочку трансформаций */
31          e := element( E(v) \ { fat(v), i } ); /* выбор ребра */
32          /* трансформация a→v→b, где e(v, a) = i, e(v, b) = e */
33          команда Изменить(i, e, t - 1);
34       }
35       if t = 2 { завершающая трансформация */
36          e := element( E(v) \ { fat(v), i } ); /* выбор ребра */
37          /* трансформация a→v→b, где e(v, a) = e, e(v, b) = i */
38          команда Изменить(e, i, t - 1);
39       }
40       if t = 1 { все трансформации закончены */
41          E(v) := E(v) ∪ { i }; /* ребро добавлено */
42          /* запуск алгоритма на построенной линейке */
```

```
43          if fat(v) = 0 { t := M(d, sson(v), j(v)); } else { t := N(d, s
            son(v), j(v)); }
44          if t ≥ 3 { посылка Начало(d(v), t) по ребру first(v); }
45          else { nson(v) := nson(v) - 1; }
46          j(v) := j(v) - 1; /* переходим к следующей линейке */
47          if fat(v) = 0 { t := M(d, sson(v), j(v)); } else { t := N(d, s
            son(v), j(v)); }
48          if j(v) > 1 { /* начинаем строить следующую линейку */
49             first(v) := i; посылка Перемещение(t) по ребру first(v);
50          }
51          if j(v) = 1 { /* все линейки построены */
52             /* запуск алгоритма на последней линейке */
53             if t ≥ 3 { посылка Начало(d(v), t) по ребру i; }
54             else { nson(v) := nson(v) - 1; }
55    }        }          }
56    if приём Конец() по ребру i {
57       nson(v) := nson(v) - 1;
58       if nson(v) = 0 { /* ветвь построена */
59          посылка Конец() по ребру fat(v);
60          if fat(v) = 0 { конец алгоритма; } /* v = root */
61 }}    }          }□
```

Утверждение 10. Алгоритм $\mathcal{B}$ на классе линейных деревьев с n вершинами с корнем в одном из листов не нарушает ограничения d ($d \ge 3$) на степень вершины и заканчивает работу с выполнением постусловия алгоритма за время $t(n) \le 2n - 2$.

Доказательство. Мы будем доказывать, что для линейки $G(v)$, ведущей из вершины v и содержащей n вершин, алгоритм после получения вершиной v сообщения *Начало* по ребру $fat(v)$ строит хорошее дерево, если $fat(v) = 0$, или почти хорошее дерево, если $fat(v) \ne 0$. В обоих случаях не нарушается ограничение d на степени вершин, корнем построенного дерева является вершина v, в каждой вершине w значение переменной $E(w)$ корректно, в конце работы алгоритма из вершины v по ребру $fat(v)$ посылается сообщение *Конец* (), и время работы алгоритма не превышает $2n - 2$. Доказательство будем вести индукцией по числу n. Напомним, что время вычислений в вершине не учитывается, и время выполнения алгоритма слагается из времени последовательных пересылок сообщений, включая сообщение *Изменение* при атомарной трансформации.

Для $n = 1$ дерево $G(v)$ состоит из одной изолированной вершины v и является как почти хорошим, так и хорошим деревом. Алгоритм не меняет дерево, $E(v) = \varnothing$, из корня по ребру с номером $fat(v)$ послано сообщение *Конец*. Время работы алгоритма без учёта пересылок сообщений *Начало* и *Конец*, очевидно, равно нулю: $t(1) = 0 = 2*1 - 2$.

Пусть утверждение верно для любого числа вершин не больше n, где $n \ge 1$, и докажем его для числа вершин $n + 1$. Сначала покажем, что алгоритм работает время не более $2(n + 1) - 2 = 2n$.

Пусть x – число соседей корня (почти) хорошего дерева с числом вершин n, а числа вершин ветвей соседей корня равны $l_1 \le \ldots \le l_x$. Очевидно, $1 \le l_1$ и $n = l_1 + \ldots + l_x$. Тогда $n \ge (x - 1) + l_x$, что влечёт $l_x \le n - x + 1$. Время $t_1(l_j)$ построения j-й линейки (почти) хорошего звездообразного дерева, содержащей l_j вершин, для $j = x \,..\, 2$, состоит из времени одного перемещения и l_j трансформаций. Поэтому $t_1(l_j) \le 1 + l_j$, а общее время этапа 1 равно $t_1(n + 1) \le (1 + l_1) + \ldots + (1 + l_{x-1}) = x - 1 + (n - l_x)$. На этапе 2 алгоритм трансформирует каждую линейку (почти) хорошего звездообразного дерева в почти хорошее дерево, начиная с пересылки сообщения *Начало* из вершины v в первую вершину линейки, соседней с корнем, и заканчивая пересылкой сообщения *Конец* в обратном направлении. На j-ую линейку тратится время $t_2(l_j) \le t(l_j) + 2$. Алгоритм выполняется на всех линейках параллельно, поэтому

общее время этапа 2 равно $t_2(n + 1) = max\{ t_2(l_j) \mid j = 1 .. x \} \le \le max\{ t(l_j) + 2 \mid j = 1 .. x \}$. Так как $l_j \le n$, по предположению шага индукции $t_2(n + 1) \le max\{ 2l_j - 2 + 2 \mid j = 1 .. x \} = 2l_x$. Тем самым, $t(n + 1) = = t_1(n + 1) + t_2(n + 1) \le x - 1 + (n - l_x) + 2l_x = x + n + l_x - 1$. Так как $l_x \le n - x + 1$, имеем $t(n + 1) \le x + n + (n - x + 1) - 1 = 2n$, что и требовалось доказать.

Докажем выполнение остальных условий. На этапе 1 строится хорошее звездообразное дерево, если $fat(v) = 0$, или почти хорошее звездообразное дерево, если $fat(v) \ne 0$. На этапе 2 по предположению шага индукции каждая из линеек этого дерева трансформируется в почти хорошее дерево. По определению (почти) хорошего звездообразного дерева, а также по утверждениями 7 и 5 на этапе 2 строится хорошее дерево, если $fat(v) = 0$, или почти хорошее дерево, если $fat(v) \ne 0$.

На этапе 1 ограничение d на степени вершин не нарушается, так как степень вершины v увеличивается ровно до того значения, которое она должна иметь в (почти) хорошем звездообразном дереве, а степени остальных вершин увеличиваются не более чем до 3. На этапе 2 ограничение d на степени вершин не нарушается по предположению шага индукции.

На этапе 1 корнем строящегося звездообразного дерева остаётся вершина v и она остаётся корнем строящегося на этапе 2 (почти) хорошего дерева. Значение переменных $E(w)$ поддерживается корректным в процессе всей работы алгоритма: при удалении (добавлении) ребра его номер удаляется из (добавляется в) $E(w)$. Алгоритм заканчивается посылкой сообщения *Конец*() из вершины v по ребру $fat(v)$.□

6. Заключение

В статье предложены два алгоритма самотрансформации дерева, лежащего в основе распределенной сети, с помощью локальных атомарных трансформаций, выполняемых по «командам» от узлов дерева. Это алгоритм трансформации любого дерева в линейное дерево и алгоритм трансформации линейного дерева в хорошее дерево, имеющее минимальный индекс Винера при том же числе вершин. Трансформации не меняют множество вершин дерева и не нарушают ограничение d ($d \ge 3$) на степени вершин.

Оба алгоритма имеют верхнюю оценку времени работы $2n - 2$, где n число вершин дерева. Однако, если для алгоритма $\mathcal{A}$ трансформации в линейное дерево эта оценка достижима, то для алгоритма $\mathcal{B}$ трансформации в хорошее дерево это не так. Это объясняется тем, что для алгоритма $\mathcal{A}$ оценка достигается на линейном дереве, а хорошее дерево алгоритм $\mathcal{A}$ трансформирует в линейное за меньшее время. Соответственно, и «обратный» алгоритм $\mathcal{B}$ трансформирует линейное дерево в хорошее меньше чем за $2n - 2$ тактов. Поэтому верхняя оценка для алгоритма $\mathcal{B}$ нуждается в уточнении (в сторону уменьшения).

Для минимизации индекса Винера дальнейшие исследования могут быть связаны с расширением класса графов, когда допускаются циклы. Предложенный подход самотрансформации графов, основанный на атомарной трансформации, может быть использован не только для максимизации или минимизации индекса Винера, но и для достижения других целей, когда используются другие критерии «оптимальности» графа. Кроме того, могут использоваться атомарные трансформации других типов, позволяющие удалять и добавлять вершины. Наконец, дополнительные проблемы возникают для ориентированных и смешанных графов, в которых сообщения можно пересылать по ориентированным ребрам только в направлении их ориентации.

Список литературы / References

[1]. Wiener H. Structural determination of paraffin boiling points. Journal of the American Chemical Society, vol. 69, no. 1, 1947, pp. 17–20.

[2]. Miranca Fischerman, Arne Hoffmann, Dieter Rautenbach, László Székely, Lutz Volkmann. Wiener index versus maximum degree in trees. Discrete Applied Mathematics, volume 122, issues 1–3, 2002, pp. 127–137.
[3]. L. A. Székely and Hua Wang. On Subtrees of Trees. Advances in Applied Mathematics, vol. 34, issue 1, 2005, pp. 138–155.
[4]. А.А. Кочкаров, Л.И. Сенникова, Р.А. Кочкаров. Некоторые особенности применения динамических графов для конструирования алгоритмов взаимодействия подвижных абонентов. Известия ЮФУ. Технические науки», № 1, 2015, стр. 207–214 / A.A. Kochkarov, L.I. Sennikova, R.A. Kochkarov. Some features of dynamic graphs applications for construction of mobile agents interaction algorithms. Izvestiya SFedU. Engineering sciences, № 1, 2015, pp. 207–214 (in Russian).
[5]. A.A. Kochkarov, R.A. Kochkarov, G.G. Malinetskii. Issues of dynamic graph theory. Computational Mathematics and Mathematical Physics, 2015, vol. 55, no. 9, pp. 1590–1596.
[6]. А.В. Проскочило, А.В. Воробьев, М.С. Зряхов, А.С. Кравчук. Анализ состояния и перспективы развития самоорганизующихся сетей. Научные ведомости Белгородского государственного университета, Экономика, Информатика, no. 36/1, вып. 19 (216), 2015, стр. 177-186 / A.V. Proskotchylo, A.V. Vorobyov, M.S. Zriakhov, A.S. Kravchuk. Analysis of state and development perspectives of self-organizing networks. Belgorod State University Scientific Bulletin. Economics, Information technologies, no. 19 (216), issue 36/1, 2015, pp. 177-186 (in Russian).
[7]. Zhi Chen, Shuai Li, and Wenjing Yue. SOFM Neural Network Based Hierarchical Topology Control for Wireless Sensor Networks. Journal of Sensors, vol. 2014, Article ID 121278, 6 p.
[8]. Chen Dongning, Zhang Ruixing, Yao Chengyu, and Zhao Zheyu. Dynamic topology multi force particle swarm optimization algorithm and its application. Chinese Journal of Mechanical Engineering, vol. 29, issue 1, 2016, pp. 124–135.
[9]. Simin Mo, Jian-Chao Zeng, Ying Tan. Particle Swarm Optimization Based on Self-organizing Topology Driven by Fitness. In Proc. of the International Conference on Computational Aspects of Social Networks, 2010, pp. 23–26.
[10]. Al-Sakib Khan Pathan (ed.) Security of self-organizing networks: MANET, WSN, WMN, VANET. CRC press, 2010, 638 p.
[11]. Boukerche A. (ed.) Algorithms and protocols for wireless, mobile Ad Hoc networks. John Wiley & Sons, 2008, 496 p.
[12]. Wen Chih-Yu and Hung-Kai Tang. Autonomous distributed self-organization for mobile wireless sensor networks. Sensors, vol. 9, issue 11, 2009, pp. 8961-8995.
[13]. Jaime Llorca, Stuart D. Milner, Christopher Davis. Molecular System Dynamics for Self-Organization in Heterogeneous Wireless Networks. EURASIP Journal on Wireless Communications and Networking, 2010, Article number: 548016.
[14]. Rangaswami Balakrishnan and S. Francis Raj. The Wiener number of powers of the Mycielskian. Discussiones Mathematicae Graph Theory, vol. 30, no. 3, 2010, p. 489–498.
[15]. Chen Wai-kai. Net Theory and its Applications: Flows in Networks. World Scientific Publishing, 2003, 672 p.
[16]. Hongzhuan Wang. On the Extremal Wiener Polarity Index of Hückel Graphs. Computational and Mathematical Methods in Medicine, vol. 2016, article ID 3873597. http://dx.doi.org/10.1155/2016/3873597.
[17]. Xiaoxin Xu, Yubin Gao, Yanbin Sang, and Yueliang Liang. On the Wiener Indices of Trees Ordering by Diameter-Growing Transformation Relative to the Pendent Edges. Mathematical Problems in Engineering, vol. 2019, article ID 8769428.
[18]. The On-Line Encyclopedia of Integer Sequences (OEIS). http://oeis.org/
[19]. Михаил Дехтярь. Основы дискретной математики. Лекция 10: Деревья. ИНТУИТ (национальный открытый университет). https://www.intuit.ru/studies/courses/1084/192/lecture/5017 / Mihail Dekhtyar'. The basics of discrete mathematics. Lecture 10: The trees. INTUIT (National Open University). Available at: https://www.intuit.ru/studies/courses/1084/192/lecture/5017 (in Russian).

Информация об авторе / Information about the author

Игорь Борисович БУРДОНОВ – доктор физико-математических наук, ведущий научный сотрудник. Научные интересы: формальные спецификации, генерация тестов, технология компиляции, системы реального времени, операционные системы, объектно-

ориентированное программирование, сетевые протоколы, процессы разработки программного обеспечения.

Igor Borisovich BURDONOV – Doctor of Physical and Mathematical Sciences, Leading Researcher. Research interests: formal specifications, test generation, compilation technology, real-time systems, operating systems, object-oriented programming, network protocols, software development processes.

DOI: 10.15514/ISPRAS-2019-31(4)-14

Задача поиска путей в ациклических графах с ограничениями в терминах булевых грамматик

[1] *Е.Н. Шеметова, ORCID: 0000-0002-1577-8347 <katyacyfra@gmail.com>*
[2] *С.В. Григорьев, ORCID: 0000-0002-7966-0698 <s.v.grigoriev@spbu.ru>*
[1] *Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики, 197101, Россия, г. Санкт-Петербург, Кронверкский пр., 49*
[2] *Санкт-Петербургский государственный университет, 199034, Россия, г. Санкт-Петербург, Университетская наб., д. 7/9*

Аннотация. Графовая модель данных активно используется в научных и прикладных областях, например, в графовых базах данных, биоинформатике, при анализе социальных сетей и в статическом анализе кода. Одной из основных задач, связанных с графовыми моделями, является поиск специфичных путей в графе. Естественным способом задать ограничения на пути являются формальные грамматики над метками рёбер графа, при этом запрос к графу может быть представлен в виде множества всех троек (A, v_1, v_2), для которых существует путь в графе от вершины v_1 до вершины v_2 такой, что метки на ребрах этого пути образуют строку, выводимую из нетерминала A в данной грамматике. Использование булевых грамматик позволяет формулировать более выразительные запросы по сравнению с традиционно используемыми регулярными и контекстно-свободными грамматиками. Известно, что задача выполнения запросов к графу с использованием булевых грамматик является неразрешимой. В данной работе предложен приближённый алгоритм поиска путей в ориентированных графах без циклов с ограничениями, заданными с помощью булевых грамматик. Благодаря ограничению на тип анализируемых графов, предложенный алгоритм является более асимптотически оптимальным, чем наивный итерационный алгоритм.

Ключевые слова: поиск путей с ограничениями; булевы грамматики; матричные операции; ациклический граф; булевы матрицы; произведение матриц.

Для цитирования: Шеметова Е.Н., Григорьев С.В. Задача поиска путей в ациклических графах с ограничениями в терминах булевых грамматик. Труды ИСП РАН, том 31, вып. 4, 2019 г., стр. 211-226. DOI: 10.15514/ISPRAS-2019-31(4)-14

Благодарности. Данная работа выполнена при поддержке гранта РНФ 18-11-00100 и JetBrains Research. Авторы выражают признательность Кознову Дмитрию Владимировичу за оказанную помощью при написании настоящей статьи.

Path querying on acyclic graphs using Boolean grammars

[1] *E.N. Shemetova, ORCID: 0000-0002-1577-8347 <katyacyfra@gmail.com>*
[2] *S.V. Grigorev, ORCID: 0000-0002-7966-0698 <s.v.grigoriev@spbu.ru>*

[1] *Saint Petersburg National Research University of Information Technologies, Mechanics and Optics, 49, Kronverkskiy pr., St. Petersburg, 197101, Russia*
[2] *Saint Petersburg State University, 7/9, Universitetskaya nab., St. Petersburg, 199034, Russia*

Abstract. Graph data models are widely used in different areas of computer science such as bioinformatics, graph databases, social networks and static code analysis. One of the problems in graph data analysis is querying for specific paths. Such queries are usually performed by means of a formal grammar that describes the allowed edge-labeling of the paths. Path query is said to be calculated using relational query semantics if it is evaluated to triple (A, v_1, v_2), such that there is a path from v_1 to v_2 such that the labels on the edges of this path form a string derivable from the nonterminal A. As the regular and context-free languages have limited expressive power, we focus on a more expressive languages, namely the Boolean languages that use Boolean grammars to describe the labeling of paths. Although path querying using relational query semantics and Boolean grammars is known to be undecidable, in this work we propose a path querying algorithm on acyclic graphs which uses relational query semantics and Boolean grammars and approximates the exact solution. To achieve better performance in compare with the naive algorithm, considered classes of graphs were limited to acyclic graphs.

Keywords: path querying; Boolean grammars; matrix operations; acyclic graph; DAG; boolean matrix; matrix multiplication.

For citation: Shemetova E.N., Grigorev S.V. Path querying on acyclic graphs using Boolean grammars. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 4, 2019. pp. 211-226 (in Russian). DOI: 10.15514/ISPRAS-2019-31(4)-14

Благодарности. This work was supported by a grant from the Russian Science Foundation 18-11-00100 and also by JetBrains Research. The authors are grateful to Dmitry Vladimirovich Koznov for the assistance in writing this article.

1. Введение

Графовая модель данных широко используется в различных областях, например, в графовых базах данных [1, 2], биоинформатике [3], моделировании и анализе социальных сетей [4, 5], в статическом анализе программного кода [6, 7, 8], а также в задачах на основе статического анализа в контексте проблемы промышленного реинжиниринга устаревших приложений [30].

Одной из важных задач анализа данных, представленных в виде графа, является задача поиска специфичных путей, например, с помощью формальных языков: если рёбра графа содержат метки, то путь задаёт слово, которое получается конкатенацией меток вдоль него. Таким образом, в качестве критерия для поиска можно использовать факт принадлежности полученного слова заданному языку. В результате возникает задача поиска путей с ограничениями, заданными в терминах формальных языков.

Многие современные языки запросов к графам, такие как Cypher [9], Gremlin [10], SPARQL [11], предоставляют средства задания ограничений в терминах регулярных языков или регулярных выражений. Однако этого недостаточно для решения многих задач статического анализа кода, например, для анализа иерархических зависимостей и поиска подобных элементов.

В результате возникает необходимость использовать более выразительные классы языков для задания ограничений – контекстно-свободные, конъюнктивные [12], булевы [13].

Вопрос использования контекстно-свободных грамматик в качестве ограничений активно исследуется в настоящее время [14, 15, 16]. При использовании более выразительных языков возникает вопрос о разрешимости задачи выполнения запросов к графу. Показано, что задача поиска путей в произвольных ориентированных графах с ограничениями в виде конъюнктивных и булевых языков неразрешима [15], однако для конъюнктивных языков предложен алгоритм, строящий приближение ответа сверху [20], что делает его применимым для приближённого решения прикладных задач. При этом, предложенный алгоритм является наивным итеративным алгоритмом. Возможность построения алгоритма, строящего приближённое решение для задачи поиска путей с ограничениями в виде булевых языков, не исследована.

Также отметим, что представленные выше алгоритмы работают с графами произвольной структуры. Для некоторых научных областей одними из наиболее часто анализируемых типов графов являются ациклические графы и деревья. Например, в статическом анализе кода это абстрактное синтаксическое дерево (AST), дерево вызовов с контекстами (CCT, calling context tree) [29] и другие, социальные иерархии в анализе социальных сетей представлены в виде ациклического графа [28]. Если исходный граф является ациклическим, то можно воспользоваться известными свойствами ациклических графов, чтобы предоставить более производительный алгоритм для решения задачи поиска путей с ограничениями для данной структуры графа. Пример подобного подхода рассмотрен в работе [8].

Несмотря на то, что задача является неразрешимой для произвольных ориентированных графов, для ориентированных графов без циклов она разрешима [15]. Однако для получения точного решения необходимо рассмотреть все пути в таком графе, число которых, как известно, экспоненциально зависит от числа вершин в графе. Одним из способов получения асимптотически более быстрого алгоритма является построение алгоритма, находящего приближённое решение задачи сверху. Такой алгоритм позволит значительно уменьшить множество возможных решений. Тогда, запустив на полученном множестве наивный алгоритм, можно найти точное решение задачи за меньшее время.

В данной работе предложен приближённый алгоритм поиска путей в ориентированных графах без циклов (Direct Acyclic Graph, DAG) с ограничениями в виде булевых грамматик. При этом алгоритм реализует так называемую реляционную семантику [14]: результатом работы алгоритма является отношение на вершинах графа и нетерминалах. Элемент отношения – тройка (A, v_1, v_2) означает, что в заданном графе существует путь из вершины v_1 в вершину v_2, такой, что соответствующее ему слово выводимо из нетерминала A в заданной грамматике. Доказано, что данный алгоритм строит приближение сверху. Благодаря ограничению на тип анализируемых графов удалось построить более асимптотически оптимальный, чем наивный итерационный, алгоритм, использующий идеи Валианта [21] и Охотина [22].

Работа организована следующим образом. В разд. 2 даны основные определения, связанные с задачей поиска путей с ограничениями в терминах булевых грамматик. В разд. 3 проанализированы существующие решения данной задачи. В разд. 4 представлена адаптация алгоритма Охотина [22], находящая аппроксимацию решения задачи поиска путей с ограничениями в терминах булевых грамматик с использованием реляционной семантики запросов для ациклических графов. Доказана также корректность применения данного алгоритма для поставленной задачи. В разд. 5 работа предложенного алгоритма продемонстрирована на примере. Заключение и направления будущих исследований приведены в разд. 6.

2. Основные определения

2.1 Терминология

Для начала определим задачу поиска путей с ограничениями в терминах булевых грамматик с использованием реляционной семантики запросов.

Рассмотрим ориентированный ациклический граф $D = (V, E)$ и формальную грамматику G. Пусть у каждого ребра графа есть метка; множество всех меток обозначим Σ. Тогда каждый путь в D будет обозначать слово над алфавитом из Σ, полученное конкатенацией меток рёбер, включенных в этот путь. На рис. 1 изображен помеченный ациклический ориентированный граф с $\Sigma = \{a, b, c\}$.

Для графа D и формальной грамматики $G = (\Sigma, N, P)$, для любого $A \in N$ обозначим отношения $R_A \subseteq V \times V$ следующим образом:

$$R_A = \{(n, m) \mid \exists\, n\pi m\ (l(\pi) \in L(G_A))\},$$

где $n\pi m$ — это путь из вершины n в m, $l(\pi)$ — слово, полученное конкатенацией меток рёбер, принадлежащих пути π, а $L(G_A)$ обозначает язык, порожденный грамматикой G со стартовым нетерминалом A.

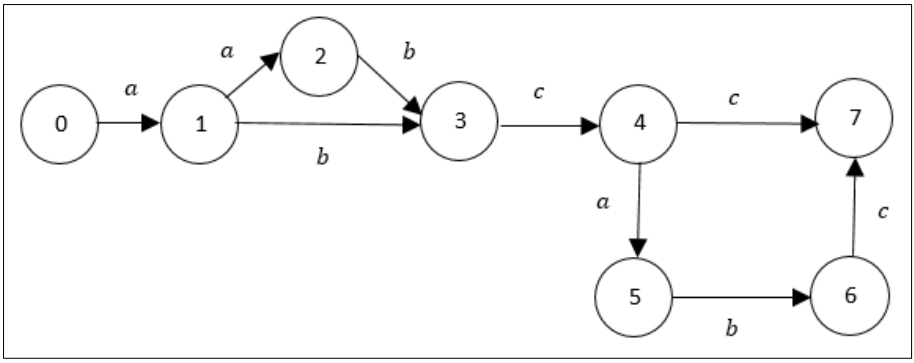


Рис. 1. Пример помеченного ориентированного ациклического графа с $\Sigma = \{a, b, c\}$
Fig. 1. Example of directed labelled acyclic graph with $\Sigma = \{a, b, c\}$

Таким образом, задача поиска путей в графе D с ограничениями в терминах формальной грамматики G и с использованием реляционной семантики запросов сводится к нахождению всех троек (A, n, m), таких, что существует путь $n\pi m$, а строка $l(\pi)$ выводима из нетерминала A в грамматике G, и это означает вычисление всех отношений R_A для любого $A \in N$.

В качестве формальной грамматики G мы будем использовать булевы грамматики [13]. Булева грамматика является классом формальных грамматик, расширяющим класс контекстно-свободных грамматик с помощью логических операций конъюнкции и отрицания.

Булева грамматика G формально определяется следующим образом: $G = (\Sigma, N, P)$, где Σ – терминальный алфавит, N – нетерминальный алфавит (множество нетерминальных символов $\{A_1, A_2, \ldots, A_n\}$), P – множество правил грамматики.

Тогда правила булевой грамматики можно представить следующим образом:

$$A \to \alpha_1 \& \alpha_2 \& \ldots \& \alpha_m \& \neg\beta_1 \& \neg\beta_2 \& \ldots \& \neg\beta_n\ ,$$

где A – это нетерминал, $m + n \geq 1$, $\alpha_1, \alpha_2, \ldots \alpha_m, \beta_1, \beta_2, \ldots \beta_n \in (\Sigma \cup N)^*$.

Отметим, что в данном определении не выделен стартовый нетерминал, так как его можно будет определить во время поиска путей с ограничениями.

На рис. 2 показана булева грамматика для языка $L(G_S) = \{a^k bc \mid k \neq 1\}$. В данном случае коньюнкт DC порождает язык $L(G_{DC}) = \{a^k b^i c^j \mid i = j = 1, k \geq 0\}$, а конъюнкт AB

порождает язык $L(G_{AB}) = \{a^k b^i c^j \mid k = i = 1,\ j \geq 0\}$. Тогда $L(G_S) = L(G_{DC}) \cap \overline{L(G_{AB})} = \{a^k b^i c^j \mid i = j = 1 \text{ и } k \neq i \neq 1\} = \{a^k bc \mid k \neq 1\}$.

$$
\begin{array}{|l|}
\hline
S \to DC\&\neg AB \\
A \to a \\
B \to b \\
C \to c \\
D \to b \\
B \to BC \\
D \to AD \\
\hline
\end{array}
$$

Рис. 2. Пример булевой грамматики
Fig. 2. Example of Boolean grammar

Булева грамматика $G = (\Sigma, N, P)$ находится в *двоичной нормальной форме*, если каждое правило в P имеет следующий вид:

- $A \to B_1 C_1 \& \ldots \& B_m C_m \& \neg D_1 E_1 \& \ldots \& \neg D_n E_n$ $(m \geq 1, n \geq 0)$, или
- $A \to a \quad (a \in \Sigma)$.

Например, грамматика на рис. 2 находится в двоичной нормальной форме.

Любая булева грамматика может быть переведена в эквивалентную ей булеву грамматику в двоичной нормальной форме [13]. Далее будем считать, что все рассматриваемые булевы грамматики находятся в двоичной нормальной форме.

2.2 Алгоритм Охотина

В данной работе предложено расширение алгоритма Охотина [22], являющегося алгоритмом синтаксического анализа для булевых грамматик. Алгоритм Охотина строит для входной строки $a_1 a_2 \ldots a_n$ и булевой грамматики $G = (\Sigma, N, P)$ в двоичной нормальной форме таблицу синтаксического анализа T, где элемент $T_{i,j}$ — это набор нетерминалов, выводящий подстроку $a_{i+1} \ldots a_j$ входной строки, а $0 \leq i < j \leq n$. Входная строка $a_1 a_2 \ldots a_n$ принадлежит языку $L(G_S)$ тогда и только тогда, когда $S \in T_{0,n}$. Построение таблицы T в данном алгоритме сведено к умножению булевых матриц различных размеров. Помимо таблицы T, алгоритм использует дополнительную структуру данных — таблицу M, каждый её элемент $M_{i,j}$ принадлежит множеству пар нетерминалов $N \times N$, таких, что:

$$M_{i,j} = \{(B, C) \mid a_{i+1}, \ldots, a_j \in L_G(B), L_G(C)\} \text{ для всех } B, C \in N \text{ и } 0 \leq i < j \leq n.$$

Используя значения $M_{i,j}$, можно получить набор нетерминалов, порождающих строку:

$$T_{i,j} = f(M_{i,j}),$$

где $f\colon 2^{N \times N} \to 2^N$ определена следующим образом:

$$f(M) = \{A \mid \exists A \to B_1 C_1 \& \ldots \& B_m C_m \& \neg D_1 E_1 \& \ldots \& \neg D_{m'} E_{m'} \in P\colon (B_t, C_t) \in M \text{ и } (D_t, E_t) \notin M \text{ для } \forall t.$$

3. Родственные работы

Наиболее популярными запросами к графам являются запросы, использующие регулярные грамматики [2, 26, 27]. Вопрос использования контекстно-свободных грамматик в качестве ограничений для поиска путей активно исследуется в настоящее время [14, 16]. Предложены эффективные алгоритмы выполнения соответствующих запросов к графам [17, 18, 19]. Алгоритм выполнения запросов с ограничениями в терминах контекстно-свободных грамматик, основанный на синтаксическом анализе «сверху вниз» [32], представлен в работе [31]. В работе [15] изучены вопросы разрешимости задачи выполнения запросов к графам для конъюнктивных и булевых грамматик. Для конъюнктивных языков предложен алгоритм, строящий приближение ответа сверху [20], что делает его применимым для приближённого

решения прикладных задач. В работе [7] описан алгоритм, работающий с линейными конъюнктивными грамматиками, которые имеют не более одного нетерминального символа в каждом конъюнкте правила.

Несмотря на то, что булевы грамматики являются наиболее выразительными по сравнению с вышеупомянутыми грамматиками [13], возможность построения алгоритма, строящего приближённое решение для задачи поиска путей с ограничениями в виде булевых языков, не исследована.

4. Алгоритм

В работе [22] предложен алгоритм синтаксического анализа для булевых грамматик, основанный на матричных операциях. В данном разделе будет предложено расширение этого алгоритма для нахождения приближенного решения задачи поиска путей в ациклических графах с ограничениями в терминах булевых грамматик и использованием реляционной семантики, а также показана его корректность.

Матричные алгоритмы синтаксического анализа позволяют достичь более высокой производительности за счет использования эффективных методов перемножения матриц [21, 22]. Также за счет использования матриц возможно более компактное хранение данных. Например, в случае если входной ациклический граф является деревом, между любой парой вершин графа будет существовать всего один путь. Если же ациклический граф не является деревом, то между одной парой вершин может быть в худшем случае $O(2^n)$ путей, поэтому хранить и обрабатывать данные для каждого пути отдельно неэффективно. Так как информация о всех путях из вершины i в вершину j хранится в одной ячейке, то с помощью конъюнкции могут быть объединены нетерминалы, соответствующие разным путям из i в j, но при этом будут объединены также нетерминалы, соответствующие одному и тому же пути, что и требуется для решения задачи. В итоге результат работы алгоритма будет содержать приближение ответа сверху.

4.1 Описание алгоритма

Рассматриваемый алгоритм решает задачу на следующих входных данных:

- помеченном ориентированном ациклическом графе $D = (V, E)$ с n вершинами, без потери общности предположим, что n является степенью двойки и
- $G = (\Sigma, N, P)$ — булевой грамматике в двоичной нормальной форме.

Результатом работы алгоритма является матрица T; каждый её элемент $T_{i,j}$ содержит множество нетерминалов $\{A_1, A_2, \dots, A_m\}$, где $A_k \in N$, являющееся надмножеством множества нетерминалов, таких, что $(i, j) \in R_{A_k}$.

Алгоритм использует следующие структуры данных.

- **Таблица T** размером $n \times n$, представляющая собой верхнетреугольную матрицу, где каждый элемент $T_{i,j}$ является множеством нетерминалов.

  Пусть $X \in (2^N)^{m \times l}$ и $Y \in (2^N)^{l \times n}$ — матрицы, элементами которой являются подмножества N, а $m, l, n \geq 1$. Тогда *произведением матриц* $X \times Y$ будет матрица $Z \in (2^N)^{m \times n}$, такая, что каждый её элемент $Z_{i,j}$ вычисляется по следующей форме:

  $$Z_{i,j} = \bigcup_{k=1}^{l} X_{i,k} \times Y_{k,j}$$

  Данное произведение можно представить в виде произведения $|N|^2$ булевых матриц: для всех пар нетерминалов $B, C \in N$ рассмотрим булеву матрицу Z^{BC}, элемент $Z^{BC}_{i,j}$ которой означает наличие пары (B, C) в ячейке матрицы $Z_{i,j}$. При этом Z^{BC} можно получить как произведение булевых матриц Z^B и Z^C. Благодаря такому переходу становится

возможным свести решение задачи к вычислению произведения булевых матриц, для осуществления которого существуют весьма эффективные алгоритмы [23, 24].

- **Таблица M**, каждый элемент $M_{i,j}$ которой принадлежит множеству пар нетерминалов $N \times N$.

```
/*   D — входной помеченный ориентированный
     ациклический граф
/*   G — входная булева грамматика в двоичной
     нормальной форме
1:   Main:
2:   n ⟵ число вершин в D
3:   E ⟵ {(i, j, a) | ребро из вершины i в j,
     помеченное символом a входит во множество рёбер
     D}
4:   N ⟵ множество нетерминалов грамматики G
5:   P ⟵ множество правил грамматики G
6:   T ⟵ пустая матрица размером n × n
7:   M ⟵ пустая матрица размером n × n
8:   D ⟵ TopologicalSorting(D)
9:   for each (i, j, a) ϵ E
10:    T_{i,j} ⟵ {A | A → a ϵ P}
11:  compute(0, n + 1)

12:  compute(l, m):
13:  if m − l ≥ 4 then
14:      compute(l, (l+m)/2)
15:      compute((l+m)/2, m)
16:  complete(l, (l+m)/2, (l+m)/2, m)

17:  complete(l, m, l′, m′)
18:  if m − l = 1 and m < l′ then
19:      T_{l,l′} ⟵ f(M_{l,l′})
20:  else if m-l > 1 then
21:      B ⟵ (l, (l+m)/2, (l+m)/2, m)
22:      B′ ⟵ (l′, (l′+m′)/2, (l′+m′)/2, m′)
23:      C ⟵ ((l+m)/2, m, l′, (l′+m′)/2)
24:      D ⟵ (l, (l+m)/2, l′, (l′+m′)/2)
25:      D′ ⟵ ((l+m)/2, m, (l′+m′)/2, m′)
26:      E ⟵ (l, (l+m)/2, (l′+m′)/2, m′)
27:      complete(C)
28:      M_D ⟵ M_D ∪ (T_B × T_C)
29:      complete(D)
30:      M_{D′} ⟵ M_{D′} ∪ (T_C × T_{B′})
31:      complete(D′)
32:      M_E ⟵ M_E ∪ (T_B × T_{D′})
33:      M_E ⟵ M_E ∪ (T_D × T_{B′})
34:      complete(E)
```

Листинг. 1. Алгоритм поиска путей в ациклических графах с ограничениями в терминах булевых грамматик
Listing. 1. Path querying on acyclic graphs using Boolean grammars

Используя значения $M_{i,j}$ и правила грамматики, из множества пар нетерминалов можно получить множество нетерминалов:

$$T_{i,j} = f(M_{i,j}),$$

где $f: 2^{N\times N} \to 2^N$ для булевых грамматик определяется как

$f(M) = \cup_{k=1}^{2^{|M|}}\{A \mid \exists\, A \to B_1C_1 \& \ldots \& B_mC_m \& \neg D_1E_1 \& \ldots \& \neg D_{m'}E_{m'} \in P: (B_t, C_t) \in M^k$ и $(D_t, E_t) \notin M^k$ для $\forall t$,

где M^k является подмножеством множества M.

Псевдокод модифицированного алгоритма Охотина приведен в листинге 1.

Элементы матрицы M рассчитываются группами с помощью выше определенного произведения подматриц из таблицы T, дающих аналогичный результат с поэлементным умножением. Схема расположения подматриц представлена на рис. 3 и 4.

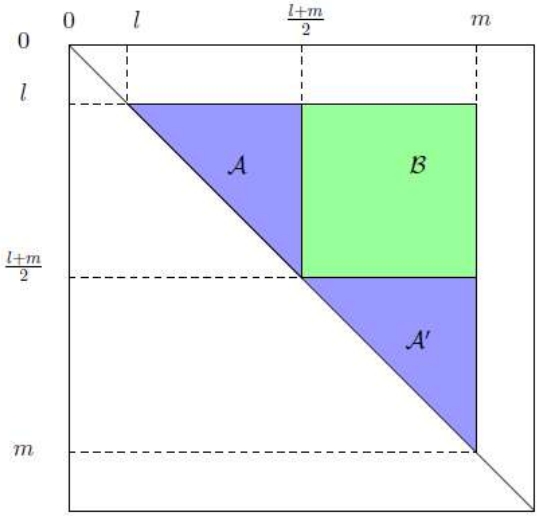


Рис. 3. Расположение подматриц для расчёта процедурой compute элементов таблицы M
Fig. 3. Submatrices for calculating elements of M by «compute» procedure

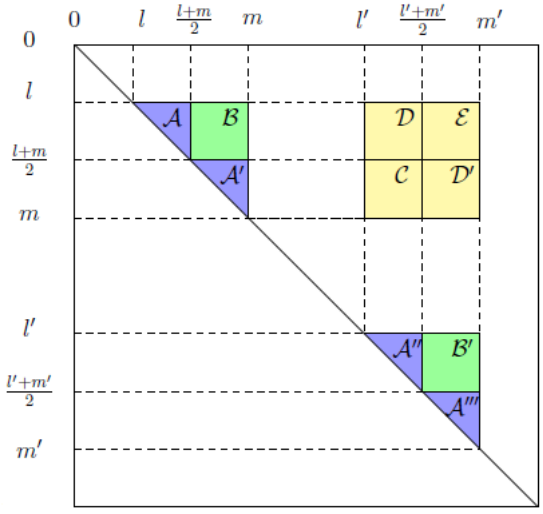


Рис. 4. Расположение подматриц для расчёта процедурой complete элементов таблицы M
Fig. 4. Submatrices for calculating elements of M by «complete» procedure

Алгоритм состоит из двух следующих рекурсивных процедур.

- $compute(l, m)$ – рассчитывает значения $T_{i,j}$ для любых $l \le i < j < m$.
- $complete(l, m, l', m')$ – определена для $l \le m < l' \le m'$ при $m - l = m' - l'$ и $m - l$ без потери общности являющимися степенью двойки.

Четыре входных параметра процедуры обозначают координаты подматрицы матрицы T, содержащей все элементы $T_{i,j}$, где $l \le i < j < m$ и $l' \le j < m'$.

Координаты интерпретируются следующим образом: l, m – это индексы начальной и конечной строки в таблице T, l', m' — индексы начального и конечного столбца. Координаты обозначают пути в графе, номер начала которых находится между l и m, а конец — между l' и m'.

Так как ранее уже были вычислены $T_{i,j}$ для $l \le i < j < m$ и $l' \le i < j < m'$, а также $M_{i,j}$ для $l \le i < m$ и $l' \le j < m'$, то процедура $complete(l, m, l', m')$ получает значения $T_{i,j}$ для $l \le i < m$ и $l' \le j < m'$.

Основные структуры данных, с которыми работает алгоритм, являются верхнетреугольными матрицами, поэтому на вершинах графа D должен быть задан линейный порядок, любое его ребро ведёт от вершины с меньшим номером к вершине с большим номером. Если D является ациклическим графом, то данного свойства легко добиться с помощью топологической сортировки графа. Например, алгоритм Тарьяна [25] позволяет осуществлять топологическую сортировку за время $O(V + E)$, где V — число вершин в графе, а E — число рёбер.

4.2 Корректность алгоритма и оценка сложности

Как говорилось выше, результатом работы алгоритма является матрица T, в ячейках которой будет содержаться некоторое множество нетерминалов. Покажем, что это множество будет содержать решение задачи.

Теорема 1. (*Корректность алгоритма для поиска путей в ациклических графах с ограничениями в терминах булевых грамматик с использованием реляционной семантики запросов*). Пусть даны помеченный ориентированный ациклический граф $D = (V, E)$ и булева грамматика $G = (\Sigma, N, P)$. Тогда для любых вершин i, j и для любого нетерминала $A \in N$, если $(i, j) \in R_A$, то $A \in T_{i,j}$.

Доказательство. Рассмотрим такие вершины i, j, что $(i, j) \in R_A$ и существует $i\pi j$, такой, что $l(\pi) \in L(G_A)$ (т.е. существует дерево разбора для строки $l(\pi)$ и грамматики G с корнем в нетерминале A). Докажем утверждение теоремы индукцией по высоте дерева разбора строки $l(\pi)$.

База индукции. Если (i, j) – ребро графа (высота дерева разбора равна 1), то алгоритм корректен, исходя из инициализации матрицы T (строки 9-10 листинга 1).

Индукционный переход. Предположим, что утверждение верно для всех деревьев разбора высотой h. Докажем, что теорема верна для деревьев разбора высотой $h + 1$. Рассмотрим дерево разбора для $l(\pi)$ высотой $h + 1$. Так как грамматика находится в двоичной нормальной форме, то у данного дерева будут поддеревья, выводящие подстроки $l(\pi)$ (возможно, пересекающиеся). По свойству топологической сортировки для всех индексов k, m этих подстрок (начала и конца соответствующих путей) выполняется неравенство $i \le k < m \le j$. $T_{i,j}$ вычисляется как $f(M_{i,j})$. $M_{i,j}$, исходя из построения алгоритма, может быть получено тремя способами, в зависимости от положения подматрицы, в которой оно задано (строки 28, 30, 32-33 листинга 1). Пусть $M_{i,j}$ — ячейка матрицы M_Z. В любом из случаев M_Z вычисляется как $M_Z \cup (T_X \times T_Y)$. По определению произведения $T_X \times T_Y$, для каждого пути из i в j будут получены все произведения нетерминалов из конъюнктов, выводящих все подстроки $l(\pi)$, такие,что путь $i\pi j$ разбит на две части вершиной k, такой, что $i < k < j$.

По индукционному предположению, то есть если $(i,k) \in R_B$, то $B \in T_{i,k}$ и если $(k,j) \in R_C$, то $C \in T_{k,j}$. Тогда произведение матриц $T_X \times T_Y$ даст все возможные пары нетерминалов $\{(B,C)\} \in M_{i,j}$ для всех подстрок. После применения функции f (подбора соответствующего правила грамматики), $T_{i,j}$ будет содержать нетерминалы, полученные после применения правил. Тогда, исходя из правил грамматики, получим, что если $(i,j) \in R_A$, то $A \in T_{i,j}$ (объединяем существующие поддеревья для подстрок в одно дерево). А это значит, что утверждение верно для дерева разбора высотой $h+1$. □

Сложность алгоритма вычисляется так же, как и для алгоритма Охотина [22], и составляет $O(|G|BMM(n)\log n)$, где $|G|$ — размер входной булевой грамматики, n — число вершин в графе D, $BMM(n)$ — время умножения булевых матриц размера $n \times n$.

5. Пример работы алгоритма

В данном разделе мы продемонстрируем работу предложенного алгоритма на небольшом примере.

В качестве входного графа будет использован граф, изображенный на рис. 1, а в качестве входной булевой грамматики – грамматика с рис. 2.

В строках 9-10 листинга 1 происходит инициализация таблицы T, заполнение происходит на основании информации о рёбрах графа. Состояние T после инициализации показано на рис. 5.

	0	1	2	3	4	5	6	7
0		$\{A\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1			$\{A\}$	$\{B,D\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
2				$\{B,D\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
3					$\{C\}$	$\emptyset$	$\emptyset$	$\emptyset$
4						$\{A\}$	$\emptyset$	$\{C\}$
5							$\{B,D\}$	$\emptyset$
6								$\{C\}$
7								

Рис. 5. Таблица T после инициализации
Fig. 5. The matrix T after initialization

После инициализации произойдет ряд рекурсивных вызовов процедур $compute$ и $complete$. Дерево рекурсивных вызовов показано на рис. 6.

Вызовы $compute(0,2), compute(2,4), compute(4,6)$ и $compute(6,8)$ из-за небольшого размера задачи будут обработаны тривиально – вызовом процедур $complete(0,1,1,2), complete(2,3,3,4), complete(4,5,5,6)$ и $complete(6,7,7,8)$ соответственно (строка 16 листинга 1). Так как для каждого из вызовов выполняется равенство $m = l'$ для координат подматриц, в соответствующих ячейках матрицы останутся текущие множества нетерминалов без изменений.

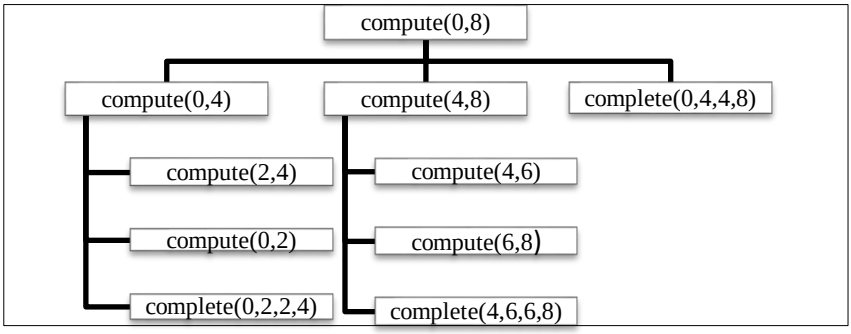


Рис. 6. Дерево рекурсивных вызовов процедур compute и complete
Fig. 6. Recursive call tree for «compute» and «complete»

На рис. 7 показаны подматрицы T, для которых будут производиться дальнейшие вычисления.

	0	1	2	3	4	5	6	7
0		$\{A\}$	complete(0,2,2,4)		complete(0,4,4,8)			
1								
2				$\{B,D\}$				
3								
4						$\{A\}$	complete(4,6,6,8)	
5								
6								$\{C\}$
7								

Рис.7. Подматрицы T, для которых будет вызвана процедура complete
Fig. 6. Submatrices of T which will be processed by «complete».

Разберем подробнее вызов $complete(0,2,2,4)$.

1. Сначала будут определены координаты подматриц матрицы T (строки 21-26). $B = (0,1,1,2), B' = (2,3,3,4), C = (1,2,2,3), D = (0,1,2,3), D' = (1,2,3,4)$.
2. Затем произойдет вызов $complete(C) = complete(1,2,2,3)$. В результате, множество в ячейке $T_{1,2}$ останется неизменным, $T_{1,2} = \{A\}$.
3. Далее вычисляем ячейку $M_{0,2}$ таблицы M (строка 28 листинга 1). $M_{0,2} = M_D = M_D \cup (T_B \times T_C) = M_{0,2} \cup (T_{0,1} \times T_{1,2}) = \{(A,A)\}$.
4. Выполняем $complete(D) = complete(0,1,2,3)$. Согласно строкам 18-19 листинга 1, $T_{0,2} = f(M_{0,2}) = \emptyset$, так как во входной грамматике отсутствует подходящее правило.
5. $M_{D'} = M_{1,3} = M_D, \cup (T_C \times T_{B'}) = M_{1,3} \cup (T_{1,2} \times T_{2,3}) = \{B,D\} \cup (\{A\} \times \{B,D\}) = \{(B),(D),(A,B),(A,D)\}$.
6. Выполняем $complete(D') = complete(1,2,3,4)$. Согласно строкам 18-19 листинга 1, $T_{1,3} = f(M_{1,3}) = \{B,D\}$.
7. Вычисляем ячейку $M_{0,3}$ (строки 32-33).
   $M_{0,3} = M_E = M_E \cup (T_B \times T_{D'}) = M_{0,3} \cup (T_{0,1} \times T_{1,3}) = \{(A,B),(A,D)\}$.
   $M_{0,3} = M_E \cup (T_D \times T_{B'}) = M_{0,3} \cup (T_{0,2} \times T_{2,3}) = \{(A,B),(A,D)\} \cup (\emptyset \times \{B,D\}) = \{(A,B),(A,D)\}$.
8. Выполняем $complete(E) = complete(0,1,3,4)$.
   $T_{0,3} = f(M_{0,3}) = \{D\}$.

Вычисление $complete(4,6,6,8)$ происходит аналогично. Состояние таблицы T после окончания работы процедур $complete(0,2,2,4)$ и $complete(4,6,6,8)$ приведено на рис. 8.

	0	1	2	3	4	5	6	7
0		$\{A\}$	$\emptyset$	$\{D\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1			$\{A\}$	$\{B,D\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
2				$\{B,D\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
3					$\{C\}$	$\emptyset$	$\emptyset$	$\emptyset$
4						$\{A\}$	$\{D\}$	$\{C,S\}$
5							$\{B,D\}$	$\{B,S\}$
6								$\{C\}$
7								

Рис. 8. Таблица T после окончания работы complete(0,2,2,4) и complete(4,6,6,8)
Fig. 8. The matrix T after complete(0,2,2,4) и complete(4,6,6,8)

Теперь всё готово для работы процедуры $complete(0,4,4,8)$. Разбиение T на подматрицы показано на рисунке 9, синим цветом выделены матрицы B (левый верхний угол) и матрица B' (правый нижний угол).

	0	1	2	3	4	5	6	7
0		$\{A\}$	$\emptyset$	$\{D\}$	$\mathcal{D}$		$\mathcal{E}$	
1			$\{A\}$	$\{B,D\}$				
2				$\{B,D\}$	$\mathcal{C}$		$\mathcal{D}'$	
3								
4						$\{A\}$	$\{D\}$	$\{C,S\}$
5							$\{B,D\}$	$\{B,S\}$
6								$\{C\}$
7								

Рис. 9. Разбиение на подматрицы таблицы T для вычисления complete(0,4,4,8)
Fig. 9. Partition of the matrix T for complete(0,4,4,8)

После определения координат подматриц будет вызвана процедура $complete(\mathrm{C}) = complete(2,4,4,6)$. Расчёт элементов T_C производится аналогично ранее описанным расчетам для матриц 2×2, поэтому сразу приведем результат:

$$T_C = \begin{pmatrix} \{B,S\} & \emptyset \\ \{C\} & \emptyset \end{pmatrix}.$$

Далее вычисляем элементы оставшихся подматриц.

1. $M_D = M_D \cup (T_B \times T_C) = \begin{pmatrix} \emptyset & \{D\} \\ \{A\} & \{B,\ D\} \end{pmatrix} \times \begin{pmatrix} \{B,S\} & \emptyset \\ \{C\} & \emptyset \end{pmatrix} =$
$\begin{pmatrix} \{(D,C)\} & \emptyset \\ \{(A,B),(A,S),(B,C),(D,C)\} & \emptyset \end{pmatrix}.$
После $complete(D) = complete(0,2,4,6)$ $T_D = \begin{pmatrix} \{S\} & \emptyset \\ \{B,S\} & \emptyset \end{pmatrix}.$
2. $M_{D'} = M_{D'} \cup (T_C \times T_{B'}) = \begin{pmatrix} \{B,S\} & \emptyset \\ \{C\} & \emptyset \end{pmatrix} \times \begin{pmatrix} \{D\} & \{C,S\} \\ \{B,D\} & \{B,S\} \end{pmatrix} =$
$\begin{pmatrix} \{(B,D),(S,D)\} & \{(B,C),(B,S),(S,C),(S,S)\} \\ \{(S,D)\} & \{(C,C),(C,S)\} \end{pmatrix}.$
После $complete(D') = complete(2,4,6,8)$ $T_{D'} = \begin{pmatrix} \emptyset & \{B\} \\ \emptyset & \emptyset \end{pmatrix}.$
3. Теперь вычисляем $M_E = M_E \cup (T_B \times T_{D'}) =$

$= \begin{pmatrix} \emptyset & \{D\} \\ \{A\} & \{B,\ D\} \end{pmatrix} \times \begin{pmatrix} \emptyset & \{B\} \\ \emptyset & \emptyset \end{pmatrix} = \begin{pmatrix} \emptyset & \emptyset \\ \emptyset & \{(A,B)\} \end{pmatrix}.$

Затем $M_E = M_E \cup (T_D \times T_{B'}) =$

$= \begin{pmatrix} \emptyset & \emptyset \\ \emptyset & \{(A,B)\} \end{pmatrix} \cup \begin{pmatrix} \{S\} & \emptyset \\ \{B,S\} & \emptyset \end{pmatrix} \times \begin{pmatrix} \{D\} & \{C,S\} \\ \{B,D\} & \{B,S\} \end{pmatrix} =$

$\begin{pmatrix} \{(S,D)\} & \{(S,C),(S,S)\} \\ \{(B,D),(S,D)\} & \{(A,B),(B,C),(B,S),(S,C),(S,S)\} \end{pmatrix}.$

После $complete(E) = complete(0,2,6,8)$ $T_E = \begin{pmatrix} \emptyset & \emptyset \\ \emptyset & \{B\} \end{pmatrix}.$

Финальное состояние таблицы T представлено на рис. 10.

	0	1	2	3	4	5	6	7
0		$\{A\}$	$\emptyset$	$\{D\}$	$\{S\}$	$\emptyset$	$\emptyset$	$\emptyset$
1			$\{A\}$	$\{B,D\}$	$\{B,S\}$	$\emptyset$	$\emptyset$	$\{B\}$
2				$\{B,D\}$	$\{B,S\}$	$\emptyset$	$\emptyset$	$\{B\}$
3					$\{C\}$	$\emptyset$	$\emptyset$	$\emptyset$
4						$\{A\}$	$\{D\}$	$\{C,S\}$
5							$\{B,D\}$	$\{B,S\}$
6								$\{C\}$
7								

Рис. 10. Результирующее состояние таблицы T
Fig. 10. Final state of the matrix T

Данный пример демонстрирует тот факт, что алгоритм не всегда находит точное решение. Например, согласно таблице на рисунке 10, между вершинами 4 и 7 (ячейка $T_{4,7}$) проходит путь, конкатенация меток которого составляет слово из языка $L(G_S)$. На самом деле, между вершинами 4 и 7 проходят два пути: первый путь — ребро графа, помеченное символом “c”, а второй путь вида $4 \to 5 \to 6 \to 7$, метки рёбер которого составляют строку “abc”. И если нетерминал $C \in T_{4,7}$ действительно выводит символ “c”, то $S \in T_{4,7}$ не выводит строку “abc” по причине того, что она выводится конъюнктом AB, отрицание которого есть в правиле $S \to DC\&\neg AB$. Это получается из-за того, что, хотя пара (A,B) и присутствовала в множестве нетерминалов $\{(A,B),(D,C)\}$ в ячейке $M_{4,7}$, функция f, которая рассматривает все подмножества данного множества, на подмножестве $\{(D,C)\}$ вернёт нетерминал S.

Несмотря на то, что стратегия рассматривания отдельных подмножеств пар нетерминалов дает лишние ответы, она гарантирует наличие в результатах настоящего ответа на задачу. Продемонстрируем это на следующем примере.

Рассмотрим пару вершин (0, 4) в графе на рис. 1. Между этой парой вершин есть два пути: путь $0 \to 1 \to 3 \to 4$, метки которого составляют строку «abc» и путь $0 \to 1 \to 2 \to 3 \to 4$, метки которого составляют строку «aabc». Заметим, что «abc» $\in L(G_{AB})$, $L(G_{DC})$, а “aabc” $\in L(G_{DC})$, поэтому $(0,\ 4) \in R_S$. Во время работы алгоритма, в ячейке $M_{0,4}$ окажется множество $\{(A,B),(D,C)\}$. Если бы мы рассматривали всё множество сразу, то из-за отрицания (A,B) не нашлось бы подходящего правила, и тогда ячейка $T_{0,4}$ в результате содержала бы пустое множество. Но, как показано выше, в графе между парой вершин 0 и 4 есть уникальный путь, слово из меток рёбер которого выводится конъюнктом DC, а конъюнкт AB на самом деле выводит слово из меток рёбер другого пути между этими же вершинами. Именно поэтому, чтобы рассмотреть все возможные случаи и не отсечь настоящее решение, алгоритм рассматривает все подмножества, что дает аппроксимацию ответа сверху.

6. Заключение

В данной работе был предложен алгоритм, находящий приближённое решение задачи поиска путей в ациклических графах с ограничениями в терминах булевых грамматик и использованием реляционной семантики запросов. Была показана применимость алгоритма для поиска приближённого решения данной задачи. Благодаря тому, что предложенный алгоритм находит приближённое решение задачи, была получена более оптимальная асимптотика по сравнению с наивным точным алгоритмом, которому необходимо было бы рассмотреть все пути в графе, число которых экспоненциально от количества вершин в графе. Так как предложенный алгоритм приближает решение сверху, на полученном решении возможно запустить точный алгоритм, который найдет точное решение за гораздо меньшее время. Также за счёт ограничения на тип входного графа и использования матричных операций, алгоритм является более асимптотически оптимальным, чем приближённый наивный итерационный алгоритм.

Определим несколько направлений для будущих исследований. Так как алгоритм работает только с ациклическими графами, открытым остается вопрос, возможно ли модифицировать алгоритм для работы с произвольными графами и сохранением такой же асимптотики. Также интерес представляет вопрос возможности эффективного преобразования произвольного входного графа в форму, позволяющую обработку данным алгоритмом, например, какой-либо вариант декомпозиции графа.

Список литературы / References

[1]. Barceló Baeza P. Querying graph databases. In Proc. of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems (PODS '13), 2013, pp. 175-188.

[2]. Mendelzon A., Wood P. Finding Regular Simple Paths in Graph Databases. SIAM Journal on Computing, vol. 24, № 6, 1995, pp. 1235-1258.

[3]. Anderson J. W. et al. Quantifying variances in comparative RNA secondary structure prediction. BMC bioinformatics, vol. 14, 2013.

[4]. Chaudhary A., Faisal A. Role of graph databases in social networks. 2016, available at: https://www.researchgate.net/publication/304462174_Role_of_graph_databases_in_social_networks, accessed 02.06.2019.

[5]. Warchał, Ł. Using Neo4j graph database in social network analysis. Studia Informatica, vol. 33, № 2A, 2012, pp. 271-279.

[6]. Reps T. Program analysis via graph reachability. In Proc. of the 1997 international symposium on Logic programming (ILPS '97), 1997, pp. 5-19.

[7]. Zhang Q., Su Z. Context-sensitive data-dependence analysis via linear conjunctive language reachability. In Proc. of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, 2017, pp. 344–358.

[8]. Yuan H., Eugster P. An Efficient Algorithm for Solving the Dyck-CFL Reachability Problem on Trees. In Proc. of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009 (ESOP '09), 2009, pp. 175-189.

[9]. Neo4j's Graph Query Language: An Introduction to Cypher. Available at: https://neo4j.com/developer/cypher-query-language/, accessed 02.06.2019.

[10]. Rodriguez M. A. The gremlin graph traversal machine and language (invited talk). In Proc. of the 15th Symposium on Database Programming Languages, 2015, pp. 1–10.

[11]. Eric Prud'hommeaux, Andy Seaborne (eds). SPARQL query language for RDF. W3C Recommendation 15 January 2008. Available at: https://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/, accessed 02.06.2019.

[12]. Okhotin A. Conjunctive grammars. Journal of Automata, Languages and Combinatorics, vol. 6, № 4, 2001, pp. 519–535.

[13]. Okhotin A. Boolean grammars. Information and Computation, vol. 194, issue 1, 2004, pp. 19–48.

[14]. Hellings J. Querying for Paths in Graphs using Context-Free Path Queries. CoRR abs/1502.02242, 2015.

[15]. Hellings J. Conjunctive context-free path queries. In Proc. of the International Conference on Database Theory (ICDT'14), 2014, pp.119–130.

[16]. Sevon P., Eronen L. Subgraph queries by context-free grammars. Journal of Integrative Bioinformatics, vol. 5, № 2, 2008.
[17]. Zhang X. et al. Context-free path queries on RDF graphs. In Proc. of the International Semantic Web Conference, 2016, pp. 632–648.
[18]. Grigorev S., Ragozina A. Context-free path querying with structural representation of result. In Proc. of the 13th Central & Eastern European Software Engineering Conference in Russia (CEE-SECR '17), article 10.
[19]. Azimov R.Sh., Grigorev S.V. Context-Free Path Querying by Matrix Multiplication. CoRR, abs/1707.01007, 2017.
[20]. Азимов Р.Ш., Григорьев С.В. Синтаксический анализ графов с использованием конъюнктивных грамматик. Труды ИСП РАН, том 30, вып. 2, 2018 г., стр. 149-166 / Azimov R.Sh., Grigorev S.V. Path querying using conjunctive grammars. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue. 2, 2018, pp. 149-166 (in Russian). DOI: 10.15514/ISPRAS-2018-30(2)-8.
[21]. Valiant L.G. General context-free recognition in less than cubic time. Journal of computer and system sciences, vol. 10, № 2, 1975, pp. 308–315.
[22]. Okhotin A. Parsing by matrix multiplication generalized to Boolean grammars. Theoretical Computer Science, vol. 516, 2014, pp. 101–120.
[23]. Арлазаров В.Л., Диниц Е.А., Кронрод М.А., Фараджев И.А. Об экономном построении транзитивного замыкания ориентированного графа. Доклады АН СССР, том 194, № 3, 1970, pp. 487–488 / Arlazarov V.L., Dinitz Y.A., Kronrod M.A., Faradzhev I.A. On economical construction of the transitive closure of an oriented graph. Doklady Akademii Nauk SSSR, vol. 194, № 3, 1970, pp. 487–488 (in Russian).
[24]. Vassilevska Williams V. Multiplying matrices faster than Coppersmith-Winograd. In Proc. of the 44th Symposium on Theory of Computing (STOC2012), 2012, pp. 887–898.
[25]. Tarjan R.E. Depth-first search and linear graph algorithms. In Proc. of the 12th Annual Symposium on Switching and Automata Theory (SWAT 1971), 1971, pp. 114-121.
[26]. Koschmieder André, Leser Ulf. Regular Path Queries on Large Graphs. In Proc. of the 24th International Conference on Scientific and Statistical Database Management (SSDBM'12), 2012, pp. 177–194.
[27]. Reutter Juan L., Romero Miguel, Vardi Moshe Y. Regular Queries on Graph Databases. Theory of Computing Systems, vol. 61, № 1, 2017, pp. 31–83.
[28]. Lu C., Yu J. X., Li R., Wei H. Exploring Hierarchies in Online Social Networks. IEEE Transactions on Knowledge and Data Engineering, vol. 28, № 8, 2016, pp. 2086-2100.
[29]. Sridharan, M., Gopan, D., Shan, L., Bodık, R. Demand-driven points-to analysis for Java. In Proc. of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA2005), 2005, pp. 59–76.
[30]. Терехов А.Н., Эрлих Л.А., Терехов А.А. История и архитектура проекта RescueWare. В сб. «Автоматизированный реинжиниринг программ», Издательство Санкт-Петербургского государственного университета, 2000 г., стр. 7-19 / Terekhov A.N., Erlikh L.A., Terekhov A.A. History and architecture of a project RescueWare. In «Automated software reengineering», Publishing House of St. Petersburg State University, 2000, pp. 7-19 (in Russian).
[31]. Medeiros Ciro M., Musicante Martin A., Costa Umberto S. Efficient evaluation of context-free path queries for graph databases. In Proc. of the 33rd Annual ACM Symposium on Applied Computing (SAC '18), 2018, pp. 1230-1237.
[32]. Rosenkrantz D.J., Stearns R.E. Properties of deterministic top-down grammars. Information and Control, vol. 17, № 3, 1970, pp. 226-256.

Информация об авторах / Information about authors

Екатерина Николаевна ШЕМЕТОВА получила степень магистра в области разработки программного обеспечения в университете ИТМО. Области научных интересов: теория формальных языков и её приложения, статический анализ кода, информационная безопасность, теория сложности вычислений.

Ekaterina Nikolaevna SHEMETOVA got master's degree in Software Engineering from ITMO University. Her research interests include formal language theory and applications, static code analysis, information security and computational complexity theory.

Семён Вячеславович ГРИГОРЬЕВ, кандидат физико-математических наук, доцент кафедры информатики СПбГУ. Области научных интересов: теория формальных языков и её приложения, графовые базы данных, статический анализ кода, параллельные вычисления.

Semyon Vyatcheslavovitch GRIGOREV is PhD in Phisical and Mathematical sciences, associated professor of the Department of Informatics at St Petersburg University. His research interests include formal language theory and applications, graph databases, static code analysis and parallel computing.