ISSN 1818–1015 (Print)
ISSN 2313–5417 (Online)

Министерство образования и науки Российской Федерации
Ярославский государственный университет им. П. Г. Демидова

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ

Том 24 № 6(72) 2017

Основан в 1999 году
Выходит 6 раз в год

Адрес редакции: ЯрГУ, ул. Советская, 14, г. Ярославль, 150003, Россия
Website: http://mais-journal.ru, e-mail: mais@uniyar.ac.ru; телефон (4852) 79-77-73

Научные статьи в журнал принимаются по электронной почте. Статьи должны содержать УДК, аннотации на русском и английском языках и сопровождаться набором текста в редакторе LaTeX. Плата с аспирантов за публикацию рукописей не взимается.

12+

СОДЕРЖАНИЕ

Моделирование и анализ информационных систем. Т. 24, №6. 2017

Свидетельство о регистрации СМИ ПИ № ФС 77 – 66186 от 20.06.2016 выдано Федеральной службой по надзору в сфере связи, информационных технологий и массовых коммуникаций. Учредитель – Федеральное государственное бюджетное образовательное учреждение высшего образования "Ярославский государственный университет им. П. Г. Демидова". Подписной индекс – 31907 в Объединенном каталоге "Пресса России". Редактор, корректор А.А. Аладьева. Редактор перевода Э.И. Соколова. Подписано в печать 10.12.2017. Дата выхода в свет 28.12.2017. Формат 60x84$^1/_8$. Усл. печ. л. 17,0. Уч.-изд. л. 15,0. Объем 145 с. Тираж 46 экз. Свободная цена. Заказ 094/017. Адрес типографии: ул. Советская, 14, оф. 109, г. Ярославль, 150003 Россия. Адрес издателя: Ярославский государственный университет им. П. Г. Демидова, ул. Советская, 14, г. Ярославль, 150003 Россия.

ISSN 1818–1015 (Print)
ISSN 2313–5417 (Online)

P.G. Demidov Yaroslavl State University

MODELING AND ANALYSIS OF INFORMATION SYSTEMS

Volume 24 No 6(72) 2017

Founded in 1999
6 issues per year

Editorial Office Address: P.G. Demidov Yaroslavl State University,
14 Sovetskaya str., Yaroslavl 150003, Russia
Website: http://mais-journal.ru, e-mail: mais@uniyar.ac.ru

Contents

Modeling and Analysis of Information Systems. Vol. 24, No 6. 2017

От редактора специального выпуска

В. А. Захаров

В этом выпуске журнала представлены статьи, подготовленные на основе научных докладов, которые были сделаны на восьмом международном семинаре «Семантика, спецификация и верификация программ: теория и приложения» (8-th Workshop on Program Semantics, Specification and Verification: Theory and Applications, PSSV 2017). Этот семинар проводился 26 июня 2017 г. в Москве, на факультете вычислительной математики и кибернетики Московского государственного университета имени М.В. Ломоносова.

Программа семинара PSSV 2017 включала 8 регулярных докладов и 3 лекции приглашенных докладчиков. В выступлениях участников семинара были представлены результаты завершенных и продолжающихся исследований разнообразных задач в области математического моделирования, статического анализа и верификации программных систем, а также задач, относящихся к теории автоматов и теории сетей Петри. Значительное внимание было уделено методам дедуктивной проверки правильности программ, верификации моделей программ, применения методов теории автоматов к тестированию программ, а также ряду вопросов построения и анализа формальных моделей информационных систем. Данный выпуск журнала включает 6 статей участников семинара PSSV 2017.

В статье А.О. Алексюка и В.М. Ицыксона описан автоматизированный метод переноса программного обеспечения из одной библиотеки в другую и эксперименты по его применению. Предложенный метод использует спецификацию библиотек в виде систем взаимодействующих автоматов. Он был успешно опробован на нескольких семействах библиотек.

Т. Баар (T. Baar) в статье, представленной в этом выпуске журнала, предложил подход к количественной оценке соответствия реализаций программных систем тем UML моделям, на основе которых эти системы были построены. В этом подходе предполагается, что UML модели сопутствует набор тестов и соответствующих им трасс. Для каждого теста реализация модели подвергается особого рода видоизменениям (мутациям), и для каждой мутации проводится проверка этого теста. Доля тех мутаций, которые успешно прошли тестовые испытания, и является количественной оценкой уровня абстракции UML модели по отношению к ее реализации.

Исследовательская группа, в состав которой входят Д. де Карвальо, М. Маццара, Б. Мингела, Л. Сафина, А. Чичигин, Н. Трошков (D. de Carvalho, M. Mazzara, B. Mingela, L. Safina, A. Tchitchigin, N. Troshkov), решала задачу проверки соответствия типов данных для языка программирования Jolie с использованием ранее известных правил вывода типов. Авторы статьи описывают метод решения этой задачи путем сведения ее к проблеме выполнимости и последующего применения автоматических средств проверки выполнимости формул в логических теориях.

А.В. Когтенков в своей статье провел сравнительный анализ нескольких методов статического анализа программ, предназначенных для проверки свойств безопасности при инициализации объектов в объектно-ориентированных языках программирования. Сравнение этих методов проведено на основе предложенных автором критериев выразительности и полноты.

Статья Н.Г. Кушик, Н.В. Евтушенко, И.Б. Бурдонова и А.С. Косачева посвящена методам сведения задачи синтеза так называемых синхронизирующих и установочных последовательностей для специального класса читающих и пишущих автоматов к аналогичным задачам для классов автоматов, для которых соответствующие задачи уже хорошо изу-

чены. Выделен класс автоматов, для которых установлены необходимые и достаточные условия существования синхронизирующих и установочных последовательностей, и получены оценки длины таких последовательностей. Результаты исследований этой работы могут найти применение при решении задач тестирования информационных систем.

Статья И.В. Марьясова, В.А. Непомнящего и Д.А. Кондратьева продолжает цикл исследований авторов по развитию дедуктивного метода верификации программ, реализующих итерационные алгоритмы над составными структурами данных. Особенность предложенного метода состоит в том, что верификация проводится в условиях, когда итерационный цикл не снабжен явно заданным подходящим инвариантом. Для преодоления этой трудности авторы предлагают использовать новое правило вывода в логике хоаровского вида, в котором вычислительный эффект цикла выражается специальной функцией. Для реализации предложенного метода авторы используют систему интерактивного автоматизированного доказательства теорем CVC4.

Моделирование и анализ информационных систем. Т. 24, № 6 (2017), с. 677–690
Modeling and Analysis of Information Systems. Vol. 24, No 6 (2017), pp. 677–690

DOI: 10.18255/1818-1015-2017-6-677-690

УДК 004.416.3+004.4'242

Семантически-ориентированная миграция Java-программ: опыт практического применения

Алексюк А. О., Ицыксон В. М.

получена 3 сентября 2017

Аннотация. Данная статья посвящена разработке процедуры автоматизированной миграции Java-программ на новый набор библиотек. Задача миграции (портирования) кода часто встречается в современных программных проектах. Например, такая задача может возникнуть, когда проект необходимо перенести на более безопасную или функциональную библиотеку, на новую платформу или на новую версию уже используемой в проекте библиотеки.

В данной работе представлена процедура автоматизированной миграции, основанная на семантическом подходе. Для процедуры миграции была разработана метамодель библиотеки, использующая предложенный ранее авторами формализм и предназначенная для описания библиотек на объектно-ориентированных языках. Формализм описывает поведение библиотек с помощью системы расширенных конечных автоматов (РКА). Процедура миграции разбита на пять шагов, каждый шаг подробно описан в тексте статьи. В процедуре используется алгоритм вычисления эквивалентной трассы на основе поиска в ширину, расширенный для решения задач миграции.

Предложенная процедура реализована в прототипе инструмента миграции. Инструмент включает в себя модули извлечения трассы выполнения программ, визуализации моделей библиотек, взаимодействия с пользователем и непосредственно миграции. Для инструмента был разработан язык описания библиотек. Прототип инструмента был протестирован как на искусственных примерах, так и на существующем проекте. В статье подробно описаны проведенные эксперименты, отдельно отмечены сложности, возникающие в процессе миграции тестовых примеров, и то, как они решаются в предложенной процедуре. В качестве библиотек в экспериментах используются реализации протокола HTTP и библиотеки протоколирования. Результаты тестирования показали, что миграция кода может быть успешно автоматизирована с использованием разработанной процедуры.

Ключевые слова: программная библиотека, миграция программ, поведенческое описание, трансформация программ

Для цитирования: Алексюк А. О., Ицыксон В. М., "Семантически-ориентированная миграция Java-программ: опыт практического применения", *Моделирование и анализ информационных систем*, **24**:6 (2017), 677–690.

Об авторах:
Алексюк Артем Олегович, orcid.org/0000-0001-5087-5567, студент,
Санкт-Петербургский политехнический университет Петра Великого,
ул. Политехническая, 29, г. Санкт-Петербург, 195251 Россия, e-mail: aleksyuk@kspt.icc.spbstu.ru

Ицыксон Владимир Михайлович, orcid.org/0000-0003-0276-4517, канд. техн. наук, доцент,
Санкт-Петербургский политехнический университет Петра Великого,
ул. Политехническая, 29, г. Санкт-Петербург, 195251 Россия, e-mail: vlad@icc.spbstu.ru

Введение

Данная работа посвящена автоматизированной миграции программного кода. Миграция кода — актуальная задача в современной индустрии разработки ПО. Обычно разработчики сталкиваются с этой задачей, когда собираются обновить свой проект до новой версии библиотеки или перенести его на более эффективную, безопасную или функциональную библиотеку. Как правило, исходный проект был в достаточной степени протестирован и разработчик хочет быть уверен, что портированный проект сохранит такой же уровень качества. Ручная миграция не гарантирует ожидаемого уровня качества, разработчики должны протестировать мигрированный проект с нуля как новый.

Целями этого исследования являются:

- Разработка нового метода миграции на основе формальных спецификаций библиотек, который полностью автоматизирован и сохраняет качество проекта.
- Разработка инструмента миграции.
- Проверка применимости метода и инструмента на практике.

Предлагаемый подход основан на созданном ранее авторами формализме для описания спецификации библиотек [1]. Данный формализм описывает библиотеки как набор расширенных конечных автоматов (РКА). Каждый РКА отражает жизненный цикл всей библиотеки или ее отдельной сущности. Примерами таких сущностей являются статически или динамически созданные файлы, сокеты, семафоры, потоки, мьютексы, потоки и так далее. Состояния в РКА соответствуют состояниям сущностей в программе, а ребра представляют собой вызовы функций библиотек. Полную спецификацию формализма можно найти в [1].

Одной из основных целей этой работы является разработка прототипа инструмента, который может быть использован для подтверждения возможности автоматической миграции на основе семантических спецификаций. Для этого был разработан простой предметно-ориентированный язык (Domain-specific language, DSL), который позволяет определять поведение библиотек. Спецификации используются в процедуре миграции, которая трансформирует исходную программую в новую в соответствии с моделями старой и новой библиотек.

Для оценки применимости созданного подхода был проведен ряд экспериментов с синтетическими примерами программ и реальным проектом с открытым исходным кодом. Все эксперименты завершились успешно, что свидетельствует о применимости подхода.

Работа состоит из четырех разделов. Первый раздел содержит обзор существующих подходов к автоматизированной миграции программного кода. Во втором разделе описывается предлагаемый подход к миграции. Третий раздел посвящен реализации прототипа инструмента миграции. Четвертый раздел посвящен экспериментальным исследованиям разработанного инструмента. В заключении анализируются полученные результаты и обсуждаются возможные направления дальнейшего развития.

1. Анализ существующих подходов к миграции

Рассмотрим существующие подходы к автоматизации миграции программных проектов.

Использование готовых программных модулей (библиотек) является стандартной практикой в современном программировании. Многие библиотеки стремятся к тому, чтобы стать всеобъемлющими и универсальными. Библиотеки помогают программисту писать меньше кода и при этом всё глубже интегрируются в проект.

Однако при необходимости отказаться от использования библиотеки в проекте преимущества её использования превращаются в недостатки. Как правило, чем активнее использовались средства библиотеки в проекте, тем больше усилий необходимо затратить на миграцию кода. Чтобы оценить сложность процедуры миграции, перечислим основные задачи переноса кода на другую библиотеку:

- замена сигнатур вызовов библиотечных методов и функций,
- преобразование используемых в программе типов и структур данных,
- удаление ссылок на константы и глобальные структуры библиотеки,
- добавление/удаление вызовов API библиотеки,
- устранение зависимости от внутреннего состояния библиотеки,
- исправление функций обратного вызова (callback),
- и т.п.

Все перечисленные задачи являются весьма рутинными, содержащими повторяющиеся действия, и отнимают время у разработчиков. Было решено рассмотреть подходы к миграции относительно их способов решения указанных задач. Ниже перечислены группы подходов к миграции программного кода:

- создание программ-обёрток (эмуляция API),
- синтаксический подход,
- семантический подход.

Программа-обертка (англ. wrapper, оболочка) — это программный модуль, который соответствует программному интерфейсу (API) одной библиотеки, но реально не содержит никакой логики. Вместо этого он передает управление другой библиотеке [10]. Программы-обертки являются примитивной прослойкой между двумя библиотеками и по сути представляют собой транслятор вызовов, работающий на уровне текста программ.

Сложность программ-оберток может сильно варьироваться. Простейшие программы-обертки просто перенаправляют вызовы функций в другую библиотеку. Этот способ миграции является достаточным, если целевая библиотека имеет функции с похожей сигнатурой и той же семантикой, что и исходная библиотека. Более сложные программы-обертки могут также выполнять преобразование форматов данных, то есть приводить передаваемые в функцию данные к такому формату, с которым может работать целевая библиотека.

Существует несколько недостатков подхода к миграции с использованием программ-оберток:

- Программы-обертки обычно пишутся для фиксированных исходных и целевых библиотек.
- Сложные программы-обертки могут серьезно повлиять на производительность.

- Программы-обертки, как правило, ограничивают доступ к структурам и/или методам целевой библиотеки.
- Если одна часть программы мигрирована на целевую библиотеку вручную или одним из методов, который непосредственно меняет код, а другая часть использует библиотеку через программу-обертку, могут возникнуть сложности с обменом данными между этими частями программы.

Несмотря на все перечисленные выше ограничения, подход с использованием программ-оберток широко применяется в промышленных проектах. Программы-обертки могут применяться тогда, когда исходный код мигрируемого проекта недоступен. Во многих случаях проще создать программу-обертку, чем переносить все проекты, использующие исходную библиотеку. Ниже перечислены некоторые известные проекты-обертки:

- ANGLE — реализация программного интерфейса OpenGL ES, работающая поверх библиотеки Direct3D [12].
- SLF4J (Simple Logging Facade for Java) — фреймворк для прокотолирования, который может использовать различные библиотеки в качестве бэкенда (модуля для непосредственного формирования и вывода сообщений) [13].

Методы, основанные на синтаксическом подходе, в процессе миграции кода используют информацию о синтаксисе языка программирования. В отличие от программ-оберток, при использовании синтаксического подхода изменения вносятся непосредственно в программный код мигрируемого проекта. К этой группе относятся методы на основе шаблонных преобразований и перезаписи термов.

Примером средства, использующего шаблонные преобразования, является интегрированная среда разработки IntelliJ IDEA. Система шаблонных преобразований в IntelliJ IDEA изначально разрабатывалась для поиска и исправления ошибок в коде, но может использоваться и для миграции программ. Довольно большое количество шаблонов изначально содержится в IDEA, пользователь неявно обращается к ним, когда статический анализатор IDEA находит потенциальную ошибку в коде и предлагает для неё исправление. Также имеется возможность добавить свои шаблоны с помощью функции Structural Search and Replace [9]. Данная функция во многом похожа на обычный диалог замены текста (Search and Replace), однако учитывает синтаксис кода. Например, при поиске вхождения не учитывается разница в форматировании между шаблоном поиска и текстом программы, также не учитывается порядок объявления переменных, методов, полей и некоторых других синтаксических конструкций.

Другой метод, реализующий синтаксический подход, использует так называемые правила перезаписи термов (term rewriting). Примером реализации этого метода является инструмент TXL [6]. Он позволяет с помощью специального языка задавать правила изменения элементов программы. Для описания грамматики языка, на котором написана программа, используется расширенная форма Бэкуса—Наура. Ещё одним похожим вариантом синтаксического подхода является метод стратегий перезаписи. Этот метод реализован в системе Stratego/XT [3] и инструменте DMS [2].

В статье [4] рассмотрено использование синтаксического подхода непосредственно для портирования программ. Авторы статьи составили набор правил перезаписи и в рамках тестирования частично портировали проект kdelibs с библиотеки Qt версии 3 на Qt 4.

Основной недостаток синтаксических подходов — малая гибкость. С помощью этих подходов можно выполнить только простейшие преобразования — замена вызова одной функции на другую, перестановка двух аргументов функции местами и другие преобразования, затрагивающие лишь один или несколько операторов [5]. Более масштабные изменения в случае синтаксических подходов либо требуют написания крайне сложных шаблонов, либо вовсе невозможны.

Подходы, описанные выше, имеют ограниченные возможности для автоматической миграции исходного кода в случае существенной разницы между исходной и целевой библиотеками. В этом случае необходима дополнительная информация, такая как семантика библиотеки.

Добавление новой информации в синтаксический подход приводит к появлению группы семантических подходов. На данный момент только начинают появляться инструменты, реализующие указанный подход. Одна из недавних статей [11] расширяет основанный на шаблонах подход информацией о семантике для лучшего поиска программных элементов, которые необходимо перенести. Тем не менее, процесс замены по-прежнему контролируется набором шаблонов и поэтому не способен выполнять сложные изменения в исходном коде. В этой работе используется семантический подход с более мощными и универсальными механизмами.

2. Предлагаемый подход

Любая система миграции оперирует не самим кодом, а его моделью в том или ином виде. Структура моделей задается метамоделью. Метамодель является важной частью подхода к миграции программного кода, именно от неё во многом зависит спектр возможностей подхода. При разработке метамодели учитывались следующие критерии:

- сложность анализа,
- способность отображать семантику библиотек,
- легкость создания моделей библиотек.

В качестве основы для метамодели был использован ранее созданный формализм [1]. Указанный формализм был создан не только для задач миграции, но и для использования внутри других областей программной инженерии, включая задачи обнаружения дефектов и тестирования программного обеспечения[1]. Формализм не задает конкретную метамодель библиотеки и её границы, но является основой для неё и описывает элементы, которые должны быть включены в метамодель.

Метамодель представляет собой набор РКА, каждый из которых соответствует семантическому объекту программы, который библиотека может обрабатывать или использовать для своих действий. Например, библиотека файлового ввода/вывода может иметь автоматы «Файл», «Имя файла», «Поток» и тому подобное. Обычно каждый класс (в терминах ООП) в библиотеке имеет соответствующий РКА в модели. РКА определяются кортежем $< Q, Q_0, X, V, C, C^A, C^D, U, F, T >$.

Каждый РКА в библиотеке включает в себя множество состояний Q. Например, автомат «Файл» может иметь такие состояния, как «Открыто», «Закрыто» и

[1]Следует отметить, что формализм в настоящее время дорабатывается и в контексте данной статьи не должен рассматриваться как окончательная версия.

«Только для чтения». Каждый автомат определяет набор переходов T, которые соответствуют операциям над объектами, и непустое множество начальных состояний Q_0. X — множество финальных состояний объекта. C — набор вызовов функций, конструкторов и других элементов кода, действующих как стимулы для переходов между состояниями. Также C содержит зависимости, то есть набор объектов-сущностей, необходимых для выполнения перехода.

Некоторые элементы программы возвращают новые объекты после выполнения. C_i^D — это набор дочерних автоматов, создаваемых при активации элемента C_i.

Так как большинство сложных библиотек не могут быть в достаточной степени описаны только с помощью состояний и переходов, метамодель была расширена атрибутами автоматов V и семантическими действиями (actions) C^A. Каждый РКА может иметь набор атрибутов (properties, свойств), идентифицированных по имени и содержащих произвольное значение (строки, целые числа и так далее). Атрибуты могут быть изменены функциями обновления U в процессе выполнения перехода. Переход между состояниями возможен только в том случае, если защитный предикат $F(V)$ истинен. Атрибуты могут использоваться в тех случаях, когда с помощью состояний сложно или полностью невозможно выразить семантику библиотеки.

Семантические действия в модели библиотеки определяют семантически значимые события, которые не могут быть выражены атрибутами или сменой состояния. Действия реализованы как атрибут перехода и специфицируются сигнатурой (именем и списком аргументов). Более подробную информацию о семантических действиях и атрибутах можно найти в статье [1].

Предлагаемая процедура миграции состоит из пяти шагов:

Первый шаг — **извлечение трассы**. Трасса — это последовательность операций в анализируемой программе. Трасса содержит список элементов кода, расположенных в порядке их выполнения.

Второй шаг — это **отображение трассы на модель**. Задача этого шага — преобразование трассы программы в трассу на модели, которая является упорядоченной последовательностью переходов в автоматах.

Третий шаг — **расчет эквивалентной трассы** — это непосредственно процесс поиска замены для трассы из исходной модели. Элементы результирующей трассы — переходы на модели целевой библиотеки. Так как предложенная процедура является реализацией семантического подхода, преобразование выполняется в контексте моделей библиотек.

Рассчитанная трасса должна быть синтаксически и семантически эквивалентна исходной. Замена ищется в соответствии с двумя критериями. Во-первых, если исходная трасса содержит создание сущности, результирующая трасса также должна создавать её. Например, если исходная программа получает длину файла, то есть создает сущность *FileLength*, результирующая трасса также должна создавать эту сущность. Это требование обусловлено тем, что исходная программа может передать созданный объект в другую функцию или сохранить его в поле класса, и корректно перенесенная программа должна сделать то же самое. Во-вторых, результирующая трасса должна содержать тот же набор семантических действий, что и исходная.

Алгоритм, используемый в процедуре миграции, основан на поиске в ширину (breadth-first search, BFS). Модель библиотеки рассматривается как граф, и крат-

чайший путь до искомой вершины можно рассматривать как последовательность эквивалентных операций. Поиск одновременно запускается из нескольких вершин (все доступные в области видимости сущности программы). Если эквивалентный путь не найден, необходимо обратиться к пользователю за помощью.

Сначала библиотечная модель преобразуется в графовое представление. Каждый расширенный конечный автомат преобразуется в отдельный граф (состояния превращаются в вершины, переходы — в ребра), после чего все такие графы объединяются в один. Полученный граф содержит несколько кластеров вершин, которые соответствуют каждому РКА.

Однако путь, найденный на таком графе, не всегда формирует подходящую трассу–замену. Во-первых, подобное графовое представление не учитывает атрибуты автоматов и связанные с ними ограничения на переходы. Во-вторых, в таком графе не отражены семантические действия. В-третьих, подобный граф не содержит информацию о зависимостях.

Чтобы учесть эти ограничения, в процессе миграции определяется новый граф G', в котором каждая вершина является возможным состоянием поиска. Граф G' включает в себя ребро $u \to v$, если в состоянии u можно добавить переход к трассе, в результате чего будет достигнуто состояние v. Именно этот граф используется в алгоритме поиска. Состояние поиска включает в себя контекст, то есть множество инстанцированных РКА и их текущее состояние.

Классический алгоритм BFS имеет две очереди: очередь посещения, которая содержит уже посещенные вершины, и очередь ожидания, содержащую вершины, которые будут обрабатываться. Алгоритм, используемый в данной работе, добавляет третью очередь под названием «Состояния с неразрешенными зависимостями». В него добавляются состояния, которые не могут выполнить переход из-за отсутствующей в контексте зависимости. Если в очередь посещения добавляется состояние поиска, которое имеет необходимую зависимость в контексте, то эти два состояния объединяются (складываются их трассы и контексты) и результат помещается в очередь ожидания.

Очередь ожидания сортируется, модели с самыми короткими путями имеют наивысший приоритет. Это, а также отсутствие весов у ребер графа обусловливает минимальную длину получаемого пути.

В некоторых случаях следующий переход в трассе начинается не с конечного состояния предыдущего перехода. Например, программа оперирует с одной сущностью, а после этого обращается к другой, с которой она работала до этого. Поиск по графу не сможет найти такой путь, так как он рассматривает только операции с последним использованным РКА. В таких случаях генерируются дополнительные состояния поиска, в которых «активным» РКА назначаются автоматы из контекста. Все созданные состояния помещаются в очередь ожидания. Этот шаг позволяет вернуться к любому объекту, доступному в контексте. В некоторых случаях число генерируемых состояний может быть огромным, поэтому этот шаг применяется только в исключительных случаях, например, в отсутствие другого решения.

Четвертый шаг — **отображение новой трассы на модель кода**. Для каждого элемента новой трассы рассчитывается место его вставки в модель кода.

Пятый шаг — непосредственно **трансформация программы** с учетом полученных на предыдущем шаге элементов.

3. Разработка прототипа

Чтобы продемонстрировать возможность автоматизированной миграции кода и протестировать метамодель, был разработан прототип инструмента для миграции программ [2].

Инструмент предназначен для миграции кода на языке Java. Язык Java на данный момент является одним из самых популярных[3], значительное количество новых проектов разрабатывается именно на нем.

Были рассмотрены следующие подходы для извлечения трассы программы:

- использование программных интерфейсов JVM для инструментирования кода,
- взаимодействие с существующими отладчиками Java,
- использование аспектно-ориентированного программирования (АОП) для внедрения инструментирующего кода в программу.

Разработанный инструмент использует подход, основанный на аспектно-ориентированном программировании, потому что существует несколько широко распространенных и хорошо протестированных реализаций АОП, которые позволяют с минимальными усилиями проинструментировать код [8]. Для инструментирования в работе использована библиотека *AspectJ*.

Инструмент миграции использует библиотеку *JavaParser* для анализа исходного кода на языке Java. Эта библиотека содержит средства для построения AST (которое используется как модель кода) и выполнения обратного преобразования в текстовое представление. В процессе преобразования библиотека сохраняет внешний вид кода (форматирование и комментарии), что очень важно при трансформации программ. Особенно это важно при портировании кода, находящегося под управлением системы контроля версиями, такой как Git.

Специально для прототипа инструмента был создан предметно-ориентированный язык (DSL) для описания библиотек. DSL основан на языке программирования *Kotlin* [14] и позволяет разрабатывать модели библиотек без глубокого знания архитектуры инструмента.

Разработанный инструмент содержит модуль визуализации. Если есть визуальное представление библиотечной модели, появляется возможность вовлечь в процесс разработки описания тех людей, которые имеют опыт работы с библиотекой, но не имеют достаточных знаний о языке описания. Кроме того, визуальное представление может быть крайне полезно при отладке описания библиотек.

4. Экспериментальная проверка инструмента миграции

Так как одна из целей данной статьи — проверить работоспособность подхода на реальных проектах, то необходимо провести набор экспериментов.

Для тестирования подхода необходимо выбрать такие библиотеки, которые решали бы похожую задачу, но имели бы различные программные интерфейсы (API).

[2] Исходный код инструмента доступен по адресу https://github.com/h31/LibraryMigration
[3] https://www.tiobe.com/tiobe-index/

В ходе работы были проведены две серии экспериментов: первая связана с библиотеками, реализующими протокол HTTP, а вторая — с библиотеками протоколирования. В качестве источников трасс выполнения использовались девять синтетических примеров и один реальный проект. Разработанные синтетические примеры доступны в упомянутом ранее репозитории с исходным кодом.

4.1. Эксперименты с библиотеками, реализующими протокол HTTP

Были созданы модели для следующих библиотек:

- *Apache HttpClient* из проекта *HttpComponents*[4];
- Клиент из Java Class Library (*java.net.HttpURLConnection* и связанные с ним классы)[5];
- *OkHttp* client[6].

В качестве иллюстрации в листингах приведен пример выполнения простейшего GET-запроса с использованием двух разных библиотек.

Листинг 1. Пример использования класса HttpURLConnection из Java Class Library

```
1 URL url = new URL("http://api.ipify.org/");
2 URLConnection conn = url.openConnection();
3 InputStream is = conn.getInputStream();
4 BufferedReader br = new BufferedReader(new InputStreamReader(is))
5 String response = br.lines().collect(Collectors.joining("\n", "", "\n"));
```

Листинг 2. Пример использования библиотеки Apache HttpCliennt

```
1 HttpClient httpclient = HttpClients.createDefault();
2 HttpGet httpget = new HttpGet("http://api.ipify.org/");
3 HttpResponse httpResponse = httpclient.execute(httpget);
4 String response = EntityUtils.toString(httpResponse.getEntity());
```

Все три перечисленные библиотеки имеют набор общих сущностей (URL-адрес, HTTP-заголовок, HTTP-статус ответа). Однако есть и различия. Например, встроенный клиент Java включает в себя класс HttpURLConnection, который предоставляет доступ как к запросу, так и к ответу. Если же говорить про библиотеки Apache HttpClient и OkHttp, то они содержат отдельные классы запросов и ответов, однако и в случае этой пары библиотек нет абсолютного соответствия между классами. Кроме того, в двух этих библиотеках есть объект Client, который должен быть создан для выполнения HTTP-запросов. Встроенный клиент Java не имеет такого класса.

Указанных различий достаточно, чтобы стать проблемой для инструментов миграции, реализующих синтаксический подход. Если в исходной программе, использующей встроенный клиент Java, происходит обращение к HTTP-ответу через объект класса URLConnection, то в эквивалентной программе для библиотеки Apache HttpClient необходимо сначала явно выполнить запрос с помощью метода execute, получить объект HttpResponse и уже через него обращаться к ответу. Однако при повторном обращении к HTTP-ответу нет необходимости заново отправлять запрос,

[4] https://hc.apache.org/httpcomponents-client-ga/index.html
[5] https://docs.oracle.com/javase/8/docs/api/java/net/HttpURLConnection.html
[6] https://square.github.io/okhttp/

необходимо воспользоваться уже полученным ранее ответом. Более того, выполнение повторного запроса вызывает неопределенное поведение, так как в протоколе HTTP имеются неидемпотентные методы, такие как POST, и поэтому выполняющая повторный запрос программа не может считаться эквивалентной. Инструмент миграции должен не просто выполнять замену по шаблону, а учитывать контекст выполнения и в соответствии с ним вставлять новые конструкции в код программы.

В модели данные особенности библиотек отражены с помощью механизма зависимостей. Если в трассе программы найдено обращение к HTTP-ответу, например, доступ к содержимому ответа с помощью метода `getOutputStream()`, то для выполнения соответствующего перехода в модели целевой библиотеки необходимо предварительно создать сущности HTTP-клиента и HTTP-ответа. При отображении на модель программы данные переходы превращаются в создание объекта HttpClient и явное выполнение запроса. Модель по своему характеру является декларативной и не описывает конкретные операции, которые необходимо выполнить при миграции кода. Если указанные сущности были созданы ранее в процессе поиска (или найдены в трассе исходной программы), то новые экземпляры не создаются и вместо них переиспользуются имеющиеся.

Некоторые операции с библиотекой невозможно отразить только с помощью механизма зависимостей. Пример — выполнение POST-запроса с установленной нагрузкой (payload). С точки зрения алгоритма поиска, установка нагрузки не приводит к созданию новых сущностей и поэтому может быть убрана из целевой программы. Чтобы решить эту проблему, к переходам, связанным с установкой нагрузки, были привязаны семантические действия `usePost` и `setPayload`. Если в трассе выполнения исходной программы были найдены переходы, помеченные соответствующими семантическими действиями, то в целевой программе будут выполняться эквивалентные (в соответствии с моделью библиотек) действия. Устанавливаемая программой нагрузка отражена в модели как ещё одна сущность, от которой зависят переходы, связанные с выполнением POST-запроса.

Аналогично с помощью механизма семантических действий в модели библиотеки были описаны операции установки HTTP-заголовков. Имя заголовка и его содержимое фиксируется в модели как аргумент семантического действия. Инструмент миграции корректно обрабатывает ситуации, когда в программе устанавливается несколько заголовков (то есть выполняется несколько семантических действий одного типа с разными аргументами) — в таком случае инструмент миграции добавляет в целевую трассу столько же операций, сколько было в исходной, и сохраняя все найденные наборы аргументов.

Ещё один важный тест для инструмента миграции и метамодели библиотек — способность к переупорядочиванию операций. Примером ситуации, при которой необходимо изменить порядок выполнения операции, в очередной раз служит код установки нагрузки для POST-запросов. В случае класса HttpURLConnection установка нагрузки выполняется после формирования запроса, а в случае библиотеки OkHttp — во время формирования запроса. Для обработки подобных случаев в инструменте имеется буфер операций. Инструмент осуществляет поиск эквивалентной замены для перехода только в тот момент, когда это невозможно откладывать дальше, а до этого накапливает переходы в буфере. Благодаря такому решению имеет-

ся возможность переупорядочивать операции в буфере и соответственно в целевой программе.

После того как первая версия инструмента была закончена, она была протестирована на наборе простых синтетических примеров, каждый из которых занимает менее ста строк исходного кода. Каждый тестовый пример представляет собой метод в Java-классе, использующий возможности библиотеки HTTP-клиента. Ниже приведены несколько сценариев, для которых написаны тестовые методы:

- выполнение GET-запроса (в нескольких вариантах, включая пример кода из оф. документации библиотеки),
- выполнение POST-запроса с установкой нагрузки,
- GET-запрос с установкой дополнительных заголовков,
- условный переход в зависимости от содержимого ответа HTTP-запроса.

Были написаны примеры для всех трех выбранных библиотек: HttpClient, OkHttp и HttpURLConnection. Все тестовые примеры были успешно перенесены на новые библиотеки. Дополнительно была протестирована обратная миграция, при которой мигрированная ранее программа переносится обратно на свою исходную библиотеку.

Для более глубокого тестирования разработанная система была применена для миграции реального проекта. Одним из важных требований для оцениваемого проекта было большое тестовое покрытие — это необходимо для динамического извлечения трасс. Для данного эксперимента был выбран проект под названием instagram-java-scraper[7], предназначенный для извлечения данных из веб-страниц сервиса Instagram. Проект для выполнения HTTP-запросов использует клиент OkHttp. В ходе своей работы программный проект устанавливает собственные заголовки, использует и GET, и POST-запросы, а также извлекает различную информацию из ответов. В качестве целевых библиотек использовались HttpURLConnection и Apache HttpClient.

Эксперимент показал, что все операции были сохранены и корректно портированы на целевую библиотеку. Все тесты из тестового набора проекта корректно выполнялись, и ручной просмотр кода показал, что портированный код корректен.

4.2. Эксперименты с библиотеками протоколирования

Во второй серии экспериментов использовались библиотеки протоколирования (logging). Протоколирование — ещё одна область, где имеется несколько конкурирующих Java-библиотек, решающих похожую задачу. Были созданы модели для следующих библиотек:

- Log4j 1.2[8]
- Log4j 2.0[9]
- SLF4J (универсальный интерфейс протоколирования)

Для возможности автоматизированной проверки корректности миграции необходимо унифицировать формат протоколирования разных библиотек, а также убрать из него информацию, являющуюся атрибутом конкретного запуска: текущее время,

[7] https://github.com/postaddictme/instagram-java-scraper
[8] https://logging.apache.org/log4j/1.2/
[9] https://logging.apache.org/log4j/2.x/

поток исполнения, PID процесса и тому подобное. После такой унификации в случае корректного портирования файл протокола, созданный исходной программой, и файл, созданный целевой программой, должны совпадать.

Были подготовлены синтетические тестовые примеры, использующие вызовы библиотек протоколирования. Все разработанные тестовые примеры были успешно мигрированы на новые библиотеки. Было проведено шесть экспериментов, в которых в качестве исходной и целевой библиотеки поочередно использовались все три библиотеки протоколирования.

Все рассмотренные тестовые сценарии были реализованы в наборе автоматических регрессионных тестов инструмента миграции. Тестовый набор выполняет миграцию рассмотренных ранее проектов и примеров и контролирует корректность полученных целевых программ. Исходный код тестовых примеров доступен в репозитории проекта.

Заключение

В этой статье были представлены результаты наших исследований в области миграции программного кода на новые библиотеки. В ходе исследования был разработан инструмент миграции, который использует формальные спецификации библиотек для автоматического переноса Java-программ на новые библиотеки. Разработанный инструмент протестирован на наборе синтетических примеров программ и на реальном проекте с открытым исходным кодом. Все искусственные программы и реальный проект были успешно мигрированы.

В дальнейшем планируется улучшить представленный подход и инструмент миграции. Ключевыми направлениями будущих исследований являются:

- дальнейшее развитие формализма для создания спецификации библиотек,
- создание более удобного языка спецификации моделей библиотек,
- расширение возможностей по управлению процессом миграции со стороны пользователя,
- проведение дополнительных экспериментов, на большем числе библиотек и на большем количестве индустриальных проектов.

Список литературы / References

[1] Ицыксон В. М., “Формализм и языковые инструменты для описания семантики программных библиотек”, *Моделирование и анализ информационных систем*, **23**:6 (2016), 754–766; [Itsykson V. M., “The Formalism and Language Tools for Semantics Specification of Software Libraries”, *Modeling and Analysis of Information Systems*, **23**:6 (2016), 754–766, (in Russian).]

[2] Baxter I.D., Pidgeon C., Mehlich M., “DMS/spl reg: program transformations for practical scalable software evolution”, *Proceedings of 26th International Conference on Software Engineering*, IEEE, 2004, 625–634.

[3] Bravenboer M. et al., “Stratego/XT 0.17. A language and toolset for program transformation”, *Science of computer programming*, **72**:1 (2008), 52–70.

[4] Broeksema B., Telea A., “Visual support for porting large code bases”, *Visualizing Software for Understanding and Analysis (VISSOFT), 2011 6th IEEE International Workshop on*, IEEE, 2011, 1–8.

[5] Christoph A., Müller M.M., “GREAT: UML transformation tool for porting middleware applications”, *International Conference on the Unified Modeling Language*, Springer, 2003, 18–30.

[6] Cordy J.R., “The TXL source transformation language”, *Science of Computer Programming*, **61**:3 (2006), 190–210.

[7] Eisenbarth T., Koschke R., Vogel G., “Static trace extraction”, *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, IEEE, 2002, 128–137.

[8] Filman R.E., Havelund K., “Source-code instrumentation and quantification of events”, *Workshop on Foundations of Aspect-Oriented Languages (FOAL) at AOSD Conference*, Twente, Netherlands, 2002, 45–49.

[9] Jemerov D., “Implementing refactorings in IntelliJ IDEA”, *Proceedings of the 2nd Workshop on Refactoring Tools*, ACM, 2008, 13.

[10] Marosi A.C., Balaton Z., Kacsuk P., “GenWrapper: A generic wrapper for running legacy applications on desktop grids”, *2009 IEEE International Symposium on Parallel Distributed Processing*, ACM, 2009, 1–6.

[11] Wu L. et al., “Transforming Code with Compositional Mappings for API-Library Switching”, *2015 IEEE 39th Annual Computer Software and Applications Conference*, **2**, 2015, 316–325.

[12] “A conformant OpenGL ES implementation for Windows, Mac and Linux”, https://github.com/google/angle, visited on 03.09.2017.

[13] “Simple Logging Facade for Java (SLF4J)”, https://www.slf4j.org/, visited on 03.09.2017.

[14] JetBrains, “Statically typed programming language for the JVM, Android and the browser”, https://kotlinlang.org/, visited on 03.09.2017.

Aleksyuk A. O., Itsykson V. M., "Semantics-Driven Migration of Java Programs: a Practical Experience", *Modeling and Analysis of Information Systems*, **24**:6 (2017), 677–690.

DOI: 10.18255/1818-1015-2017-6-677-690

Abstract. The purpose of the study is to demonstrate the feasibility of automated code migration to a new set of programming libraries. Code migration is a common task in modern software projects. For example, it may arise when a project should be ported to a more secure or feature-rich library, a new platform or a new version of an already used library. The developed method and tool are based on the previously created by the authors a formalism for describing libraries semantics. The formalism specifies a library behaviour by using a system of extended finite state machines (EFSM). This paper outlines the metamodel designed to specify library descriptions and proposes an easy to use domain-specific language (DSL), which can be used to define models for particular libraries. The mentioned metamodel directly forms the code migration procedure. A process of migration is split into five steps, and each step is also described in the paper. The procedure uses an algorithm based on the breadth-first search extended for the needs of the migration task. Models and algorithms were implemented in the prototype of an automated code migration tool. The prototype was tested by both artificial code examples and a real-world open source project. The article describes the experiments performed, the difficulties that have arisen in the process of migration of test samples, and how they are solved in the proposed procedure. The results of experiments indicate that code migration can be successfully automated.

Keywords: software library, code migration, behavioral description, program transformation

On the authors:
Artyom O. Aleksyuk, orcid.org/0000-0001-5087-5567, student,
Peter the Great St. Petersburg Polytechnic University,
29 Polytechnicheskaya str., St. Petersburg 195251, Russia, e-mail: aleksyuk@kspt.icc.spbstu.ru

Vladimir M. Itsykson, orcid.org/0000-0003-0276-4517, PhD,
Peter the Great St. Petersburg Polytechnic University,
29 Polytechnicheskaya str., St. Petersburg 195251, Russia, e-mail: vlad@icc.spbstu.ru

Моделирование и анализ информационных систем. Т. 24, № 6 (2017), с. 691–703
Modeling and Analysis of Information Systems. Vol. 24, No 6 (2017), pp. 691–703

DOI: 10.18255/1818-1015-2017-6-691-703

UDC 519.686.2

Towards Measuring the Abstractness of State Machines based on Mutation Testing

Thomas Baar

Received October 30, 2017

Abstract. The notation of state machines is widely adopted as a formalism to describe the behaviour of systems. Usually, multiple state machine models can be developed for the very same software system. Some of these models might turn out to be equivalent, but, in many cases, different state machines describing the same system also differ in their level of abstraction.

In this paper, we present an approach to actually *measure the abstractness level* of state machines w.r.t. a given implemented software system. A state machine is considered to be less abstract when it is conceptionally closer to the implemented system. In our approach, this distance between state machine and implementation is measured by applying coverage criteria known from software mutation testing.

Abstractness of state machines can be considered as a new metric. As for other metrics as well, a known value for the abstractness of a given state machine allows to assess its quality in terms of a simple number. In model-based software development projects, the abstract metric can help to prevent model degradation since it can actually measure the semantic distance from the behavioural specification of a system in form of a state machine to the current implementation of the system.

In contrast to other metrics for state machines, the abstractness cannot be statically computed based on the state machine's structure, but requires to execute both state machine and corresponding system implementation.

The article is published in the author's wording.

Keywords: model-based software development, metric, state machine, mutation testing

For citation: Thomas Baar, "Towards Measuring the Abstractness of State Machines based on Mutation Testing", *Modeling and Analysis of Information Systems*, **24**:6 (2017), 691–703.

On the author:
Thomas Baar, orcid.org/0000-0002-8443-1558, PhD,
University of Applied Sciences (Hochschule für Technik und Wirtschaft (HTW) Berlin)
Wilhelminenhofstrasse 75 A, D-12459, Berlin, Germany, e-mail: thomas.baar@htw-berlin.de

1. Introduction

The notation of *state machines* [7, 8] is widely used in industry to specify the behaviour of software systems and is supported by numerous tools [15, 8, 9, 2]. Though the notation comes in different flavours, the core concepts *states*, *events*, *transitions*, *actions*, *state variables*, and *guards* are ubiquitous. Though there are some tools like Yakindu [9] available, which offer the generation of code into a target programming language such as C, C++, Java, etc., we will assume in this paper the (still common) situation that the

implementation of the system has been implemented independently from the behaviour specification in form of a state machine.

Once both the specification and the implementation of the system has been finalized, the natural question arises whether the actual behaviour of the system is actually reflected by the specification given as a state machine.

It is tempting to tackle this *correctness question* by using formal methods to actually prove that the implementation behaves according to the specification. Abadi and Lamport show in [1], that for each correct implementation there exists a formally definable refinement mapping. However, Abadi and Lamport assume the implementation also be given in form of a state machine. Their refinement mapping connects the two artefacts *specification* and *implementation*, which both have been written in the same formalism of state machine.

Our situation is quite different. While the specification is again given in form of a state machine, the implementation can be written in *any* programming language. Even worse, we do not rely on having any formal semantics of the programming language available, but just the compiler/interpreter allowing us to execute the implementation. Thus, we actually consider the implementation as a gray box, i.e. a combination of white box and black box in the following sense: The white box characteristic is due to the fact that we can execute the implementation and, moreover, we are able to stop the execution at any time in order to inspect the internal state. For example, the internal state might be given by the values of global variables together with the set of all existing objects including the values of their attribute in case of object-orientation has been used as a programming paradigm. The black box characteristic is due to the fact that we do not have a formal semantics of the used implementation language at hand. As a consequence, there is no chance to extract and to analyze any further artefacts such as control-flow graph from the implementation.

Due to this gray box characteristic, we cannot formally prove that the actual behaviour of the implemented system is correctly described by the state machine specification. What we can do is to test the correct realization in concrete scenarios. We can stimulate the implemented system with events and then check, whether the implemented system changes its internal state the same way the state machine specification has prescribed. However, this requires to define a formal correspondence between the states of the state machine (together with the current values of the state variables) to the internal states of the implementation. This correspondence between the state spaces of state machine and implementation is done in terms of formal predicates and is called *bridge* in our approach (cmp. Section 4.).

The second question we are interested in is how precisely the state machine reflects the behaviour of the implementation. In this paper we propose a novel metric to measure the *abstractness* of a state machine w.r.t. a given implementation. While the motivation for this metric originated from assessing the efforts students have made to re-model an existing application (see Section 2.), this measurement is also helpful to detect *model degradation*. A well-known problem of model-based software development is that models tend to degrade over time: if modeling artefacts are not maintained properly while the system implementation evolves, they become less and less valuable since they do not reflect any longer structure and/or behaviour of the implementation [16].

Technically, the abstractness is measured using the technique of mutation testing.

For a predefined list of input events, both the state machine and the implementation are executed in parallel. After each event has been processed, it is checked whether both systems are in comparable states (this is checked using the bridge-predicates, which relate both state spaces). For measuring the abstractness, the parallel executions of implementation and state machine are repeated for the chosen list of input events, but now the implementation has been mutated, i.e. at some point in the implementation a statement has been changed (for example, an operator '+' has been changed to '-'). If the trace of the manipulated implementation can still be mapped correctly to the trace of the state machine, then this is a sign that the state machine does not specify the behaviour of the original implementation *precisely* (since a change in the implementation is not detected) and thus, this model is considered to be rather abstract. In the opposite case of diverged traces, we have a witness that the state machine prescribes the behaviour of the original implementation precisely, thus the state machine is considered to be rather detailed.

The remaining of the paper is organized as follows. Section 2 presents a motivating example, which already shows some pitfalls when applying the state machine notation for the specification of real-world software. Section 3 introduces the notation of state machine formally, while Section 4 formally defines the bridge for connecting the state space of state machine and implementation. The core of our approach, i.e. the technique to measure the abstractness of state machines w.r.t. a given implementation, is detailed in Section 5 In Section 6 we review relevant literature while Section 7 concludes the paper.

2. Motivating Example

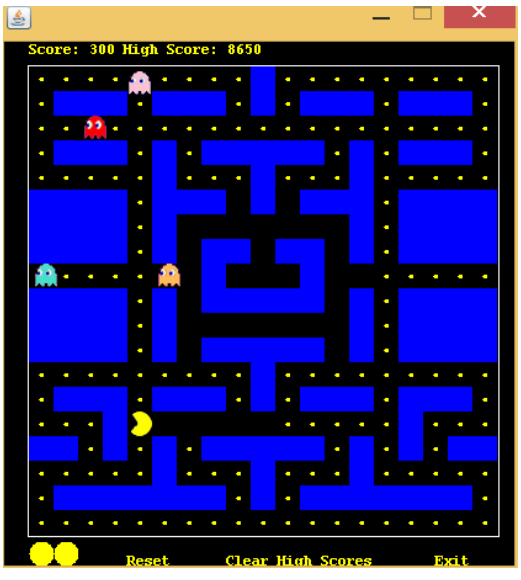

Fig 1. Screenshot Pac-Man

Suppose, you had to describe the behaviour of the classic computer game Pac-Man[1].

[1]For a detailed description see https://en.wikipedia.org/wiki/Pac-Man

Using informal language, you might solve this task be referring to Figure 1, which shows a screenshot from a open-source implementation in Java[2]. You might continue by saying that the user can move the Pac-Man via keyboard or joystick through the labyrinth and its task is to eat as many dots as possible while avoiding any collision with the enemies (the four ghosts). The Pac-Man has three lifes and a life ends by colliding with a ghost. The game is over when either the Pac-Man lost his last life or when it has eaten sufficiently many dots.

Suppose someone urges you now to use instead of informal language a modeling notation such as the ones bundled by the Unified Modeling Language (UML) [12]. Still, your model should be easily understood by a software engineer, so that she does not need to look into the implementation code to understand the rules of the game.

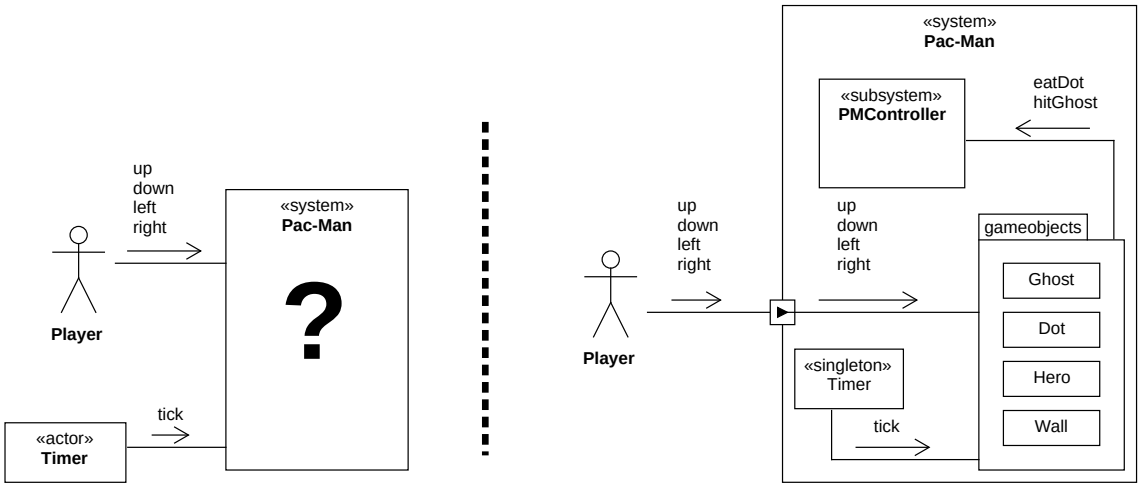

Fig 2. Naive(left) and elaborated(right) environment model – Determining the events for the state machine

Surely, the classic state machine notation seems the right formalism to be used, but before you can start you have to determine the external events to be taken into account.

Environment models as shown in Figure 2 can help a lot in this regard. You might start with a basic version shown at the left side and observe, that the system receives as input events from `Player` the four possible directions for moving Pac-Man. Furthermore, since the ghosts change their position independently of the player's input, it is quite obvious to model also an external Timer sending events `tick` to the system.

Unfortunately, it turns out to be impossible to develop a state machine which is purely based on these five events. A compelling argument is that the state machine has to reflect that Pac-Man has three lifes at the beginning and loses a life whenever it collides with a ghost. But how can the state machine detect such a collision? For this, the state machine had to keep track of the position of both Pac-Man and all ghosts.

To solve this problem you might restructure the Environment model as done at the right side of Figure 2. We split the system into a subsystem `PMController` and other

[2]The Java source code is available from https://github.com/dtschust/javapacman.

components for the individual game objects and the timer (whether `Timer` is an internal or external component remains a matter of taste). The original events from the player are now forwarded to the game objects. Also the event `tick` is forwared to them. Once the game objects detect a collision or that a dot was eaten by the Pac-Man, they issue new events `eatDot` and `hitGhost`, which are sent to `PMController`.

Having found now the right level of abstraction for the input events, a compact state machine reflecting the rules of the Pac-Man game can be developed quite easily, as shown in Figure 3, right side[3].

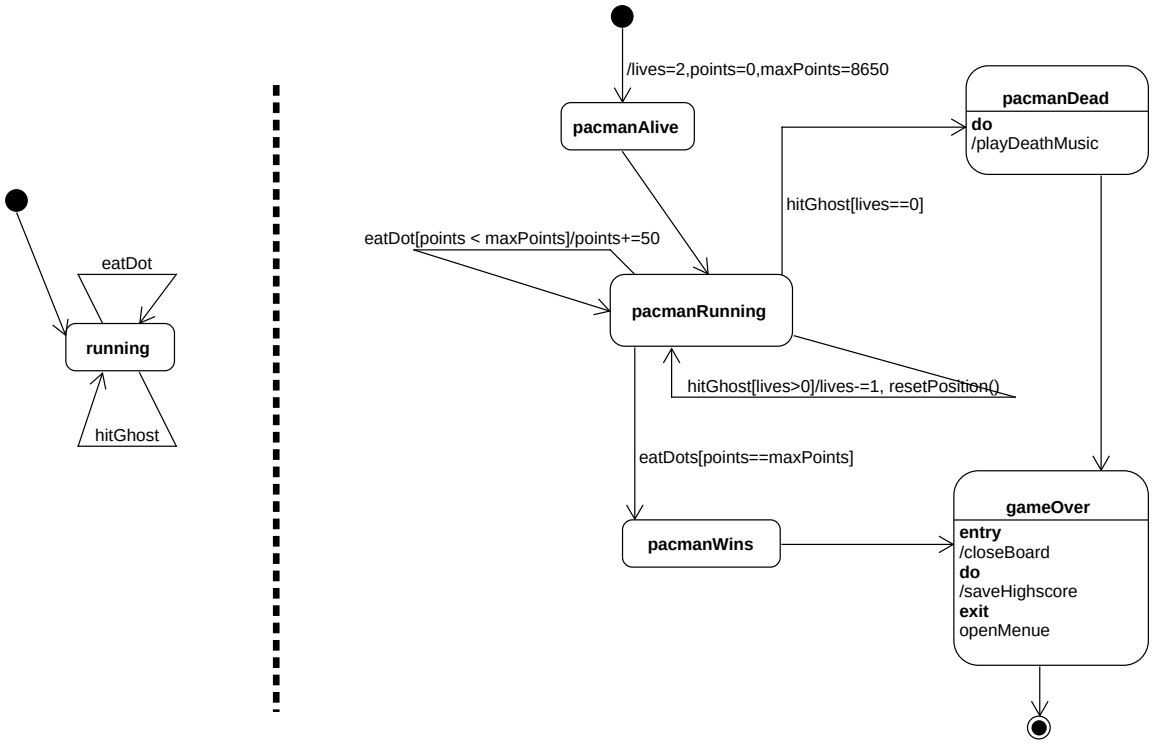

Fig 3. Useless(left) and elaborated(right) state machine

This one picture (state machine) has the same value as a lot of words. It reveals immediately how the Pac-Man game is organized and how the user can win the game. However, to decide whether this state machine actually describes the given implementation correctly, is rather challenging. Note that the implementation can only be stimulated with the four external events from the user, but that the state machine refers to the events `eatDot` and `hitGhost`, which are generated internally. When testing on the correctness of the implementation, it is crucial to find such event sequences that let the implementation generate these internal events.

A striking example, why one could be interested in measuring the abstractness of state machines w.r.t. an implementation is given in Figure 3, left side. One could argue that this state machine also describes the given implementation and it is pretty obvious,

[3]This state model is based on a student solution submitted by Fiona Brömer, Thorsten Vaterrodt, and Josefine Keller.

that the description is correct (if we do not require that after the game has been finished, the state machine must result in a final state). However, this state machine is useless since it correctly describes any other implementation dealing with the events `eatDot` and `hitGhost` as well.

In order to have a formal criterion on how we can distinguish useful from useless state machines, the metric for abstractness of a state machine is developed in this paper.

3. Background: State Machines

The state machine notation comes in many different variants and some of the advanced concepts such as *hierarchical state*, *parallel state*, *history state*, *state variable*, *time-triggered event*, *spontaneous transition*, *action*, *entry-/exit-action* are not supported by every tool.

In the remainder of this section, we define the version of state machines used in this paper[4]. The definition is done both in terms of syntax and semantics.

3.1. Syntax

We use a rather basic version of state machines. Only the concepts *state*, *start state* (always unique), *event*, *transition*, *guard*, *state variable*, and *action* (in form of parallel assignments of arithmetic expressions to state variables) are supported.

Definition 1 (State Machine)**.** *A state machine SM is a tuple $(S, Ev, V, init, trans)$ where*

- *S, Ev, V are pairwise disjoint, non-empty sets of states, events, and (integer) variables, respectively*
- *$init \in (S \times Bind)$ denotes the initial state and the initial binding of variables. Here, $Bind = \{b | \, b : V \to \mathbb{Z} \cup \{\bot\}\}$ denotes the set of all possible bindings of variables to an integer value or to undefined ($\bot$)*
- *$trans \subseteq S \times S \times Ev \times Exp_{bool} \times (V \nrightarrow Exp_{int})$ denotes the set of transitions. For $(s_{pre}, s_{post}, e, g, asgmnt) \in trans$ we call s_{pre} the pre-state, s_{post} the post-state, e the event, g the guard, and $asgmnt$ the assignment of the transition*

$Exp = Exp_{bool} \cup Exp_{int}$ denotes the set of all arithmetic/boolean expressions over the set of variables V. The expressions of Exp are built using the usual arithmetic and boolean operators and must obey the usual type rules. By definition, all variables from V are of type int.

As an example, we explore the state machine SM_{stack} describing the behaviour of a stack as defined in Fig. 4. Here the variable *num* encodes the number of elements currently residing on the stack. Fig. 4 shows two equivalent definitions of SM_{stack}. In

[4]Tool support for the described version of state machine is available in form of the toolset SSMA (Simple State Machine Analyzer) [2]. Deduction-based analysis techniques implemented by SSMA are discussed in detail in [3].

$SM_{stack} = (S, Ev, V, init, trans)$ with
- $S = \{e, ne\}$
- $Ev = \{push, pop\}$
- $V = \{num\}$
- $init = (e, \{num \mapsto 0\})$
- $trans = \{$
$(e, ne, push, true, \{num \mapsto num + 1\}$,
$(ne, ne, push, true, \{num \mapsto num + 1\}$,
$(ne, ne, pop, num > 1, \{num \mapsto num - 1\}$,
$(ne, e, pop, num == 1, \{num \mapsto num - 1\}$
$\}$

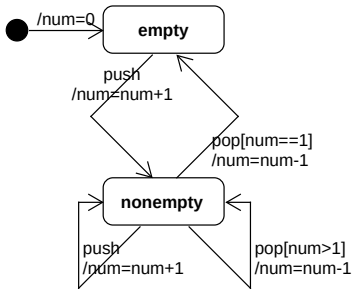

Fig 4. Example for State Machine: Stack

the left part the definition is done in a mathematical style following the definition Def. 1 whereas in the right part, a graphical version using the well-established graphical concrete syntax for state machines known from the Unified Modeling Language (UML) is used[5].

3.2. Semantics

The semantics of a state machine is given by describing how a state machine changes its state and the value of variables upon receiving a sequence of events. The resulting behaviour is called a *trace*.

Definition 2 (Trace)**.** *Let SM = (S, Ev, V, init, trans) be a state machine and* $inp = (e_1, e_2, e_3, \ldots, e_n)$ *with* $e_i \in Ev$ *for* $i \in \{1, \ldots, n\}$ *a finite sequence of (input) events. Let* $eval : Exp \times Bind \to \{true, false, \bot\}$ *be the usual evaluation function of expressions under a given variable binding. The evaluation w.r.t. undefinedness is strict. Then, a trace is a sequence of execution states* $(es_0, es_1, \ldots, es_n)$ *where each execution state is of form* $es_i = (s_i, b_i) \in S \times Bind$ *and each of the following constraints is satisfied:*

1. $es_0 = (s_0, b_0) = init$
2. $es_{i+1} = es_i$ *if the event is rejected since there is no outgoing transition from the pre-state* s_i *for the event* e_{i+1}*. Formally:* $\{t \mid t = (s_{pre}, s_{post}, e, g, asgmnt) \in trans \text{ and } s_{pre} = s_i \text{ and } e = e_{i+1}\} = \emptyset$
3. $es_{i+1} = es_i$ *if the transition cannot fire due to guard evaluation: For all transitions which are outgoing from* s_i *and which are annotated with event* e_{i+1}*, the evaluation of the guard does not yield true. Formally:* $\{t \mid t = (s_{pre}, s_{post}, e, g, asgmnt) \in trans \text{ and } s_{pre} = s_i \text{ and } e = e_{i+1} \text{ and } eval(g, b_i) = true\} = \emptyset$
4. *Transition is fired and bindings for variables are updated: There exists a transition* $t = (s_i, s_{i+1}, e_{i+1}, g, asgmnt) \in trans$ *with* $eval(g, b_i) = true$*. The new binding* b_{i+1} *is obtained by* $b_{i+1} = b_i \leftarrow asgmnt$ *as the update of the previous binding* b_i *according to the transition's assignment asgmnt*

[5]Mathematical and graphical definition differ solely in the chosen names for states. Thanks to the space efficiency of the graphical notation, we can afford here longer and more expressive state names, e.g. nonempty vs. ne.

Example: For the above stack example SM_{stack} and for the input event sequence *inp = (pop, push, push)*, the sequence of execution states $((e, num \mapsto 0), (e, num \mapsto 0), (ne, num \mapsto 1), (ne, num \mapsto 2))$ would be a trace. Every other sequence of execution states would be not a trace for the chosen input sequence *inp*.

4. Bridging State Machines and Implementations

Due to the semantics as defined in the previous section, state machines can be actually used as a graphical notation to program. Tools such as Yakindu [9] offer the generation of implementation code for a given state machine. However, a feature missed in many state machine tools (including Yakindu) is the possibility to set the edited state machine in relation to a given implementation.

Recall that an (object-oriented) implementation is executed by invoking methods on objects (or, rarely, classes) and that the system state is determined by the set of currently existing objects and the values for their attributes. In contrast, the execution of state machines is determined by the list of incoming events. Moreover, the execution state of a state machine consists of the currently active state and the current binding of state variables to values.

We relate the state space of both implementation and state machine by defining a mapping as follows, the mapping is called a *bridge*.

Definition 3 (Bridge)**.** *Let $SM = (S, Ev, V, init, trans)$ be a state machine, Impl an implementation, M_{Impl} the set of its methods, and S_{Impl} the set of all possible states the implementation can reach.*

Then, a bridge $B(SM, Impl)$ for SM and $Impl$ is a tuple $(Q, map_{ev}, Pred)$ consisting of

- *a set Q of queries $q_i : S_{Impl} \to \mathbb{Z} \cup \{\bot\}$*
- *a partial mapping $map_{ev} : M_{Impl} \nrightarrow Ev$ from methods to events*
- *a set $Pred \subseteq Exp_{bool}$ of predicates. The Boolean expressions are build with the usual boolean and arithmetic operators over variables from V, but quantifiers $(\forall, \exists)$ are not allowed to occur. Boolean expressions can, in addition, contain subexpressions q_i for accessing the current state of the implementation and the atomic predicate $inState(s)$ where $s \in S$*

Definition 4 (Valid Bridge)**.** *Let SM be a state machine, $Impl$ an implementation, and $B(SM, Impl) = (Q, map_{ev}, Pred)$ a bridge. Let es be an execution state of SM and ies an implementation state of $Impl$.*

We call $B(SM, Impl)$ a valid bridge *w.r.t. the pair (es, ies), iff all predicates $p \in Pred$ are evaluated to true in (es, ies). The evaluation of p in (es, ies) is defined via structural induction on the terms occurring in p as usual. The subterm q_i is evaluated to $q_i(ies)$. The subterm $inState(s)$ is evaluated to true, if and only if es is of form (s, b) for an arbitrary binding b.*

```
7  public class Stack1 {
8
9      private List<Item> items = new ArrayList<Item>();
10
11     public void push(Item i){
12         items.add(i);
13     }
14
15     public Item pop(){
16         if (items.isEmpty())
17             return null;
18         int lastIndex = items.size()-1;
19
20         return items.remove(lastIndex);
21     }
22
23     // allow the adapter to get necessary information
24     public int getLength(){
25         return items.size();
26     }
27 }
```

Fig 5. Java-Implementation of Stack

Example: We consider a possible implementation in Java of the stack as shown in Fig. 5. A possible bridge between the state machine defined in Fig. 4 and the implementation from Fig. 5 could look as follows:

- $Q = \{c\_s\}$ where c_s evaluates in a concrete implementation state to the return value of the invocation of the method getLength()
- $map_{ev} = \{push() \mapsto push, pop() \mapsto pop\}$
- $Pred = \{inState(empty)\, implies\, c\_s = 0,$
  $inState(nonempty)\, implies\, c\_s > 0\}$

Informally speaking, the bridge maps the method calls $push()/pop()$ to the corresponding events. The two predicates actually relate the execution states of the state machine with those of the implementation and stipulate, whenever the state machine is in state *empty*, the implementation is in a state where $getLength()$ returns zero. Furthermore, if the state machine is in state *nonempty*, the invocation of $getLength()$ yields a number greater than zero.

Definition 5 (Valid Abstraction Trace)**.** *Let SM be a state machine, $Impl$ an implementation and $B(SM, Impl)$ a bridge for them.*

We call the trace of the state machine $(es_0, \ldots, es_n)$ induced by an event sequence $(e_1, \ldots, e_n)$ a valid abstraction trace *iff for the corresponding implementation trace $(ies_0, \ldots, ies_n)$ the following holds: For each state pair (es_i, ies_i) for $i \in \{0, \ldots, n\}$ the bridge $B(SM, Impl)$ is actually a valid bridge.*

The notion *valid abstraction trace* witnesses that the state machine has specified the behaviour of an implementation correctly for a given list of input events.

In the remainder of the paper, we call a state machine a *presumably valid abstraction* w.r.t. a given implementation, if we cannot find any sequence of events, for which the induced trace of the state machine is not a valid abstraction trace.

5. Measuring the Abstractness of State Machines

After we have clarified (i) syntax and semantics of the basic version of state machines used in this paper and (ii) how a state machine trace can relate to a trace of an implementation, we now present the core of our approach to measure the abstractness of a state machine.

The basic idea can be stated as follows: Let a state machine, an implementation and a bridge be given. Due to successful tests on many input event sequences we are convinced that the state machine is a presumably valid abstraction of the implementation w.r.t. the bridge.

In order to measure the abstractness of the state machine, we repeat the same tests but we use now a slightly changed implementation, a so-called *mutation* of the original implementation. If the test fails now, this means that the state machine is detailed in the sense that it is not an abstraction of the changed implementation. If the test is still successful, the opposite is true: the state machine is abstract enough to ignore the change in the implementation (at least for the current input sequence).

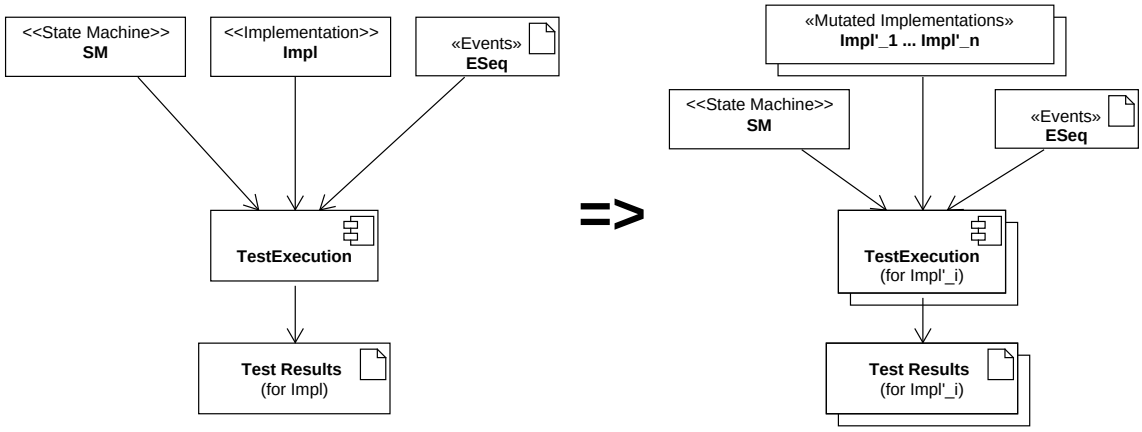

Fig 6. Repeating test on mutated implementations

Figure 6 illustrates this approach. The left part shows the parallel execution of state machine SM and implementation $Impl$ for the given input event sequence $ESeq$. Since we assume that the state machine is a presumably valid abstraction, the test must have been successful, because otherwise we had a counterexample for a valid abstraction trace.

In the right part, Figure 6 shows the test executions on the mutated implementations $Impl'_1 \ldots Impl'_n$. For each mutation $Impl'_i$, the test is rerun for the same sequence of input events $ESeq$. If the test fails, this is a sign that the state machine SM was not more abstract than the original implementation $Impl$. If the test succeeds, the state machine was more abstract, since the current test is also a valid abstraction trace w.r.t. the mutated implementation $Impl'_i$. This brings us to the central definition of this paper:

Definition 6 (Abstractness)**.** *Let a state machine SM, an implementation $Impl$ and a bridge $B = (SM, Impl)$ be given. Let SM be a presumably valid abstraction of $Impl$ w.r.t. B. Let $Mut(Impl)$ be a fixed set of mutations of $Impl$, let $n = \#Mut(Impl)$ be the number of the considered mutations.*

The abstractness of SM for a given input event sequence $ESeq$ (denoted by $abstr_{ESeq}(SM)$) is a number between 0 and 1 and computed as

$$\boxed{abstr_{ESeq}(SM) = \frac{k}{n}}$$

where k is the number of succeeding tests when running the test for input sequence $ESeq$ on all mutated implementations from $Mut(Impl)$.

A mutation of implementation $Impl$ is obtained by applying a mutation operator on $Impl$. Mutation operators have been thoroughly studied in mutation testing [10]. For Java implementations, mutation operators have been made generally available by the Java compiler in terms of command line options, but one can also use dedicated frameworks such as PIT [14] to mutate an existing implementation. Note that mutating a given implementation is a repeatable transformation. When applied on the same location within the implementation, a mutation operator will always produce the same mutated implementation. The number of mutants ($n = \#Mut(Impl)$) for a given implementation $Impl$ depends on i) the length and complexity of $Impl$ and ii) the set of selected mutation operators.

Modern mutation frameworks such as PIT [14] allow to control on which locations within the implementation code the mutation operators should be applied. For the Pac-Man example given in Section 2. it would desirable to allow mutations only in the implementation classes controlling the behaviour of the game objects and to avoid such mutations, for example, in GUI classes.

6. Related Work

Measuring the quality of modeling artefacts is traditionally done by metrics. Zhang and Hölzl propose in [18] a set of metrics for measuring the complexity of UML state machines. They apply criteria known from traditional object-oriented programming metrics [6] such as FanIn/FanOut to assess the complexity of state machines. Furthermore, they propose the refactoring of complex constructs such as Junction by a set of states and transitions before measuring the complexity to make the measured values comparable. In contrast to our approach, the metric is computed statically and does not take system executions into account.

Model-based testing (MBT) [17, 5, 4] is a widely recognized approach to take models as input for testing an executable system implementation. Compared to traditional pre-/post-state specifications as used by unit tests, the test specification is much more compact. Tools supporting MBT such as ParTeG [13] or SpecExplorer [11] support state machines (or similar notations) as a description of the expected system behaviour. From this input model, most MBT-tools generate traditional unit tests, which are then executed. If all tests succeed, then the implemented system hehaves (presumably) as specified by the state machine. MBT tools provide the ability to define a bridge between

state machine and implementation. They use elaborated techniques to generate the 'right' test cases by analysing the implementation code, what makes it a white-box approach.

7. Conclusion

This paper presented a novel approach to measure the abstractness of state machines. For this measurement, beside the state machine, a correct implementation and a bridge from the state machine to the implementation is needed. The abstractness is a value between 0 and 1 and encodes the 'semantic distance' between state machine and implementation.

Our approach is highly flexible since it allows to combine any state machine with any possible implementation thanks to the flexible bridging mechanism. To realize a bridge, the user has to implement an adapter allowing the state machine to access the internal state of the implementation at runtime. A second adapter realizes the mapping of input events for the state machines to method calls (or any other signals) of the implementation.

References

[1] Martin Abadi and Leslie Lamport, "The existence of refinement mappings", *Theoretical Computer Science*, **82** (1991), 253–284.

[2] Thomas Baar, "SSMA – Simple State Machine Analyzer", https://github.com/thomasbaar/simplesma.

[3] Thomas Baar, "Verification Support for a State-Transition-DSL Defined with Xtext", *Proceedings of the 10th International Andrei Ershov Informatics Conference, PSI*, Lecture Notes in Computer Science, **9609**, Springer, 2015, 50–60.

[4] Robert V. Binder, Bruno Legeard, Anne Kramer, "Model-based testing: where does it stand?", *Communications of the ACM*, **58** (2015), 52–56.

[5] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, Alexander Pretschner, *Model-Based Testing of Reactive Systems, Advanced Lectures*, Lecture Notes in Computer Science, **3472**, Springer, 2005.

[6] Shyam R. Chidamber, Chris F. Kemerer, "A metrics suite for object oriented design", *IEEE Transactions on Software Engineering*, **20**:6 (1994), 476–493.

[7] David Harel, "Statecharts: A visual formalism for complex systems", *Science of Computer Programming*, **8**:3 (1987), 231–274.

[8] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, Mark B. Trakhtenbrot, "STATEMATE: A working environment for the development of complex reactive systems", *IEEE Transactions on Software Engineering*, **16**:4 (1990), 403–414.

[9] Itemis, "Yakindu", http://statecharts.org/.

[10] Yue Jia, Mark Harman, "An analysis and survey of the development of mutation testing", *IEEE Transactions on Software Engineering*, **37**:5 (2011), 649–678.

[11] Microsoft, "SpecExplorer", https://msdn.microsoft.com/en-us/library/ee620411.aspx.

[12] Object Management Group, "Unified Modeling Language (UML), version 2.5", http://www.omg.org/spec/UML/2.5/.

[13] Stephan Weißleder, "Partition Test Generator (ParTeG)", http://parteg.sourceforge.net/.

[14] Pitest, "PIT Mutation Testing", http://pitest.org/.

[15] QuantumLeaps, “QM ™”, http://www.state-machine.com/qm/.

[16] Bernhard Schätz, *Model-Based Development of Software Systems: From Models to Tools*, Technical University Munich, 2009.

[17] Mark Utting and Bruno Legeard, *Practical Model-Based Testing: A Tools Approach*, Morgan Kaufmann, 2007.

[18] Gefei Zhang and Matthias Hölzl, “A set of metrics for states and transitions in UML state machines”, *Proceedings of the 2014 Workshop on Behaviour Modelling – Foundations and Applications*, ACM, 2014.

Баар Т., "К вопросу об измерении уровня абстракции диаграмм состояний на основе тестирования мутаций", *Моделирование и анализ информационных систем*, **24**:6 (2017), 691–703.

DOI: 10.18255/1818-1015-2017-6-691-703

Аннотация. Система обозначений диаграмм состояний (state machines) широко применяется в качестве формального средства описания поведения систем. Обычно для одной и той же программной системы можно создать много разных диаграмм состояний. Некоторые из этих моделей могут оказаться эквивалентными, но во многих случаях разные диаграммы состояний описывают одну и ту же систему на разных уровнях абстракции. В этой статье мы предлагаем подход, позволяющий провести фактическое измерение уровня абстракции диаграмм состояний по отношению к заданной реализации программной системы. Диаграмма состояний считается тем менее абстрактной, чем ближе она концептуально к реализованной системе. Согласно нашему подходу эта отдаленность диаграммы состояний от реализации системы измеряется путем применения критерия покрытия, используемого для тестирования мутации программного обеспечения. Уровень абстракции диаграмм состояний можно рассматривать как новый вид метрики. Что касается других метрик, то знание значения уровня абстракции заданной диаграммы состояний дает возможность оценить ее качество в числовых терминах. В тех проектах по разработке программного обеспечения, которые начинаются с построения модели, метрика абстракции может помочь избежать деградации моделей, поскольку она позволяет измерить фактическое отдаление спецификации поведения системы, представленной в виде диаграммы состояний, от текущей реализации системы. В отличие от прочих метрик для диаграмм состояний уровень абстракции нельзя вычислить статически, основываясь лишь на структуре самой диаграммы; для этого нужно сравнивать выполнения диаграмм состояний и соответствующую реализацию системы. Статья публикуется в авторской редакции.

Ключевые слова: разработка программного обеспечения на основе моделей, метрика, диаграмма состояний, тестирование мутаций

Об авторе:
Баар Томас, orcid.org/0000-0002-8443-1558, профессор,
Берлинская Высшая школа техники и экономики,
Wilhelminenhofstrasse 75 A, D-12459, Berlin, Germany,
e-mail: thomas.baar@htw-berlin.de

Моделирование и анализ информационных систем. Т. 24, №6 (2017), с. 704–717
Modeling and Analysis of Information Systems. Vol. 24, No 6 (2017), pp. 704–717

DOI: 10.18255/1818-1015-2017-6-704-717

UDC 519.686.4

Jolie Static Type Checker: a Prototype

de Carvalho D., Mazzara M., Mingela B., Safina L., Tchitchigin A., Troshkov N.

Received September 8, 2017

Abstract. Static verification of a program source code correctness is an important element of software reliability. Formal verification of software programs involves proving that a program satisfies a formal specification of its behavior. Many languages use both static and dynamic type checking. With such approach, the static type checker verifies everything possible at compile time, and the dynamic one checks the remaining. The current state of the Jolie programming language includes a dynamic type system. Consequently, it allows avoidable run-time errors. A static type system for the language has been formally defined on paper but lacks an implementation yet. In this paper, we describe a prototype of Jolie Static Type Checker (JSTC), which employs a technique based on a SMT solver. We describe the theory behind and the implementation, and the process of static analysis. The article is published in the authors' wording.

Keywords: microservice, static analysis, Jolie programming language

For citation: de Carvalho D., Mazzara M., Mingela B., Safina L., Tchitchigin A., Troshkov N., "Jolie Static Type Checker: a Prototype", *Modeling and Analysis of Information Systems*, **24**:6 (2017), 704–717.

About the authors:
Daniel de Carvalho, orcid.org/0000-0003-1473-8551, PhD, Innopolis University,
1 Universitetskaya ul., Innopolis, Respublika Tatarstan, 420 000 Russia, e-mail: d.carvalho@innopolis.ru

Manuel Mazzara, orcid.org/0000-0002-3860-4948, PhD,
Innopolis University, Russia, e-mail: m.mazzara@innopolis.ru

Bogdan Mingela, orcid.org/0000-0003-1384-7078, student
Innopolis University, Russia, e-mail: b.mingela@innopolis.ru

Larisa Safina, orcid.org/0000-0002-4490-7451, researcher,
Innopolis University, Russia, e-mail: l.safina@innopolis.ru

Alexander Tchitchigin, orcid.org/0000-0001-9286-6116,
Typeable.io LLC, Russia, email: sad.ronin@gmail.com

Nikolay Troshkov, orcid.org/0000-0002-5151-9940, student,
Innopolis University, Russia, e-mail: n.trshkv@gmail.com

1. Introduction

The microservice architecture is a style inspired by service-oriented computing that promises to change the way in which software is perceived, conceived and designed [17]. The trend of migrating monolithic architectures into microservices to reap benefits of scalability is growing fast today [7, 5]. Jolie [20] is the only language natively supporting microservice architectures [9] and, currently, has dynamic type checking only.

Static type checking is generally desirable for programming languages improving software quality, lowering the number of bugs and preventing avoidable errors. The

idea is to allow compilers to identify as many issues as possible before actually run the program, and therefore avoid a vast number of trivial bugs, catching them at a very early stage. Despite the fact that, in the general case interesting properties of programs are undecidable [26], static type checking, within its limits, is an effective and well established technique of program verification. If a compiler can prove that a program is well-typed, then it does not need to perform dynamic safety checks, allowing the resulting compiled binary to run faster.

A static type system for Jolie has been exhaustively and formally defined only on paper [25], but still lacks an implementation. The obstacles of programming in a language without a static type analyzer have been witnessed by Jolie developers, especially by newcomers. However, implementing such system is a non trivial task due to technical challenges both of general nature and specific to the language. In this paper, we introduce and describe the Jolie Static Type Checker (JSTC), building on top of the previous work on the Jolie programming language [20]. Our approach follows the formal derivation rules as defined in [25]. The project is built as a Java implementation of source code processing and verification via Z3 SMT solver [21] and it has to be intended as our community contribution to the Jolie programming language [2].

Section 2. recalls the basic of Jolie and section 3. discusses related work. The description of the static type-checking and the system architecture can be found in Section 4., while Section 6. draws conclusive remarks and discusses open issues.

2. Background

Microservices [6] is an architectural style evolved from Service-Oriented Architectures [12]. According to this approach, applications are composed by small independent building blocks that communicate via message passing. These composing parts are indeed called microservices. This paradigm has seen a dramatic growth in popularity in recent years [24]. Microservices are not limited to a specific technology. Systems can be built using a wide range of technologies and still fit the approach. In this paper, however, we support the idea that a paradigm-based language would bring benefit to development in terms of simplicity and development cost.

Jolie is the first programming language constructed above the paradigm of microservices: each component is autonomous service that can be deployed separately and operated by running in parallel processes. Jolie comprises formally-specified semantics, inspired by process calculi such as CCS [18] and the π-calculus [19]. As for practical side, Jolie was inspired by standards for Service-Oriented Computing such as WS-BPEL [34] and the attempts of formalizing it [13]. The composition of both theoretical and practical aspects allows Jolie to be the preferred candidate for the application of modern research methodologies, e.g. runtime adaptation, process-aware web applications, or correctness-by-construction of concurrent software.

The basic abstraction unit of Jolie is the microservice [6]. It is based on a recursive model where every microservice can be easily reused and composed to obtain, in turn, other microservices. Such approach allows distributed architecture and guarantees simple management of all components, which reduces maintenance and development effort. Microservices communicate and work together by sending messages to each other. In

Jolie, messages are represented in tree structure. A variable in Jolie is a path in a data tree and the type of a data tree is a tree itself with the same shape but containing types of corresponding data. Equality of types must therefore be handled with that in mind. Variables do not require declaration before use, therefore the manipulation of the program state must be inferred. Communications are type checked at runtime, when messages are sent or received. Type checking of incoming messages (at runtime?) is especially relevant, since it could moderate the consequences of errors.

The Jolie language is constructed in three layers: the behavioural layer operates with the internal actions of a process and the communication it performs as seen from the process' point of view, the service layer deals with the underlying architectural instructions and the network layer deals with connecting communicating services.

Other workflow languages are capable of expressing orchestration of (micro)services the same way Jolie can do, for example WS-BPEL [34]. WS-BPEL allows developers to describe workflows of services and other communication aspects (such as ports and interfaces), and it has been also shown how dynamic workflow reconfiguration can be expressed [15]. However, WS-BPEL has been designed for high-level orchestration, while programming the internal logic of a single micro-service requires fine-grained procedural constructs. Here Jolie works better.

3. Related work

The implementation of a static type checker for Jolie is part of a broader attempt to enhance the language for practical use. Previous work on the type system has been done, however focusing mostly on dynamic type checking. Safina extended the dynamic type system as described in [28], where type choices have been added in order to move computation from a process-driven to a data-driven approach.

The idea to integrate dynamic and static type checking with the introduction of refinement types, verified via SMT solver, has been explored in [31]. The integration of the two approaches allows a scenario where the static verification of internal services and the dynamic verification of (potentially malicious) external services cooperates in order to reduce testing effort and enhancing security.

The idea of using SMT Solvers for static analysis, in particular in combination with other techniques, has been successfully adopted before for other programming languages, for example LiquidHaskell and F*. LiquidHaskell [11][1] is a notable example of implementation of Liquid Types (*Logically Qualified Data Types*) [27]. It is a static verification technique combining *automated deduction* (SMT solvers), *model checking* (Predicate Abstraction), and *type systems* (Hindley-Milner inference). Liquid Types have been implemented for several other programming languages. The original paper presented an OCaml implementation. F* [36] instead an ML-like functional programming language specifically designed for program verification. The F* type-checker uses a combination of SMT solving and manual proofs to guarantee correctness.

Another direction in developing static type checking for Jolie is creating a verified type checker[2] by means of proof assistant instead of SMT solver [1]. Proof assistant

[1]Online demo at http://goto.ucsd.edu/~rjhala/liquid/haskell/demo/
[2]https://github.com/ak3n/jolie

is a software tool needed to assist with the development of formal proofs by human-machine collaboration and helps to ascertain the correctness of them. The type checker is expressed as well-typed program with dependent types in Agda [35]. If the types are well formed, all required invariants and properties are described and expressed in the types of the program meaning that the program is correct. This work is currently in progress and evolves in parallel with ours.

4. Static type-checking implementation

This paper builds on top of Julie Meinicke Nielsen's work [25] at the Technical University of Denmark implementing the type system of the Jolie language. The thesis represents the theoretical foundation for the type checking of the core fragment of the language, which excludes recursive types, arrays, subtyping of basic types, faults and deployment instructions such as architectural primitives. The work of Nielsen presents the first attempt at formalizing a static type checker for the core fragment of Jolie, and the typing rules expressed there are the core theory behind our static checker.

In Nielsen's work typing rules are represented in the style of type theory where the rules take form of inference rules describing how a type system assigns a type to a syntactic construct of the language [4]. Important addition to classical inference rules is that we use typing context Γ both left and right of a turnstile. This accounts for Jolie-specific possibility to extend a variable's inferred typing tree while analyzing program text which is rooted in dynamic nature of the language. The rules are then applied by the type system to determine if a program is well typed or not. The main typing rules will be presented in the following of this paper.

The implementation of JSTC consists of two system components. Firstly, a Java program accepts the source code of a Jolie program, builds an abstract syntax tree (AST), visits it and produces a set of logical assertions written in SMT Lib [3] language. At the second phase, the generated assertions are feed into Z3 solver. The basic idea is to implement, for each Jolie node[3], methods containing statements expressed in the SMT Lib syntax. These statements can then be processed via a solver. In Figure 1 the overall process is pictorially represented and details are described in section 4.4.

The concept of SMT solvers is closely related to logical theorems. Logic, especially in the field of proof theory, considers theorems as statements of a formal language. Existence of such logical expressions allows to formulate a set of axioms and inference rules to formalize the typing rules for each of Jolie syntax nodes and then perform the validation of the nodes using constructed theorems. Consequently, the Jolie typing rules are the specific cases of logical theorems, that are used in the project. The concept is implied from software verification fundamentals [10].

Since Jolie program may contain complex expressions with function calls, it is also necessary to consider data structures representing a match between names and expressions, in order to be able to avoid inconsistency and redundancy, that are likely to cause conflicts during type-checking. The project implementation considers using a stack during the recursive checking of the nodes as illustrated in section 4.4.

[3]Any syntax unit is considered a node. It can be a logical or arithmetic expression, an assignment; a condition; a loop etc. Those nodes comprise the abstract syntax tree.

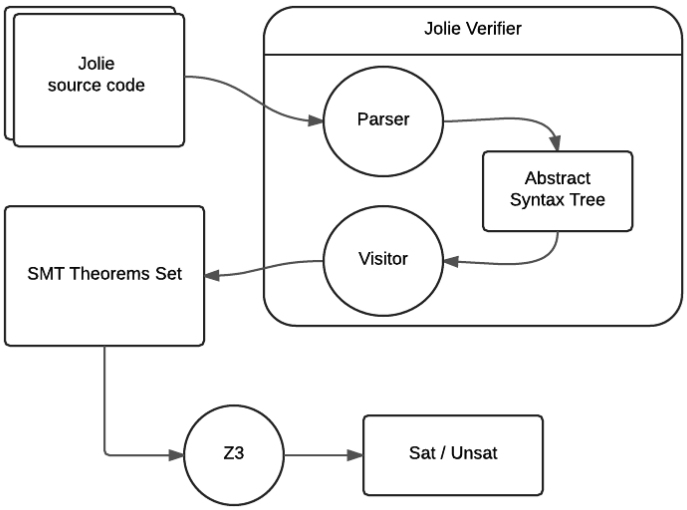

Fig 1. Process of Type Checking in the Jolie Verifier

The decision of using an SMT-solver, instead of more lightweight techniques, was made in order to allow a future straightforward integration of Refinement Types into the type checker, objective on which our team is already working [28, 31]. Furthermore, relying on a solid existing technology allowed us to prototype and release a proof of concept of the type checker in a shorter period of time.

4.1. Jolie verifier

The Java program reuses an existing structure of a Visitor pattern developed for Jolie interpreter and also used in a previous project for formatting Jolie source code[4]. It accepts processed Jolie program source code in the form of AST and performs traversing. For each kind of node the system creates one or more logical formulas written in SMT Lib syntax, which are then stored into a file[5]. The verifier targets assignments, conditions, and other cases of variables usage where type consistency can be violated.

4.2. SMT Solver

Z3 carries out the main burden of program verification. Z3 is an SMT solver from Microsoft Research [21]. It is targeted at solving problems that arise in software verification and software analysis. Given a set of formulas that was previously created by the verifier in Java, Z3 processes it and returns whether this set is satisfiable or not. In case of any contradiction in the set, the solver will signal that the overall theorem is not satisfiable, therefore alerting that the input program is not consistent in terms of types usage.

[4] https://github.com/nickaleks/jolie
[5] At the current implementation state the theorems are collected in a single data element.

4.3. Typing rules

Our objective is to accurately translate Jolie typing rules into SMT statements, therefore allowing static type checking[6]. The foreground activity so far is producing the set of statements for the constructs of the behavioural layer of Jolie. The layer describes the internal actions of a process and the communications it performs as seen from the process' point of view. The layer is chosen for the first phase of the development because it forms foundation of the syntactical structures of Jolie. Also there is a similarity of the layer with common programming languages in a sense of the abstraction level.

All statements at the behavioural layer of Jolie are called behaviours. We write $\Gamma \vdash_B B \triangleright \Gamma'$ to indicate a behaviour B, typed with respect to an environment Γ, which updates Γ to Γ' during type checking [25]. The context Γ consists of variables known to the type checker so far and their respective types.

There are some core rules presented and described below.

T-Nil. The typing rule for a nil behaviour is an axiom. In the conclusion the typing environment is not changed, since the nil statement doesn't affect the typing environment.

$$\frac{}{\Gamma \vdash_B 0 \triangleright \Gamma}$$

T-If-Then-Else. The rule for typing an if statement is standard: an if statement is typeable if its condition has type bool, and the type checking of its branches perform the same updates to the environment. We require the branches to perform the same updates because we do not know which branch will be taken. The else part may also be omitted and B_2 may be replaced by an empty behaviour. The conditional typing statement is the following:

$$\frac{\Gamma \vdash e : bool \quad \Gamma \vdash_B B_1 \triangleright \Gamma' \quad \Gamma \vdash_B B_2 \triangleright \Gamma'}{\Gamma \vdash_B if(e)\ B_1\ else\ B_2 \triangleright \Gamma'}$$

T-While. The rule for typing a while statement is also standard: a while statement is typeable if its condition has type bool, and type checking its body has no influence on the typing environment.

$$\frac{\Gamma \vdash e : bool \quad \Gamma \vdash_B B \triangleright \Gamma}{\Gamma \vdash_B while(e) B \triangleright \Gamma}$$

Above, it is required that the body of a while loop does not change the typing of variables because we do not know whether the body will be executed at all, and for how many times. We also require that expression e is type checked against type bool.

T-Seq. A sequence statement typed with respect to an environment is typeable if its first component is typeable with respect to the environment and its second component is typeable with respect to the update of the environment performed by the first component. The update of the environment performed by the sequence statement is the update performed by the second component with respect to the update performed by the first component.

[6]Please note that, at the moment, not all the rules in [25] have been implemented.

$$\frac{\Gamma \vdash_B B_1 \triangleright \Gamma' \quad \Gamma' \vdash_B B_2 \triangleright \Gamma''}{\Gamma \vdash_B B_1; B_2 \triangleright \Gamma''}$$

Thus, fundamental typing rules of the behavioural layer of Jolie programming language are presented and explained for further topic revelation.

4.4. Typing rules to SMT translation

Here we will examine an example of the conditional rule translation in order to understand the procedure in detail.

The typing rule of the *if* statement does not contradict intuition. The statement is typeable when its condition expression is boolean, and the execution of both its branches brings the same updates to the environment. This means that the set of matches between expressions and variables with their types remains the same with no difference from a branch choice. This is necessary since it is not possible to predict what branch will be executed at runtime[7].

The full implementation is available on Github[8]. In Figure 2 we show the Java fragment that builds the corresponding SMT statement.

```
(declare-const key Term)

(declare-const $$__term_id_4 Term)
(assert (hasType $$__term_id_4 string))
(assert (sameType key $$__term_id_4))
(assert (hasType key string))
(declare-const $$__term_id_5 Term)
(declare-const animals Term)
(declare-const animals.cat Term)

(declare-const $$__term_id_10 Term)
(assert (hasType $$__term_id_10 string))
(assert (sameType animals.cat $$__term_id_10))
(assert (hasType animals.cat string))
(declare-const $$__term_id_11 Term)
(declare-const animals.DYNAMIC_PATH_$$__term_id_14 Term)

(declare-const $$__term_id_19 Term)
(assert (hasType $$__term_id_19 int))
(assert (sameType animals.DYNAMIC_PATH_$$__term_id_14 $$__term_id_19))
(assert (hasType animals.DYNAMIC_PATH_$$__term_id_14 int))
```

Fig 2. Code in Java for checking conditionals

The code structure represents basic steps to achieve a record with corresponding SMT statements of the block as a result. Firstly, a condition of the *if* statement is separated from the body. Then the condition is sent to be checked using the same visitor class. Eventually after the last 'recursion' step the condition is put in the stack of terms, which contains any terms (expressions, variables etc.) processed during the checking. So

[7]The *else* part may also be omitted and B_2 may be replaced by an empty behavior.
[8]https://github.com/innopolis-jolie-smt-typechecker/jolie

the term corresponding to the condition is expected to be on top of the stack. Then an assertion that says the condition term is boolean is written. Afterwards the body is processed using one of the other overloads of the visitor. These steps can be repeated in case of existence of nested conditional statements. In the end of the method the *else* branch body of the very first *if* is processed if it is present. There is also an important note that the conditional statement does not impose any other direct type restrictions besides the condition term that is confirmed by the mentioned typing rule. Other implemented nodes can be seen in the source mentioned above.

The Jolie verifier takes some input for processing. Let us consider a simple piece of Jolie code of Figure 3 with a conditional statement.

```
1 a = 2;
2 b = 3;
3 if ( a > b ) {
4   println@Console( a + b )()
5 } else{
6   println@Console( "Hello!" )()
7 }
```

Fig 3. Code in Jolie

```
a = 2;
b = 3;
if ( 5 ) {
  println@Console( a + b )()
} else{
  println@Console( "Hello!" )()
}
```

Fig 4. Variation of the code of Figure 3

In the case everything works, none of the typing rules are violated. Z3 agrees with the opinion and results in 'sat', that means the program state is satisfiable. Figure 5 lists the SMT statements representing the condition processing.

```
(declare-const $$__term_id_10 Term)
(assert (hasType $$__term_id_10 bool))

(assert (hasType $$__term_id_10 bool))
```

Fig 5. Z3 code for the conditional of Figure 3

The first assertion is made based on an expression type determination: the expression $a > b$ is boolean. The second one is imposed by the typing rule: the condition expression must be boolean. In this case there is no contradiction between the two assertions.

If the condition is replaced with some other expression the typing rule may be violated. The corresponding example of incorrect conditional expression is shown in Figure 4 and the constructed SMT statements for the example are given in Figure 6.

```
(declare-const $$__term_id_10 Term)
(assert (hasType $$__term_id_10 int))

(assert (hasType $$__term_id_10 bool))
```

Fig 6. Z3 code for the conditional of Figure 4

Now the contradiction between the assertions is notable. The checker decided the expression to be an integer, which is correct. But the restriction on a condition type from the typing rule simply contradicts the actual type. Consequently Z3 results in 'unsat'. This means the assertions representing program's typing are unsatisfiable and incorrect in terms of the considered static type checking analysis.

5. Evaluation

The question of how to prove correctness of verification tools has always been widely discussed. How can we be sure that the output of such tool is correct? It can be poorly written, or the hardware could malfunction. However, in most cases we tend to trust verification tools, and in our project we have to make sure that this tool is as trustworthy as any other. The general solution is testing. Verification of the correctness the code being written was continuously performed during the development process. The Jolie Team created a collection of examples of Jolie programs[9]. The verification results of some of them are presented in this section.

5.1. An unsatisfiable model

The overall purpose of the type checker is to find inconsistency in (assumed) type usage. The program listed in Figure 7 is the most basic example of a program with inconsistent types. The variable *myInt* is assigned an integer first, and then a string. The current design of the type checker disallows this behavior.

```
main
{
   myInt = 15;
   myInt = "fifteen"
}
```

Fig 7. Jolie code

```
1  (declare-const myInt Term)
2
3  (declare-const $$__term_id_4 Term)
4  (assert (hasType $$__term_id_4 int))
5  (assert (sameType myInt $$__term_id_4))
6  (assert (hasType myInt int))
7
8  (declare-const $$__term_id_10 Term)
9  (assert (hasType $$__term_id_10 string))
10 (assert (sameType myInt $$__term_id_10))
11 (assert (hasType myInt string))
```

Fig 8. Z3 code corresponding to Figure 7

The resulting set of SMT theorems is listed in Figure 8.

The model is unsatisfiable. Assertions on the lines 4-6 restrict type of *myInt* to integer, whereas assertions on the lines 9-11 ensure that the same variable should be of type string. These assertions cannot be evaluated to be true in the same model. Therefore the program is considered ill-typed.

[9] https://github.com/jolie/examples

5.2. A satisfiable model

In case everything in the program code is correct in terms of type consistency, the type checker should evaluate the resulting SMT model as satisfiable. It also should ignore cases that have not being processed properly yet, without giving any false positives. The program listed in Figure 9, in fact, an inconsistency in types usage. The line 8 reassigns a variable to be of type integer, whereas at the line 5 the same variable was introduced as a variable of type string. However, as long as this assignment include a statement with a dynamic key, the type checker ignores it. The reason for this is inability to determine which variable this path will point to at the moment of execution.

```
1 main
2 {
3   key = "cat";
4   animals.cat = "I am a cat";
5   animals.(key) = 13
6 }
```

Fig 9. Jolie code with dynamic key

The resulting set of SMT assertions is listed in Figure 10. The workaround for the dynamic keys issue can be seen at Lines 20 and 21. Any time a variable is used in order to construct another variable path, the whole Term is getting a unique identifier marked with the *DYNAMIC_PATH* substring. This way it will not interfere with any other assertion, thus not affecting the overall model satisfiability.

```
1  (declare-const key Term)
2
3  (declare-const $$__term_id_4 Term)
4  (assert (hasType $$__term_id_4 string))
5  (assert (sameType key $$__term_id_4))
6  (assert (hasType key string))
7  (declare-const $$__term_id_5 Term)
8  (declare-const animals Term)
9  (declare-const animals.cat Term)
10
11 (declare-const $$__term_id_10 Term)
12 (assert (hasType $$__term_id_10 string))
13 (assert (sameType animals.cat $$__term_id_10))
14 (assert (hasType animals.cat string))
15 (declare-const $$__term_id_11 Term)
16 (declare-const animals.DYNAMIC_PATH_$$__term_id_14 Term)
17
18 (declare-const $$__term_id_19 Term)
19 (assert (hasType $$__term_id_19 int))
20 (assert (sameType animals.DYNAMIC_PATH_$$__term_id_14 $$__term_id_19))
21 (assert (hasType animals.DYNAMIC_PATH_$$__term_id_14 int))
```

Fig 10. Z3 code corresponding to the Jolie code of Figure 9

6. Conclusions and future works

Jolie is the first programming language specifically oriented to the microservice architecture. It has been shown how software attributes such as extensibility, modifiability and consistency can significantly benefit from a migration into the microservice paradigm [7, 5]. Projects run by our team demonstrated the efficacy of the paradigm and of the Jolie programming language in the field of ambient intelligence and smart buildings [29, 30]. Social networks implementation would also benefit from a reorganization of the software architecture [16]. Local projects, and beyond that a number of projects worldwide involving the use of Jolie, would immensely benefit from a fully stable implementation of the Jolie Static Type Checker.

Static type checking allows compilers to identify certain programming mistakes (that violate types) at compile time, i.e. before actually running the program. Therefore a vast number of trivial bugs can be caught and fixed at a very early stage of the software life-cycle. In this paper we described JSTC, a static type checker for the Jolie programming language which natively supports microservices. A static type system for the language has been exhaustively and formally defined on paper, but so far still lacked an implementation. We introduced our ongoing work on a static type checker and presented some details of the implementation. The type checker prototype, at the moment, consists of a set of rules for the type system expressed in SMT Lib language. The actual implementation covers operations such as assignments, logical statements, conditions, literals and comparisons.

JSTC is already able to validate programs, as it has been shown in this paper. However, it works with certain assumptions. The main assumption is that programs do not contain implicit type casts. The Jolie language allows implicit type casts, however, their behavior is very complex. Handling such situations is an open issue left for future development and future versions. Two other major issues have not been addressed.

Variable types can be changed at runtime. This strictly depends on the approach that has been chosen. Generally, static typing guarantees that a variable has a type that cannot be changed after declaration or assignment. However, Jolie allows this operation. We need to determine which behavior we expect from the type checker, thus deciding how to process type changes.

Implicit type casts in Jolie are ambiguous. This is a major problem, and further research is required in order to find a solution. While Jolie allows implicit type casts, sometimes the result of a cast is not obvious. For example, casting a negative Integer to Boolean will result in a False. This is an unexpected behavior when compared to other programming languages. There may be a solid rationale for this, however, we need to investigate all cases and make sure that the type checker works accordingly to the Jolie actual behavior, and not to the expected one.

JSTC future releases will need to be validated in real-life applications. The plan is to use the Jolie programming language and the type checker as a basis for the development of future research projects, the same way was done in [29] and [30]. Potential application scenarios are cognitive architecture [32], automotive systems [8] and smart houses [23].

References

[1] Akentev E., Tchitchigin A., Safina L., Mazzara M., "Verified type checker for Jolie programming language", https://arxiv.org/pdf/1703.05186.pdf.

[2] Bandura A., Kurilenko N., Mazzara M., Rivera V., Safina L., Tchitchigin A., "Jolie Community on the Rise", *9th IEEE International Conference on Service-Oriented Computing and Applications, SOCA*, 2016.

[3] Barrett C., Stump A., Tinelli C., "The SMT-LIB Standard. Version 2.0", *Proceedings of the 8th international workshop on satisfiability modulo theories*, Edinburgh, England, 2010.

[4] Cardelli L., "Type Systems", *ACM Computing Surveys*, **28** (1996), 263–264.

[5] Dragoni N., Dustdar S., Larsen S. T., Mazzara M., "Microservices: Migration of a Mission Critical System", https://arxiv.org/abs/1704.04173.

[6] Dragoni N., Giallorenzo S., Lluch-Lafuente A., Mazzara M., Montesi F., Mustafin R., Safina L., "Microservices: yesterday, today, and tomorrow", *Present and Ulterior Software Engineering*, ed. Bertrand Meyer, Manuel Mazzara, Springer, 2017, 195–216.

[7] Dragoni N., Lanese I., Larsen S. T., Mazzara M., Mustafin R., Safina L., "Microservices: How To Make Your Application Scale", *A.P. Ershov Informatics Conference (the PSI Conference Series, 11th edition)*, Lecture Notes in Computer Science, Springer, 2017.

[8] Gmehlich R., Grau K., Loesch F., Iliasov A., Jackson M., Mazzara M., "Towards a formalism-based toolkit for automotive applications", *1st FME Workshop on Formal Methods in Software Engineering (FormaliSE 2013)*, IEEE Computer Society, 2013, 36–42.

[9] Guidi C., Lanese I., Mazzara M., Montesi F., "Microservices: a Language-based Approach", *Present and Ulterior Software Engineering*, ed. Bertrand Meyer, Manuel Mazzara, Springer, 2017, 217–225.

[10] Hoare C. A. R., "An Axiomatic Basis for Computer Programming", *Communications of the ACM*, **12** (1969), 576–583.

[11] Jhala R., "Liquid Haskell", https://ucsd-progsys.github.io/liquidhaskell-blog/.

[12] MacKenzie M. C., Laskey K., McCabe F., Brown P. F., Metz R., "Reference model for service oriented architecture 1.0", *OASIS Standard*, **12** (2006).

[13] Mazzara M., *Towards Abstractions for Web Services Composition*, Ph.D. Thesis, University of Bologna, 2006, http://www.informatica.unibo.it/it/ricerca/technical-report/2006/UBLCS-2006-08.

[14] Mazzara M., Abouzaid F., Dragoni N., Bhattacharyya A., "Toward Design, Modelling and Analysis of Dynamic Workflow Reconfigurations – A Process Algebra Perspective", *Web Services and Formal Methods*, 8th International Workshop, WS-FM 2011, Lecture Notes in Computer Science, **7176**, 2011, 64–78.

[15] Mazzara M., Abouzaid F., Dragoni N., Bhattacharyya A., "Design, Modelling and Analysis of a Workflow Reconfiguration", *Proceedings of the International Workshop on Petri Nets and Software Engineering*, Newcastle upon Tyne, UK, June 20–21, 2011, 10–24.

[16] Mazzara M., Biselli L., Greco P. P., Dragoni N., Marraffa A., Qamar N., Simona de Nicola, "Social networks and collective intelligence: a return to the agora", *Social Network Engineering for Secure Web Data and Services*, IGI Global, 2013.

[17] Mazzara M., Mustafin R., Safina L., Lanese I., "Towards Microservices and Beyond: An incoming Paradigm Shift in Distributed Computing", https://arxiv.org/pdf/1610.01778.pdf.

[18] Milner R., *Communication and concurrency*, Prentice Hall International (UK) Ltd., 1995.

[19] Milner R., *Communicating and Mobile Systems: The π-calculus*, Cambridge University Press, 1999.

[20] Montesi F., Guidi C., Zavattaro G., "Service-Oriented Programming with Jolie", *Web Services Foundations*, Springer, 2014, 81–107.

[21] Moura de L., Bjørner N., "Z3: An Efficient SMT Solver", *Tools and Algorithms for the Construction and Analysis of Systems*, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008 (Budapest, Hungary, March 29–April 6, 2008), Lecture Notes in Computer Science, **4963**, Springer, 2008, 337–340.

[22] Moura de L., Bjørner N., "Satisfiability modulo theories: An appetizer", *Formal Methods: Foundations and Applications*, 12th Brazilian Symposium on Formal Methods, SBMF 2009, Lecture Notes in Computer Science, **5902**, Springer, 2009, 23–36.

[23] Nalin M., Baroni I., Mazzara M., "A Holistic Infrastructure to Support Elderlies' Independent Living", *Encyclopedia of E-Health and Telemedicine*, IGI-Global, 2016, 591–605.

[24] Newman S., *Building microservices*, O'Reilly Media, Inc., 2015.

[25] Nielsen J. M., *A Type System for the Jolie Language*, Master thesis, Technical University of Denmark, 2013.

[26] Rice H. G., "Classes of Recursively Enumerable Sets and Their Decision Problems", *Trans. Amer. Math. Soc.*, **74** (1953), 358–366.

[27] Rondon P. M., Kawaguci M., Jhala R., "Liquid Types", *SIGPLAN Not.*, **43**:6 (2008), 159–169.

[28] Safina L., Mazzara M., Montesi F., Rivera V., "Data-driven Workflows for Microservices (genericity in Jolie)", *Proc. of the 30th IEEE International Conference on Advanced Information Networking and Applications (AINA 2016)*, 2016.

[29] Salikhov D., Khanda K., Gusmanov K., Mazzara M., Mavridis N., "Microservice-based IoT for Smart Buildings", *Proceedings of the 31st International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, 2017.

[30] Salikhov D., Khanda K., Gusmanov K., Mazzara M., Mavridis N., "Jolie Good Buildings: Internet of things for smart building infrastructure supporting concurrent apps utilizing distributed microservices", *Proceedings of the 1st International conference on Convergent Cognitive Information Technologies*, 2016.

[31] Tchitchigin A., Safina L., Mazzara M., Elwakil M., Montesi F., Rivera V., "Refinement types in Jolie", *Spring/Summer Young Researchers Colloquium on Software Engineering, SYRCoSE*, 2016.

[32] Vallverdú J., Talanov M., Distefano S., Mazzara M., Tchitchigin A., Nurgaliev I., "A cognitive architecture for the implementation of emotions in computing systems", *Biologically Inspired Cognitive Architectures*, **15** (2016), 34 – 40.

[33] "Z3", https://github.com/Z3Prover/z3.

[34] "OASIS. Web Services Business Process Execution Language", http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html.

[35] "Agda", http://wiki.portal.chalmers.se/agda/pmwiki.php.

[36] "F*", https://www.fstar-lang.org/.

де Карвальо Д., Маццара М., Мингела Б., Сафина Л., Чичигин А., Трошков Н., "Прототип статического тайп-чекера для языка программирования Jolie", *Моделирование и анализ информационных систем*, **24**:6 (2017), 704–717.

DOI: 10.18255/1818-1015-2017-6-704-717

Аннотация. Статическая верификация исходного кода программы является важным элементом надежности программного обеспечения. Под верификацией предполагается доказательство соответствия поведения программы ее спецификации. Во многих языках программирования используется как статическая, так и динамическая проверка типов. Таким образом, статический тайп-чекер старается проверить все возможное во время компиляции, а динамический проверяет оставшееся. На данный момент язык программирования Jolie имеет динамическую систему типов, что позволяет обнаруживать ошибки только во время выполнения программы. Статическая система типов для языка была формально определена на бумаге, но пока не реализована. В этой статье мы представим прототип статического тайп-чекера для языка программирования Jolie (JolieStaticTypeChecker или JSTC), основанный на SMT-решателе. Мы опишем базовую теорию, необходимую для реализации тайп-чекера, саму реализацию, а также процесс статического анализа программы. Статья публикуется в авторской редакции.

Ключевые слова: микросервисы, статический анализ кода, язык программирования Jolie

Об авторах:
де Карвальо Даниэль, orcid.org/0000-0003-1473-8551, доцент,
Университет Иннополис, Университетская ул., 1, Иннополис, Респ. Татарстан, 420500, Россия
e-mail: d.carvalho@innopolis.ru

Маццара Мануэль, orcid.org/0000-0002-3860-4948, руководитель лаборатории архитектуры и моделей ПО,
Университет Иннополис, Университетская ул., 1, Иннополис, Респ. Татарстан, 420500, Россия
e-mail: m.mazzara@innopolis.ru

Мингела Богдан, orcid.org/0000-0003-1384-7078, студент
Университет Иннополис, Университетская ул., 1, Иннополис, Респ. Татарстан, 420500, Россия
e-mail: b.mingela@innopolis.ru

Сафина Лариса, orcid.org/0000-0002-4490-7451, младший научный сотрудник лаборатории архитектуры и моделей ПО, Университет Иннополис, Университетская ул., 1, Иннополис, Респ. Татарстан, 420500, Россия
e-mail: l.safina@innopolis.ru

Чичигин Александр, orcid.org/0000-0001-9286-6116,
ООО Тайпэбл, Россия, email: sad.ronin@gmail.com

Трошков Николай, orcid.org/0000-0002-5151-9940, студент,
Университет Иннополис, Университетская ул., 1, Иннополис, Респ. Татарстан, 420500, Россия
e-mail: n.trshkv@gmail.com

Моделирование и анализ информационных систем. Т. 24, №6 (2017), с. 718–729
Modeling and Analysis of Information Systems. Vol. 24, No 6 (2017), pp. 718–729

DOI: 10.18255/1818-1015-2017-6-718-729

UDC 004.052.42, 004.4'6, 004.423.42, 004.432.2, 004.438 Eiffel, 519.681.2, 519.682.1

Towards Null Safety Benchmarks for Object Initialization

Kogtenkov A. V.

Received September 11, 2017

Abstract. Null pointer dereferencing remains one of the major issues in modern object-oriented languages. An obvious addition of keywords to distinguish between never null and possibly null references appears to be insufficient during object initialization when some fields declared as never null may be temporary null before the initialization completes. This work identifies the key reasons of the object initialization problem. It suggests scenarios and metrics to be used as the benchmarks to compare solutions of this problem. Finally, it demonstrates application of the benchmarks on the proposed solution for object initialization in Eiffel.

The article is published in the author's wording.

Keywords: null pointer dereferencing, null safety, void safety, object initialization, static analysis, null safety benchmarks

For citation: Kogtenkov A. V., "Towards Null Safety Benchmarks for Object Initialization", *Modeling and Analysis of Information Systems*, **24**:6 (2017), 718–729.

On the author:
Alexander V. Kogtenkov, orcid.org/0000-0003-4873-8306, Doctor of Sciences ETH Zurich
Independent scientist, Podolsk, Russia, e-mail: kwaxer@mail.ru
address for correspondence: MAIS, 14 Sovetskaya str., Yaroslavl, 150003 Russia

1. Introduction

Null pointer dereferencing remains one of the major day-to-day issues in software industry. To construct a sound null-safe type system, most solutions of the problem for object-oriented languages add a notion of non-null and maybe-null types, usually expressed with additional type annotations. Such annotations would be sufficient to solve the null safety problem if objects could be created in an atomic operation, so that all fields marked as non-null were initialized with object references. Unfortunately, sequential initialization of the fields breaks the solution. Several proposals solving the object initialization issue [1, 2, 7, 9] suggest extending the type systems further to identify objects that are not completely initialized.

Instead of tweaking the type system, I proposed a static-analysis-based solution [10]. This solution relies on the validity rules checked during compilation to cover the cases left by the type system checks. Combined with removal of annotations for local variables [5], the solution is very effective in practice for avoiding null dereferencing problems, whilst

having low cost in compilation time: (a) it reduces the annotation overhead compared to previous solutions for the null-safe programming; (b) it correctly accepts or rejects code of the published examples relevant to this problem [1, 2, 7, 9]; (c) it permits new scenarios, impossible with type-system-based solutions.

But is the solution good enough? How does it compare to other approaches that solve the object initialization problem? In this work I review examples from the previous publications and identify the key reasons that cause the problem. Then I focus on the detailed criteria that can be used to compare different solutions.

The main contributions of this work are:

- identification of the roots of the object initialization problem;
- development of execution scenarios and metrics for benchmarks of different null safety solutions.

2. Motivating examples

Every publication on the problem of object initialization has few examples that authors use either to demonstrate issues with their approach, or to explain how the issues are resolved. I collected all the examples from the publications as well as found in class libraries and divided them into the following cases. I use the differences between some examples to describe common scenarios in section 5.

i. Polymorphic call from a constructor. Manuel Fähndrich and Rustan Leino [1] describe a call to a virtual method on `this` in a superclass constructor. Because subclass fields of the object are not initialized yet, accessing them in the polymorphic call causes `NullReferenceException`. Xin Qi and Andrew C. Myers [7] consider a similar example with a class `Point` and its subclass `CPoint` that adds a color attribute.

ii. Polymorphic callback from a constructor. Accesses to an uninitialized object can be done indirectly. If a superclass constructor passes a reference to the current object as an argument to create another object, this "remote" constructor can call-back on the object where not all fields are initialized yet.

iii. Modification of existing structures. Convenience of the ability to invoke regular procedures inside a constructor can be demonstrated with a mediator pattern [3]. It decouples objects so that they do not know about each other, but still can communicate using an intermediate object, *mediator*. If the communicating objects register themselves with the mediator in the code of their constructors, clients do not need to clutter their code with the calls to register every new communicating object after creating it.

iv. Safety violations. In addition to valid cases, authors usually mention examples that should trigger a compiler error. This aims at the original goal: a sound solution should catch potential null dereferencing at compile time.

v. Circular references. Manuel Fähndrich and Songtao Xia [2] review a linked list example with a sentinel. When a new list is constructed, a special sentinel node is created and it should reference the original list object.

vi. Self-referencing. This is a variant of circular references when an object references itself rather than another object.

3. Practical void safety

3.1. Why is object initialization problematic?

Null safety is complicated for object initialization. To understand why, I suggest to look at how program execution can lead to the null reference exception. Firstly, the object that causes the problem should not complete its initialization, i.e., some of its fields of non-null types should be null. Secondly, this object should be accessible — either directly or through some variables. Thirdly, the information that its initialization is incomplete should be lost. Otherwise, it would be easy to report the error at compile time. Finally, the reference retrieved from the uninitialized field of the object should be dereferenced to trigger the exception. To summarize, there are the following roots of the problem:

- *Non-atomic initialization* of an object leads to the possibility to have fields with null values even when their type is non-null.
- *Aliasing* allows for accessing the same object from an arbitrary point of the program, in particular, from the code that does not expect an incompletely initialized object.
- *Uncontrollable control flow*, interrupting the regular one, makes sequential reasoning about program execution useless.
- *Dereferencing* of an uninitialized field of the incompletely initialized object triggers the exception.

My review focuses solely on the null safe frameworks that use an existing object-oriented language as the basis. This aims at reusing legacy code if possible and preserving known coding techniques and patterns. In particular, I assume that constructors are allowed to execute arbitrary code, not just a sequence of assignments of supplied arguments to the fields. This implies that the current object can be used in the constructor and to escape from it as soon as such uses and escapes are guaranteed to avoid null dereferencing.

To achieve the safety, the solutions extending the type system with new types limit the operations on incompletely initialized objects. For Eiffel, I developed the *practical void safety* solution, briefly described in the next subsection. It specifies the conditions, when dereferencing may be unsafe, and forbids such dereferencing altogether.

3.2. Solution outline

From the point of view of the solution that avoids additional types annotations, all examples from section 2 can be divided into 2 major groups:

- Examples i to iv: – Can the code be reordered to initialize all fields before use?
- Examples v and vi: – Can compile-time rules ensure an object with recursive references to itself is not used as a completely initialized one?

The issue in the first group arises because the current object is passed before all fields of this object are set. The issue in the second group arises because not all fields can be set before passing a reference to the current object. Then, on the one hand, a call on an incompletely initialized object cannot assume all attributes are properly set. On the other hand, a qualified call does not allow seeing what operations on an incompletely initialized object are performed. The *practical void safety* solution disallows qualified calls when some objects are incompletely initialized:

Validity rule. *A constructor is null-safe if it satisfies all the following conditions:*

1. *All fields of the class are set at the end of the constructor.*
2. *Every field is set before it is used.*
3. *Any of the following is true at every execution point:*
   - 3.1 *All fields are properly set.*
   - 3.2 *The reference to the current object is unused before or at the current execution point.*
   - 3.3 *The expression at the execution point is neither of*
     - *a qualified call;*
     - *a creation expression that makes a qualified call.*

Here, the term "current object" means a special entity `this` (in C# and Java) or **Current** (in Eiffel). The term "qualified call" stands for the call with an explicit target (i.e., has the form `target.message`) whereas the term "unqualified call" is used for the call on the current object (i.e., has the form `message`). The rule applies not only to the constructors declared in the class, but also to the inherited ones. The latter should be rechecked in descendants even if the constructors are already checked in the classes where they are defined.

In these conditions and because types of objects that are initialized by the constructors are known, all unqualified calls from the constructors can be resolved at compile time. Therefore, the methods, involved in these calls, can be checked using the same validity rule. This allows for reusing initialization methods without any special annotations. Although the compile-time checks take advantage of the static method resolution, the generated code for constructors can still be shared among different classes and can rely on dynamic dispatch for unqualified calls.

4. Related work

Raw types ***(solve examples i and iv with*** 2+ ***annotations).*** Manuel Fähndrich and K. Rustan M. Leino [1] denote attached types with T^{-} and detachable types with T^{+} and propose to add raw types T^{raw-} to be used for partially initialized objects. If class C has an attribute of type T and some entity has type C^{raw-} then a qualified call to this attribute has type T^{+} regardless of original attachment status of that attribute. An assignment to an entity of a raw type accepts only a source expression of a non-raw non-null type to ensure that if an object becomes fully initialized, it cannot be

uninitialized. Also, by the end of every constructor, every non-null field should be assigned. Unfortunately, the rules for super-class constructors are not directly applicable to languages with multiple class inheritance, like Eiffel. Also, this approach does not support creation of circular references.

Masked types ***(solve examples i to vi with many annotations).*** Xin Qi and Andrew C. Myers [7] address the complete object life cycle. They instrument the type system with so called "masks" representing sets of fields that are not currently initialized. For example, the notation `Node\parent!\Node.sub[l.parent] -> *[this.parent]` for an argument `l` tells that it has a type `Node` and on entry requires that its field `parent` is not set and at the same time fields declared in subclasses of `Node` are not set unless `l.parent` is initialized. On exit the actual argument conforms to the type `Node\*[this.parent]` that indicates that the node object will be completely initialized as soon as its field `parent` is set. The notation is very powerful and goes far beyond null safety, but even with its complexity authors complain that it is not sufficient for real programs.

With masked types the results of a flow-sensitive type analysis are checked against provided specifications, while in *practical void safety* approach they are used to check the Validity rule conditions.

Free and committed types ***(solve examples i and iv to vi with*** 1+ ***annotations).*** Alexander J. Summers and Peter Müller [9] set the following design goals:

1. *Modularity:* the type system can check each class type separately.
2. *Soundness:* the type system is safe, i.e., null pointer exceptions are impossible at run-time.
3. *Expressiveness:* the type system handles common initialization patterns. In particular, it allows objects to escape from constructors and supports the initialization of cyclic structures.
4. *Simplicity:* the type system is simple and requires little annotation overhead.

The authors distinguish just two object states: under initialization and completely initialized. A newly allocated object has a so called "free" type. When an object is deeply initialized, i.e., all its fields are set to deeply initialized objects, it is said to have a "committed" type. The commitment point logically changes the type of an object from free to committed and is defined as the end of a constructor that takes only committed arguments. Possible aliasing between free and committed types is prevented by not having a subtyping relation between them. This differs from the convention for raw types [1].

The Validity rule is very close in spirit to the idea of free and committed types. But it relies on a flow-sensitive analysis and ceases free type status when all attributes are set. This allows the *practical void safety* to handle cyclic data structures without explicit annotations.

Other approaches ***(solve examples i to vi with*** 0 ***annotations, non-modular).*** Additional annotations are avoided by Bertrand Meyer in [6] using so called "targeted expressions" and creation-involved features. The analysis is somewhat similar to the abstract interpretation approach used by Fausto Spoto [8] and should be applied

to the system as a whole, thus sacrificing modularity. The advantage of the non-modular checks is in accepting larger code base as correct, i.e., in better expressiveness.

5. Benchmark criteria

The most important goal of the null safety design is soundness. This limits the possibilities to write arbitrary code that is still null safe. Indeed, the general problem of safe object initialization is undecidable. Therefore, some restrictions should be imposed on the code to make sure it can be checked in finite time.

As usual, soundness and expressiveness work against each other: the simpler the language rules, the less code can be written without violating them. If the rules are too strict, some scenarios found in real software can become extinct. E.g., the *raw types* [1] do not allow for creation of circular structures, and the *free and committed types* [9] rule out registration of objects in existing object structures inside constructors.

I use the design goals suggested by Alexander J. Summers and Peter Müller [9] as the base criteria to evaluate usability of null safety solutions. I give a detailed analysis of every aspect of the goals and review how it applies to distinguish characteristics of different approaches.

5.1. Soundness

Authors of all the solutions [1, 2, 7, 9] from section 4 claim them to be sound, so do I [10] for the *practical void safety*, based on the Validity rule. Unfortunately, not all aspects of a real programming environment are usually reflected in the formal model. In particular, none of the null safety formalizations reflects garbage collection that is an important channel to compromise safety guarantees.

The roots of the object initialization problem, mentioned in section 3, are mapped to the programming language constructs as follows:

- *Non-atomic initialization* corresponds to the order of initialization of the object fields intermingled with other computations. This work does not consider languages that support atomic (transactional) object initialization.

- Explicit *aliasing* becomes possible when an object is assigned to a field of an existing object, either passed to the constructor as an argument or directly reachable from the current context, or when the new object is thrown as an exception. Implicit *aliasing* happens when the class declares a finalizer that gets access to the object.

- *Uncontrollable control flow* can be caused by concurrent execution, preemptive execution (with exception and signal handlers), cooperative execution (coroutines).

- *Dereferencing* is done by a qualified call of the form `target.access` where `access` stands for a field or method name and `target` is a name of a reference corresponding to one of uninitialized field of the object.

Contrary to the formal models, programming languages, supporting some null safety mechanisms, do not always handle object initialization properly. A notable example is Kotlin [4] where, at the time of writing, null safety is unsound.

```
class X create make feature
    item: detachable A
    put (value: A) do item := value end
    make
        do
            (create {A}.initialize (Current)).
                data.do_nothing
        rescue
            if attached item as value then
                value.data.do_nothing
            end
        end
end
```

```
class A inherit
    EXCEPTIONS
create initialize feature
    data: A
    initialize (x: X)
        do
            x.put (Current)
            raise (Void)
            data := Current
        end
end
```

Fig 1. An example (in Eiffel) of a buggy scenario A-I(1a)

A. Escaping of uninitialized objects

If a program context does not expect an uninitialized object, there should be no channels that allow for the object to escape to this context. The channels depend on the programming language. The most common ones are discussed next.

A-I. Registration in an existing object. A program can register a new object in an existing one. When this is done before the new object is completely initialized, there is a problem: the incompletely initialized object can be accessed via the exiting object because of aliasing. Thus, such registration should be disallowed. The scenario can be further classified by

1) the *source* of the reference to the existing object that can be either (a) passed as an argument to the constructor or (b) retrieved from the current context (static data, once functions, singleton objects);

2) the *type* of the object in which the new one is registered: it can be (a) a user-defined one or (b) a built-in one (e.g., an array).

A registration of the new object in the existing one requires a qualified call. The condition 3.3 of the Validity rule disallows any qualified calls until all objects become completely initialized. Therefore, the *practical void safety* solution guarantees that the unsafe scenario with the explicit registration is impossible.

The example corresponding to this scenario is shown in fig. 1. It can be used in the suit of benchmarks for null safety solutions. The method *make* of the class X creates an object of the type A and passes a reference to itself (let's call it x) as an argument. The constructor *initialize* of the class A saves a reference to the current object in the object x. At this point the field *data* is null. Then a raised exception breaks the execution of the constructor in *initialize*. The rescue clause in *make* intercepts the exception, retrieves the incompletely initialized object and makes a call on the field *data* of a non-null type. This causes the null reference exception.

A-II. Reclamation of incompletely initialized objects. Finalizers are the methods called before object's memory is reused. The finalizers are registered for calling by the run-time right after object's memory is allocated and before the constructor is invoked. If the object initialization does not complete (due to an uncaught exception in the constructor), the finalizer is invoked on the incompletely initialized object. Unless a programmer keeps track of object initialization, there is no way to figure out what state the object is in. Therefore, the current object in a finalizer should be treated like at the beginning of a constructor.

A-III. Out-of-order object transfer. Most programming languages allow for transferring references to objects bypassing regular control flow. The most familiar mechanism is exceptions. If a new exception object referencing an incompletely initialized one is thrown, the reference to the incompletely initialized object becomes accessible in the code that relies on the type system rules and does not expect uninitialized fields.

There is no special construct to raise an exception in Eiffel. A qualified call to a library method is used to do it. According to the condition 3 of the Validity rule, all objects should be completely initialized at this point, therefore, soundness is preserved.

B. Non-sequential control flow

Approaches solving the object initialization problem with whole-system analysis make a safe approximation about all possible execution traces. Consequently, these approaches are non-modular.

The modular approaches are limited by the local analysis only and should assume worst-case scenarios for operations on incompletely initialized objects. This is achieved by pretending that sequential execution can break anywhere. In other words, the solutions do not exclude unexpected interruption. Therefore, here I just list possible scenarios that can be used to build the benchmark tests:

B-I. *Exceptions.*

B-II. *Concurrency.*

B-III. *Cooperative execution.*

C. Dereferencing

C-I. Unqualified calls. Access to uninitialized fields should either be prohibited or result in a potentially null value. The condition 2 of the Validity rule sticks to the first variant.

C-II. Qualified calls. Before all objects are completely initialized, language rules should either disallow qualified access to fields or make sure the retrieved values are not considered as safe for use. In particular, these values may be null or (recursively) have null values in non-null fields of referenced objects. The *practical void safety* solution takes the first route and disables qualified calls until all objects are completely initialized.

5.2. Expressiveness

Any example demonstrating a soundness issue from section 5.1 can be turned into a legitimate one. To this end, one of the conditions that cause the problem should be

```
data := Current
x.put (Current)
raise (Void)
    (a)
```

```
raise (Void)
data := Current
    (b)
```

Fig 2. Possible fixes of the method *initialize* from fig. 1

made false. E.g., setting the field *data* before passing a reference to the current object as in fig. 2a or avoiding leaking this reference altogether as in fig. 2b, both make the example from fig. 1 legitimate.

D. Calls on incompletely initialized objects

Calls on incompletely initialized objects require special precaution, but they allow for more code reuse.

D-I. Unqualified calls.

1) *Method call.* One of the key uses of unqualified calls is initialization itself. The methods that set object fields can be called from different constructors to avoid code duplication.

2) *Field access.* For the same reason, access to the fields that might be unset can be useful outside of object initialization. The code would check if the field value is null and proceed as needed.

D-II. Qualified calls. In general, qualified calls on incompletely initialized objects are unsafe. Therefore, additional restrictions should be imposed on the code of the called methods as well as on the code of the callers. They both cannot rely on the regular type system rules.

1) *Method call.* The called methods cannot assume that fields of non-null types are not null.

2) *Field access.* Safe access to object fields can be guaranteed only if they have a basic type without nested references.

E. Circular references.

Object structures with circular references appear to be a common design pattern. They can be classified by the structure kind:

1) *Self-referencing.* A new object references itself after creation.

2) *Mutual references.* (a) Two objects or (b) n objects make a circular structure after creation, where n is not necessary fixed at compile time.

3) *Complex structures.* A combination of cases 1) and 2).

F. Regular use of a completely initialized object from the constructor

F-I. Callback. When an object is completely initialized, it can be passed from the constructor as an argument to create another object that makes a callback later. This pattern is used in some portable GUI libraries.

F-II. Escaping of initialized objects. The scenarios repeat the scenarios A:

1) *Registration in an existing object.*

2) *Out-of-order object transfer.*

But now the object is completely initialized and does not cause a problem.

G. Genericity

Solutions can differ by the ability to use a formal generic type as a creation type. For the *practical void safety* solution, special convention should be introduced to indicate whether the constructor of the actual generic parameter satisfies the Validity rule requirements.

5.3. Modularity

H. Scope. The solution is modular if it is sufficient to analyze (recursively) ancestors and suppliers of the class to be checked.

The Validity rule depends on the properties of the constructors from the other classes. Because these constructors are known at compile time, the checks do not depend on the classes that are not directly reachable from the one being checked. Therefore, a library can be checked as a standalone entity without the need to recheck it after inclusion in some other project.

I. Incrementality. The metric tells if changes to previously checked code require a partial recheck rather than a complete one.

With the *practical void safety*, fast recompilation is supported if information about reachable constructors and whether they perform qualified calls is recorded for every class.

5.4. Simplicity

This group of metrics tells how accessible is the solution.

J. Number of additional annotations. This is the number of different type marks that need to be added besides the marks “non-null” and “maybe-null”.

K. Ease. This metric describes whether few new simple rules are added to the language to make object initialization null safe.

L. Performance. The metrics describes the resource consumption to support the additional checks. This includes:

1) **space complexity** – how much additional memory is required;

2) **time complexity** – how much time the new language checks take.

M. Availability. This metric tells if there is a tool (compiler/framework) that supports the rules.

6. Preliminary results

The table below summarizes results of selected benchmarks for different solutions (a plus sign indicates a positive result of the benchmark, a minus sign indicates a negative one):

Solution	**ABC**	*D-I(1)*	*D-I(2)*	*D-II*	**E**	**F**	**H**	**J**	**K**	**M**
Raw types	+	+	+	+	−	−	+	2+	+	+
Masked types	+	+	−	+	+	+	+	many	−	−
Free/committed types	+	+	+	+	+	−	+	1+	+	+
Targeted expressions	+	+	−	+	+	+	−	0	±	−
Practical void safety	+	+	−	−	+	+	+	0	+	+

All the solutions are sound (if the scenario *A-II* is not taken into account) and support simple use cases. With modularity and annotation overhead in mind, the best two solutions are *free & committed types* and *practical void safety*. Unqualified access to uninitialized attributes (*D-I(2)*) can be easily supported in the latter solution. But inability to support registration of a newly created object from the constructor in an existing object (**F**) with the former solution cannot be fixed easily. Therefore, *practical void safety* seems to be a better choice to solve the object initialization problem.

7. Conclusion and future work

In this work, I identify the roots of the object initialization problem in object-oriented languages and propose a list of benchmarks to compare different null safety solutions. The benchmarks demonstrate very good results for the *practical void safety* compared to other solutions.

The work reveals the following areas of future development:

- formalization of null safety models taking into account finalizers and possible interruption of execution due to asynchronous signals;
- improvement of the *practical void safety* solution by supporting access to uninitialized attributes and by supporting qualified calls in the context with incompletely initialized objects;
- creation of a test suit with executable examples to compare different implementation of null safe programming languages.

References

[1] Fähndrich Manuel, Leino K. Rustan M., “Declaring and Checking Non-null Types in an Object-oriented Language”, *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA ’03, ACM, New York, NY, USA, 2003, 302–312, http://doi.acm.org/10.1145/949305.949332.

[2] Fähndrich Manuel, Xia Songtao, “Establishing Object Invariants with Delayed Types”, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA ’07, ACM, New York, NY, USA, 2007, 337–350, http://doi.acm.org/10.1145/1297027.1297052.

[3] Gamma Erich, Helm Richard, Johnson Ralph, Vlissides John, *Design Patterns: Elements of Reusable Object-oriented Software*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[4] JetBrains, *Kotlin Language Specification*, https://jetbrains.github.io/kotlin-spec/kotlin-spec.pdf, visited on 2017-01-31.

[5] Kogtenkov A. V., "Mechanically Proved Practical Local Null Safety", *Proceedings of the Institute for System Programming of the RAS*, **28**:5 (2016), 27–54, DOI: 10.15514/ISPRAS-2016-28(5)-2.

[6] Bertrand Meyer, *Targeted expressions: safe object creation with void safety*, 2017, http://se.ethz.ch/ meyer/publications/online/targeted.pdf, visited on 2017-05-08.

[7] Qi Xin, Myers Andrew C., "Masked Types for Sound Object Initialization", *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, ACM, New York, NY, USA, 2009, 53–65, http://doi.acm.org/10.1145/1480881.1480890.

[8] Spoto Fausto, "Precise null-pointer analysis", *Software & Systems Modeling*, **10**:2 (2011), 219–252, http://dx.doi.org/10.1007/s10270-009-0132-5.

[9] Summers Alexander J., Müller Peter, "Freedom Before Commitment: A Lightweight Type System for Object Initialisation", *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, ACM, New York, NY, USA, 2011, 1013–1032, http://doi.acm.org/10.1145/2048066.2048142.

[10] Kogtenkov Alexander, "Practical Void Safety", *Verified Software. Theories, Tools, and Experiments*, 2017, http://dx.doi.org/10.1007/978-3-319-72308-2_9.

Когтенков А. В., "К критериям оценки безопасности нулевых ссылок при инициализации объекта", *Моделирование и анализ информационных систем*, **24**:6 (2017), 718–729.

DOI: 10.18255/1818-1015-2017-6-718-729

Аннотация. Разыменование нулевого указателя остаётся одной из основных проблем в современных объектно-ориентированных языках. Очевидное добавление ключевых слов, чтобы различать всегда ненулевые и возможно нулевые ссылки, оказывается недостаточным во время инициализации объекта, когда некоторые поля, объявленные всегда ненулевыми, могут временно быть нулевыми до окончания инициализации. Данная работа устанавливает ключевые причины проблемы инициализации объектов. Она предлагает сценарии и метрики в качестве эталонных тестов для сравнения решений этой проблемы. Наконец, она демонстрирует применение этих тестов к предложенному решению инициализации объектов в Eiffel. Статья публикуется в авторской редакции.

Ключевые слова: разыменование нулевого указателя, безопасность нулевых ссылок, безопасность пустых ссылок, инициализация объектов, статический анализ, эталонные тесты безопасности нулевых ссылок

Об авторе:
Когтенков Александр Валентинович, orcid.org/0000-0003-4873-8306, кандидат наук (Doctor of Sciences ETH Zurich), Независимый ученый, г. Подольск, Россия, e-mail: kwaxer@mail.ru
адрес для корреспонденции: редакция журнала МАИС, ул. Советская, 14, г. Ярославль, 150003 Россия

Моделирование и анализ информационных систем. Т. 24, №6 (2017), с. 730–742
Modeling and Analysis of Information Systems. Vol. 24, No 6 (2017), pp. 730–742

DOI: 10.18255/1818-1015-2017-6-730-742

УДК 519.713

К синтезу синхронизирующих и установочных последовательностей для входо-выходных полуавтоматов

Кушик Н. Г., Евтушенко Н. В., Бурдонов И. Б., Косачев А. С.

получена 3 сентября 2017

Аннотация. В работе рассматриваются задачи проверки существования и синтеза синхронизирующих и установочных последовательностей для конечных входо-выходных полуавтоматов. Соответствующие последовательности могут быть использованы при идентификации состояния проверяемой системы после подачи подходящей входной последовательности. В модели, исследуемой в работе, действия разделены на входные и выходные, однако отсутствуют выделенные явно семейства начальных и финальных состояний. В статье определяются понятия синхронизирующей и установочной последовательностей и предлагаются методы их синтеза для специального класса входо-выходных полуавтоматов, у которых в каждом состоянии определены переходы или только по входным, или только по выходным действиям; кроме того, в соответствующем графе переходов отсутствуют циклы по выходным символам. Для описанного класса входо-выходных полуавтоматов устанавливаются необходимые и достаточные условия существования синхронизирующих и установочных последовательностей и оценивается длина таких последовательностей. Выделяются подклассы полуавтоматов, для которых худшие (в основном экспоненциальные) оценки сложности не являются достижимыми.

Ключевые слова: входо-выходные полуавтоматы, синхронизирующая последовательность, установочная последовательность

Для цитирования: Кушик Н. Г., Евтушенко Н. В., Бурдонов И. Б., Косачев А. С., "К синтезу синхронизирующих и установочных последовательностей для входо-выходных полуавтоматов", *Моделирование и анализ информационных систем*, **24**:6 (2017), 730–742.

Об авторах:
Кушик Наталья Геннадьевна, д-р физ.-мат. наук,
Университет Телеком Южный Париж / Париж Сакле,
ул. Ш. Фурье, 9, г. Еври, 91000 Франция, e-mail: natalia.kushik@telecom-sudparis.eu

Евтушенко Нина Владимировна, orcid.org/0000-0002-4006-1161, д-р техн. наук, профессор,
Институт системного программирования РАН, ул. Солженицына, 25, г. Москва, 109004 Россия
Томский государственный университет, ул. Ленина, 36, г. Томск, 634050 Россия, e-mail: nyevtush@gmail.com

Бурдонов Игорь Борисович, orcid.org/0000-0001-9539-7853, д-р физ.-мат. наук, старший научный сотрудник,
Институт системного программирования РАН, ул. Солженицына, 25, г. Москва, 109004 Россия, e-mail: igor@ispras.ru

Косачев Александр Сергеевич, orcid.org/0000-0001-5316-3813, канд. физ.-мат. наук, старший научный сотрудник,
Институт системного программирования РАН, ул. Солженицына, 25, г. Москва, 109004 Россия, e-mail: kos@ispras.ru

Благодарности:
Работа частично поддержана проектами РНФ № 16-49-03012 и РФФИ № 17-07-00682.

Введение

Проблема идентификации состояний в конечных автоматах (Finite State Machines или FSMs в англоязычной литературе) активно исследуется, начиная с первой работы Мура, посвященной синтезу так называемых «умозрительных» экспериментов для конечных автоматов [12]. Результаты Мура были в дальнейшем развиты исследователями для различных классов конечных автоматов (см., например, [2, 6, 11]). При идентификации текущего/финального состояния на исследуемую систему подается заранее сформированная синхронизирующая или установочная последовательность. Если последовательность синхронизирующая, то из любого состояния исследуемая система переходит в известное состояние. Если последовательность установочная, то по наблюдаемой реакции системы на установочную последовательность делается вывод о состоянии, достигнутом системой после подачи последовательности. В настоящей работе мы не рассматриваем адаптивные входные последовательности, т.е. входная синхронизирующая (или установочная) последовательность сформирована заранее, до проведения эксперимента с системой.

Идентификация состояний используется в различных приложениях. Одним из таких приложений являются автоматные методы синтеза проверяющих тестов для дискретных управляющих систем [4, 7, 13], и существует достаточно много публикаций, каким образом можно построить такие последовательности для детерминированных и недетерминированных (в том числе ненаблюдаемых), частичных и полностью определенных автоматов [1, 5, 7, 13]. Помимо так называемого «активного» тестирования, идентификация текущего состояния проверяемой системы используется также в задачах мониторинга (пассивного тестирования) при наблюдении поведения проверяемой реализации. В частности в работе [8] авторы исследуют возможность ускорения/оптимизации мониторинга при наличии в спецификации системы синхронизирующих и/или установочных последовательностей. Если текущее состояние системы известно, то можно существенно сократить множество проверяемых свойств при “пассивной” верификации системы.

Тем не менее, при описании поведения дискретных управляющих систем с целью проверки их надежности и корректности функционирования достаточно часто используются не только модель конечного автомата, но также и входо-выходные полуавтоматы. Причиной является тот факт, что выходная реакция системы не обязательно появляется непосредственно после каждого входного символа, и, вообще говоря, на один входной символ возможна реакция в виде последовательности выходных символов. Входо-выходные полуавтоматы позволяют описать такие ситуации подходящей математической моделью, однако методы анализа таких моделей с точки зрения идентификации текущего/финального состояния авторам работы не известны. Соответственно, предлагаемые методы построения установочных и синхронизирующих последовательностей для такой модели являются новыми.

Следует отметить, тем не менее, что синхронизирующие эксперименты для полуавтоматов рассматривались и ранее, однако в известных случаях действия, помечающие переходы, не разделены на входные и выходные [1, 13, 15]. Иными словами, при синтезе эксперимента во внимание принимается тот факт, что в каждом состоянии полуавтомата может быть подан любой из определенных в данном состоянии символов, которые можно понимать как входные символы; выходных символов

среди действий нет. В данной работе мы рассматриваем более широкий класс полуавтоматов, а именно полуавтоматы, в которых в каждом состоянии может быть или принят входной, или произведен выходной символ. Таким образом, мы распространяем полученные результаты по синтезу синхронизирующих экспериментов на исследуемый класс полуавтоматов.

Мы отмечаем, что представленные в работе результаты частично вошли в публикацию [16] по результатам Восьмого симпозиума по спецификации и верификации программ "Program Semantics, Specification and Verification: Theory and Applications" (2017 г.). В настоящей статье мы "наследуем" развитые в предыдущей статье алгоритмы. Научная новизна данной работы заключается в 1) доказательстве корректности предложенных алгоритмов, 2) установлении оценок длин синхронизирующей и установочной последовательностей для рассматриваемого класса входо-выходных полуавтоматов и 3) выделении специальных подклассов входо-выходных полуавтоматов, для которых худшие (в основном экспоненциальные) оценки сложности не являются достижимыми.

Структура работы следующая. В первом разделе приводятся основные определения и обозначения. Методы синтеза установочных и синхронизирующих последовательностей для входо-выходных полуавтоматов приведены во втором разделе. Третий раздел содержит заключение, в котором, в частности, освещаются перспективы дальнейших научных исследований.

1. Основные определения и обозначения

Под *входо-выходным полуавтоматом* (или просто *полуавтоматом*) в этой статье понимается четверка $\boldsymbol{S} = (S, I, O, T_S)$, где S есть конечное непустое множество состояний; I и O суть конечные непересекающиеся алфавиты входных и выходных символов; $T_S \subseteq S \times I \times S \cup S \times O \times S$ есть *отношение переходов*, где тройки $(s, i, s') \in T_S$ и $(s, o, s') \in T_S$ суть *переходы* [1].

В этой статье мы рассматриваем специальный класс полуавтоматов, для которых выполняются следующие условия.

1. В каждом состоянии полуавтомата определены только входные символы или только выходные символы, т.е. $S = S_1 \cup S_2$, $S_1 \cap S_2 = \emptyset$ и $T_S \subseteq S_1 \times I \times S \cup S_2 \times O \times S$, причем в полуавтомате отсутствуют состояния, из которых нет переходов.

2. Диаграмма переходов полуавтомата не содержит циклов, помеченных только выходными символами, т.е. язык полуавтомата не содержит последовательностей с бесконечным суффиксом из выходных символов.

3. В полуавтомате есть специальный выходной символ $\delta \notin O$, представляющий так называемое "молчание" (англ. – "quiescence [14]") в состояниях, где определены переходы по входным символам: соответственно в каждом состоянии $s \in S_1$ определен переход-петля, помеченный символом δ, т.е. $(s, \delta, s) \in T_S$.

[1] Таким образом, в данной работе рассматриваются полуавтоматы без переходов по ненаблюдаемым действиям и без выделения множеств начальных и финальных состояний.

Пример входо-выходного полуавтомата $\boldsymbol{S}$ показан на рис. 1. Полуавтомат имеет 5 состояний, т.е. $S = \{s_1, \ldots, s_5\}$, для которых $S_1 = \{s_1, s_2, s_5\}$ и $S_2 = \{s_3, s_4\}$. В каждом состоянии из S_1 полуавтомат "принимает" входные символы i_1 и i_2. В состояниях s_3 и s_4 полуавтомат не принимает входные символы, а выдает выходные реакции o_1 или o_2.

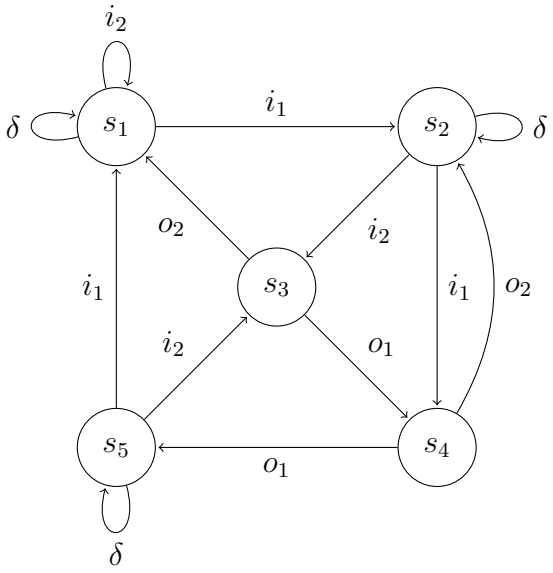

Рис. 1. Входо-выходной полуавтомат $\boldsymbol{S}$
Fig. 1. An Input/Output automaton $\boldsymbol{S}$

Как обычно, синхронизирующие и установочные входные последовательности используются для идентификации состояния полуавтомата после подачи такой последовательности, при условии, что начальное состояние полуавтомата не известно. Основываясь на [16], мы далее вводим определения таких последовательностей для входо-выходных полуавтоматов. Отметим, что подача входной последовательности и идентификация состояния после подачи последовательности осуществляются согласно следующему **предположению**.

Перед подачей первого входного символа тестер ожидает поступления выходного символа в течение определенного выходного таймаута t. Если полуавтомат выдает выходной символ, то таймер обнуляется, и тестер ждет очередной выходной символ в течение последующих t единиц времени. Если в течение t единиц времени выходной символ полуавтоматом не производится, то на тестер подается следующий входной символ или тестирование заканчивается, и таймер обнуляется. Такой порядок подачи входных символов объясняет наличие специального выходного символа $\delta \notin O$; когда выходной символ не наблюдается в течение t единиц времени, мы полагаем, что производится выходной символ δ. Выходной таймаут обычно определяется, исходя из самой длинной возможной выходной последовательности на поданный входной символ.

Как обычно, трассой в полуавтомате называется допустимая последовательность действий, и γ-преемник состояния $s \in S$ есть множество состояний, которые достижимы из состояния по трассе γ, т.е. γ-преемник множества S содержит каждое

состояние из S, достижимое из некоторого состояния $s \in S$ по трассе γ. Синхронизирующей последовательностью называется входная последовательность, после подачи которой полуавтомат переходит в одно и то же состояние из любого начального состояния. Другими словами, последовательность $\alpha = i_1 i_2 \dots i_k$ называется *синхронизирующей* для полуавтомата $\boldsymbol{S}$, если существует состояние $s \in S$, такое что для любой последовательности действий $\beta_1 i_1 \beta_2 i_2 \dots \beta_k i_k \beta_{k+1}$ в полуавтомате, где p есть длина самой длинной последовательности только из выходных символов и $\beta_j \in (O \cup \{\delta\})^p$, $j = 1, \dots, k+1$, справедливо, что $\beta_1 i_1 \beta_2 i_2 \dots \beta_k i_k \beta_{k+1}$-преемник множества S (множество состояний $\beta_1 i_1 \beta_2 i_2 \dots \beta_k i_k \beta_{k+1}$-state-after-$S$) есть пустое множество или $\{s\}$. Таким образом, синхронизирующая последовательность позволяет единственным образом идентифицировать состояние полуавтомата после ее подачи без наблюдения производимых полуавтоматом выходных реакций.

Установочная последовательность позволяет единственным образом идентифицировать состояние полуавтомата после ее подачи и наблюдения производимых полуавтоматом выходных реакций. Таким образом, последовательность $\alpha = i_1 i_2 \dots i_k$ называется *установочной* для полуавтомата $\boldsymbol{S}$, если для каждой трассы $\beta_1 i_1 \beta_2 i_2 \dots \beta_k i_k \beta_{k+1}$, $\beta_j \in (O \cup \{\delta\})^p$, $j = 1, \dots, k+1$, справедливо, что $\beta_1 i_1 \beta_2 i_2 \dots \beta_k i_k \beta_{k+1}$-преемник множества S есть пустое множество или синглетон. Непосредственной проверкой можно убедиться, что для автомата $\boldsymbol{S}$ на рис. 1 $\alpha = i_1 i_1$ является установочной последовательностью.

2. Методы синтеза синхронизирующих и установочных последовательностей для входо-выходных полуавтоматов

В данном разделе мы предлагаем алгоритмы построения синхронизирующих и установочных последовательностей для входо-выходных полуавтоматов из описанного выше класса. Кроме того, мы уточняем оценки сложности построения таких последовательностей и рассматриваем классы полуавтоматов с пониженными оценками длин синхронизирующих и установочных последовательностей.

2.1. Построение синхронизирующих последовательностей

Как отмечалось ранее, синхронизирующие последовательности и методы их синтеза хорошо изучены для “классических” полуавтоматов, в которых алфавит действий не разделен на подмножества входных (стимулов) и выходных (реакций) действий соответственно. Если в полуавтомате $\boldsymbol{S}$ определены переходы только по входным действиям, то последовательность действий α называется *синхронизирующей*, если существует состояние полуавтомата, в которое α переводит полуавтомат из любого начального состояния. Алгоритмы проверки существования и построения синхронизирующих последовательностей (если существуют) для таких, в том числе недетерминированных и, возможно, частичных, полуавтоматов опубликованы в [1,5,13,15]. При построении синхронизирующей последовательности для входо-выходного полуавтомата $\boldsymbol{S}$, который удовлетворяет условиям 1–3, описанным выше, мы предлагаем

переход к "классическому" полуавтомату без выходных действий, который обладает тем же самым множеством синхронизирующих последовательностей.

Алгоритм 1: синтез полуавтомата $\boldsymbol{A}$

Вход : Входо-выходной полуавтомат $\boldsymbol{S} = (S, I, O, T_S)$
Выход: Полуавтомат $\boldsymbol{A}$
Шаг 1 Строим полуавтомат $\boldsymbol{A} = (S_1, I, T_A)$ с пустым множеством переходов, т.е. $T_A = \emptyset$.
Шаг 2. Для каждого перехода $(s, i, s'') \in T_S$, где $s, s'' \in S_1$, добавляем к T_A переход (s, i, s''); для каждого перехода (s, i, s''), где $s \in S_1$ и $s'' \in S_2$, добавляем к T_A переход (s, i, s'''), где $s''' \in S_1$ и s''' является β-преемником состояния s'' в полуавтомате $\boldsymbol{S}$, $\beta \in O^*$.

Утверждение 1. *Множества синхронизирующих последовательностей для полуавтоматов $\boldsymbol{A}$ (построен по алгоритму 1) и $\boldsymbol{S}$ совпадают.*

Доказательство. По определению, последовательность $\alpha = i_1 \dots i_k$ переводит полуавтомат $\boldsymbol{A}$ из состояния $s' \in S_1$ в состояние $s \in S_1$, если и только если существуют $\beta_j \in (O \cup \{\delta\})^p$, $j = 1, \dots, k+1$, такие, что $\beta_1 i_1 \beta_2 i_2 \dots \beta_k i_k \beta_{k+1}$-преемник состояния s' во входо-выходном полуавтомате $\boldsymbol{S}$ есть состояние s. Последовательность α является синхронизирующей в полуавтомате $\boldsymbol{A}$, если и только если существует состояние $s \in S_1$, такое, что α-преемник любого состояния полуавтомата $\boldsymbol{A}$ равен $\{s\}$. Последнее означает, что $\beta_1 i_1 \beta_2 i_2 \dots \beta_k i_k \beta_{k+1}$-преемник любого состояния $s' \in S_1$ входо-выходного полуавтомата $\boldsymbol{S}$, где p есть длина самой длинной последовательности только из выходных символов и $\beta_j \in (O \cup \{\delta\})^p$, $j = 1, \dots, k+1$, есть пустое множество или $\{s\}$. Обратно, если α является синхронизирующей последовательностью во входо-выходном полуавтомате $\boldsymbol{S}$, то существует состояние $s \in S_1$, такое что $\beta_1 i_1 \beta_2 i_2 \dots \beta_k i_k \beta_{k+1}$-преемник любого состояния входо-выходного полуавтомата $\boldsymbol{S}$ есть пустое множество или $\{s\}$, т.е. α есть синхронизирующая последовательность в полуавтомате $\boldsymbol{A}$. □

Следствие 1. *Входо-выходной полуавтомат $\boldsymbol{S}$ обладает синхронизирующей последовательностью, если и только если полуавтомат $\boldsymbol{A}$, построенный по алгоритму 1, обладает синхронизирующей последовательностью.*

Отметим, что определенный выше класс входо-выходных полуавтоматов не требует полной определенности функции поведения в состояниях из множества S_1 по входным символам. Поэтому полуавтомат $\boldsymbol{A}$, синтезируемый по алгоритму 1, может оказаться частично определенным, если в исходном полуавтомате в некотором состоянии из множества S_1 определены переходы не по всем входным символам. С другой стороны, множество классических полуавтоматов образует подкласс исследуемого класса входо-выходных автоматов, и, вместе с тем, для частичных детерминированных полуавтоматов (в англоязычной литературе Partial Deterministic Automaton или PDA) задача проверки существования синхронизирующей последовательности является PSPACE-полной [3]. Этот факт обосновывает PSPACE-трудность задачи проверки существования синхронизирующей последовательности для входо-выходных полуавтоматов, исследуемых в данной статье.

Высокие оценки сложности задач проверки существования последовательностей, идентифицирующих текущее состояние полуавтомата, как известно, тесно связаны с максимальными оценками минимальных длин этих последовательностей. Следует отметить, что для входо-выходных полуавтоматов из исследуемого класса эта оценка является экспоненциальной даже в случае, когда поведение полуавтомата определено по всем входным символам в состояниях из множества S_1.

Утверждение 2. *Для синхронизируемого входо-выходного полуавтомата* $\boldsymbol{S}$*, поведение которого определено по всем входным символам в каждом из состояний из множества* S_1*, длина кратчайшей синхронизирующей последовательности не превосходит* $O(2^{|S_1|})$.

Доказательство. Полуавтомат $\boldsymbol{A}$, синтезируемый по алгоритму 1, имеет $|S_1|$ состояний и $|I|$ действий. Этот полуавтомат является полностью определенным, однако он может быть недетерминированным, поскольку для одного и того же входного символа в некотором состоянии возможно существование различных выходных последовательностей, переводящих исходный входо-выходной полуавтомат $\boldsymbol{S}$ в различные состояния из одного и того же состояния из множества S_1. Длина кратчайшей синхронизирующей последовательности полуавтомата $\boldsymbol{A}$ в худшем случае экспоненциально зависит от числа состояний, в частности, для "классического" полуавтомата с множеством состояний S_1 может иметь длину порядка $2^{|S_1|}$ [5], что доказывает утверждение. □

В качестве примера рассмотрим проверку существования синхронизирующей последовательности для полуавтомата на рис. 1. Соответствующий "классический" полуавтомат $\boldsymbol{A}$ (без выходных действий) показан на рис. 2. Непосредственной проверкой можно убедиться, что для полуавтомата $\boldsymbol{A}$ на рис. 2 не существует синхронизирующей последовательности. Таким образом, и для входо-выходного полуавтомата на рис. 1 не существует синхронизирующей последовательности.

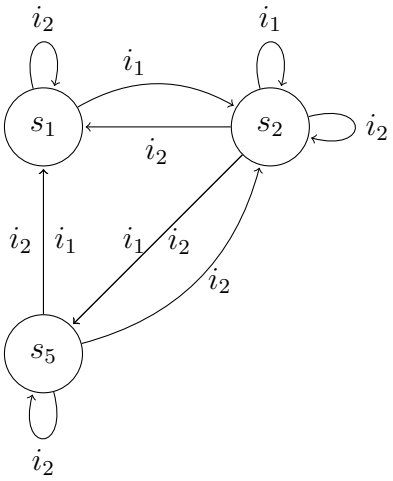

Рис. 2. Полуавтомат $\boldsymbol{A}$, полученный после применения алгоритма 1
Fig. 2. Automaton $\boldsymbol{A}$ returned by Algorithm 1

2.2. Построение установочных последовательностей

В настоящей работе задача проверки существования и синтеза установочных последовательностей для входо-выходных полуавтоматов сводится к задаче синтеза установочных последовательностей для классических, возможно, частичных и недетерминированных автоматов, для которых такая задача хорошо исследована [6–8].

Рассмотрим конечный автомат $\boldsymbol{M} = (M, I, O, h_M)$ [2], в котором M – конечное непустое множество состояний, I и O - конечные непересекающиеся входной и выходной алфавиты. В отличие от рассмотренной выше модели входо-выходного полуавтомата переходами в автомате являются четверки (s, i, o, s'), т.е. в любой трассе автомата за каждым входным символом следует выходной символ, и каждая трасса заканчивается выходным символом. Автомат называется *полностью определенным*, если в каждом состоянии определен переход по каждому входному символу, иначе, автомат называется частично определенным. Автомат называется *детерминированным*, если в каждом состоянии по каждому входному символу определено не более одного перехода, иначе, автомат называется *недетерминированным*. Обычным образом отношение переходов автомата распространяется на входные и выходные последовательности. Автомат называется *сильно связным*, если для любых двух состояний m_1 и m_2 существует входная последовательность, переводящая автомат из состояния m_1 в состояние m_2. Детерминированный автомат называется *приведенным*, если для любых двух состояний существует входная последовательность, выходные реакции на которую в этих состояниях различны.

Входная последовательность α называется *установочной* в автомате, если для любой трассы γ, проекция которой на входной алфавит равна α, γ-преемник каждого подмножества состояний имеет мощность не более единицы, причем одинаковые выходные реакции на последовательность α гарантируют достижимость одного и того же состояния. Методы синтеза установочных последовательностей для конечных автоматов хорошо исследованы, в частности, для полностью определенных детерминированных автоматов – в [6, 13]; для недетерминированных автоматов – в [8]; для частичных автоматов – в [17]. Подобно методу построения синхронизирующих последовательностей, мы предлагаем достаточно простую процедуру построения конечного автомата по заданному входо-выходному полуавтомату, для которых множества установочных последовательностей совпадают.

Алгоритм 2: синтез автомата $\boldsymbol{M}$

Вход : Входо-выходной полуавтомат $\boldsymbol{S} = (S, I, O, T_S)$

Выход: Автомат $\boldsymbol{M}$

Шаг 1 Строим автомат $\boldsymbol{M} = (S_1, I, O \cup O^2 \cup \cdots \cup O^p \cup \{\delta\}, T_M)$ с пустым множеством переходов, т.е. $T_M = \emptyset$, где p есть длина самой длинной последовательности из выходных символов в полуавтомате $\boldsymbol{S}$.

Шаг 2. Для каждого состояния $s \in S_1$, такого что $(s, i, s') \in T_S$, $s' \in S_1$, добавляем к T_M переход (s, i, δ, s').

Шаг 3. Для каждого состояния $s \in S_1$, такого что $(s, i, s') \in T_S$, $s' \in S_2$, добавляем к T_M переход $(s, i, o_1 o_2 \dots o_k, s'')$, $k \le p$, где $s'' \in S_1$ есть $o_1 o_2 \dots o_k$-преемник состояния s'.

Утверждение 3. *Множества установочных последовательностей автомата* $\boldsymbol{M}$ *(построен по алгоритму 2) и полуавтомата* $\boldsymbol{S}$ *совпадают.*

Доказательство. По определению, последовательность $\alpha = i_1 \dots i_k$ является установочной в автомате $\boldsymbol{M}$, если для любой трассы γ, проекция которой на входной алфавит равна α, γ-преемник каждого подмножества состояний имеет мощность не более единицы, причем одинаковые выходные реакции на последовательность α гарантируют достижимость одного и того же состояния. По построению автомата $\boldsymbol{M}$, последнее означает, что $\beta_1 i_1 \beta_2 i_2 \dots \beta_k i_k \beta_{k+1}$-преемник любого состояния входо-выходного полуавтомата $\boldsymbol{S}$, где p есть длина самой длинной последовательности только из выходных символов и $\beta_j \in (O \cup \{\delta\})^p$, $j = 1, \dots, k+1$, есть пустое множество или $\{s\}$ для подходящего $s \in S$, причем одинаковые трассы $\beta_1 i_1 \beta_2 i_2 \dots \beta_k i_k \beta_{k+1}$ гарантируют достижимость одного и того же состояния из различных состояний. Обратно, если α является установочной последовательностью во входо-выходном полуавтомате $\boldsymbol{S}$, то $\beta_1 i_1 \beta_2 i_2 \dots \beta_k i_k \beta_{k+1}$-преемник любого состояния входо-выходного полуавтомата $\boldsymbol{S}$ не более чем одноэлементен, т.е. α есть установочная последовательность в автомате $\boldsymbol{M}$. □

Следствие 2. *Входо-выходной полуавтомат* $\boldsymbol{S}$ *обладает установочной последовательностью, если и только если автомат* $\boldsymbol{M}$*, построенный по алгоритму 2, обладает установочной последовательностью.*

Как отмечалось ранее, для проверки существования и построения установочной последовательности в автомате $\boldsymbol{M}$ можно воспользоваться алгоритмами из [6, 8, 9, 13], которые "обрабатывают" детерминированные и недетерминированные, полностью и частично определенные автоматы. Отметим также, что в случае ненаблюдаемости недетерминизма полученного конечного автомата $\boldsymbol{M}$ в силу высокой сложности задачи идентификации финального состояния автомата удобно воспользоваться различными эвристиками усечения так называемого установочного дерева (получается из дерева преемников, представляющего "развертку" поведения автомата). Например, в работе [8] предлагается использовать поиск установочной последовательности для частичного, возможно, ненаблюдаемого автомата с ограничением на максимальную длину требуемой последовательности. Этот факт обосновывается оценкой длины кратчайшей установочной последовательности для недетерминированных автоматов даже в случае, когда исходный полуавтомат является определенным по всем входным символам в состояниях из множества S_1.

Известные результаты по достижимости оценки длины кратчайшей установочной последовательности для наблюдаемого полностью определенного автомата обосновывают достижимость экспоненциальной оценки длины установочной последовательности для входо-выходного полуавтомата. В этом случае автомат из работы [10] может быть естественным образом преобразован в полуавтомат "разверткой" каждого перехода i/o в последовательность io длины два. Таким образом, имеет место следующее утверждение.

Утверждение 4. *Для любого* $n > 3$ *существует входо-выходной полуавтомат* $\boldsymbol{S}$*, для которого* $|S_1| = n$ *и длина кратчайшей установочной последовательности составляет* $2^{n-1} - 1$.

Возможность описанного выше преобразования конечного автомата во входо-выходной полуавтомат также обосновывает и PSPACE-трудность задачи проверки существования установочной последовательности, поскольку задача проверки существования установочной последовательности является PSPACE-полной даже для частично определенного детерминированного автомата. PSPACE-полноту задачи проверки существования установочной последовательности для входо-выходных полуавтоматов нужно исследовать дополнительно, поскольку в результате перехода (полиномиальной сложности) к классическому автомату можно в худшем случае получить частичный ненаблюдаемый автомат, для которого принадлежность подходящей задачи классу PSPACE остается под вопросом.

В качестве примера проверим наличие установочной последовательности в полуавтомате $\boldsymbol{S}$ на рис. 1. В этом полуавтомате нет синхронизирующей последовательности, однако непосредственной проверкой с использованием методов из [6, 7, 13] можно убедиться, что соответствующий автомат $\boldsymbol{M}$ (рис. 3) обладает установочной последовательностью $\alpha = i_1 i_1$.

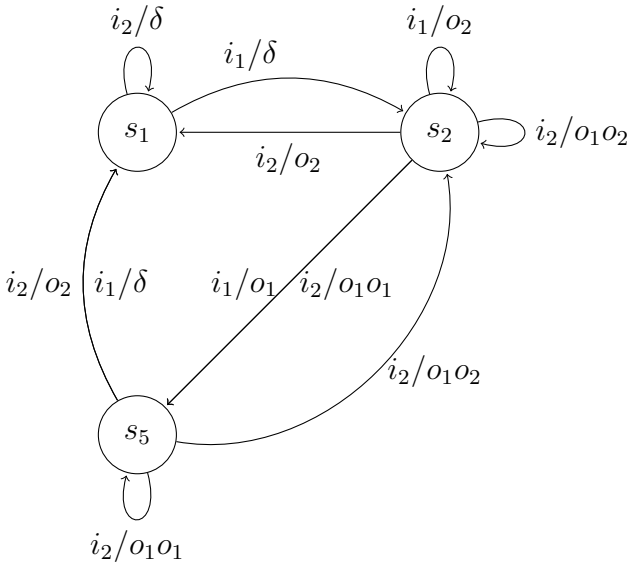

Рис. 3. Автомат $\boldsymbol{M}$, полученный из $\boldsymbol{S}$ после применения алгоритма 2
Fig. 3. Automaton $\boldsymbol{M}$ returned for $\boldsymbol{S}$ by Algorithm 2

Согласно приведенным выше рассуждениям и утверждениям, задачи проверки существования и построения синхронизирующих и установочных последовательностей для рассматриваемого класса входо-выходных полуавтоматов имеют достаточно высокую сложность. Однако в ряде случаев эта сложность может быть понижена за счет наложения дополнительных ограничений на исследуемый полуавтомат. Например, вновь обратимся к полуавтоматам, в которых в каждом допустимом состоянии определен переход по каждому входному символу. Договоримся также, что в каждом состоянии из множества S_2 определен единственный выходной символ, т.е. после каждого входного символа следует единственная последовательность выходных символов.

В этом случае подходящий автомат $\boldsymbol{M}$ и полуавтомат $\boldsymbol{A}$, синтезируемые по алгоритмам 1 и 2, представляют собой “хорошие” детерминированные, всюду опреде-

ленные системы переходов. Задачи проверки существования и синтеза установочных и синхронизирующих последовательностей для полностью определенных сильно связных приведенных детерминированных автоматов (и "классических" полуавтоматов соответственно) могут быть решены за полиномиальное время, поэтому проблема идентификации финального/текущего состояния такого входо-выходного полуавтомата может быть решена за полиномиальное время. Более того, длины соответствующих входных последовательностей ограничиваются полиномом второй и третьей степеней относительно мощности подмножества состояний полуавтомата, в которых определены переходы по входным символам. Примером "хорошего" класса входо-выходных полуавтоматов может служить подмножество детерминированных по выходам полуавтоматов, у которых в состояниях из множества S_1 определен переход по каждому входному символу $i \in I$, а в состояниях из множества S_2 – переход только по одному выходному символу. Если для каждого входо-выходного полуавтомата из данного подкласса подходящий автомат $\boldsymbol{M}$ является приведенным и сильно связным, то всякий исследуемый входо-выходной полуавтомат обязательно обладает установочной последовательностью, причем длина этой последовательности не превосходит $|S_1|(|S_1| - 1)/2$. Исследование расширения множества таких "хороших" подклассов полуавтоматов является частью дальнейшей работы.

3. Заключение

В настоящей работе исследуется задача синтеза синхронизирующих и установочных последовательностей для специального класса входо-выходных полуавтоматов, в которых в каждом состоянии определены только входные или только выходные действия, и в диаграмме переходов отсутствуют циклы, помеченные только выходными действиями. Мы показываем, каким образом задачу синтеза синхронизирующей и установочной последовательностей для полуавтоматов из этого класса можно свести к решению подобных задач для классических автоматов и полуавтоматов, для которых такие алгоритмы существуют. Кроме того, в статье получены оценки сложности проверки существования и построения таких последовательностей. Описан один из классов входо-выходных полуавтоматов, для которых полученные оценки сложности могут быть понижены. В дальнейшем мы планируем продолжить исследование классов входо-выходных полуавтоматов с пониженными оценками сложности для синхронизирующих и установочных последовательностей, в частности, за счет использования адаптивных "умозрительных" экспериментов. Мы также планируем провести исследования по расширению класса рассматриваемых входо-выходных полуавтоматов, добавив ненаблюдаемое действие τ, а также допущение о существовании композитных состояний, в которых допустимы как входные, так и выходные действия.

Список литературы / References

[1] Мартюгин П. В., "Нижние оценки длины кратчайших бережно синхронизирующих слов для двух- и трёхбуквенных частичных автоматов", *Дискретн. анализ и исслед. опер.*, **15**:4 (2008), 44–56; [Martjugin P. V., "Nizhnie ocenki dliny kratchajshih berezhno

sinhronizirujushhih slov dlja dvuh- i trjohbukvennyh chastichnyh avtomatov", *Diskretn. analiz i issled. oper.*, **15**:4 (2008), 44–56, (in Russian).]

[2] Gill A., "State-identification experiments in finite automata", *Information and Control*, **4** (1961), 132–154.

[3] Martyugin P., "Computational Complexity of Certain Problems Related to Carefully Synchronizing Words for Partial Automata and Directing Words for Nondeterministic Automata", *Theory Comput. Syst.*, **54**:2 (2014), 293–304.

[4] Hierons R. M., Jourdan G. V., Ural H., Yenigün H., "Using adaptive distinguishing sequences in checking sequence constructions", *Proc. of the 2008 ACM symposium on Applied computing (SAC)*, ACM, 2008, 682–687.

[5] Ito M., Shikishima-Tsuji K., "Some Results on Directable Automata", *Theory Is Forever: Essays Dedicated to Arto Salomaa on the Occasion of His 70th Birthday*, Lecture Notes in Computer Science, **3113**, Springer, Berlin, Heidelberg, 2004, 125–133.

[6] Kohavi Z., *Switching and Finite Automata Theory*, McGraw-Hill, New York, 1978.

[7] Kushik N., El-Fakih K., Yevtushenko N., Cavalli A.R., "On adaptive experiments for nondeterministic finite state machines", *International Journal on Software Tools for Technology Transfer*, **18**:3 (2016), 251–264.

[8] Kushik N., López J., Cavalli A.R., Yevtushenko N., "Improving Protocol Passive Testing through 'Gedanken' Experiments with Finite State Machines", *Proc. 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, IEEE, 2016, 315–322.

[9] Kushik N., El-Fakih K., Yevtushenko N., "Preset and adaptive homing experiments for nondeterministic finite state machines", *Implementation and Application of Automata. CIAA 2011*, Lecture Notes in Computer Science, **6807**, 2011, 215–224.

[10] Kushik N., Yevtushenko N., "On the Length of Homing Sequences for Nondeterministic Finite State Machines", *Implementation and Application of Automata. CIAA 2013*, Lecture Notes in Computer Science, **7982**, 2013, 220–231.

[11] Lee D., Yannakakis M., "Testing finite-state machines: state identification and verification", *IEEE Trans. on Computers.*, **43**:3 (1994), 306–320.

[12] Moore E.F., "Gedanken-experiments on sequential machines", *Automata Studies (Annals of Mathematical Studies)*, Princeton University Press, 1956, 129–153.

[13] Sandberg S., "Homing and Synchronizing Sequences", *Model-Based Testing of Reactive Systems*, Lecture Notes in Computer Science, **3472**, 2005, 5–33.

[14] Tretmans J., "Test Generation with Inputs, Outputs and Repetitive Quiescence", *Software – Concepts and Tools*, **17**:3 (1996), 103–120.

[15] Volkov M.V., "Synchronizing Automata and the Černý Conjecture", *Language and Automata Theory and Applications. LATA 2008*, Lecture Notes in Computer Science, **5196**, 11–27.

[16] Kushik N.G, Yevtushenko N.V, Burdonov I.B., Kossatchev A.S, "Synchronizing and Homing Experiments for Input/output Automata", *System Informatics*, 2017, № 10, 1–9.

[17] Yenigün H., Yevtushenko N., Kushik N., "The complexity of checking the existence and derivation of adaptive synchronizing experiments for deterministic FSMs", *Information Processing Letters*, **127** (2017), 49–53.

Kushik N. G., Yevtushenko N. V., Burdonov I. B., Kossatchev A. S., "Deriving Synchronizing and Homing Sequences for Input/Output Automata", *Modeling and Analysis of Information Systems*, **24**:6 (2017), 730–742.

DOI: 10.18255/1818-1015-2017-6-730-742

Abstract. In this paper, we study the problem of existence check and derivation of synchronizing and homing sequences for finite input/output automata. Corresponding sequences can be effectively

used for the current state identification of a system under test / verification, after the input sequence is applied. In the model considered in the paper, the alphabet of actions is divided into disjoint sets of inputs and outputs; however, no sets of possible initial or final states are defined. We introduce the notions of homing and synchronizing sequences for a specific class of such machines for which at each state the transitions only under inputs or under outputs are defined, and the machine transition diagram does not contain cycles labeled by outputs, i.e. the language of the machine does not contain traces with infinite postfix of outputs. For such a class of input/output automata, we establish necessary and sufficient conditions for the existence of synchronizing and homing sequences and discuss the length of such sequences. We also define some subclasses of automata for which the worst-case upper bounds (normally, exponential) are not reachable.

Keywords: input/output automata, synchronizing sequence, homing sequence

On the authors:
Natalia G. Kushik, PhD, Assistant Professor
Télécom SudParis/Université Paris-Saclay,
9 Charles Fourier str., Évry 91000, France, e-mail: natalia.kushik@telecom-sudparis.eu

Nina V. Yevtushenko, orcid.org/0000-0002-4006-1161, PhD, Professor,
Institute for System Programming RAS, 25 Solzhenitsyn str., Moscow 109004, Russia,
Tomsk State Univeristy, 36 Lenin str., Tomsk 634050, Russia, nyevtush@gmail.com

Igor B. Burdonov, orcid.org/0000-0001-9539-7853, PhD, Senior Researcher,
Institute for System Programming RAS,
25 Solzhenitsyn str., Moscow 109004, Russia, e-mail: igor@ispras.ru

Alexander S. Kossatchev, orcid.org/0000-0001-5316-3813, PhD, Senior Researcher,
Institute for System Programming RAS,
25 Solzhenitsyn str., Moscow 109004, Russia, e-mail: kos@ispras.ru

Acknowledgments:
The work was partially supported by the Russian Science Foundation (RSF), project № 16-49-03012 and № 17-07-00682 of Russian Foundation for Basic Research.

Моделирование и анализ информационных систем. Т. 24, № 6 (2017), с. 743–754
Modeling and Analysis of Information Systems. Vol. 24, No 6 (2017), pp. 743–754

DOI: 10.18255/1818-1015-2017-6-743-754

UDC 004.052.42

Invariant Elimination of Definite Iterations over Arrays in C Programs Verification

Maryasov I. V.[1], **Nepomniaschy V. A.**, **Kondratyev D. A.**[1]

Received September 8, 2017

Abstract. This work represents the further development of the method for definite iteration verification [7]. It extends the mixed axiomatic semantics method [1] suggested for C-light program verification. This extension includes a verification method for definite iteration over unchangeable arrays with a loop exit in C-light programs. The method includes an inference rule for the iteration without invariants, which uses a special function that expresses loop body. This rule was implemented in verification conditions generator, which is the part of our C-light verification system. To prove generated verification conditions an induction is applied which is a challenge for SMT-solvers. At proof stage the SMT-solver CVC4 is used in our verification system. To overcome mentioned difficulty a rewriting strategy for verification conditions is suggested. A method based on theory extension by new theorems to prove verification conditions is suggested. An example, which illustrates the application of these methods, is considered. The article is published in the authors' wording.

Keywords: C-light, loop invariants, mixed axiomatic semantics, definite iteration, arrays, CVC4, specification, verification, Hoare logic

For citation: Maryasov I. V., Nepomniaschy V. A., Kondratyev D. A., "Invariant Elimination of Definite Iterations over Arrays in C Programs Verification", *Modeling and Analysis of Information Systems*, **24**:6 (2017), 743–754.

On the author:
Ilya V. Maryasov, orcid.org/0000-0002-2497-6484, PhD,
A. P. Ershov Institute of Informatics Systems SB RAS,
6 Akademik Lavrentyev av., Novosibirsk 630090, Russia, e-mail: ivm@iis.nsk.su

Valery A. Nepomniaschy, orcid.org/0000-0003-1364-5281, PhD,
A. P. Ershov Institute of Informatics Systems SB RAS,
6 Akademik Lavrentyev av., Novosibirsk 630090, Russia, e-mail: vnep@iis.nsk.su

Dmitry A. Kondratyev, orcid.org/0000-0002-9387-6735, postgraduate student,
A. P. Ershov Institute of Informatics Systems SB RAS,
6 Akademik Lavrentyev av., Novosibirsk 630090, Russia, e-mail: apple-66@mail.ru

Acknowledgments:
[1]This research is partially supported by RFBR grant 15-01-05974.

Introduction

C program verification is an important task nowadays. A lot of projects (e. g. [3, 4]) propose different solutions. But none of the mentioned above suggests any methods for loop verification without invariants whose construction is a challenge. Therefore, the user has to provide these invariants. In many cases it is a difficult task. Tuerk [13] suggested

to use pre- and post-conditions for while-loops but the user still has to construct them himself.

Our method describes a class of loops, which can be verified without invariants or pre- and post-conditions for loops. It deals with a definite iteration of a special form. We extend our mixed axiomatic semantics of the C-light language [1] with a new rule for verification of such iterations. This extension includes a verification method for definite iteration over unchangeable arrays with a loop exit in C-light programs. The method includes an inference rule for the iteration without invariants, which uses a special function that expresses loop body. This rule was implemented in verification conditions generator, which is the part of our C-light verification system.

At the proof stage, the SMT-solver CVC4 [2] is used. A rewriting strategy for the induction based verification conditions is suggested to prove them in CVC4.

Also an algorithm based on theory extension by special implications to prove verification conditions is suggested. It allows a source formula to be proved successfully. The induction processing approach [12] implemented in CVC4 is too constrained by orientation to inductive data types. The suggested algorithm allows to overcome this difficulty.

1. Definite Iteration and Replacement Operation

The method of loop invariants elimination for definite iteration was suggested in [11]. It includes four cases:

1. Definite iteration over unchangeable data structures without loop exits.
2. Definite iteration over unchangeable data structures with a loop exit.
3. Definite iteration over changeable data structures with a loop exit.
4. Definite iteration over hierarchical data structures with a loop exit.

The first case was considered in [7]. Our paper deals with the second case.
Consider the statement

$$\textbf{for x in S do v} := \textbf{body}(\mathbf{v}, \mathbf{x}) \textbf{ end}$$

where S is a structure, x is the variable of the type "an element S", v is a vector of loop variables which does not contain x and *body* represents the loop body computation, which does not modify x and S, and which terminates for each $x \in memb(S)$, where $memb(S)$ is the multiset of elements of the structure S. The loop body can contain only the assignment statements, the **if** statements and the **break** statements. Such **for** statement is named a definite iteration.

To express the effect of the iteration let us define a replacement operation $rep(v, S, body, n)$, where $rep(v, S, body, 0) = v$, $rep(v, S, body, i) = body(rep(v, S, body, i-1), s_i)$ for all $i = 1, 2, \ldots, n$ if $\neg empty(S)$.

A number of theorems, which express important properties of the replacement operation, were proved in [11].

The inference rule [10] for definite iterations has the form [8]:

$$\frac{E, SP \vdash \{\exists v'\ P(v \leftarrow v')\ \wedge v = rep(v', S, body)\}\ \mathbf{A};\{Q\}}{E, SP \vdash \{P\}\ \mathbf{for\ x\ in\ S\ do\ v} := \mathbf{body(v, x)\ end\ A};\{Q\}}$$

Here A are program statements after the loop. We use forward tracing: we move from the program beginning to its end and eliminate the leftmost operator (at the top level) applying the corresponding rule of the mixed axiomatic semantics [1] of C-light. E is the environment which contains an information about current function (its identifier, type and body) which is verified, an information about current block and label identifier if **goto** statement occurred earlier. SP is program specification which includes all preconditions, postconditions, and invariants of loops and labeled statements.

2. Definite Iteration over Arrays with a Loop Exit

Let S be a one-dimensional array of n elements. Consider the special case of definite iteration

$$\mathbf{for}\ (\mathbf{i} = \mathbf{0}; \mathbf{i} < \mathbf{n}; \mathbf{i}++)\ \mathbf{v} := \mathbf{body(v, i)}\ \mathbf{end}$$

where $\mathbf{v} := \mathbf{body(v, i)}$ consists of assignment statements, **if** statements (possibly nested) and **break** statements.

In order to generate verification conditions we have to determine v, $body(v, i)$, and the function rep [8].

Let the loop body has the form

$$\begin{gathered}\{\mathbf{x_1} = \mathbf{expr_1}(\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_k});\\ \mathbf{x_2} = \mathbf{expr_2}(\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_k});\\ \ldots\\ \mathbf{x_k} = \mathbf{expr_k}(\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_k});\}\end{gathered}$$

where $expr_j (j = 1, 2, \ldots k)$ are some C-light expressions.

The vector v of loop variable consists of all variables from left parts of assignment statements: $v = (x_1, x_2, \ldots, x_k)$. From the statements before the loop, we can get the initial value of v and obtain the first axiom for rep:

$$rep(v, S, body, 0) = (x_{1_0}, x_{2_0}, \ldots, x_{k_0})$$

where $x_{j_0}, j = 1, 2, \ldots, k$ are the initial values of x_j before the loop execution.

At the next step, we make consecutive substitutions

$$\begin{gathered}x_1 = expr_1(x_1, x_2, \ldots, x_k);\\ x_2 = expr_2(expr_1(x_1, x_2, \ldots, x_k), x_2, \ldots, x_k);\\ \ldots\\ x_k = expr_k(expr_1(x_1, x_2, \ldots, x_k), expr_2(expr_1(x_1, x_2, \ldots, x_k), x_2, \ldots, x_k), \ldots, x_k);\end{gathered}$$

And then in the right parts $rep((x_1, x_2, \ldots, x_k), S, body, i-1)$ is substituted for x_j.

For each **if** statement of the form $\mathbf{if}\ (\mathbf{e(i, x_1, x_2, \ldots, x_k)})\{\mathbf{A};\}\ \mathbf{else}\ \{\mathbf{B};\}$, where **A** and **B** are compound statements consisting of assignment statements, two axioms are added to the output of verification conditions generator:

$$\forall x_1 \forall x_2 \ldots \forall x_k \; e(i, x_1, x_2, \ldots, x_k) \Rightarrow A^*$$
$$\forall x_1 \forall x_2 \ldots \forall x_k \; \neg e(i, x_1, x_2, \ldots, x_k) \Rightarrow B^*$$

where A^* and B^* are obtained by consecutive substitutions as described above.

The **break** statement could appear at the top level of the loop or in the **if** statement. The first case is obvious, it means that the loop iterates no more than once and all the statements after **break** in the loop body are ignored. Therefore, the function *rep* is defined for $i = 0, 1$.

The second case means that for some j such that $0 < j \leq n$ a loop exit occurs and such j is defined by the condition of the **if** statement. Therefore, for all i such that $j \leq i \leq n$

$$rep((x_1, x_2, \ldots, x_k), S, body, i) = rep((x_1, x_2, \ldots, x_k), S, body, j)$$

In this case the following axiom is added:

$$\forall x_1 \forall x_2 \ldots \forall x_k \; e(i, x_1, x_2, \ldots, x_k) \Rightarrow (A^* \wedge (\forall l \, i < l \Rightarrow A^*))$$

For the case when the **break** statement is located in the **else** statement, the negation of e is used.

For each nested **if** statements of the form $\mathbf{if}\,(\mathbf{e_1}(\mathbf{i}, \mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_k}))\{\mathbf{A_1}; \mathbf{if}\,(\mathbf{e_2}(\mathbf{i}, \mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_k}))\mathbf{A_2}\,\mathbf{else}\,\mathbf{A_3}; \mathbf{A_4}\}\,\mathbf{else}\,\{\mathbf{B};\}$, where $\mathbf{A_1}, \mathbf{A_2}, \ldots \mathbf{A_4}$ are compound statements consisting of assignment statements, the following axioms are added to the output of verification conditions generator:

$$\forall x_1 \forall x_2 \ldots \forall x_k \; ((e_1(i, x_1, x_2, \ldots, x_k) \Rightarrow A_1^*) \wedge e_2(i, x_1, x_2, \ldots, x_k)) \Rightarrow (A_2; A_4)^*$$
$$\forall x_1 \forall x_2 \ldots \forall x_k \; ((e_1(i, x_1, x_2, \ldots, x_k) \Rightarrow A_1^*) \wedge \neg e_2(i, x_1, x_2, \ldots, x_k)) \Rightarrow (A_3; A_4)^*$$
$$\forall x_1 \forall x_2 \ldots \forall x_k \; \neg e_1(i, x_1, x_2, \ldots, x_k) \Rightarrow B^*$$

where A_1^*, $(A_2; A_4)^*$, $(A_3; A_4)^*$ and B^* are obtained by consecutive substitutions as described above. For multiple nested **if** statements we make axioms in the similar way.

$\forall i (1 \leqslant i \leqslant k)$ rep_i is automatically generated to simplify the proof of rep-based verification conditions in CVC4. Note that $\forall i (1 \leqslant i \leqslant k)$ rep_i corresponds to variable x_i. This generation is based on substitution of the i-th *rep* by rep_i.

3. Extending Theory to Prove Verification Conditions

To prove by induction some proposition $\forall n \; P(n)$ we use Leino approach [6]. It is to add an extra axiom (induction step) of the form $\forall j \; P(j) \Rightarrow P(j+1)$ and to modify the verification condition by adding a base case of induction $P(1)$. In our case of definite iteration over unchangeable one-dimensional arrays the inductive variable is the length of array. Therefore, the verification conditions generator is able to rewrite the verification condition which contains a *rep* function automatically.

Let us consider the following method of proving formula ϕ, which has the form:

$$\forall x_1 x_2 ... x_{m-1} x_m \quad a(x_1 x_2 ... x_{m-1} x_m) \implies b(x_1 x_2 ... x_{m-1} x_m), \tag{1}$$

where

$$a = H_1(x_{K_{1_1}} x_{K_{1_2}} ... x_{K_{1_{l_1-1}}} x_{K_{1_{l_1}}}) \wedge H_2(x_{K_{2_1}} x_{K_{2_2}} ... x_{K_{2_{l_2-1}}} x_{K_{2_{l_2}}}) \wedge ...$$
$$\wedge H_{n-1}(x_{K_{n-1_1}} x_{K_{n-1_2}} ... x_{K_{n-1_{l_{n-1}-1}}} x_{K_{n-1_{l_{n-1}}}}) \wedge H_n(x_{K_{n_1}} x_{K_{n_2}} ... x_{K_{n_{l_n-1}}} x_{K_{n_{l_n}}}), \quad (2)$$

$K_1 \cup K_2 \cup ... \cup K_{n-1} \cup K_n = \{1, ..., m\}$ and $|K_1| = l_1, |K_2| = l_2, ..., |K_{n-1}| = l_{n-1}, |K_n| = l_n$.

The formula ϕ and set of axioms and theorems are the input arguments of the algorithm.

Let us consider the set of axioms and theorems:

$$\forall y_{11} y_{12} ... y_{1p_1-1} y_{1p_1} \ f_1, \quad \forall y_{21} y_{22} ... y_{2p_2-1} y_{2p_2} \ f_2 ...$$
$$\forall y_{q-11} y_{q-12} ... y_{q-1p_{q-1}-1} y_{q-1p_{q-1}} \ f_{q-1}, \quad \forall y_{q1} y_{q2} ... y_{qp_q-1} y_{qp_q} \ f_q \quad (3)$$

The message "The formula ϕ is true" or "unknown" is the output value of the algorithm.

1. Let i:=1.

2. Let us consider $\forall y_{i1} y_{i2} ... y_{ip_i-1} y_{1p_i} \ f_i$.

   If the structure of f_i is of the form of $c(y_{i1} y_{i2} ... y_{ip_i-1} y_{1p_i}) \implies d(y_{i1} y_{i2} ... y_{ip_i-1} y_{1p_i})$ then go to the step 3 else go to the step 9.

3. If the structure of c is in the form of

$$G_1(y_{R_{1_1}} y_{R_{1_2}} ... y_{R_{1_{s_1-1}}} y_{R_{1_{s_1}}}) \wedge G_2(y_{R_{2_1}} y_{R_{2_2}} ... y_{R_{2_{s_2-1}}} y_{R_{2_{s_2}}}) \wedge ...$$
$$\wedge G_{t-1}(y_{R_{t-1_1}} y_{R_{t-1_2}} ... y_{R_{t-1_{s_{t-1}-1}}} y_{R_{t-1_{l_{t-1}}}}) \wedge G_t(y_{R_{t_1}} y_{R_{t_2}} ... y_{R_{t_{s_t-1}}} y_{R_{t_{l_t}}}), \quad (4)$$

   where $t \leqslant n$, $R_1 \cup R_2 \cup ... \cup R_{t-1} \cup R_t = \{i1, i2, ..., ip_i\}$ and $|R_1| = s_1, |R_2| = s_2, ..., |R_{t-1}| = s_{t-1}, |R_t| = s_t$ then go to the step 4 else go to the step 9.

4. Let us consider a subformula of ϕ $a' = H'_1(x') \wedge H'_2(x') \wedge ... \wedge H'_{n-1}(x') \wedge H'_n(x')$ where each conjunct $H'_i(x')$ results from conjunct $H_i(x_{K_{i_1}} x_{K_{i_2}} ... x_{K_{i_{l_i-1}}} x_{K_{i_{l_i}}})$ by substitution of all occurrences of $x_{K_{i_1}} x_{K_{i_2}} ... x_{K_{i_{l_i-1}}} x_{K_{i_{l_i}}}$ by unique identifier x'.

   Let us consider subformula $c' = G'_1(x') \wedge G'_2(x') \wedge ... \wedge G'_{t-1}(x') \wedge G'_t(x')$ where each conjunct $G'_i(x')$ results from conjunct $G_i(y_{R_{i_1}} y_{R_{i_2}} ... y_{R_{i_{s_i-1}}} y_{R_{i_{s_i}}})$ by substitution of all occurrences of $y_{R_{i_1}} y_{R_{i_2}} ... y_{R_{i_{s_i-1}}} y_{R_{i_{s_i}}}$ by unique identifier x'.

   Let us consider bijection set $\{e_1, e_2, ..., e_{v-1}, e_v\}$ where $\forall i$ e_i is bijection from the set $\{1, .., t\}$ of conjunct indexes of subformula c to the subset U of conjunct indexes of a. Note that $e_i(w) = u \iff G'_w(x')$ is syntactically equal to $H'_u(x')$.

5. Let $j := 1$.

6. Generate a table of correspondence w between variables y of formula c and variables x of the following formula

$$H_{e_j(1)} \wedge H_{e_j(2)} \wedge ... \wedge H_{e_j(t-1)} \wedge H_{e_j(t)}. \quad (5)$$

using matching between G_v and $H_{e_j(v)}$ subformulas. If there is not such correct table w then go to the step 8 else go to the step 7.

7. Let us consider the following formula

$$d'(w(y_{i1})w(y_{i2})...w(y_{ip_i-1})w(y_{1p_i})) = d[y_{i1} \leftarrow w(y_{i1}), y_{i2} \leftarrow w(y_{i2}), ..., y_{ip_i-1} \leftarrow w(y_{ip_i-1}), y_{1p_i} \leftarrow w(y_{1p_i})] \quad (6)$$

It has been generated using simultaneous substitution of $y_{i1}, y_{i2}, ..., y_{ip_i-1}, y_{1p_i}$ by corresponding variables $w(y_{i1}), w(y_{i2}), ...w(y_{ip_i-1}), w(y_{1p_i})$.

Let us consider the following formula

$$\forall x_1 x_2 ... x_{m-1} x_m \quad a \wedge d' \implies b \quad (7)$$

It may be proved using Leino approach based on induction. The SMT solver CVC4 may be used in such case. If such proving results in "unsat" then go to the step 11 else go to the step 8.

8. Let $j := j + 1$. If $j \leqslant v$ then go to the step 6 else go to the step 9.

9. Let $i := i + 1$. If $i \leqslant q$ then go to the step 2 else go to the step 10.

10. The algorithm results in "unknown".

11. The algorithm results in "The formula ϕ is true".

The suggested algorithm allows the theory for proving verification conditions to be extended by new theorems. They may be used to simplify proof.

4. Example: Array Searching Program

Let us demonstrate the application of our method. Consider the following function **search_count**. For a given integers *key* and *entr* it returns 1 if not less than *entr* elements of the given array of integers *arr* are equal to *key*, where *length* is *arr* length. Otherwise the function returns 0.

The annotated (in SMT-LIB v2 syntax of CVC4) C-light [5] program has the form:

```
/* (assert (and (> length 0) (> entr 0))) */
int search_count(int* arr, int length, int key, int entr){
    auto int result = 0, cnt = 0, i;
    for (i = 0; i < length; i++){
        if ((key == arr[i]) && (entr > (cnt + 1))){
            cnt++;
        }
        else if ((key == arr[i]) && (entr == (cnt + 1))){
            cnt++;
            result = 1;
```

```
            break;
        }
    }
    return result;
}
/* (assert (and
             (=> (<= entr (COUNT length arr key entr)) (= result 1))
             (=> (> entr (COUNT length arr key entr)) (= result 0))) */
```

The logical function **COUNT** returns the number of occurrences of *key* in *arr* from 0 to $length - 1$. Note that in CVC4 all functions must be total, so we also need to consider the case for $i \leqslant 0$. The other axioms describe the common case and define **COUNT** recursively. Let us consider some of axioms of the function **COUNT**:

```
(declare-fun COUNT (Int (Array Int Int) Int Int) Int)

(assert (forall ((i Int) (arr (Array Int Int)) (key Int) (entr Int))
   (=>
      (and (< 0 i) (= (select arr (- i 1)) key))
      (= (COUNT i arr key entr) (+ (COUNT (- i 1) arr key entr) 1)))))

(assert (forall ((i Int) (arr (Array Int Int)) (key Int) (entr Int))
   (=>
       (and (< 0 i) (not (= (select arr (- i 1)) key)))
       (= (COUNT i arr key entr) (COUNT (- i 1) arr key entr)))))
```

In *rep* function $v = (cnt, result)$ and its initial value before the iteration is $(0, 0)$. The approach from section 2 allows axioms of *rep* to be generated. Let us consider some of these axioms:

```
(declare-fun rep1 (Int (Array Int Int) Int Int) Int)

(declare-fun rep2 (Int (Array Int Int) Int Int) Int)

(assert (forall ((i Int) (arr (Array Int Int)) (key Int) (entr Int))
    (=>
        (and
             (< 0 i)
             (= key (select arr (- i 1)))
             (> entr (+ (rep1 (- i 1) arr key entr) 1)))
        (= (rep1 i arr key entr) (+ (rep1 (- i 1) arr key entr) 1)))))

(assert (forall ((i Int) (arr (Array Int Int)) (key Int) (entr Int))
    (=>
         (and
              (< 0 i)
              (not (= key (select arr (- i 1))))
```

```
            (> entr (+ (rep1 (- i 1) arr key entr) 1)))
        (= (rep1 i arr key entr) (rep1 (- i 1) arr key entr)))))

(assert (forall ((i Int) (arr (Array Int Int)) (key Int) (entr Int))
    (=>
        (and
            (< 0 i)
            (> entr (+ (rep1 (- i 1) arr key entr) 1)))
        (= (rep2 i arr key entr) (rep2 (- i 1) arr key entr)))))

(assert (forall ((i Int) (arr (Array Int Int)) (key Int) (entr Int))
    (=>
        (and
            (< 0 i)
            (= key (select arr (- i 1)))
            (= entr (+ (rep1 (- i 1) arr key entr) 1)))
        (= (rep2 i arr key entr) 1))))
```

CVC4 is the SMT-solver but our task is to check a validity of verification conditions, not satisfiability. Therefore, the verification conditions generator produces the negation of the verification condition:

```
(assert (not (forall ((length Int) (arr (Array Int Int))
                      (key Int) (entr Int) (result Int))
    (=>
        (and (> length 0) (> entr 0) (= result (rep2 j arr key entr)))
        (and
            (=>
                (<= entr (COUNT length arr key entr))
                (= result 1))
            (=>
                (> entr (COUNT length arr key entr))
                (= result 0)))))))
```

And then we expect the answer “unsat” which means that the negation is unsatisfiable therefore the verification condition is true.

However, SMT solvers do not support proofs by induction, which appears inevitably in our verification method. In this example we get the answer “unknown” which means that CVC4 is not able to determine whether the formula is satisfiable or not. Rustan Leino suggested a rewriting strategy and a heuristic for when to apply it to verify simple inductive theorems [6].

The simplification of verification condition allows it to be considered as a conjunction of the first conjunct

```
(assert (forall ((length Int) (arr (Array Int Int))
                 (key Int) (entr Int) (result Int))
                 (=>
```

```
                    (and
                       (> length 0)
                       (> entr 0)
                       (= result (rep2 length arr key entr))
                       (> entr (COUNT length arr key entr)))
   (= result 0))))
```

and the second conjunct

```
(assert (forall ((length Int) (arr (Array Int Int))
                 (key Int) (entr Int) (result Int))
                (=>
                   (and
                       (> length 0)
                       (> entr 0)
                       (= result (rep2 length arr key entr))
                       (<= entr (COUNT length arr key entr)))
                   (= result 1))))
```

They are referred to as the first and the second part of verification condition.

Algorithm from section 3 allows the first conjunct to be proved. The theory has been extended by the following theorem:

```
(assert (forall ((j Int) (arr (Array Int Int))
                 (key Int) (entr Int))
                 (=>
                    (and
                       (> j 0)
                       (> entr 0)
                       (> entr (COUNT j arr key entr)))
                    (> entr (rep1 j arr key entr)))))
```

It has been proved using CVC4. This proof is based on Leino approach.

Note that renaming variables can result in equality of premises of this formula and some of premises of the first part of the verification condition. Thus, it can result in extension of premises of the first part of the verification condition by conclusion of the considered formula.

Leino approach allows extended verification condition to be proved using CVC4. The base case of induction is trivial. Let us consider the extension of theory by negation of the induction step:

```
(assert (not (forall ((length Int))
     (=>
           (forall ((arr (Array Int Int)) (key Int)
                    (entr Int) (result Int))
                 (=>
                     (and
```

```
                    (> length 0)
                    (> entr 0)
                    (= result (rep2 length arr key entr))
                    (> entr (COUNT length arr key entr))
                    (> entr (rep1 length arr key entr)))
                (= result 0)))
        (forall ((arr (Array Int Int)) (key Int)
                 (entr Int) (result Int))
                (=>
                    (and
                        (> (+ length 1) 0)
                        (> entr 0)
                        (= result (rep2 (+ length 1) arr key entr))
                        (> entr (COUNT (+ length 1) arr key entr))
                        (> entr (rep1 (+ length 1) arr key entr))
                    (= result 0)))))))
```

This theorem has been proved using CVC4.

The proof of the second part of verification condition is similar to the proof of the first one. The theory has been extended by the following theorem:

```
(assert (forall ((j Int) (arr (Array Int Int))
                 (key Int) (entr Int))
                 (=>
                    (and
                        (> j 0)
                        (> entr 0)
                        (<= entr (COUNT j arr key entr)))
                  (<= entr (rep1 j arr key entr)))))
```

Leino approach allows it to be proved using CVC4. Also this approach has been applied at the step 7 of the algorithm from section 3. Thus, this algorithm allows the second part of verification condition to be proved.

Therefore the verification condition has been proved.

5. Conclusion

This paper represents an extension of our system [9] for C-light program verification. In the case of definite iteration over unchangeable arrays with a loop exit, this extension allows us to generate verification conditions without loop invariants. This generation is based on the described inference rule for the C-light **for** statement which introduces the replacement operation. It expresses definite iteration in the special form described in the paper.

The suggested rewriting strategy for induction-based formulas allowed us to prove obtained verification conditions using CVC4 [2]. The proposed algorithm of extending the theory allows verification condition to be proved. Proving formula $a \implies b$ is the

goal of the algorithm execution. Note that this algorithm is based on using implication $c \implies d$ where c is a subformula of a after variables renaming. The proof of formula $a \implies b$ is reduced to proof of formula $(a \wedge d) \implies b$. Using unique identifier allows a and c to be matched. Also the table of correspondence between variables of a and c is created by the algorithm. Note that this table is used for renaming variables of formula d.

The rewriting strategy allowed CVC4 to prove the partial correctness of the example from [7]. It iterates over an array of integers and for a given integer computes the number of its occurrences in this array. Another successfully proved example is the program which for a given integer finds its first occurrence in the given array of integers [8].

We plan to consider the case of elimination of loop invariant for changeable data structures and to verify classical array sorting programs without invariants.

References

[1] I. S. Anureev, I. V. Maryasov, V. A. Nepomniaschy, "C-programs Verification Based on Mixed Axiomatic Semantics", *Automatic Control and Computer Sciences*, **45**:7 (2011), 485–500.

[2] C. Barrett, C. L., Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, C. Tinelli, "CVC4", 23rd Int. Conf. CAV, *Lecture Notes in Computer Science*, **6806** (2011), 171–177.

[3] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, S. Tobies, "VCC: A Practical System for Verifying Concurrent C", 22nd Int. Conf. TPHOLs, *Lecture Notes in Computer Science*, **5674** (2009), 23–42.

[4] J.-C. Filliâtre, C. Marché, "Multi-prover Verification of C Programs", 6th ICFEM, *Lecture Notes in Computer Science*, **3308** (2004), 15–29.

[5] D. A. Kondratyev, "The Extension of the MetaVCG Approach by Semantic Mark-up Concept", *Int. Workshop-conf. "Tools & Methods of Program Analysis"*, St. Petersburg, 2015, 107–118.

[6] K. R. M. Leino, "Automating Induction with an SMT Solver", 13th Int. Conf. VMCAI, *Lecture Notes in Computer Science*, **7148** (2012), 315–331.

[7] I. V. Maryasov, V. A. Nepomniaschy, "Loop Invariants Elimination for Definite Iterations over Unchangeable Data Structures in C Programs", *Modeling and Analysis of Information Systems*, **22**:6 (2015), 773–782.

[8] I. V. Maryasov, V. A. Nepomniaschy, D. A. Kondratyev, "Verification of Definite Iteration over Arrays with a Loop Exit in C Programs", *System Informatics*, 2017, № 10, 57–66.

[9] I. V. Maryasov, V. A. Nepomniaschy, A. V. Promsky, D. A. Kondratyev, "Automatic C Program Verification Based on Mixed Axiomatic Semantics", *Automatic Control and Computer Sciences*, **48**:7 (2014), 407–414.

[10] M. Moriconi, R. L. Schwarts, "Automatic Construction of Verification Condition Generators From Hoare Logics", Automata, Languages and Programming, *Lecture Notes in Computer Science*, **115** (1981), 363–377.

[11] V. A. Nepomniaschy, "Verification of Definite Iteration over Hierarchical Data Structures", FASE/ ETAPS, *Lecture Notes in Computer Science*, **1577** (1999), 176–187.

[12] A. Reynolds, V. Kuncak, "Induction for SMT solvers", 16th Int. Conf. VMCAI, *Lecture Notes in Computer Science*, **8931** (2015), 80–98.

[13] T. Tuerk, "Local Reasoning about While-Loops", *VSTTE*, 2010, 29–39.

Марьясов И. В., Непомнящий В. А., Кондратьев Д. А., "Элиминация инвариантов финитных итераций над массивами при верификации Си программ", *Моделирование и анализ информационных систем*, **24**:6 (2017), 743–754.

DOI: 10.18255/1818-1015-2017-6-743-754

Аннотация. Данная работа представляет дальнейшее развитие метода верификации финитной итерации [7]. Он расширяет метод смешанной аксиоматической семантики [1], предложенный для верификации C-light программ. Это расширение включает метод верификации для финитной итерации над неизменяемыми массивами с выходом из цикла в C-light программах. Метод содержит правило вывода для итерации без инвариантов, которое использует специальную функцию, выражающую действие тела цикла. Данное правило было реализовано в генераторе условий корректности, являющемся частью нашей системы верификации C-light программ. Для доказательства порождённых условий корректности применяется метод математической индукции, вызывающий сложности у SMT-решателей. В нашей системе верификации на стадии доказательства используется SMT-решатель CVC4. Для преодоления упомянутой трудности применяется стратегия переписывания условий корректности. Для доказательства условий корректности предложен метод, основанный на расширении теории новыми теоремами. Рассмотрен пример, иллюстрирующий применение данных методов. Статья публикуется в авторской редакции.

Ключевые слова: C-light, Си, инварианты циклов, смешанная аксиоматическая семантика, финитная итерация, массивы, CVC4, спецификация, верификация, логика Хоара

Об авторе:
Марьясов Илья Владимирович, orcid.org/0000-0002-2497-6484, канд. физ.-мат. наук,
Институт систем информатики им. А. П. Ершова СО РАН,
пр. Академика Лаврентьева, 6, г. Новосибирск, 630090 Россия, e-mail: ivm@iis.nsk.su

Непомнящий Валерий Александрович, orcid.org/0000-0003-1364-5281, канд. физ.-мат. наук,
Институт систем информатики им. А. П. Ершова СО РАН,
пр. Академика Лаврентьева, 6, г. Новосибирск, 630090 Россия, e-mail: vnep@iis.nsk.su

Кондратьев Дмитрий Александрович, orcid.org/0000-0002-9387-6735, аспирант,
Институт систем информатики им. А. П. Ершова СО РАН,
пр. Академика Лаврентьева, 6, г. Новосибирск, 630090 Россия, e-mail: apple-66@mail.ru

Благодарности:
[1]Работа частично поддержана грантом РФФИ 15-01-05974.

Моделирование и анализ информационных систем. Т. 24, № 6 (2017), с. 755–759
Modeling and Analysis of Information Systems. Vol. 24, No 6 (2017), pp. 755–759

DOI: 10.18255/1818-1015-2017-6-755-759

УДК 517.9

Семантические средства обеспечения безопасности в программно-конфигурируемых сетях

Антошина Е. Ю.[1], **Чалый Д. Ю.**[2]

получена 15 марта 2017

Аннотация. Программно-конфигурируемые сети являются многообещающей технологией построения коммуникационных сетей, в которой управление сетью в отличие от традиционного подхода, основанного на конфигурировании отдельных устройств, представляет собой программу, автоматически задающую конфигурацию сети. Это управляющее программное обеспечение позволяет динамически настраивать маршрутизацию информационных потоков внутри сети в зависимости от требований к качеству обслуживания. Однако при этом возникает задача обеспечения безопасности передачи данных [2,3], например, чтобы потоки от конфиденциальных узлов не могли достигать открытого сегмента сети. В статье показано, как эта задача может быть решена с использованием семантических средств.

Ключевые слова: безопасность, семантика, программно-конфигурируемые сети

Для цитирования: Антошина Е. Ю., Чалый Д. Ю., "Семантические средства обеспечения безопасности в программно-конфигурируемых сетях", *Моделирование и анализ информационных систем*, **24**:6 (2017), 755–759.

Об авторах:
Антошина Екатерина Юрьевна, orcid.org/0000-0003-1081-1758, аспирант, ассистент,
Ярославский государственный университет им. П.Г. Демидова,
ул. Советская, 14, г. Ярославль, 150003 Россия, e-mail: kantoshina@gmail.com

Чалый Дмитрий Юрьевич, orcid.org/0000-0003-0553-7387, канд. физ.-мат. наук, декан факультета информатики и вычислительной техники
Ярославский государственный университет им. П.Г. Демидова,
ул. Советская, 14, г. Ярославль, 150003 Россия, e-mail: chaly@uniyar.ac.ru

Благодарности:
[1] Работа частично поддержана грантом РФФИ № 17-07-00823-а «Разработка и анализ моделей и алгоритмов адаптивной организации передачи данных в коммуникационных сетях динамической структуры».
[2] Работа выполнена в рамках инициативного проекта АААА-А16-116070610022-6.

Введение

Программно-конфигурируемые сети (ПКС) — перспективное направление в развитии компьютерных сетей. Управление сетью, отделенное от аппаратной реализации, предоставляет новые возможности для реализации сетевых приложений, используя современные практики разработки программного обеспечения. Программные решения имеют сложную природу, в связи с чем возникает необходимость в дополнительных инструментах верификации программ как со стороны корректности, так и со стороны безопасности. В данной работе представлена модель безопасности ПКС, построенных на базе протокола OpenFlow [5].

1. Модель коммуникационной сети

В работе рассматривается модель коммуникационной сети, в которой сеть состоит из конечных узлов, генерирующих трафик, а также промежуточных узлов, осуществляющих его маршрутизацию. Мы предполагаем, что конечными узлами являются процессы, использующие виртуальные порты транспортного уровня, а промежуточные узлы – это OpenFlow-коммутаторы [5]. В такой модели передача данных может осуществляться внутри отдельного вычислительного узла между выполняющимися на нем процессами, что ставит дополнительные задачи в обеспечении безопасности всей системы. В коммуникационной сети существует выделенный управляющий узел – контроллер, который по защищённым каналам связи взаимодействует со всеми коммутаторами. Контроллер обладает динамически обновляемой информацией о структуре и топологии сети. Основываясь на этой информации, он генерирует и передает команды для каждого коммутатора, решая задачи, поставленные при создании коммуникационной сети.

Коммутатор использует для работы одну или несколько таблиц потоков и групповую таблицу. Таблицы потоков заполняются только на основании полученных от контроллера команд. При помощи этих таблиц происходит обработка и перенаправление пакетов. Таблицы групп в данной работе не рассматриваются, так как могут быть промоделированы при помощи таблиц потоков.

Контроллер отправляет набор команд для коммутатора в ответ на события, которые произошли в сети (например, приход пакета на коммутатор) либо являются событиями окружающей среды. Каждая запись этого списка состоит из набора логических условий ($matchfields$), инструкций ($action$) и счётчиков. Этот набор заносится в таблицу потоков коммутатора.

Набор логических условий $match$ накладывает условия на отдельные заголовки пакета, включая Ethernet, IP и TCP заголовки. Весь предикат match можно представить как $match = m_{src} \wedge m_{dst}$, где m_{src} и m_{dst} – предикаты, соответствующие условиям для адресов источника и назначения.

Если обрабатываемый пакет удовлетворяет набору логических условий $match$, то для него выполняются инструкции из соответствующего списка команд. В данной модели рассматриваются команды: $Output$, $Drop$, Set и $Delete$. Если пакет не соотносится ни с одной из записей таблицы потоков, то в рамках данной модели он передаётся на контроллер.

Предыдущий подход заключался в разработке модели безопасного воздействия в ПКС [6], акцентируя внимание на политике конфиденциальности для ПКС сети. Сделать обоснованный вывод о соблюдении этой политики стало возможно благодаря сформированной формальной системе правил, составляющих семантическую модель [1].

2. Построение семантической модели безопасности

В этой работе мы предлагаем общий принцип, согласно которому можно определить семантики, отражающие различные свойства безопасности. На предмет соблюдения политики безопасности рассматривается список команд, который контроллер в ответ на событие отправляет на коммутатор. Этот упорядоченный набор записей в

рамках данного подхода можно представить как пары $match \times action$. Само событие определяет первую команду в списке, называемую контекстом обработки пакета.

Для решения этой задачи необходимо построить множество правил задания типов S, согласно которому устанавливается, какой уровень безопасности должен быть назначен списку команд.

Для определения уровней безопасности для потоков данных задаются уровни безопасности конечных узлов сети, которые представляют пользовательские процессы. Каждому конечному узлу p ставится в соответствие уровень безопасности (высокий, $p : high$, или низкий, $p : low$). Мы предполагаем, что уровень безопасности всех узлов сети известен контроллеру и может быть определен, например, при помощи протокола аутентификации.

Поскольку отдельные записи таблиц потоков в коммутаторах ПКС-сети могут перенаправлять трафик между отдельными подсетями, которые состоят из многих узлов, для определения того, на какие потоки влияют эти записи, мы введем предикат $forall$, позволяющий определить уровень безопасности множества потоков.

Начнем формирование множества правил задания типов S: внесём в него правила типизации предиката $forall$, который помогает определять свойства, связанные с уровнями безопасности множеств узлов:

$$\frac{\{p_1 : low, \ldots, p_n : low\}}{\vdash forall(p_1, \ldots, p_n) : low}, \tag{1}$$

$$\frac{\{p_1 : high, \ldots, p_n : high\}}{\vdash forall(p_1, \ldots, p_n) : high}. \tag{2}$$

Таким образом подсеть в целом типизируется в соответствии с типом безопасности узлов, которые в нее входят. Если в подсеть входят узлы разных типов, то типизировать предикат $forall$ нельзя. Это позволяет жестко разделить потоки данных [4], относящиеся к разным уровням безопасности, хотя и ограничивает практическую применимость нашего метода. Впоследствии мы покажем, как можно ослабить эти требования с целью большей практичности.

Продолжим формирование множества S правилами, которые справедливы для желаемой политики безопасности. В это множество добавляются правила определения типа безопасности для каждого возможного события. Тип безопасности события определяет контекст, в котором происходит установка команд. Чтобы задать политику безопасности сети, для каждой команды необходимо определить, над какими информационными потоками возможно её применение в каждом контексте безопасности событий.

Возьмём абстрактную команду и определим для неё политику безопасности: для каждого контекста и информационного потока укажем, нарушает или соблюдает данная команда требуемое свойство безопасности.

Теперь можно определить семантические правила для этой команды. Если в контексте безопасности $[C]$ разрешен поток типа $t_{src} \to t_{dst}$, то правило принимает вид:

$$\frac{\vdash forall(src(match)) : t_{src} \quad \vdash forall(dst(match)) : t_{dst}}{[C] \vdash match \times action}. \tag{3}$$

Имеет смысл попытаться представить эти правила в более ёмком виде. Для этого предлагается построить дерево решений, которое можно редуцировать, и использовать для построения более компактного множества S. Полученные правила вывода для каждой команды добавляются в множество S.

Также в это множество необходимо добавить правило, позволяющее типизировать композиции команд:

$$\frac{[pc] \vdash A \quad [pc] \vdash B}{[pc] \vdash A; B}.$$

Совокупность правил из полученного множества S представляет собой формальную систему безопасности для ПКС, с помощью которой можно сделать вывод о соблюдении или нарушении заданной политики безопасности: если весь список команд может быть типизирован при помощи представленной системы, то данный список соблюдает установленную политику безопасности.

Заключение

Данная работа предлагает подход, основанный на разработанной формальной модели безопасности ПКС. Сформированная согласно представленному алгоритму система гарантирует, что приложение на контроллере не нарушает требуемую политику безопасности. После создания модели предлагается расширение: введение правила деклассификации, которое увеличивает её практическую ценность. Система безопасности может быть реализована как программный модуль контроллера, который будет проверять, не нарушают ли приложения, запускаемые на контроллере, свойства безопасности.

В качестве дальнейших направлений исследований интересно рассмотреть проблему безопасности ПКС в более комплексном виде. Поскольку контроллер может реагировать на события внешнего мира и является сложным программным средством, то перспективным подходом является разработка семантической системы безопасности, которая учитывает также поток управления контроллера.

Список литературы / References

[1] Sabelfeld A, Myers A.C., “Language-Based Information-Flow Security”, *IEEE Journal on Selected Areas in Communications*, 2003, № 21, 5–19.

[2] Smeliansky R.L., “SDN for Network Security”, *Proc. of Int. Conf. "Modern Networking Technologies (MoNeTec)*, 2014, 86–95.

[3] Casado M., Foster N., Guha A., “Abstractions for Software-Defined Networks”, *Communications of the ACM*, **57**:10 (2014), 86–95.

[4] Foster N., Freedman M.J., Monasanto Ch., Rexford J., Story A., Walker D., “Splendid Isolation: A Slice Abstraction for Software-Defined Networks”, *CM Int. Conf. on Functional Programming*, 2011, 279—291.

[5] McKeown N., Anderson T., Balakrishnan H., Parulkar G., Kozen D., Peterson L., Rexford J., Shenker S., Turner J., “OpenFlow: Enabling Innovation in Campus Networks”, *SIGCOMM Computer Communications Review*, **38**:2 (2008), 69–74.

[6] Antoshina E.Ju., Nikitin E.S., Chalyy D.Ju., Sokolov V.A., "End-to-end Information Flow Security Model for Software-Defined Networks", *Modeling and Analysis of Information Systems*, **22**:6 (2015), 735–749.

Antoshina E. Ju., Chalyy D. Ju., "Semantic Security Methods for Software-Defined Networks", *Modeling and Analysis of Information Systems*, **24**:6 (2017), 755–759.

DOI: 10.18255/1818-1015-2017-6-755-759

Abstract. Software-defined networking is a promising technology for constructing communication networks where the network management is the software that configures network devices. This contrasts with the traditional point of view where the network behaviour is updated by manual configuration uploading to devices under control. The software controller allows dynamic routing configuration inside the net depending on the quality of service. However, there must be a proof that ensures that every network flow is secure, for example, we can define security policy as follows: confidential nodes can not send data to the public segment of the network. The paper shows how this problem can be solved by using a semantic security model. We propose a method that allows us to construct semantics that captures necessary security properties the network must follow. This involves the specification that states allowed and forbidden network flows. The specification is then modeled as a decision tree that may be reduced. We use the decision tree for semantic construction that captures security requirements. The semantic can be implemented as a module of the controller software so the correctness of the control plane of the network can be ensured on-the-fly.

Keywords: security, semantics, software-defined networks

On the authors:
Ekaterina Ju. Antoshina, orcid.org/0000-0003-1081-1758, gradeuate student,
P.G. Demidov Yaroslavl State University,
14 Sovetskaya str., Yaroslavl 150003, Russia, e-mail: kantoshina@gmail.com

Dmitry Ju. Chalyy, orcid.org/0000-0003-0553-7387, PhD,
P.G. Demidov Yaroslavl State University,
14 Sovetskaya str., Yaroslavl 150003, Russia, e-mail: chaly@uniyar.ac.ru

Acknowledgments:
[1] The work was partially supported by the RFBR grant № 17-07099823-a.
[2] This work was partially supported by the initiative project AAAA-A16-116070610022-6.

Моделирование и анализ информационных систем. Т. 24, №6 (2017), с. 760–771
Modeling and Analysis of Information Systems. Vol. 24, No 6 (2017), pp. 760–771

DOI: 10.18255/1818-1015-2017-6-760-771

УДК 519.248.6

Об определении уровня полезных сигналов при расшифровке магнитных и вихретоковых дефектограмм

Кузьмин Е. В.[1], **Горбунов О. Е., Плотников П. О., Тюкин В. А.**

получена 16 октября 2017

Аннотация. Для обеспечения безопасности движения на железнодорожном транспорте регулярно проводится неразрушающий контроль рельсов с применением различных подходов и методов, включая методы магнитной и вихретоковой дефектоскопии. Статья посвящена задаче автоматического определения порогового уровня амплитуд полезных сигналов (от дефектов и конструктивных элементов рельсового пути) при расшифровке дефектограмм магнитных и вихретоковых дефектоскопов. Сигнал считается полезным (и подлежит дальнейшему анализу), если отклонение его значения от среднего значения всех сигналов как минимум в два раза превосходит пороговый уровень шума рельсов. Вероятность появления сигнала с некоторой амплитудой в бездефектных рельсах на участке без конструктивных элементов, т. е. являющегося рельсовым шумом, характеризуется законом нормального распределения. Таким образом, для вычисления порогового уровня шума может быть задействовано правило трех сигм. А удвоение порога шума дает уровень, превышение которого по амплитудному отклонению от выборочного среднего означает, что сигнал является полезным. В статье предлагается алгоритм нахождения порогового уровня шума рельсов и дается его теоретическое обоснование, а также рассматриваются примеры его работы на нескольких фрагментах реальных магнитных и вихретоковых дефектограмм.

Ключевые слова: неразрушающий контроль рельсов, магнитная и вихретоковая дефектоскопия, обнаружение дефектов, автоматический анализ магнитных и вихретоковых дефектограмм

Для цитирования: Кузьмин Е. В., Горбунов О. Е., Плотников П. О., Тюкин В. А., "Об определении уровня полезных сигналов при расшифровке магнитных и вихретоковых дефектограмм", *Моделирование и анализ информационных систем*, **24**:6 (2017), 760–771.

Об авторах:
Кузьмин Егор Владимирович, orcid.org/0000-0003-0500-306X, д-р физ.-мат. наук, профессор кафедры теоретической информатики, Ярославский государственный университет им. П.Г. Демидова,
ул. Советская, 14, г. Ярославль, 150003 Россия, e-mail: kuzmin@uniyar.ac.ru, kuzminev@nddlab.com

Горбунов Олег Евгеньевич, orcid.org/0000-0001-6274-9971, канд. физ.-мат. наук, генеральный директор,
ООО «Центр инновационного программирования», NDDLab, ул. Союзная, 144, г. Ярославль, 150008 Россия,
e-mail: gorbunovoe@nddlab.com

Плотников Петр Олегович, orcid.org/0000-0001-5687-7969, инженер-технолог, ООО «Центр инновационного программирования», NDDLab, ул. Союзная, 144, г. Ярославль, 150008 Россия, e-mail: plotnikovpo@nddlab.com

Тюкин Вадим Александрович, orcid.org/0000-0001-9149-7435, руков. сектора разработки, ООО «Центр инновационного программирования», NDDLab, ул. Союзная, 144, г. Ярославль, 150008 Россия, e-mail: tyukinva@nddlab.com

Благодарности:
[1] Работа выполнена при финансовой поддержке программы развития ЯрГУ на период 2017–2021 гг. как опорного вуза Ярославской области. Мероприятие «Развитие инновационно активных подразделений университета в приоритетных сферах экономики региона». Направление «Модернизация научно-исследовательской и инновационной деятельности, включая развитие инновационной экосистемы университета».

Введение

Для обеспечения безопасности движения на железнодорожном транспорте регулярно проводится неразрушающий контроль рельсов с применением различных подходов и методов, включая методы магнитной и вихретоковой дефектоскопии. При неразрушающем контроле рельсов с использованием магнитных и вихретоковых дефектоскопов проверяется каждый миллиметр участка пути. Это приводит к необходимости автоматического анализа большого массива данных (дефектограмм), которые поступают от соответствующего оборудования. Под анализом понимается процесс определения по дефектограммам наличия дефектных участков наряду с выявлением конструктивных элементов рельсового пути. При этом в условиях значительных объемов поступающей на обработку информации наибольший интерес представляют быстрые алгоритмы анализа данных.

Эта статья посвящена задаче автоматического определения порогового уровня амплитуд полезных сигналов при расшифровке дефектограмм магнитных и вихретоковых дефектоскопов. Результаты работы предполагается использовать при построении аппаратно-программных комплексов рельсовой дефектоскопии [1–4].

В статье рассматриваются обобщения реальных устройств, которые активно применяются на практике, в виде абстрактных 8-разрядного магнитного и 10-разрядного вихретокового дефектоскопов. Сигналы, регистрируемые этими дефектоскопами, можно разделить на два типа — 1) рельсовый шум и 2) сигналы от конструктивных элементов и дефектов.

Сигналы, являющиеся шумом от рельсов, обычно составляют более 90% всех сигналов дефектограммы. Полезными являются сигналы второго типа, т. е. сигналы от дефектов и конструктивных элементов рельсового пути.

При автоматическом анализе удобно разбивать дефектограммы на фрагменты, которые соответствуют 50-метровым участкам пути. Это означает, что при снятии показаний дефектоскопа с каждого миллиметра пути блок анализа представляет собой массив из 50000 элементов, где элемент массива — это значение амплитуды сигнала. В случае 8-разрядного дефектоскопа имеем не более 256 значений амплитуд, а от 10-разрядного вихретокового дефектоскопа может поступать не более 1024 различных значений сигналов.

Будем полагать, что значения амплитуд сигналов регистрируются дефектоскопами в виде натуральных чисел от 1 до 1024 в 10-разрядном случае и от 1 до 256 в случае 8-разрядного дефектоскопа.

Небольшое отклонение значения амплитуды от выборочного среднего значения соответствует слабому сигналу, а большое — сильному. Сильные сигналы означают наличие дефектов или конструктивных элементов и по амплитудному отклонению значительно превосходят рельсовый шум.

Практический опыт работы с данными магнитных и вихретоковых дефектоскопов показал, что вероятность появления сигнала с некоторой амплитудой в бездефектных рельсах на участке без конструктивных элементов подчиняется (в приближении) закону нормального распределения.

Сигнал считается полезным (и подлежит дальнейшему анализу), если отклонение его значения от среднего значения всех сигналов как минимум в два раза превосходит пороговый уровень шума рельсов.

Под пороговым уровнем шума будем понимать отклонение Level от среднего значения μ сигналов рассматриваемого фрагмента дефектограммы (данные с 50-метрового участка), при котором сигналы со значениями амплитуд из диапазона $[\mu - \text{Level}; \mu + \text{Level}]$ являются шумом рельсов.

Таким образом, поскольку рельсовый шум имеет нормальный закон распределения, для вычисления его порогового уровня может быть задействовано правило трех сигм. А удвоение порога шума дает уровень, превышение которого по амплитудному отклонению от выборочного среднего означает, что сигнал является полезным.

Отметим, что если пороговый уровень шума меньше 10 единиц, запись считается плохой (например, по причине неправильной калибровки оборудования) и не подлежит анализу. Вопрос автоматической оценки качества записи дефектограмм в данной статье не рассматривается. Далее будем полагать, что все магнитные и вихретоковые дефектограммы, которые передаются на анализ, записаны качественно.

В следующих разделах предлагается алгоритм нахождения порогового уровня шума рельсов и дается его теоретическое обоснование, а также рассматриваются примеры его работы на нескольких фрагментах реальных магнитных и вихретоковых дефектограмм.

1. Алгоритм

Несмотря на то, что шум рельсов характеризуется нормальным законом распределения вероятностей, из-за наличия сильных сигналов (от дефектов и конструктивных элементов) не представляется возможным применение в чистом виде правила трех сигм, согласно которому около 99,73% сигналов шума лежит в интервале $[\mu - 3\sigma; \mu + 3\sigma]$, где μ — это выборочное среднее, а σ — среднее квадратическое отклонение. Другими словами, сильные сигналы оказывают значительное влияние на результат вычисления среднего квадратического отклонения по всей выборке.

Для построения приемлемого уровня шума $\text{Level} \approx 3\sigma$ необходимо при вычислении приближенного значения σ исключить из рассмотрения (сильные) сигналы анализируемой выборки, которые не укладываются в рамки закона нормального распределения. Таким образом, для нахождения порога Level предлагается использовать следующий итерационный алгоритм, реализующий эту идею.

Основные этапы алгоритма нахождения порогового уровня шума рельсов Level:

1. Получить массив значений амплитуд сигналов $X[1 \,..\, 50000]$ (данные от дефектоскопа с 50-метрового участка рельсового пути).

2. Вычислить среднее арифметическое значение μ элементов массива X.

3. По элементам массива X, значения которых лежат в диапазоне $[\mu - i; \mu + i]$, где $i \in \mathbb{N}$ изначально равно 1, построить среднее квадратическое отклонение σ от значения μ. Увеличить i на 1.

4. Повторять пункт 3 до тех пор, пока не будет выполнено условие $3 \cdot \sigma > 1{,}3 \cdot i$ при числе элементов со значением из $[\mu - i; \mu + i]$ более 30% от общего числа.

5. Присвоить переменной Level значение $3 \cdot \sigma$, а OldLevel — значение 0.

6. Повторять следующую процедуру до тех пор, пока Level − OldLevel > 0.

   (a) По элементам массива X, значения которых лежат в измененном диапазоне $[\mu - \text{Level}; \mu + \text{Level}]$, построить новое среднее квадратическое отклонение σ от значения μ.

   (b) Присвоить переменной OldLevel значение Level, а Level — значение $3 \cdot \sigma$.

7. Выдать значение Level в качестве искомого порогового уровня шума.

Ниже представлена реализация описанного алгоритма в псевдокоде.

```
PROGRAM NoiseLevel
CONST N=1024; L=50000;
VAR X: array [1..L] of integer;     /* массив значений амплитуд сигналов          */
    A: array [1..N] of integer = 0; /* A[k] -- количество сигналов с амплитудой k */
    h, i, k, s, f, cnt, cntall: word;
    mu, sig, sum, Level, OldLevel: double;
BEGIN_PROGRAM
  load(X); h=size(X); cntall=0;
  for i=1:h
    A[X[i]]=A[X[i]]+1; cntall=cntall+1;
  end;
  s=1; f=N; cnt=0; sum=0;
  for k=s:f
    cnt=cnt+A[k]; sum=sum+A[k]*k;
  end;
  mu=sum/cnt; k=round(mu); /* round() -- округление до ближайшего целого */
  sum=(k-mu)*(k-mu)*A[k]; cnt=A[k];
  for i=1:N
    s=round(mu-i); f=round(mu+i);
    if s>=1 cnt=cnt+A[s]; sum=sum+(s-mu)*(s-mu)*A[s]; end;
    if f<=N cnt=cnt+A[f]; sum=sum+(f-mu)*(f-mu)*A[f]; end;
    sig=sqrt(sum/cnt);
    if cnt>0.3*cntall & 3*sig>1.3*i break; end;
  end;
  Level=3*sig; OldLevel=0;
  while (Level-OldLevel)>0
    s=round(mu-Level); s=max(s,1); f=round(mu+Level); f=min(f,N); cnt=0; sum=0;
    for k=s:f
      cnt=cnt+A[k]; sum=sum+(k-mu)*(k-mu)*A[k];
    end;
    sig=sqrt(sum/cnt); OldLevel=Level; Level=3*sig;
  end;
END_PROGRAM.
```

2. Обоснование алгоритма

Пусть непрерывная случайная величина X имеет нормальное распределение вероятностей. Для простоты вычислений будем полагать, что математическое ожидание $M[X]$ равно 0. Тогда функция плотности распределения вероятности имеет вид

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}}\, e^{-\frac{x^2}{2\sigma^2}}.$$

Построим дисперсию $\sigma'^2(z)$ случайной величины X на участке $[-z;\,+z]$:

$$\sigma'^2(z) = \int\limits_{-z}^{+z} x^2 \frac{1}{\sigma\sqrt{2\pi}}\, e^{-\frac{x^2}{2\sigma^2}}\, dx = 2\int\limits_{0}^{z} x^2 \frac{1}{\sigma\sqrt{2\pi}}\, e^{-\frac{x^2}{2\sigma^2}}\, dx = \begin{bmatrix} x = \sigma\sqrt{2}\, t \\ dx = \sigma\sqrt{2}\, dt \end{bmatrix} =$$

$$= 2\int\limits_{0}^{\frac{z}{\sigma\sqrt{2}}} 2\sigma^2 t^2 \frac{1}{\sigma\sqrt{2\pi}}\, e^{-\frac{2\sigma^2 t^2}{2\sigma^2}}\, \sigma\sqrt{2}\, dt = 2\int\limits_{0}^{\frac{z}{\sigma\sqrt{2}}} \frac{2\sigma^2}{\sqrt{\pi}}\, t^2\, e^{-t^2}\, dt =$$

$$= \frac{2\sigma^2}{\sqrt{\pi}} \int\limits_{0}^{\frac{z}{\sigma\sqrt{2}}} t\, 2\, t\, e^{-t^2}\, dt = -\frac{2\sigma^2}{\sqrt{\pi}} \int\limits_{0}^{\frac{z}{\sigma\sqrt{2}}} t\, de^{-t^2} = \begin{bmatrix} dv = de^{-t^2} \\ du = dt \end{bmatrix} =$$

$$= \frac{2\sigma^2}{\sqrt{\pi}} \left(\int\limits_{0}^{\frac{z}{\sigma\sqrt{2}}} e^{-t^2}\, dt \;-\; t\, e^{-t^2} \Big|_{0}^{\frac{z}{\sigma\sqrt{2}}} \right);$$

$$\int\limits_{0}^{\frac{z}{\sigma\sqrt{2}}} e^{-t^2}\, dt = \int\limits_{0}^{\frac{z}{\sigma\sqrt{2}}} \left(1 + \frac{(-t^2)}{1!} + \frac{(-t^2)^2}{2!} + \frac{(-t^2)^3}{3!} + \frac{(-t^2)^4}{4!} + \frac{(-t^2)^5}{5!} + \frac{(-t^2)^6}{6!} + \dots\right) dt =$$

$$= \left(t - \frac{t^3}{3(1!)} + \frac{t^5}{5(2!)} - \frac{t^7}{7(3!)} + \frac{t^9}{9(4!)} - \frac{t^{11}}{11(5!)} + \frac{t^{13}}{13(6!)} - \dots\right) \Big|_{0}^{\frac{z}{\sigma\sqrt{2}}} =$$

$$= \left(t - \sum_{n=1}^{\infty} \frac{t^{4n-1}}{(4n-1)(2n-1)!} + \sum_{n=1}^{\infty} \frac{t^{4n+1}}{(4n+1)(2n)!}\right) \Big|_{0}^{\frac{z}{\sigma\sqrt{2}}};$$

$$\sigma'^2(z) = \frac{2\sigma^2}{\sqrt{\pi}} \left(t - \sum_{n=1}^{\infty} \frac{t^{4n-1}}{(4n-1)(2n-1)!} + \sum_{n=1}^{\infty} \frac{t^{4n+1}}{(4n+1)(2n)!} - t\, e^{-t^2}\right) \Big|_{0}^{\frac{z}{\sigma\sqrt{2}}}.$$

Известно, что

$$\int\limits_{-\infty}^{+\infty} x^2 \frac{1}{\sigma\sqrt{2\pi}}\, e^{-\frac{x^2}{2\sigma^2}}\, dx = \sigma^2,$$

т. е. при $z \to \infty$ имеем $\sigma'^2(z) \to \sigma^2$.

Рассмотрим следующую функцию φ на участке $(0; 3]$:

$$\varphi(x) = \frac{3\sqrt{\sigma'^2(x\sigma)}}{x\sigma} \approx \frac{3\sqrt{\sigma''^2(x)}}{x}, \text{ где}$$

$$\sigma''^2(x) = \frac{2}{\sqrt{\pi}} \left(t - \sum_{n=1}^{9} \frac{t^{4n-1}}{(4n-1)(2n-1)!} + \sum_{n=1}^{9} \frac{t^{4n+1}}{(4n+1)(2n)!} - t\, e^{-t^2}\right) \Big|_{0}^{\frac{x}{\sqrt{2}}}.$$

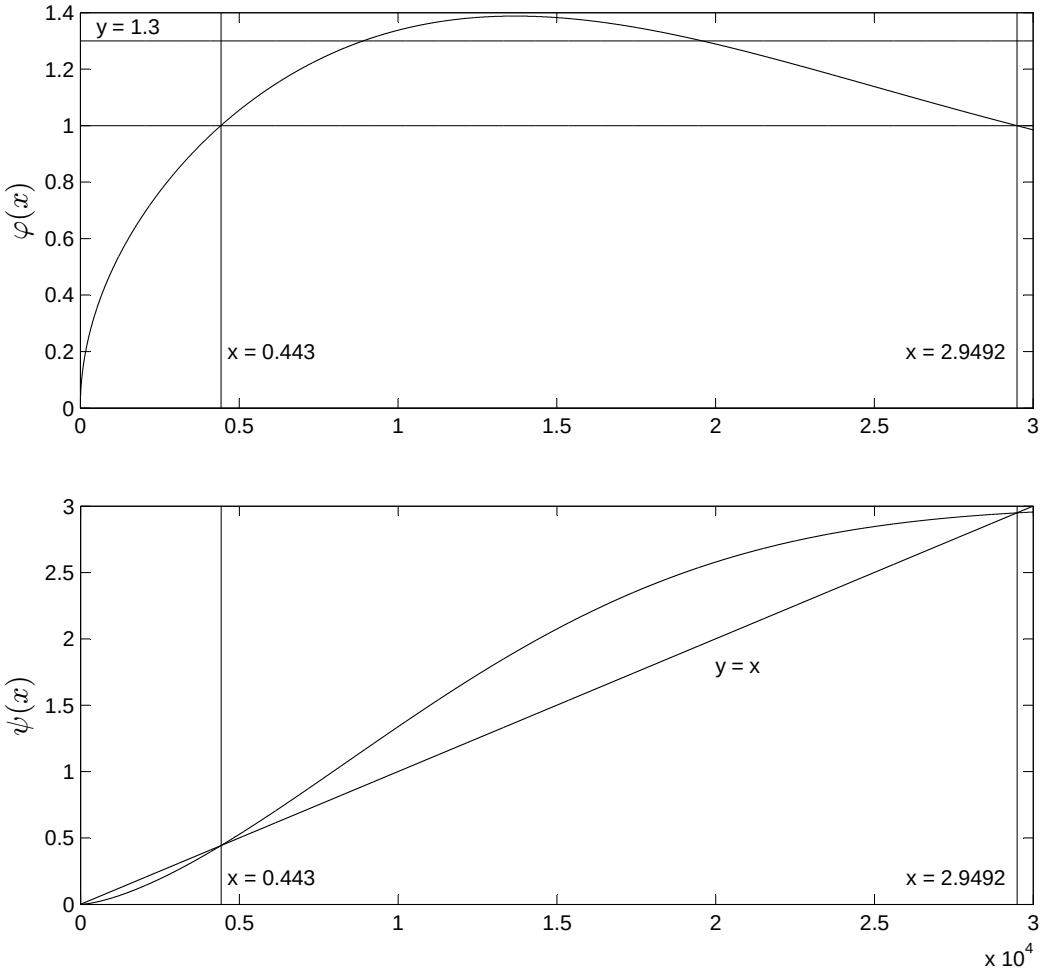

Рис. 1. Графики функций $\varphi(x)$ и $\psi(x)$ на участке $(0;3]$ при $\Delta x = 0{,}0001$
Fig. 1. Graphs of functions $\varphi(x)$ and $\psi(x)$ on the section $(0;3]$ with $\Delta x = 0{,}0001$

Вычислим приближенные значения функции $\varphi(x)$ для всех $x = k \cdot \Delta x$, где шаг $\Delta x = 0{,}0001$, а $k = 1\,..\,30000$. График функции $\varphi(x)$, построенный по вычисленным значениям, представлен на рис. 1 (верхний график).

Функция $\varphi(x)$ позволяет оценить, во сколько раз три значения среднего квадратического отклонения, построенного по элементам выборки со значениями из диапазона $[-x\sigma; +x\sigma]$, больше значения $x\sigma$, где σ — среднее квадратическое отклонение, построенное только по тем элементам выборки, которые являются шумом (соответствуют нормальному закону распределения вероятностей).

Факт, что $\varphi(x) \geqslant 1{,}3$ только при $x \in [0{,}8911; 1{,}9584]$, обеспечивает обоснование шагов с 3 по 5 алгоритма определения уровня шума, которые находят стартовую точку для следующей итеративной части этого алгоритма. Таким образом, выполнив пятый шаг, алгоритм (в идеальном случае) в качестве стартового Level будет иметь значение приближенно равное $1{,}16\,\sigma$.

На практике, очевидно, и это будет показано далее на примерах, реальное стартовое значение Level будет отклоняться от указанного. Но целью первых шагов алгоритма является выход на коэффициент 1,3, позволяющий поначалу увеличивать почти на треть диапазон амплитуд для рассматриваемых элементов выборки.

Для оценки погрешности вычисления значений рассматриваемой функции $\varphi(x)$ отметим, что при $x = 3$ и $n = 9$

$$\frac{t^{35}}{35(17!)}\Bigg|_0^{\frac{3}{\sqrt{2}}} \approx 0{,}00002; \qquad \frac{t^{37}}{37(18!)}\Bigg|_0^{\frac{3}{\sqrt{2}}} \approx 0{,}000005.$$

Отметим также, что выбранной дискретности с шагом $\Delta x = 0{,}0001$ вполне достаточно, так как 10-разрядный дефектоскоп выдает значения из интервала $[1; 1024]$, т. е. среднее квадратическое отклонение не может быть более 512, а по условию корректной записи дефектограмм оно не должно быть менее 3.

Рассмотрим еще одну новую функцию $\psi(x) = x \cdot \varphi(x)$ для $x = k \cdot \Delta x$, $x \in (0; 3]$. Эта функция соответствует шагу 6 алгоритма определения порогового уровня шума рельсов Level. На нижнем графике рис. 1 видно, что для всех рассматриваемых x значение $\psi(x)$ меньше 3. Поскольку $\psi(x) > 1$ на промежутке $(0{,}443;\ 2{,}9492)$, то, стартовав из любой точки этого промежутка, в идеальном случае алгоритм сходится в точке $x = 2.9492$. Заметим, что для $x \in (2{,}9492; 3]$ выполняется условие $x < \psi(x)$.

Запустим алгоритм с шестого шага в стартовой точке $x_0 = 1{,}16$. Получим следующую последовательность (с точностью вычислений до 0,0001): $x_1 = \psi(x_0) = 1{,}5922$; $x_2 = \psi(x_1) = 2{,}1861$; $x_3 = \psi(x_2) = 2{,}7021$; $x_4 = \psi(x_3) = 2{,}9041$; $x_5 = \psi(x_4) = 2{,}9427$; $x_6 = \psi(x_5) = 2{,}9483$; $x_7 = \psi(x_6) = 2{,}9491$; $x_8 = \psi(x_7) = 2{,}9492$; $x_9 = \psi(x_8) = 2{,}9492$; где $x_8 = x_9$. Таким образом, эта последовательность дает примерное представление о количестве повторений наиболее трудоемкого шестого шага алгоритма.

В заключение отметим, что вероятность $P(|X| < x\sigma)$ попадания нормально распределенной случайной величины X на участок $(-x\sigma;\ +x\sigma)$, где $x \in (0{,}443;\ 0{,}8911]$, принадлежит промежутку от 0,34 до 0,63. Этот участок соответствует значениям функции $\varphi(x)$ из интервала $(1;\ 1{,}3]$. Таким образом, одно из условий выхода из цикла на шаге 4 алгоритма, требующее обработки более 30% всех сигналов, ориентировано на дефектограммы, которые содержат от 90% до 50% сигналов шума.

3. Примеры

Пример 1. На верхнем графике рис. 2 представлена дефектограмма, записанная 8-разрядным магнитным дефектоскопом на 50-метровом участке рельсового пути. Средний график — фрагмент записи, который соответствует сварному стыку (соединению) рельсов. Регистрация данных проводилась каждый миллиметр пути (ось X). Амплитудное значение сигнала, полученное на одном шаге сканирования, откладывается по оси Y. Нижний график рис. 2 представляет собой оценку плотности распределения вероятности появления некоторой амплитуды на рассматриваемом участке рельсов. На всех графиках рис. 2 показаны результаты алгоритма в виде линий отсечки шума рельсов и уровня начала полезных сигналов. Для вычисления порогового уровня шума рельсов Level потребовалось 6 основных итераций алгоритма: $\text{Level}_0 = 9{,}5706$, $\text{Level}_1 = 17{,}1632$, $\text{Level}_2 = 26{,}1227$, $\text{Level}_3 = 29{,}6068$, $\text{Level}_4 = 29{,}9479$, $\text{Level}_5 = 29{,}9726$, $\text{Level}_6 = 29{,}9726 = \text{Level}$.

Пример 2. На верхнем графике рис. 3 представлена дефектограмма 8-разрядного магнитного дефектоскопа (запись данных с 50-метрового участка пути). Средний график — фрагмент записи, соответствующий болтовому стыку рельсов с боковыми соединительными накладками. Нижний график рис. 3 — оценка плотности

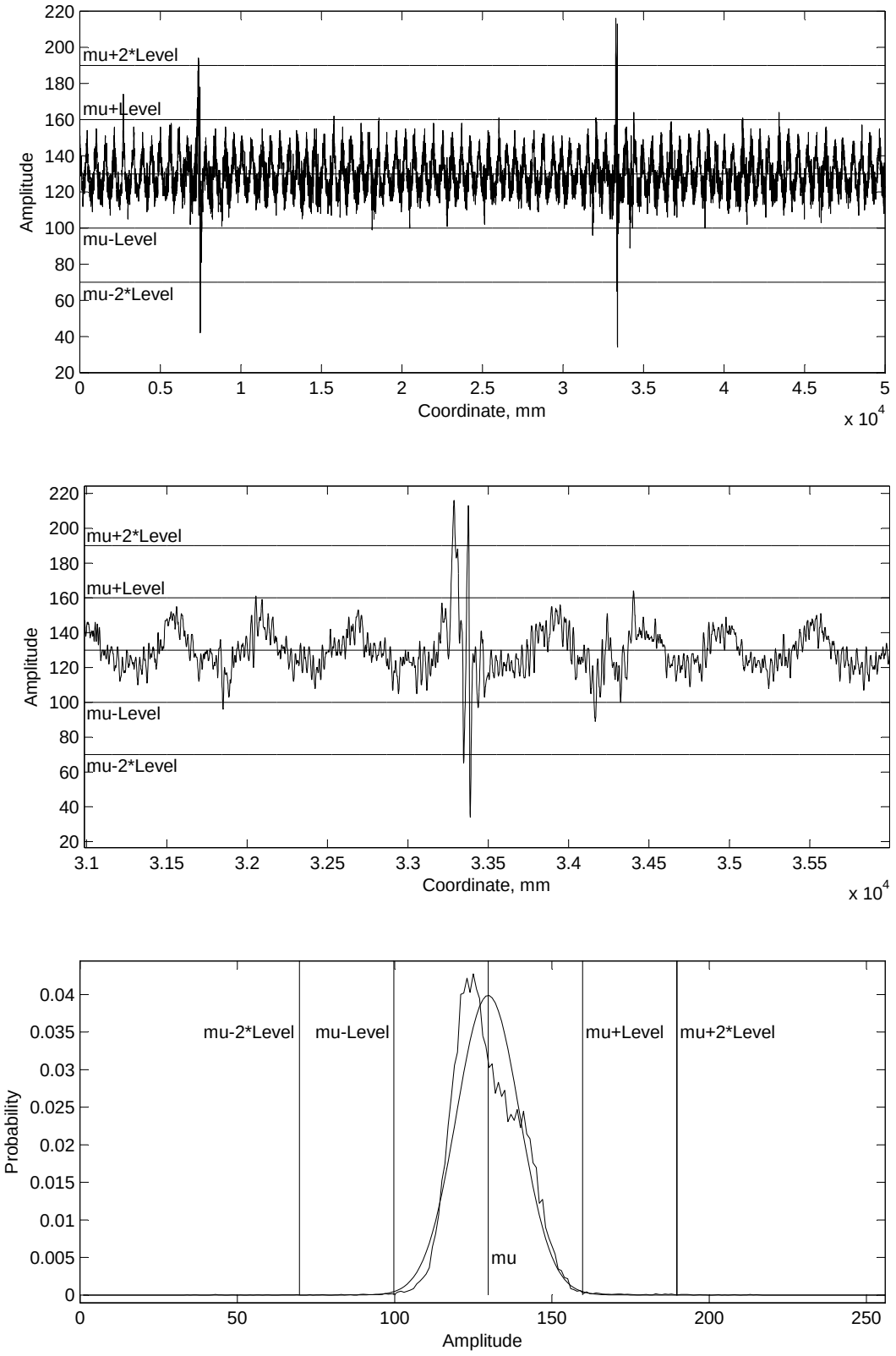

Рис. 2. Представления данных дефектограмм с результатами работы алгоритма
Fig. 2. Representations of flaw detector data and algorithm work results

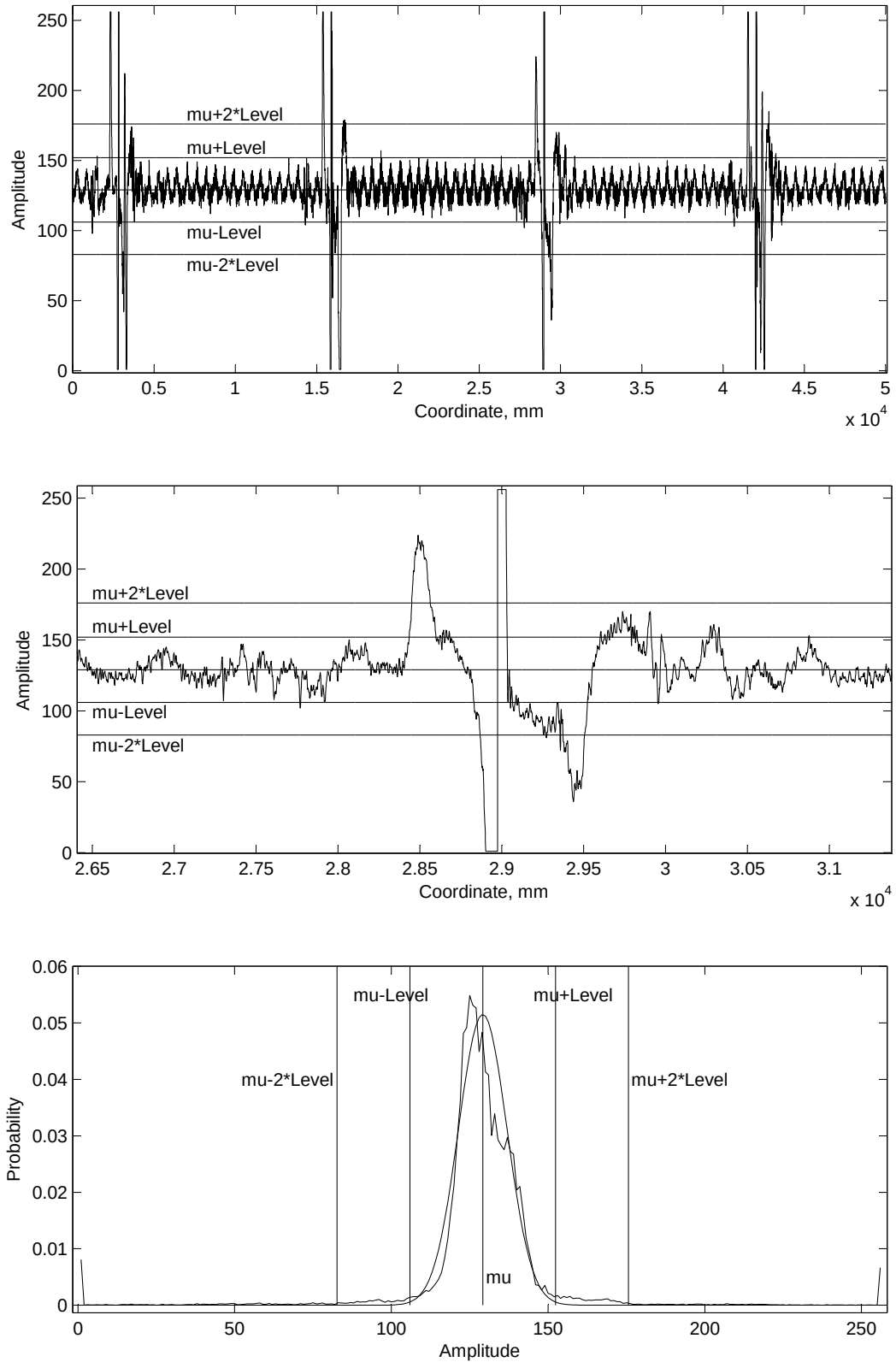

Рис. 3. Представления данных дефектограмм с результатами работы алгоритма
Fig. 3. Representations of flaw detector data and algorithm work results

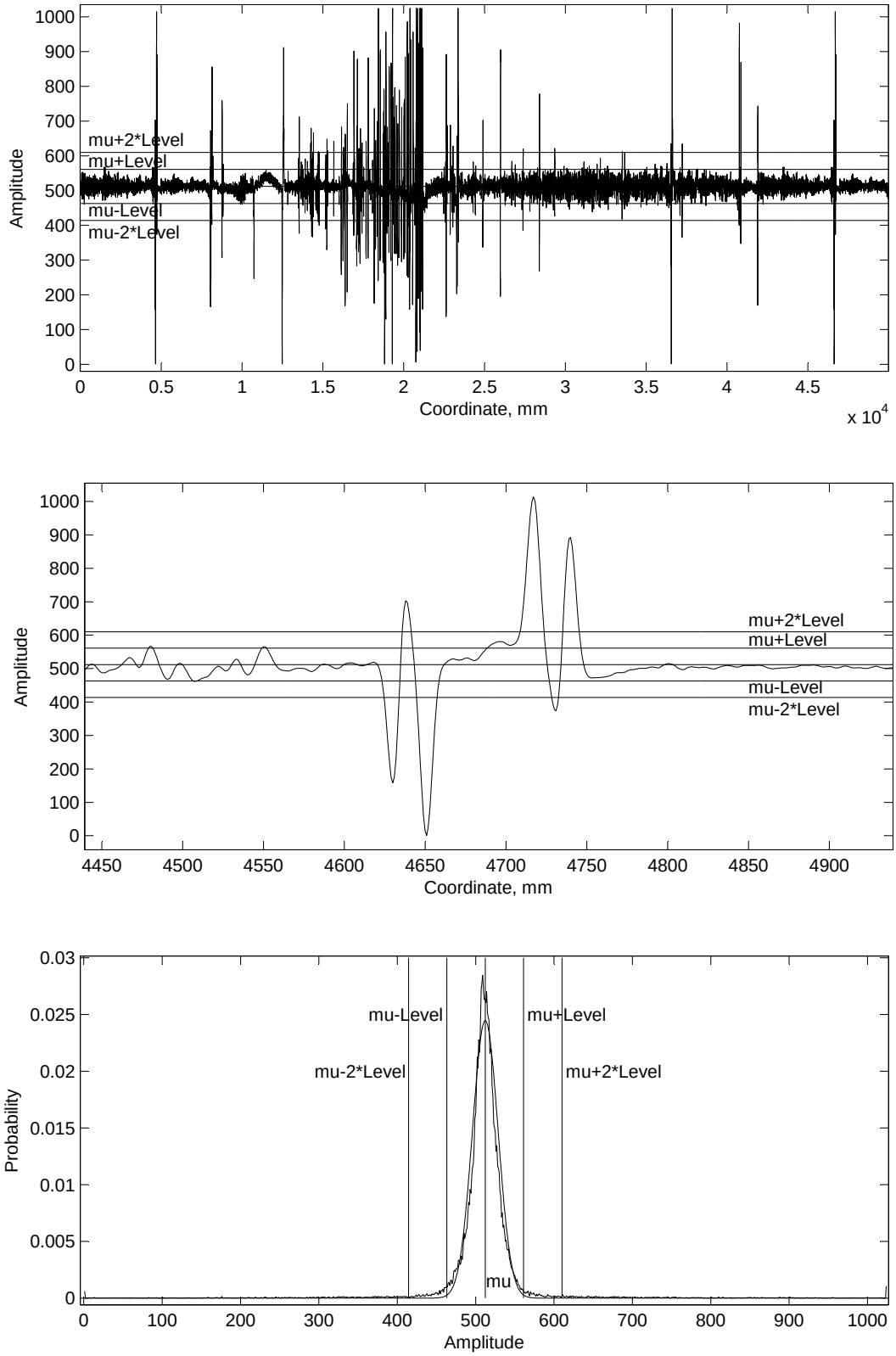

Рис. 4. Представления данных дефектограмм с результатами работы алгоритма
Fig. 4. Representations of flaw detector data and algorithm work results

распределения вероятности появления некоторой амплитуды на рассматриваемом участке рельсов. На всех графиках рис. 3 показаны результаты алгоритма в виде линий отсечки шума рельсов и уровня начала полезных сигналов. Для вычисления порогового уровня шума рельсов Level потребовалось 7 основных итераций алгоритма: $\text{Level}_0 = 6{,}0095$, $\text{Level}_1 = 11{,}0208$, $\text{Level}_2 = 17{,}2490$, $\text{Level}_3 = 21{,}1645$, $\text{Level}_4 = 22{,}5926$, $\text{Level}_5 = 23{,}0529$, $\text{Level}_6 = 23{,}1936$, $\text{Level}_7 = 23{,}1936 = \text{Level}$.

Пример 3. На верхнем графике рис. 4 представлена дефектограмма, которая была записана 10-разрядным вихретоковым дефектоскопом на 50-метровом участке рельсового пути. Средний график — фрагмент записи, соответствующий сварному стыку рельсов. Шаг сканирования — 1 мм. Нижний график рис. 4 — оценка плотности распределения вероятности появления некоторой амплитуды на рассматриваемом участке рельсов. На всех графиках рис. 4 показаны результаты алгоритма в виде линий отсечки шума рельсов и уровня начала полезных сигналов. Для вычисления порогового уровня шума рельсов Level потребовалось 8 основных итераций алгоритма: $\text{Level}_0 = 11{,}0542$, $\text{Level}_1 = 18{,}6767$, $\text{Level}_2 = 28{,}0950$, $\text{Level}_3 = 37{,}2262$, $\text{Level}_4 = 43{,}4621$, $\text{Level}_5 = 46{,}7912$, $\text{Level}_6 = 48{,}2453$, $\text{Level}_7 = 48{,}7251$, $\text{Level}_8 = 48{,}7251 = \text{Level}$.

На рассмотренных примерах видно, что шум рельсов (в приближении) подчиняется закону нормального распределения вероятностей. График оценки плотности распределения шума близок к графику плотности нормального распределения с параметрами μ и Level, где μ — математическое ожидание, а Level — среднее квадратическое отклонение.

Список литературы / References

[1] Марков А. А., Кузнецова Е. А., *Дефектоскопия рельсов. Формирование и анализ сигналов. Кн. 1. Основы*, КультИнформПресс, СПб., 2010; [Markov A. A., Kuznetsova E. A., *Rails flaw detection. Formation and analysis of signals. Book 1. Principles*, KultInformPress, St. Petersburg, 2010, (in Russian).]

[2] Марков А. А., Кузнецова Е. А., *Дефектоскопия рельсов. Формирование и анализ сигналов. Кн. 2. Расшифровка дефектограмм*, Ультра Принт, СПб., 2014; [Markov A. A., Kuznetsova E. A., *Rails flaw detection. Formation and analysis of signals. Book 2. Data interpretation*, Ultra Print, St. Petersburg, 2014, (in Russian).]

[3] Тарабрин В. Ф., Зверев А. В., Горбунов О. Е., Кузьмин Е. В., "О фильтрации данных при автоматической расшифровке дефектограмм АПК «АСТРА»", *В мире неразрушающего контроля*, **64**:2 (2014), 5–9; [Tarabrin V. F., Zverev A. V., Gorbunov O. E., Kuzmin E. V., "About Data Filtration of the Defectogram Automatic Interpretation by Hardware and Software Complex "ASTRA"", *NDT World*, **64**:2 (2014), 5–9, (in Russian).]

[4] Тарабрин В. Ф., Кузьмин Е. В., Горбунов О. Е., Зверев А. В., "Об определении динамического порога уровня сигналов при автоматической расшифровке дефектограмм АПК «АСТРА»", *Сборник тезисов научных докладов XX Всероссийской научно-техн. конф. по неразрушающему контролю и технической диагностике*, ИД «Спектр», М., 2014, 145–147; [Tarabrin V. F., Kuzmin E. V., Gorbunov O. E., Zverev A. V., "Ob opredelenii dinamicheskogo poroga urovnya signalov pri avtomaticheskoy rasshifrovke defektogramm APK "ASTRA"", *Sbornik tezisov nauchnykh dokladov XX vserossiyskoy nauchno-technicheskoy konferentsii po nerazrushayuschemu kontrolyu I tekhnicheskoy diagnostike*, Izdatlskiy dom "Spektr", Moscow, 2014, 145–147, (in Russian).]

[5] Вентцель Е. С., Овчаров Л. А., *Теория вероятностей и ее инженерные приложения*, Высшая школа, М., 2000; [Ventcel E. S., Ovcharov L. A., *Probability Theory and Its Engineering Applications*, Vysshaya Shkola Publishers, Moscow, 2000, (in Russian).]

[6] Гмурман В. Е., *Теория вероятностей и математическая статистика*, Высшая школа, М., 2004; [Gmurman V. E., *Probability Theory and Mathematical Statistics*, Vysshaya Shkola Publishers, Moscow, 2004, (in Russian).]

[7] Феллер В., *Введение в теорию вероятностей и ее приложения*, Пер. с англ. Т. 1–2, Мир, М., 1984; [Feller W., *An Introduction to Probability Theory and Its Applications*. V. I–II, John Willey & Sons, 1970–1971.]

Kuzmin E. V., Gorbunov O. E., Plotnikov P. O., Tyukin V. A., "On Finding a Threshold of Useful Signals in the Analysis of Magnetic and Eddy Current Defectograms", *Modeling and Analysis of Information Systems*, **24**:6 (2017), 760–771.

DOI: 10.18255/1818-1015-2017-6-760-771

Abstract. To ensure traffic safety of railway transport, non-destructive testing of rails is regularly carried out by using various approaches and methods, including magnetic and eddy current flaw detection methods. The paper is devoted to the problem of automatic determination of a threshold level of amplitudes of useful signals (from defects and structural elements of a railway track) during the analysis of defectograms (records) of magnetic and eddy current flaw detectors. A signal is considered useful (and is subject to further analysis) if a deviation of its value from an average of all signals is at least twice the threshold noise level of rails. The probability of obtaining a signal from a section without structural elements (a rail noise signal) is characterized by the normal distribution law. Thus, the rule of three sigma can be used to calculate the threshold noise level. And a signal is useful if its amplitude deviation from a sample mean exceeds twice the threshold noise level. The paper proposes an algorithm for finding the threshold level of a rail noise and gives its theoretical justification, and it also examines examples of its operation on several fragments of real magnetic and eddy current defectograms.

Keywords: nondestructive testing, magnetic and eddy current testing, rail flaw detection, automated analysis of magnetic and eddy current defectograms

On the authors:
Egor V. Kuzmin, orcid.org/0000-0003-0500-306X, doctor of science, associate professor,
P.G. Demidov Yaroslavl State University,
14 Sovetskaya str., Yaroslavl, 150003 Russia, e-mail: kuzmin@uniyar.ac.ru, kuzminev@nddlab.com

Oleg E. Gorbunov, orcid.org/0000-0001-6274-9971, PhD, general director,
Center of Innovative Programming, NDDLab,
144 Soyuznaya str., Yaroslavl, 150008 Russia, e-mail: gorbunovoe@nddlab.com

Petr O. Plotnikov, orcid.org/0000-0001-5687-7969, production engineer,
Center of Innovative Programming, NDDLab,
144 Soyuznaya str., Yaroslavl, 150008 Russia, e-mail: plotnikovpo@nddlab.com

Vadim A. Tyukin, orcid.org/0000-0001-9149-7435, head of software development,
Center of Innovative Programming, NDDLab,
144 Soyuznaya str., Yaroslavl, 150008 Russia, e-mail: tyukinva@nddlab.com

Acknowledgments:
[1] This work was supported by the Program of YSU development for the period 2017-2021 as a basic university of the Yaroslavl region. The event “Development of Innovative Active Divisions of the University in Priority Areas of the Region’s Economy”. Direction “Modernization of Research and Innovation Activities, Including the Development of the University’s Innovative Ecosystem”.

Моделирование и анализ информационных систем. Т. 24, №6 (2017), с. 772–787
Modeling and Analysis of Information Systems. Vol. 24, No 6 (2017), pp. 772–787

DOI: 10.18255/1818-1015-2017-6-772-787
УДК 004.912

Анализ использования различных типов связей между терминами тезауруса, сгенерированного с помощью гибридных методов, в задачах классификации текстов

Лагутина Н. С., Лагутина К. В., Щитов И. А., Парамонов И. В. [1,2]

получена 16 октября 2017

Аннотация. Цель данной статьи — проанализировать, насколько эффективно могут применяться различные типы тезаурусных связей в задачах классификации текстов. Основой исследования является автоматически сгенерированный тезаурус предметной области, содержащий три типа связей: синонимические, иерархические и ассоциативные. Для генерации тезауруса используется гибридный метод, основанный на нескольких лингвистических и статистических алгоритмах выделения семантических связей и позволяющий создать тезаурус с достаточно большим числом терминов и связей между ними. Авторы рассматривают две задачи: тематическая классификация текстов и классификация больших новостных статей по тональности. Для решения каждой из них авторами были использованы два подхода, каждый из которых дополняет стандартные алгоритмы процедурой, применяющей связи тезауруса для определения семантических особенностей текстов. Подход к тематической классификации включает в себя стандартный алгоритм BM25 вида «обучение без учителя» и процедуру, использующую синонимические и иерархические связи тезауруса предметной области. Подход к классификации по тональности состоит из двух шагов. На первом шаге создается тезаурус, тональные веса терминов которого считаются в зависимости от частоты встречаемости в обучаемой выборке или от веса соседей по тезаурусу. На втором шаге тезаурус применяется для вычисления признаков слов из текстов и классификации текстов методом опорных векторов или наивным байесовским классификатором. В экспериментах с корпусами BBCSport, Reuters, PubMed и корпусом статей об американских иммигрантах авторы варьировали типы связей, которые участвуют в классификации, и степень их использования. Результаты экспериментов позволяют оценить эффективность применения тезаурусных связей для классификации текстов на естественном языке и определить, при каких условиях те или иные связи имеют бóльшую значимость. В частности, наиболее полезными тезаурусными связями оказались синонимические и иерархические, так как они обеспечивает лучшее качество классификации.

Ключевые слова: тезаурус, семантические отношения, тезаурусные связи, тематическая классификация, классификация по тональности

Для цитирования: Лагутина Н. С., Лагутина К. В., Щитов И. А., Парамонов И. В., "Анализ использования различных типов связей между терминами тезауруса, сгенерированного с помощью гибридных методов, в задачах классификации текстов", *Моделирование и анализ информационных систем*, **24**:6 (2017), 772–787.

Об авторах: Лагутина Надежда Станиславовна, orcid.org/0000-0002-6137-8643, канд. физ.-мат. наук, доцент,
Ярославский государственный университет им. П.Г. Демидова,
ул. Советская, 14, г. Ярославль, 150003 Россия, e-mail: lagutinans@rambler.ru

Лагутина Ксения Владимировна, orcid.org/0000-0002-1742-3240, студент,
Ярославский государственный университет им. П.Г. Демидова,
ул. Советская, 14, г. Ярославль, 150003 Россия, e-mail: lagutinakv@mail.ru

Щитов Иван Андреевич, orcid.org/0000-0002-5027-5024, аспирант,
Ярославский государственный университет им. П.Г. Демидова,
ул. Советская, 14, г. Ярославль, 150003 Россия, e-mail: ivan.shchitov@e-werest.org

Парамонов Илья Вячеславович, orcid.org/0000-0003-3984-8423, канд. физ.-мат. наук, доцент,
Ярославский государственный университет им. П.Г. Демидова,
ул. Советская, 14, г. Ярославль, 150003 Россия, e-mail: Ilya.Paramonov@fruct.org

Благодарности:
[1]Работа выполнена при финансовой поддержке гранта Президента Российской Федерации для государственной поддержки молодых российских ученых (государственный контракт № МК-5456.2016.9).
[2] Авторы благодарят заведующую кафедрой иностранных языков гуманитарных факультетов ЯрГУ, доцента Наталию Николаевну Касаткину за подготовку корпуса статей об американских иммигрантах.

Введение

Автоматическая обработка текстов на естественных языках является неотъемлемой частью современных информационных технологий. Необходимость анализа текстов возникает при решении таких задач, как поиск информации, поиск ответов на вопросы, рубрикация, классификация, аннотирование документов, машинный перевод и многое другое.

Огромное количество информации содержится как в коллекциях документов из Интернета, так и в книгах, статьях, узкоспециализированных информационных базах и т. п. Для качественного и эффективного использования этих ресурсов недостаточно рассматривать текст как набор независимых терминов, требуется учитывать структуру предложений и связь между словами и терминами предметной области. Это означает, что необходима модель, объединяющая знания о языке и особенностях предметной области. Такой моделью может служить тезаурус, включающий в себя термины предметной области и связи между ними.

Изначально тезаурусы создавались для индексирования документов вручную. В сфере автоматизации одно из первых применений тезауруса произошло в области машинного перевода в 1954 году [1]. Однако при автоматической обработке текстов между текстом и результатом работы нет человека-посредника, использующего не только тезаурус или словарь, но и свои знания о предмете исследования. Есть только автоматический процесс и тезаурус, который должен включать как набор общеупотребительных терминов и связей между ними, так и те знания, которые использует эксперт для анализа текста. Таким образом, тезаурус, предназначенный для автоматической обработки текстов, должен содержать значительно больше информации о предметной области, в частности, большее количество терминов и большее количество связей [2].

Расширение базы тезауруса ведет к увеличению и усложнению отношений между понятиями тезауруса. Поэтому, кроме развития и совершенствования методов выделения таких связей, необходимо понимание того, как разные виды связей влияют на эффективность использования тезаурусов в тех или иных задачах автоматической обработки текстов. Во многих современных исследованиях отношения между терминами активно используются, однако практически нигде виды связей не дифференцируются, и, тем более, не анализируется влияние разных типов связей на

качество решения рассматриваемой задачи. Поэтому авторы статьи в рамках своих исследований в области классификации текстов по тональности поставили вопрос о степени влияния связей терминов примененного тезауруса. В данной работе описаны проведенные в этой области эксперименты и их результаты.

1. Тезаурусы и виды связей

Тезаурус представляет собой словарь, включающий в себя термины или понятия, организованный таким образом, чтобы установить явные отношения между этими терминами [3]. Термин тезауруса — это слово или словосочетание, обычно в форме существительного или именной группы, являющееся точным обозначением определённого понятия какой-либо области знания. Опишем более подробно различные виды отношений между терминами тезауруса. Одним из основных является отношение синонимии — тождественность или близость значения слов, а также морфем, синтаксических конструкций, словосочетаний. В классических тезаурусах каждая группа синонимов формирует отдельную единицу словаря — синсет. Среди синонимов выбирается дескриптор — термин, который представляет отдельное понятие предметной области. Другие термины из синонимического ряда, включённые в синсет, называются аскрипторами [4].

В соответствии со стандартом основными типами отношений между синсетами тезауруса являются следующие:

- род — вид;
- часть — целое;
- причина — следствие;
- сырьё — продукт;
- процесс — объект;
- процесс — субъект;
- свойство — носитель свойства;
- функциональное сходство;
- административная иерархия;
- антонимия.

В тезаурусах, используемых для автоматической обработки текста, все эти отношения, как правило, делятся на два типа: иерархические и ассоциативные, где ассоциативная связь обозначает наличие связи между понятиями, отличающейся от синонимической и иерархической. Таким образом, тезаурус представляет собой модель предметной области, состоящую из терминов и синонимических, иерархических и ассоциативных связей между ними.

2. Обзор смежных работ

Влияние тезаурусных связей на качество анализа текста редко анализируется в научной литературе. Часть исследований на данную тему касается задачи индексирования документов и близкой к ней проблемы тематического моделирования, другие области обработки текстов практически не затрагиваются. Авторы проанализировали несколько работ, посвященных изучению структуры тезауруса и ее значения для решения практических задач.

В работе [5] анализируется влияние структуры тезауруса на качество предметного индексирования. Авторы разработали метод индексирования, который включает в себя алгоритм случайного блуждания. Данный алгоритм обрабатывает различные семантические связи с различными вероятностями, и, таким образом, связи из тезауруса влияют на результат в большей или меньшей степени. Авторы экспериментировали с четырьмя ручными тезаурусами: AGROVOC, HEP, NALT и MeSH — и вероятностями для иерархических и ассоциативных связей. Полученная средняя точность 19.01% достаточно низкая. Она достигается несколькими возможными комбинациями параметров метода: для трех тезаурусов необходимо задать высокую вероятность для ассоциаций и низкую для остальных связей. С тезаурусом HEP возникает противоположная ситуация, гиперонимы влияют сильнее других связей.

Авторы статьи [6] также применяют тезаурус для индексирования документов. Они разработали полуавтоматический инструмент DigiDoc MetaEdit, который позволяет пользователю соотносить термины из тезауруса с HTML-документами. Инструмент составляет набор возможных ключевых слов в зависимости от частоты появления и релевантности термина и дополняет этот набор тезаурусными синонимами, гипонимами, гиперонимами и ассоциациями. При этом параметры алгоритма задаются пользователем.

Эксперименты проводились со 100 испанскими журнальными статьями с портала BiD и тезаурусом TLIS (Thesaurus on Library and Information Science). Авторы сравнивали эффективность автоматической части инструмента для случаев, когда он применяет различные виды тезаурусных связей. Лучшая полнота 73% достигается при использовании всех видов связей. Использование только горизонтальных связей позволяет достичь только 64%, иерархических — 58%. Алгоритм без тезауруса демонстрирует самую низкую полноту — 49%.

Н. Лукашевич с соавторами [7] предлагают метод решения задачи тематического моделирования, который дополняет наборы близких тем терминами из тезауруса, созданного вручную. Авторы провели несколько экспериментов с алгоритмами без тезауруса, с синонимами и с синонимами и гиперонимами, используя для измерения качества метрику уникальности ядра. Она описывает, насколько различные темы, определяемые группами связанных тезаурусных терминов, отличаются друг от друга. В результате комбинация двух связей обеспечивает лучшее значение данной метрики: 0.4–0.7, тогда как в случае алгоритма без тезауруса эта характеристика равна 0.3–0.5.

Данные исследования демонстрируют, что различные способы использования связей из тезауруса существенно влияют на результат анализа текста на естественном языке. К сожалению, значимость тезаурусных связей в данной области изучена

недостаточно, в том числе для задач классификации текстов по темам или тональности. В методах решения этих задач обычно применяется один тип связей или все связи унифицируются и обрабатываются одинаково как ассоциации.

В работе [8] решается задача классификации текстов по темам при помощи тезауруса. Тезаурус OpenOffice бразильского варианта португальского языка является вспомогательным инструментом для соотнесения текстов и терминов из онтологии. Алгоритм ищет для каждого слова из текста ближайшие связанные термины из тезауруса и отбирает те из них, которые содержатся также и в онтологии. На основе этих данных работает предложенный авторами классификатор. Точность и полнота результата вырастают на 7–10% (до 60–70%) по сравнению с методом опорных векторов без тезауруса. На следующем этапе исследования тезаурус и онтология дополняются терминами и связями вручную при помощи экспертов, а также авторы изменяют меру близости термина и текста. Это позволяет достичь одних из лучших результатов по области: 96% точности и полноты.

В статье [9] исследуется проблема классификации отзывов пользователей Интернета на позитивные и негативные. Авторы предлагают классификатор, основанный на применении автоматически созданного тезауруса. Тезаурус строится по всему корпусу текстов, на котором впоследствии будет обучаться и работать классификатор. Слова маркируются на положительные и отрицательные в зависимости от тональности текстов, в которых они чаще встречаются. Термины, появляющиеся в одном предложении, считаются связанными ассоциативной связью, причем вес данной связи также зависит от частоты встречаемости терминов в одном и том же тексте. Далее вычисляются веса пар термин–отзыв в зависимости от количества и веса связей в тезаурусе между данным термином и другими словами отзыва. Эти характеристики формируют векторы отзывов, которые в итоге поступают на вход классификатору, использующему метод максимальной энтропии. В результате получены следующие результаты для 800 позитивных и 800 негативных отзывов с Amazon: точность без тезауруса 62–72%, точность с тезаурусом 72–87% — одна из лучших для данной задачи. Предлагаемый метод позиционируется как независимый от предметной области, т.е. его потенциально можно использовать для текстов отзывов по любой теме.

Таким образом, несмотря на то, что большинство исследователей не дифференцируют семантические связи различных типов между терминами тезауруса для классификации текстов, данная идея была успешно применена для некоторых задач анализа текстов на естественном языке, поэтому ее можно попытаться распространить и на сферу классификации.

3. Классификация текстов по темам с применением тезауруса

Тематическая классификация текстов — это задача разделения корпуса текстов на несколько классов (возможно пересекающихся), каждый из которых обозначает основную тему текста. Для целей данного исследования авторы используют алгоритм тематической классификации, который дополняет существующий алгоритм

BM25 [10] процедурой, применяющей специализированный тезаурус, разработанный авторами.

3.1. Генерация тезауруса

Специализированный тезаурус генерируется полностью автоматически на корпусе текстов заданной предметной области, которые затем будут классифицироваться. Это позволяет создать модель предметной области, описывающую основные темы данной области и связи между ними. Подробная характеристика разработанного гибридного алгоритма и получающегося в результате тезауруса дана в предыдущей статье авторов [11]. Данный алгоритм объединяет в себе несколько существующих статистических и лингвистических алгоритмов выделения тезаурусных связей, за счет этого он позволяет построить значительное количество различных семантических связей между терминами. Краткое описание шагов алгоритма выглядит следующим образом:

1. Выделение терминов алгоритмом TextRank [12].
2. Построение семантических связей между терминами (ассоциативных, синонимических и гипонимо-гиперонимических) гибридным методом.
3. Фильтрация терминов, не имеющих связей.

В результате работы алгоритма создается тезаурус с большим набором терминов и различных связей между ними. Также основным достоинством алгоритма является то, что он работает полностью автоматически и не требует вмешательства эксперта ни на одном этапе, поэтому тезаурус генерируется достаточно быстро. Сравнение с существующим тезаурусом, построенным вручную, показало, что автоматический тезаурус показывает хорошие точность и полноту терминов и связей, поэтому он может успешно применяться для обработки текстовой информации [11].

3.2. Классификация текстов

Построенный тезаурус применяется авторами для тематической классификации текстов на естественном языке. Автоматическая классификация осуществляется при помощи стандартного алгоритма BM25 вида «обучение без учителя», который дополнительно учитывает связи между терминами тезауруса. Входными данными для алгоритма являются корпус текстов, не размеченных по классам, список классов и автоматический тезаурус.

Алгоритм состоит из следующих шагов:

1. Выделение всех существительных и прилагательных из текстов и вычисление частот их встречаемости.
2. Создание для каждого класса списка терминов-соседей по тезаурусу.
3. Ранжирование пар текст–класс алгоритмом BM25 в зависимости от частоты встречаемости терминов класса и их соседей.

Сначала алгоритм выбирает из текстов отдельные термины, строит для каждого текста инвертированный индекс и считает частоту встречаемости каждого слова.

Затем для терминов класса находятся соседи по тезаурусу первого порядка, т.е. термины, имеющие с ними прямую связь — синонимическую, гипонимическую или гиперонимическую. Ассоциации не обрабатываются, так как они часто обозначают довольно слабую семантическую связь и даже могут связывать слова из разных тем, например, «тромб» и «сердечный клапан». С синонимическими и иерархическими связями ситуация обратная: они в большинстве случаев отражают связи между терминами из одной и той же темы, так что лучше подходят для тематического сопоставления текстов и классов.

На последнем шаге применяется BM25 для пар текст–класс, чтобы ранжировать тексты в зависимости от терминов классов и их терминов-соседей по тезаурусу. Данный алгоритм популярен при решении различных задач анализа текстов, в том числе классификации [13]. Также он не требует настройки дополнительных параметров и обучения на существующих корпусах данных, поэтому он легко дополняется обработкой информации из тезауруса, так что хорошо подходит для целей данного исследования.

4. Классификация текстов по тональности с применением тезауруса

Проблема классификации текстов по тональности подразумевает разделение корпуса текстов на два или более класса в зависимости от тональности текста в целом: на положительные или отрицательные, или на положительные, отрицательные и нейтральные. В данной работе рассматривается задача классификации больших новостных статей на два класса с использованием автоматически сгенерированного тезауруса.

Предлагаемый авторами подход состоит из следующих двух шагов: создание тонального тезауруса и классификация текстов. На первом шаге полностью автоматически создается тезаурус, на втором статьи классифицируются при помощи метода опорных векторов и наивного байесовского классификатора, причем векторы признаков составляются на основе весов терминов в тезаурусе. Оба шага получают на вход корпус необработанных новостных статей, предварительно разделенный на обучающую и тестовую выборки. Обучающая выборка изначально была промаркирована экспертами-филологами.

4.1. Генерация тезауруса

В качестве терминов тонального тезауруса выбираются все слова, которые могут иметь положительную или отрицательную семантику: существительные, прилагательные, глаголы и наречия. Связи между ними строятся по алгоритму, описанному в предыдущей главе. Результатом данного алгоритма является специализированный тезаурус с большим количеством синонимических, ассоциативных и гипонимо-гиперонимических связей. Большое количество связей может позволить вычислить

более точную числовую характеристику тональности термина, так как тональность в данном случае будет зависеть сразу от нескольких соседей термина по тезаурусу.

На последнем этапе генерации терминам сопоставляются тональные веса, обозначающие тональность: от -1 до 0 для отрицательных и от 0 до 1 для положительных терминов. Сначала считаются веса тех терминов, которые встречаются в обучающей выборке текстов. Поскольку тексты в данной выборке уже промаркированы по тональности, её можно распространить и на термины.

Авторы предположили, что положительные термины чаще встречаются в положительных текстах, а отрицательные — в отрицательных, поэтому использовали для веса термина следующую формулу: $w = (p-n)/(p+n)$, где p обозначает, сколько раз термин появился в положительных текстах, n — в отрицательных. Данная формула позволяет назначить тональности из диапазона от -1 до 1, описанного выше.

Далее необходимо вычислить веса терминов, которые не встречаются в обучающей выборке, но появляются в тестовой. Авторы предположили, что термины, имеющие общую тезаурусную связь, имеют и близкую тональность. Поэтому тональность можно вычислить, опираясь на веса соседей по тезаурусу. Для реализации данной идеи каждому типу связи был назначен свой коэффициент для преобразования тональности, который варьировался от 0.1 до 1.0.

Алгоритм вычисления весов по тезаурусным связям следующий. Если у термина есть синонимы, маркированные по тональности, берется их средний вес и умножается на коэффициент связи, результат записывается как вес термина. Если у термина таких синонимов нет, но есть гиперонимы, аналогично поступают с ними. И если у термина имеются только ассоциации, среднее значение, умноженное на коэффициент, считается для них. Так как связей в тезаурусе достаточно много, у каждого термина будет хотя бы один маркированный сосед.

Таким образом, сгенерированный тезаурус содержит набор терминов из предметной области, семантические связи между ними и тональные веса всех терминов.

4.2. Классификация текстов

Тональность терминов созданного тезауруса используется для вычисления векторов признаков, используемых далее в алгоритме классификации. Каждому тексту ставится в соответствие числовой вектор, размер которого равен количеству терминов в тезаурусе. Каждый элемент вектора соответствует конкретному тезаурусному термину и считается как произведение $w \cdot F$, где w — это вес термина, а F — некоторая стандартная статистическая характеристика пары термин—текст, зависящая от частоты встречаемости термина в корпусе или тексте и/или от его веса в тезаурусе. Авторы использовали четыре различные статистические характеристики: TF*IDF, коэффициент Джини, расстояние Кульбака—Лейблера, взаимная информация и критерий χ^2 [14].

На последнем этапе векторы признаков подаются на вход одному из стандартных бинарных классификаторов. Многие исследования показывают, что лучшими классификаторами для вычисления тональности текстов являются алгоритмы машинного обучения [15]. Среди данных алгоритмов одними из самых эффективных

считаются метод опорных векторов и наивный байесовский классификатор [16], поэтому они были выбраны авторами для реализации предлагаемого подхода.

5. Корпуса текстов и методика проведения экспериментов

Предложенные авторами алгоритмы классификации тестировались на нескольких корпусах англоязычных текстов из различных областей знаний:

- Корпус PubMed (https://www.nlm.nih.gov/databases/download/pubmed_medline.html) содержит 1000 медицинских статей из 63 классов, общее количество слов — 154 850.

- Корпус Reuters (http://www.daviddlewis.com/resources/testcollections/reuters21578/) содержит 1534 экономические статьи из 15 классов, общее количество слов — 294 813.

- Корпус BBCSport (http://mlg.ucd.ie/datasets/bbc.html) содержит 737 новостных спортивных статей из 5 классов, общее количество слов — 253 667.

- Корпус статей об американских иммигрантах из газет The New York Times, The New York Post и The Los Angeles Times. Он содержит 56 статей, из них 34 положительных и 22 отрицательных, общее количество слов — 37 669. Корпус был создан и размечен по тональности экспертами-филологами.

Первые три корпуса текстов использовались для тематической классификации, последний — для классификации по тональности. Также четвертый корпус был разделен на обучающую и тестовую выборки, по 17 положительных и 11 отрицательных статей в каждой.

Оба алгоритма классификации реализованы на языке программирования Python с использованием библиотеки NLTK (http://www.nltk.org). Данная библиотека содержит реализации стандартных алгоритмов обработки текстов и машинного обучения, в том числе наивного байесовского классификатора и метода опорных векторов.

Процедура оценки результатов классификации также была написана на языке Python. Авторы применили для оценки стандартные метрики качества: точность, полноту, F-меру и долю правильных ответов [17].

Следует отметить, что для тематической классификации были выбраны микроусредненные метрики, которые считают полноту и точность одновременно по всем классам, так как классов у текстов достаточно много и подсчет отдельных значений метрик для каждого класса усложнит анализ результата. Задача второго типа, классификация по тональности, подразумевает деление всего на два класса, поэтому точность, полнота и F-мера считались отдельно для положительных и отрицательных текстов.

6. Результаты экспериментов

Авторы поставили несколько экспериментов с разработанными алгоритмами и при этом варьировали их параметры: подключали различные типы связей по отдельности и в комбинации. Для алгоритма тематической классификации также изменялся порог фильтрации: если результат BM25 был меньше определенного порога (0, 2 или 4), пара класс–текст отвергалась. Для алгоритма классификации по тональности рассматривались различные коэффициенты связей от 0.1 до 1.0 с шагом 0.1. В данном случае коэффициенты для разных типов связей использовались, чтобы проанализировать изменение степени влияния каждого типа связи в отдельности.

Таблица 1. Тематическая классификация корпуса BBCSport
Table 1. Topical classification of the BBCSport corpus

Связи	Порог фильтрации	P	R	F	A
нет	4	0.835	0.096	0.173	0.815
синонимы	4	0.828	0.098	0.175	0.815
гипонимы	4	0.790	0.107	0.189	0.816
гиперонимы	0	0.403	0.479	0.438	0.754
синонимы и гипонимы	4	0.784	0.109	0.191	0.816
синонимы и гиперонимы	0	0.399	0.482	0.437	0.751

Таблица 2. Тематическая классификация корпуса Reuters
Table 2. Topical classification of the Reuters corpus

Связи	Порог фильтрации	P	R	F	A
нет	4	0.738	0.246	0.369	0.933
синонимы	2	0.519	0.559	0.538	0.924
синонимы	4	0.749	0.276	0.404	0.935
все	0	0.329	0.746	0.457	0.859

Таблица 3. Тематическая классификация корпуса PubMed
Table 3. Topical classification of the PubMed corpus

Связи	Порог фильтрации	P	R	F	A
нет	4	0.234	0.065	0.101	0.943
синонимы	4	0.216	0.072	0.109	0.941
гиперонимы	2	0.169	0.169	0.169	0.917
все	0	0.125	0.201	0.154	0.890

В таблицах 1, 2, 3 представлены лучшие результаты классификации по темам для каждого корпуса текстов. Символы P, R и F обозначают точность, полноту и F-меру соответственно. A — это доля правильных ответов (от accuracy). «Нет»

в первом столбце значит, что данный эксперимент проводился с алгоритмом без связей, а «все» — что подключались все связи: гипонимы, гиперонимы и синонимы.

Из результатов классификации новостей BBCSport видно, что лучшую точность 0.835, но самую низкую полноту 0.096 обеспечивает алгоритм, который не использует тезаурус. Лучшая полнота 0.482 и вместе с тем худшая точность 0.399 у алгоритма, использующего синонимы и гиперонимы и вместе с тем не использующего фильтрацию. Примечательно, что его F-мера 0.437 отличается от лучшей всего на 0.1, которая является результатом алгоритма с гиперонимами. Лучшая доля правильных ответов 0.816 у алгоритмов с синонимами и гипонимами, т.е. они дают больше всего правильных ответов, но F-мера у всех них низкая: 0.175–0.191. Таким образом, лучшие характеристики для корпуса BBCSport достигаются за счет использования иерархических связей.

Для корпуса Reuters самыми значимыми связями оказались синонимы. Именно алгоритмы с ними обеспечивают лучшую F-меру 0.538 и лучшую долю правильных ответов 0.935.

Результаты классификации PubMed получились низкими для всех комбинаций тезаурусных связей. Алгоритм выдал слишком мало пар класс–текст, поэтому точность и полнота не превышают 0.25. Доля правильных ответов высока за счет того, что она учитывает, сколько текстов было верно не соотнесено с неподходящими классами, и за счет большого количества классов (63) это число достаточно большое.

Качество классификации по тональности выше тематической в абсолютных числах. Ее результаты представлены в таблицах 4–10. P, R и F обозначают точность, полноту и F-меру, нижние индексы neg и pos — отрицательный и положительный классы текстов. Слова Hyp, Syn и $Assoc$ подразумевают использование в эксперименте гиперонимов, синонимов и ассоциаций.

Таблица 4. Классификация по тональности с методом опорных векторов и коэффициентом Джини
Table 4. Sentiment classification with SVM and Gini Index

Связи	Коэффициенты	A	P_{neg}	R_{neg}	F_{neg}	P_{pos}	R_{pos}	F_{pos}
нет	0	0.643	0.571	0.364	0.444	0.667	0.824	0.737
Hyp	0.1, 0.3, 0.7, 0.9	0.750	1.000	0.364	0.533	0.708	1.000	0.829
Syn и $Assoc$	1.0 и 0.6	0.714	0.800	0.364	0.500	0.696	0.941	0.800

Таблица 5. Классификация по тональности с наивным байесовским классификатором и коэффициентом Джини
Table 5. Sentiment classification with Naive Bayes and Gini Index

Связи	Коэффициенты	A	P_{neg}	R_{neg}	F_{neg}	P_{pos}	R_{pos}	F_{pos}
нет	0	0.607	0.500	0.273	0.353	0.636	0.824	0.718
Syn	0.5	0.714	0.800	0.364	0.500	0.696	0.941	0.800
$Assoc$	0.6	0.679	0.667	0.364	0.471	0.682	0.882	0.769
Syn и Hyp	1.0 и 1.0	0.679	0.667	0.364	0.471	0.682	0.882	0.769

Таблица 6. Классификация по тональности с методом опорных векторов и расстоянием Кульбака—Лейблера
Table 6. Sentiment classification with SVM and information gain

Связи	Коэффициенты	A	P_{neg}	R_{neg}	F_{neg}	P_{pos}	R_{pos}	F_{pos}
нет	0	0.571	0.444	0.364	0.400	0.632	0.706	0.667
Syn и *Hyp*	1.0 и 0.5, 0.6, 0.9	0.750	0.833	0.455	0.588	0.727	0.941	0.821
Syn и *Assoc*	1.0 и 0.5	0.714	0.714	0.455	0.556	0.714	0.882	0.789

Таблица 7. Классификация по тональности с наивным байесовским классификатором и расстоянием Кульбака—Лейблера
Table 7. Sentiment classification with Naive Bayes and information gain

Связи	Коэффициенты	A	P_{neg}	R_{neg}	F_{neg}	P_{pos}	R_{pos}	F_{pos}
нет	0	0.643	0.667	0.182	0.286	0.640	0.941	0.762
Syn	1.0	0.714	0.800	0.364	0.500	0.696	0.941	0.800
Syn и *Hyp*	1.0 и 0.1, 0.9	0.750	1.000	0.364	0.533	0.708	1.000	0.829
Syn и *Assoc*	1.0 и 0.3	0.714	0.800	0.364	0.500	0.696	0.941	0.800

Таблица 8. Классификация по тональности с методом опорных векторов и взаимной информацией
Table 8. Sentiment classification with SVM and mutual information

Связи	Коэффициенты	A	P_{neg}	R_{neg}	F_{neg}	P_{pos}	R_{pos}	F_{pos}
нет	0	0.607	0.500	0.364	0.421	0.650	0.765	0.703
Syn	0.1–0.9	0.643	0.571	0.364	0.444	0.667	0.824	0.737
Hyp	0.5	0.714	0.800	0.364	0.500	0.696	0.941	0.800
Syn и *Hyp*	1.0 и 0.4, 0.8	0.643	0.571	0.364	0.444	0.667	0.824	0.737

Таблица 9. Классификация по тональности с наивным байесовским классификатором и взаимной информацией
Table 9. Sentiment classification with Naive Bayes and mutual information

Связи	Коэффициенты	A	P_{neg}	R_{neg}	F_{neg}	P_{pos}	R_{pos}	F_{pos}
нет	0	0.607	0.500	0.273	0.353	0.636	0.824	0.718
Hyp	1.0	0.750	0.833	0.455	0.588	0.727	0.941	0.821
Syn и *Hyp*	1.0 и 0.5, 0.8	0.750	0.750	0.545	0.632	0.750	0.882	0.811

Каждая таблица описывает результаты классификации статей об американских иммигрантах для пар классификатор–характеристика веса термина. В таблицы не вошли пары с TF*IDF и пара байесовский классификатор — χ^2, так как их результаты оказались ниже остальных и при этом не зависели от способа использования тезауруса. Отметим, что во всех случаях результаты для алгоритмов с тезуарусом

Таблица 10. Классификация по тональности с методом опорных векторов и χ^2
Table 10. Sentiment classification with SVM and χ^2

Связи	Коэффициенты	A	P_{neg}	R_{neg}	F_{neg}	P_{pos}	R_{pos}	F_{pos}
нет	0	0.643	0.571	0.364	0.444	0.667	0.824	0.737
Hyp	0.2, 0.6	0.679	0.667	0.364	0.471	0.682	0.882	0.769
Syn и *Hyp*	1.0 и 0.4	0.679	0.667	0.364	0.471	0.682	0.882	0.769
Syn и *Hyp*	1.0 и 0.8	0.679	0.625	0.455	0.526	0.700	0.824	0.757
Syn и *Assoc*	1.0 и 0.1–0.9	0.679	0.625	0.455	0.526	0.700	0.824	0.757

оказались выше результатов алгоритма без него, а в большинстве случаев — значительно выше.

В сочетании с методом опорных векторов и коэффициентом Джини (таблица 4) лучший результат по всем характеристикам дало использование гиперонимов, причем коэффициент гиперонимов практически не играет роли: он может быть 0.1, 0.3, 0.7 или 0.9. Для сочетания байесовский классификатор — коэффициент Джини (таблица 5) то же можно сказать про синонимы с коэффициентом 0.5.

При экспериментах с комбинацией метода опорных векторов и расстояния Кульбака—Лейблера (таблица 6) снова выделяется конкретный случай с синонимами и гиперонимами, который обеспечивает лучшие результаты. Следует отметить, что случай с синонимами и ассоциациями незначительно хуже, почти все его метрики отличаются от лучших не больше чем на 5%. И для комбинации байесовский классификатор — расстояние Кульбака—Лейблера (таблица 7) лучшие результаты обеспечиваются именно синонимами с коэффициентом 1.0.

Для комбинаций метод опорных векторов — взаимная информация (таблица 8) и байесовский классификатор — взаимная информация (таблица 9) являются значимыми и синонимы, и гиперонимы, так как именно они и их пара позволяют достичь самых высоких значений метрик качества 0.75–0.95. Примечательно, что коэффициент синонимов был равен 1.0, а коэффициент гиперонимов колебался от 0.4 до 1.0.

При экспериментах с комбинацией метода опорных векторов и χ^2 (таблица 10) было обнаружено, что тип используемой тезаурусной связи не играет большой роли: хорошие результаты были обеспечены как гиперонимами и их парой с синонимами, так и парой синонимы — ассоциации, причем коэффициент ассоциаций оказался не значимым.

Подводя итоги всех экспериментов, авторы обнаружили следующие закономерности. Использование тезаурусных связей существенно увеличивает качество результата по всем характеристикам по сравнению с алгоритмами без тезауруса. Самыми значимыми связями в тезаурусе оказались синонимические и иерархические, так как в решениях обеих задач классификации именно они обеспечили лучшие результаты: F-меру 0.5–0.8 и долю правильных ответов 0.75–0.9. Ассоциативные связи не сделали существенного вклада в качество результата, как видно практически из всех экспериментов с разными параметрами.

Другая закономерность заключается в зависимости способа использования тезауруса от предметной области текстов. Эксперименты показали, что для новостных

текстов BBCSport более качественный результат дают иерархические связи, а для экономических текстов Reuters — синонимические. Для больших газетных статей об иммигрантах оказались важными сразу два типа связей: синонимы и гиперонимы, причем ни одну из них нельзя выделить как наиболее значимую.

Заключение

Исследование результатов классификации текстов из разных предметных областей позволяет сделать не только общий вывод о наиболее существенном влиянии синонимических и иерархических связей по сравнению с ассоциативными, но и выделить более детальные закономерности. Наиболее важным с точки зрения авторов является то, что в ходе классификации коротких новостных текстов существенное влияние оказал один тип связей, в то время как при классификации статей большого объема наилучшие результаты оказались при использовании всех связей между терминами тезауруса. Это может быть обусловлено тем, что полноценные публицистические тексты на естественном языке используют гораздо более богатую лексику и структуру предложений, в отличие от коротких новостных публикаций. Поэтому для анализа таких текстов необходим тезаурус, обладающий большим количеством терминов и связей между ними.

Другой важный вывод заключается в том, что влияние разных типов связей между терминами зависит от конкретной предметной области. Это наблюдение нуждается в дополнительном тщательном исследовании. Еще одно направление для дальнейшей работы заключается в детализации ассоциативных связей и анализе их влияния при обработке больших текстов, как публицистических, так и научных, художественных и т.д. В любом случае, исследования показали, что связи между терминами являются важнейшей частью тезауруса как модели предметной области и что тщательный выбор способа их использования позволяет более качественно решать задачи автоматической обработки текста.

Список литературы / References

[1] Masterman M., “Semantic message detection for machine translation, using an interlingua”, *Proc. 1961 International Conf. on Machine Translation*, 1961, 438–475.

[2] Loukachevitch N., Dobrov B., “The Sociopolitical Thesaurus as a resource for automatic document processing in Russian”, *Terminology*, **21**:2 (2015), 237–262.

[3] Aitchison J., Clarke S.D., “The thesaurus: a historical viewpoint, with a look to the future”, *Cataloging and classification quarterly*, **37**:3–4 (2004), 5–21.

[4] Лукашевич Н. В., *Тезаурусы в задачах информационного поиска*, Издательство МГУ, М., 2011, 512 с.; [Lukashevich N. V., *Tezaurusy v zadachah informacionnogo poiska*, Izdatelstvo MGU, Moskow, 2011, 512 pp., (in Russian).]

[5] Willis C., Losee R., “A random walk on an ontology: Using thesaurus structure for automatic subject indexing”, *Journal of the American Society for Information Science and Technology*, **64**:7 (2013), 1330–1344.

[6] Vállez M., Pedraza-Jiménez R., Codina L., Blanco S., Rovira C, “A semi-automatic indexing system based on embedded information in HTML documents”, *Library Hi Tech*, **33**:2 (2015), 195–210.

[7] Loukachevitch N., Nokel M., Ivanov K., *Combining Thesaurus Knowledge and Probabilistic Topic Models*, 2017, https://arxiv.org/abs/1707.09816.

[8] Sanchez-Pi N., Martí L. Garcia A. C. B., "Improving ontology-based text classification: An occupational health and security application", *Journal of Applied Logic*, **17** (2016), 48–58.

[9] Bollegala D., Weir D., Carroll J., "Cross-domain sentiment classification using a sentiment sensitive thesaurus", *IEEE transactions on knowledge and data engineering*, **25**:8 (2013), 1719–1731.

[10] Sparck Jones K., Walker S., Robertson S.E., "A probabilistic model of information retrieval: development and comparative experiments: Part 2", *Information Processing and Management*, **36**:6 (2000), 809–840.

[11] Лагутина Н. С., Лагутина К. В., Мамедов Э. И., Парамонов И. В., "Методические аспекты выделения семантических отношений для автоматической генерации специализированных тезаурусов и их оценки", *Моделирование и анализ информационных систем*, **23**:6 (2016), 826–840; [Lagutina N. S., Lagutina K. V., Mamedov E. I., Paramonov I. V., "Methodological aspects of semantic relationship extraction for automatic thesaurus generation", *Modeling and Analysis of Information Systems*, **23**:6 (2016), 826–840, (in Russian).]

[12] Mihalcea R., Tarau P., "TextRank: Bringing order into texts", *Proceedings of Empirical Methods in Natural Language Processing – EMNLP*, ACL, Barcelona, Spain, 2004, 404–411.

[13] Trieschnigg D., Pezik P., Lee V., De Jong F., Kraaij W., Rebholz-Schuhmann D., "MeSH Up: effective MeSH text classification for improved document retrieval", *Bioinformatics*, **25**:11 (2009), 1412–1418.

[14] Aggarwal C., Zhai C., "A survey of text classification algorithms", *Mining text data*, Springer-Verlag, New York, 2012, 163–222.

[15] Grimmer J., Stewart B., "Text as data: The promise and pitfalls of automatic content analysis methods for political texts", *Political analysis*, **21**:3 (2013), 267–297.

[16] Ravi K., Ravi V., "A survey on opinion mining and sentiment analysis: tasks, approaches and applications", *Knowledge-Based Systems*, **89** (2015), 14–46.

[17] Junker M., Hoch R., Dengel A., "On the evaluation of document analysis components by recall, precision, and accuracy", *Proceedings of the Fifth International Conference on Document Analysis and Recognition*, IEEE, 1999, 713–716.

Lagutina N. S., Lagutina K. V., Shchitov I. A., Paramonov I. V., "Analysis of Influence of Different Relations Types on the Quality of Thesaurus Application to Text Classification Problems", *Modeling and Analysis of Information Systems*, **24**:6 (2017), 772–787.

DOI: 10.18255/1818-1015-2017-6-772-787

Abstract. The main purpose of the article is to analyze how effectively different types of thesaurus relations can be used for solutions of text classification tasks. The basis of the study is an automatically generated thesaurus of a subject area, that contains three types of relations: synonymous, hierarchical and associative. To generate the thesaurus the authors use a hybrid method based on several linguistic and statistical algorithms for extraction of semantic relations. The method allows to create a thesaurus with a sufficiently large number of terms and relations among them. The authors consider two problems: topical text classification and sentiment classification of large newspaper articles. To solve them, the authors developed two approaches that complement standard algorithms with a procedure that take into account thesaurus relations to determine semantic features of texts. The approach to topical classification includes the standard unsupervised BM25 algorithm and the procedure, that take into account synonymous and hierarchical relations of the thesaurus of the subject area. The approach to sentiment classification consists of two steps. At the first step, a thesaurus is created, whose terms

weight polarities are calculated depending on the term occurrences in the training set or on the weights of related thesaurus terms. At the second step, the thesaurus is used to compute the features of words from texts and to classify texts by the algorithm SVM or Naive Bayes. In experiments with text corpora BBCSport, Reuters, PubMed and the corpus of articles about American immigrants, the authors varied the types of thesaurus relations that are involved in the classification and the degree of their use. The results of the experiments make it possible to evaluate the efficiency of the application of thesaurus relations for classification of raw texts and to determine under what conditions certain relationships affect more or less. In particular, the most useful thesaurus connections are synonymous and hierarchical, as they provide a better quality of classification.

Keywords: thesaurus, semantic relations, thesaurus relations, topical classification, sentiment classification

On the authors:
Nadezhda S. Lagutina, orcid.org/0000-0002-6137-8643, PhD, associate professor
P.G. Demidov Yaroslavl State University,
14 Sovetskaya str., Yaroslavl, 150003 Russia, e-mail: lagutinans@rambler.ru

Ksenia V. Lagutina, orcid.org/0000-0002-1742-3240, student,
P.G. Demidov Yaroslavl State University,
14 Sovetskaya str., Yaroslavl, 150003 Russia, e-mail: lagutinakv@mail.ru

Ivan A. Shchitov, orcid.org/0000-0002-5027-5024, graduate student,
P.G. Demidov Yaroslavl State University,
14 Sovetskaya str., Yaroslavl, 150003 Russia, e-mail: ivan.shchitov@e-werest.org

Ilya V. Paramonov, orcid.org/0000-0003-3984-8423, PhD, associate professor
P.G. Demidov Yaroslavl State University,
14 Sovetskaya str., Yaroslavl, 150003 Russia, e-mail: Ilya.Paramonov@fruct.org

Acknowledgments:
This work was supported by the grant of the President of Russian Federation for state support of young Russian scientists (project MK-5456.2016.9).
[2] The authors would like to thank Natalia Kasatkina, associate professor, head of the Department of Foreign Languages of Humanities in P.G. Demidov Yaroslavl State University, for preparation of the corpus of articles about American immigrants.

Моделирование и анализ информационных систем. Т. 24, №6 (2017), с. 788–801
Modeling and Analysis of Information Systems. Vol. 24, No 6 (2017), pp. 788–801

DOI: 10.18255/1818-1015-2017-6-788-801

УДК 519.17

Задача о кратчайшем пути в кратном графе

Смирнов А. В.[1]

получена 23 августа 2017

Аннотация.

В статье вводится определение неориентированного кратного графа произвольной натуральной кратности $k > 1$. Кратный граф содержит ребра трех типов: обычные, кратные и мультиребра. Ребра последних двух типов представляют собой объединение k связанных ребер, которые соединяют 2 или $k + 1$ вершину соответственно. Связанные ребра могут использоваться только согласованно. Если вершина инцидентна какому-либо кратному ребру, то она может быть инцидентна другим кратным ребрам, а также она может быть общим концом k связанных ребер какого-либо мультиребра. Если вершина является общим концом какого-либо мультиребра, то она не может быть общим концом никакого другого мультиребра.

Отдельно рассматривается класс делимых кратных графов, основной особенностью которых является возможность выделения k частей, согласованных на всех связанных ребрах и не содержащих общих ребер. Каждая из частей является обычным графом.

Для кратного графа обобщаются понятия степени вершины, связности графа, пути, цикла, веса ребра и длины пути.

Вводится понятие множества достижимости по обычным и по кратным ребрам, определяется свойство смежности двух множеств достижимости. Показано, что проверка связности кратного графа может быть выполнена за полиномиальное время с помощью алгоритма, основанного на поиске множеств достижимости и проверки их смежности.

Рассматривается критерий существования кратного пути между двумя вершинами и ставится задача о кратчайшем кратном пути. Строится алгоритм поиска кратчайшего пути в кратном графе, который использует алгоритм Дейкстры для поиска кратчайших путей в подграфах, соответствующих отдельным множествам достижимости.

Ключевые слова: кратный граф, делимый граф, множество достижимости, связность, кратный путь, кратчайший путь

Для цитирования: Смирнов А. В., "Задача о кратчайшем пути в кратном графе", *Моделирование и анализ информационных систем*, **24**:6 (2017), 788–801.

Об авторах:
Смирнов Александр Валерьевич, orcid.org/0000-0002-0980-2507, канд. физ.-мат. наук, доцент,
Ярославский государственный университет им. П.Г. Демидова,
ул. Советская, 14, г. Ярославль, 150003 Россия, e-mail: alexander_sm@mail.ru

Благодарности:
[1] Работа выполнена при поддержке Российского фонда фундаментальных исследований (проект № 17-07-00823 А).

Введение

В данной статье мы введем понятие *кратного графа* кратности $k > 1$ и рассмотрим задачу о кратчайшем пути в нем. Кратные графы содержат три типа ребер (обычные, кратные и мультиребра) и являются обобщением обычных графов – по сути, обычный граф имеет кратность $k = 1$.

Среди других известных обобщений графов стоит выделить мультиграфы, гиперграфы (см., например, [1] – [2]), а также метаграфы (см. [3] – [4]), как наиболее близкие нам концепции. Действительно, как и в мультиграфах, в кратных графах допускается наличие нескольких ребер между парой вершин (набор таких ребер мы будем в дальнейшем называть *кратным ребром*), однако в случае кратного графа количество таких ребер должно быть строго равным k. В кратных графах присутствуют *мультиребра*, соединяющие между собой $k + 1$ вершину. Но в отличие от гиперребер гиперграфа, мультиребро представляется в виде k связанных ребер, имеющих один общий конец, причем все эти k ребер должны использоваться согласованно. По сути, понятие мультиребра близко понятию ребра между вершиной и метавершиной в метаграфе. При этом в метаграфе, напомним, метапуть между двумя метавершинами фактически моделирует причинно-следственные связи в некоторой предметной области. Однако в кратном графе используется принципиально иной подход к определению пути: *кратный путь* должен состоять ровно из k обычных путей, проходящих по обычным ребрам, а также по связанным ребрам кратных и мультиребер; при этом пути должны быть согласованы (одинаковы) на кратных и мультиребрах. Поэтому кратный граф нельзя считать частным случаем метаграфа.

Ранее (см. [5] – [6]) мы разработали обобщение теории потоков Форда–Фалкерсона (см. [7]), названное кратными сетями и кратными потоками. В статьях [8] – [10] кратные сети и потоки используются для поиска решения задачи целочисленного сбалансирования трех- и четырехмерной матрицы. Задача целочисленного сбалансирования является частным случаем многоиндексных задач целочисленного линейного программирования (см., например, [11] – [13]), при этом она NP-полна (см. [14]). В работе [8] показано, что сведение задачи целочисленного сбалансирования трехмерной матрицы к поиску наибольшего кратного потока в сети кратности 2 в большинстве случаев оказывается более эффективным, нежели применение стандартных методов решения задач целочисленного линейного программирования. Заметим, что задача целочисленного сбалансирования имеет ряд приложений в сфере экономики, управления, финансов.

Следует отметить, что кратные сети являются частным случаем кратных графов, рассматриваемых в настоящей статье, а понятие кратного пути в кратном графе близко понятию обобщенного пути прорыва, на построении которого основаны алгоритмы поиска наибольшего потока в кратной сети.

1. Кратный граф

В [5] – [6] было введено понятие *кратной сети* и рассмотрена задача о наибольшем кратном потоке в такой сети. Так же, как обычная сеть является частным случаем

обычного графа, кратная сеть является частным случаем кратного графа. Введем соответствующие определения.

Определение 1. *В качестве* кратного графа *произвольной натуральной кратности $k > 1$ рассматривается граф $G(X, E)$, между вершинами которого могут быть ребра одного из 3 видов:*

1) обычное ребро *e^o; множество обычных ребер обозначим через E^o;*

2) кратное ребро *e^k между двумя вершинами, которое состоит из k одинаковых связанных ребер; связанные ребра кратного ребра могут использоваться только согласованно; множество кратных ребер обозначим через E^k;*

3) связанное ребро *e между двумя вершинами, имеющее один общий конец с другими $k - 1$ ребрами (у любых двух из k связанных ребер только один конец является общим); множество связанных общей вершиной ребер будем называть* мультиребром *e^m; связанные ребра мультиребра могут использоваться только согласованно; множество мультиребер обозначим через E^m.*

Если вершина инцидентна какому-либо кратному ребру, то она может быть инцидентна другим кратным ребрам, а также она может быть общим концом какого-либо мультиребра.

Если вершина является общим концом какого-либо мультиребра, то она не может быть общим концом никакого другого мультиребра.

Определение 2. *Кратный граф является* ориентированным*, если хотя бы на одном обычном, кратном или мультиребре задано направление. При этом ориентация всех связанных ребер кратного или мультиребра совпадает.*

В данной работе мы ограничимся рассмотрением неориентированных кратных графов.

Определение 3. Петлей *в кратном графе является обычное или кратное ребро вида $\{x, x\}$, где $x \in X$.*

Отметим, что петля $\{x, x\}$ в кратном графе обязательно является кратным ребром, если вершина x инцидентна какому-либо кратному ребру или является общим концом мультиребра. В противном случае петля $\{x, x\}$ обязательно является обычным ребром.

Определение 4. *Для любой вершины $x \in X$ определена ее* степень $\deg x$ *– количество обычных или связанных ребер, инцидентных x.*

Таким образом, каждое обычное ребро вида $\{x, y\}$ добавляет 1 к $\deg x$, каждое кратное ребро вида $\{x, y\}$ добавляет k к $\deg x$, а каждое мультиребро вида $\{x, \{y_1, \ldots, y_k\}\}$ добавляет k к $\deg x$ и 1 к каждому из $\deg y_i$.

Определение 5. *Вершина является* висячей*, если она инцидентна только одному ребру.*

Следует отметить, что степень висячей вершины равна 1, если она инцидентна обычному ребру либо является концом одного связанного ребра какого-то мультиребра. Если же висячая вершина инцидентна кратному ребру либо является общим концом мультиребра, то ее степень будет равна k.

Теорема 1. $\sum\limits_{x \in X} \deg x = 2\left(|E^o| + k\left|E^k\right| + k\left|E^m\right|\right).$

Справедливость теоремы 1 следует непосредственно из определения степени вершины в кратном графе.

Следствие 1. *Количество вершин нечетной степени четно.*

Определение 6. Делимым кратным графом *назовем такой граф, для каждого мультиребра которого не существует пути из конца одного связанного ребра в конец другого связанного ребра того же мультиребра, проходящего только по обычным ребрам.*

Очевидно, что при удалении всех мультиребер делимый граф распадется на n компонент связности (связность здесь понимается в том же смысле, что и для обычных графов), каждая из которых содержит только кратные ребра либо только обычные ребра. При этом связанные ребра каждого мультиребра можно пронумеровать от 1 до k таким образом, что каждой компоненте связности, содержащей только обычные ребра, будут инцидентны связанные ребра мультиребер с одинаковыми номерами.

Определение 7. Частью G_i *($i \in \overline{1,k}$) делимого графа $G(X,E)$ назовем подграф, содержащий связанные ребра с номером i всех кратных и мультиребер, а также компоненты связности, состоящие из обычных ребер и инцидентные i-м связанным ребрам всех мультиребер.*

Очевидно, что каждая часть G_i является обычным графом. При этом возможность выделения частей G_i является особенностью делимых графов. В общем случае получить части G_i не удастся.

2. Кратный путь и задача о наименьшем кратном пути

Дадим теперь определение кратного пути. Основное отличие кратного пути от пути в обычном графе состоит в том, что связанные ребра каждого кратного и мультиребра должны проходиться в этом пути согласованно.

Определение 8. $S(x,y) = \cup_{i=1}^{k} S^i(x,y)$ *является* кратным путем из вершины x в вершину y *в графе $G(X,E)$, если выполнены следующие условия:*

1) $S^i(x,y) = \left(\{x, v_1^i\}, \{v_1^i, v_2^i\}, \ldots, \{v_{p-1}^i, v_p^i\}, \{v_p^i, y\}\right)$, где $p \geq 0$, – последовательность ребер, представляющая собой обычный (некратный) путь из x в y, где каждое ребро $\{a,b\}$ является либо обычным ребром в графе $G(X,E)$, либо i-м связанным ребром кратного или мультиребра. Если в путь $S(x,y)$ не входит ни одного кратного или мультиребра, то $S^2(x,y) = S^3(x,y) = \ldots = S^k(x,y) = \varnothing$;

2) вершины, не являющиеся общим концом связанных ребер мультиребра и не инцидентные кратным ребрам, могут встречаться в $S^i(x,y)$ несколько раз, то есть $S^i(x,y)$ может содержать циклы;

3) вершины, инцидентные кратным ребрам либо являющиеся общим концом мультиребра, не могут встретиться в $S^i(x,y)$ дважды;

4) любое обычное ребро может встречаться в $S^i(x,y)$ *несколько раз, причем направления, в которых оно проходится в разных вхождениях, могут не совпадать;*

5) обычное ребро, входящее в $S^i(x,y)$*, может также входить в любой* $S^j(x,y)$*,* $j \neq i$*;*

6) все пути $S^i(x,y)$ *согласованы (одинаковы) на общей части. Это условие означает, что если связанное ребро какого-то кратного или мультиребра входит в некоторый путь* $S^i(x,y)$*, то остальные связанные ребра должны входить во все* $S^j(x,y)$*,* $j \neq i$ *(по одному связанному ребру в каждый* $S^j(x,y)$*). При этом порядок вхождения всех кратных и мультиребер во все* $S^i(x,y)$ *одинаков;*

7) если $S(x,y)$ *содержит мультиребро* $\{x_0, \{x_1, \ldots, x_k\}\}$*, проходимое в направлении от общего конца, то он не может содержать никакого другого мультиребра* $\{y_0, \{x_1, \ldots, x_k\}\}$*, проходимого в том же направлении. Аналогичное условие должно выполняться и в случае движения к общему концу.*

Определение 9. *Кратный путь* $S(x,y)$ *является* кратным циклом*, если* $x = y$ *и* $S(x,y) \neq \varnothing$*.*

Условие 6 в определении 8 фактически является требованием синхронности использования связанных ребер (при движении по кратному пути соответствующие связанные ребра проходятся параллельно). Такой подход к определению кратного пути является наиболее естественным. При этом условия 2 и 4 допускают наличие циклических участков в частях пути $S^i(x,y)$, совокупность которых, однако, не образует кратного цикла. Возможность добавления таких циклов в путь вполне оправдана даже для делимых кратных графов, что можно проиллюстрировать следующим примером.

Пример 1. Пусть кратность $k = 2$ и делимый граф имеет структуру, представленную на рис. 1 (жирными линиями обозначены кратные ребра, раздваивающимися линиями обозначены мультиребра).

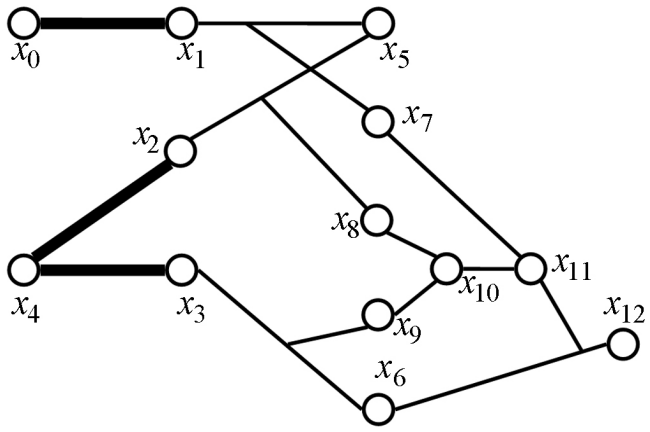

Рис. 1. Делимый граф кратности 2
Fig. 1. Divisible graph of multiplicity 2

Нетрудно убедиться, что путь $S(x_0, x_{12})$ в этом графе можно построить единственным образом (связанные ребра кратных ребер выделены однократным под-

черкиванием, мультиребер – двойным подчеркиванием):

$$S^1(x_0, x_{12}) = \Big(\underline{\{x_0, x_1\}}, \underline{\underline{\{x_1, x_5\}}}, \underline{\underline{\{x_5, x_2\}}}, \underline{\{x_2, x_4\}}, \underline{\{x_4, x_3\}}, \underline{\underline{\{x_3, x_6\}}}, \underline{\underline{\{x_6, x_{12}\}}}\Big);$$

$$S^2(x_0, x_{12}) = \Big(\underline{\{x_0, x_1\}}, \underline{\underline{\{x_1, x_7\}}}, \{x_7, x_{11}\}, \{x_{11}, x_{10}\}, \{x_{10}, x_8\}, \underline{\underline{\{x_8, x_2\}}}, \underline{\{x_2, x_4\}}, \underline{\{x_4, x_3\}}, \underline{\underline{\{x_3, x_9\}}}, \{x_9, x_{10}\}, \{x_{10}, x_{11}\}, \underline{\underline{\{x_{11}, x_{12}\}}}\Big).$$

При этом участок пути $(\{x_{10}, x_8\}, \{x_8, x_2\}, \{x_2, x_4\}, \{x_4, x_3\}, \{x_3, x_9\}, \{x_9, x_{10}\})$ в части $S^2(x_0, x_{12})$ является циклическим. Более того, ребро $\{x_{10}, x_{11}\}$ проходится в этой части дважды (в противоположных направлениях).

Отметим также, что совокупность условий 2–5 и 7 из определения 8 ограничивает наличие кратных циклов внутри пути $S(x, y)$: циклическим (в кратном смысле) участком может быть только участок пути, начинающийся и заканчивающийся в вершине z, инцидентной обычным ребрам. Если дополнительно потребовать минимальность пути, такой участок очевидно будет отброшен. Запрет всех вариантов кратного цикла внутри пути (за исключением отмеченного) также вполне оправдан, так как возможность добавления кратных циклов в путь существенно увеличит количество операций, необходимых для отыскания этого пути.

Кроме того, заметим, что для делимых графов условие 5 из определения 8 не нужно, так как каждый из путей $S^i(x, y)$ будет проходить по своей части графа G_i, при этом части G_i не имеют общих обычных ребер. По этой же причине в пути в делимом графе не может возникнуть никакого кратного цикла.

Теперь мы можем ввести понятие связности для кратного графа.

Определение 10. *Кратный граф $G(X, E)$ является* связным, *если одновременно выполнены два условия:*

1) для любых двух вершин $x \in X$, $y \in X$, каждая из которых либо инцидентна кратному ребру, либо является общим концом мультиребра, существует кратный путь $S(x, y)$;

2) невозможно выделить такой подграф $G' \subset G$, который будет содержать только обычные ребра, и при этом подграфы G' и $G \setminus G'$ не будут соединены ни одним ребром (обычным ребром или связанным ребром мультиребра).

В отличие от обычных графов связность кратного графа не предполагает наличие кратных путей из каждой вершины в каждую. Фактически в связном кратном графе между каждой парой вершин должен существовать обычный (некратный) путь, использующий связанные ребра кратных и мультиребер несогласованно, а кратные пути обязательно должны существовать только для пар вершин, каждая из которых инцидентна кратным ребрам или является общим концом мультиребра. Рассмотрим пример.

Пример 2. Пусть граф кратности $k = 2$ состоит из 4 вершин и 2 мультиребер (рис. 2).

Граф очевидно является связным, но при этом в нем не существует путей $S(x_1, x_2)$, $S(x_1, x_3)$, $S(x_4, x_2)$ и $S(x_4, x_3)$, поскольку их невозможно построить согласованно относительно связанных ребер.

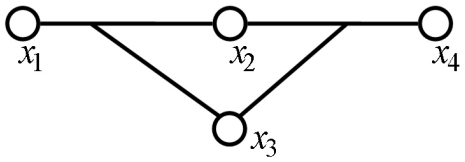

Рис. 2. Граф кратности 2
Fig. 2. Graph of multiplicity 2

Алгоритм проверки связности кратного графа будет рассмотрен в разделе 3.

Отметим, что для делимого кратного графа определение 10 может быть переписано в более простой форме, что обусловлено структурой графа.

Определение 11. *Делимый кратный граф $G(X, E)$ является* связным, *если одновременно выполнены два условия:*

1) для любых двух вершин $x \in X$, $y \in X$, каждая из которых либо инцидентна кратному ребру, либо является общим концом мультиребра, существует кратный путь $S(x, y)$;

2) каждая из частей G_i является связным графом.

Поставим задачу о наименьшем кратном пути. Сначала определим длину ребра.

Определение 12. *Целочисленная функция $l(e)$, определенная для всех ребер $e \in E$, является* длиной (весом) ребра *в кратном графе $G(X, E)$, если выполнено следующее:*

1) $l(e) > 0$ для любого ребра e;

2) если e является кратным или мультиребром, то $l(e_1) = l(e_2) = \ldots = l(e_k)$ и $l(e) = k \cdot l(e_1)$, где $e_1, \ldots, e_k$ – это связанные ребра данного ребра e.

Тогда *длина пути $S(x, y)$* будет определяться по формуле

$$l(S(x,y)) = \sum_{i=1}^{k} l(S^i(x,y)) = \sum_{i=1}^{k} \sum_{e_j \in S^i(x,y)} l(e_j),$$

при этом может оказаться $e_j = e_p$ $(j \neq p)$, то есть в сумме учитывается каждое повторное вхождение обычного ребра в $S^i(x, y)$.

Задача 1. *В кратном графе $G(X, E)$ требуется найти* кратчайший путь из вершины x в вершину y, *то есть такой путь $S_{\min}(x, y)$, что для любого пути $S(x, y)$ выполнено*

$$l(S_{\min}(x,y)) \leq l(S(x,y)).$$

Условия существования кратного пути и алгоритм решения задачи 1 будут рассмотрены в разделе 4.

3. Множества достижимости и связность кратного графа

В данном разделе мы получим полиномиальный алгоритм для проверки связности кратного графа.

Определение 13. Множеством достижимости по кратным ребрам *для некоторой вершины x назовем множество R_x^k всех вершин y таких, что существует путь из x в y, проходящий только по кратным ребрам.*

Определение 14. Множеством достижимости по обычным ребрам *для некоторой вершины x назовем множество R_x^o всех вершин y таких, что существует путь из x в y, проходящий только по обычным ребрам.*

Очевидно, что $x \in R_x^k$, $x \in R_x^o$. Если $y \in R_x^k$, то $R_y^k = R_x^k$. Если $y \in R_x^o$, то $R_y^o = R_x^o$.

Мы будем определять множества R_x^k только для вершин x, инцидентных кратным ребрам либо являющихся общим концом мультиребра, а множества R_x^o мы будем определять только для вершин x, инцидентных обычным ребрам либо являющихся концом одного связанного ребра какого-то мультиребра.

Определение 15. *Множества достижимости R_x^k и R_y^k являются* смежными, *если для произвольных вершин $a \in R_x^k$, $b \in R_y^k$ существует соединяющий их кратный путь $S(a, b)$.*

Очевидно, что отношение смежности является *рефлексивным*, *симметричным* и *транзитивным*.

Мы будем использовать множества достижимости и свойство смежности этих множеств для проверки связности кратного графа. Нетрудно убедиться, что будет справедлив следующий критерий связности.

Теорема 2. *Кратный граф является связным тогда и только тогда, когда все R_x^k являются попарно смежными и в каждом из R_y^o найдется вершина, инцидентная одному связанному ребру какого-либо мультиребра.*

Алгоритм 1 (*построение множества достижимости по кратным ребрам для вершины x*).

1. Все вершины графа являются непросмотренными. $R_x^k = \{x\}$.

2. Берем произвольную непросмотренную вершину $z \in R_x^k$. Если таких вершин нет, множество R_x^k сформировано, выход.

3. Для каждого ребра $\{z, y\} \in E^k$ включаем вершину y в R_x^k, если она является непросмотренной.

4. Вершина z становится просмотренной. Переходим на шаг 2.

Алгоритм 2 (*построение множества достижимости по обычным ребрам для вершины x*).

Аналогично алгоритму 1, но рассматриваем ребра из E^o.

Алгоритм 3 (*поиск всех пар смежных множеств достижимости по кратным ребрам*).

1. Выбираем из множества E^m пару мультиребер $\{a, \{a_1, \ldots, a_k\}\}$ и $\{b, \{b_1, \ldots, b_k\}\}$.

2. Если для всех $i \in \overline{1,k}$ выполнено $a_i \in R^o_{b_i}$, то включаем $\{R^k_a, R^k_b\}$ в список пар смежных множеств достижимости. Если остались непросмотренные пары мультиребер, переходим на шаг 1.

3. Пользуясь свойством транзитивности отношения смежности, дополняем список пар смежных множеств достижимости, пока это возможно.

Алгоритм 4 (*проверка связности кратного графа*).

1. С помощью алгоритма 2 ищем все R^o_y. Если хотя бы одно из R^o_y не содержит вершины, инцидентной связанному ребру какого-то мультиребра, то граф несвязен, выход.

2. С помощью алгоритма 1 ищем все R^k_x. После этого применяем алгоритм 3 для определения пар смежных множеств достижимости. Если хотя бы одна пара $\{R^k_a, R^k_b\}$ не вошла в этот список, то граф несвязен, выход.

3. Граф связен.

Отметим, что все приведенные в данном разделе алгоритмы выполняются за полиномиальное время.

4. Алгоритм поиска кратчайшего пути в кратном графе

Прежде чем сформулировать алгоритм поиска минимального кратного пути, определим условия, при которых этот путь существует.

Теорема 3. *В кратном графе $G(X, E)$ существует путь из вершины $x \in X$ в вершину $y \in X$ тогда и только тогда, когда выполнено одно из условий:*

1) $x \in R^o_y$;

2) $x \in R^k_y$;

3) каждая из вершин x и y инцидентна кратным ребрам либо является общим концом связанных ребер мультиребра, и при этом $R^k_x \neq R^k_y$, но R^k_x и R^k_y смежны;

4) x и y не инцидентны кратным ребрам и не являются общим концом связанных ребер мультиребра, при этом $R^o_x \neq R^o_y$, но в E^m есть два мультиребра $\{a, \{a_1, \ldots, a_k\}\}$ и $\{b, \{b_1, \ldots, b_k\}\}$ такие, что для всех $i \in \overline{1,k}$ выполнено $a_i \in R^o_x$ и $b_i \in R^o_y$, а R^k_a и R^k_b смежны;

5) x инцидентен кратным ребрам либо является общим концом связанных ребер мультиребра, а y нет, но при этом в E^m есть мультиребро $\{a, \{a_1, \ldots, a_k\}\}$ такое, что для всех $i \in \overline{1,k}$ выполнено $a_i \in R^o_y$, а R^k_a и R^k_x смежны;

6) y инцидентен кратным ребрам либо является общим концом связанных ребер мультиребра, а x нет, но при этом в E^m есть мультиребро $\{a, \{a_1, \ldots, a_k\}\}$ такое, что для всех $i \in \overline{1,k}$ выполнено $a_i \in R^o_x$, а R^k_a и R^k_y смежны.

Справедливость данной теоремы вытекает непосредственно из свойств множеств достижимости и отношения смежности.

Следует отметить, что для любых двух вершин кратного графа выполнение условия теоремы можно проверить за полиномиальное время, используя алгоритмы 1–3 из предыдущего раздела. При этом для делимых кратных графов достаточно проверять только первые три условия, поскольку структура таких графов делает невозможным выполнение оставшихся условий.

Сформулируем теперь алгоритм решения задачи 1. На отдельных шагах этого алгоритма для нахождения минимальных участков пути без мультиребер мы будем использовать известный алгоритм Дейкстры (см. [15]).

В алгоритме мы будем использовать следующие структуры данных:

– множества достижимости R_z^k и R_z^o;

– минимальный на текущий момент путь S_0 длины $L_{\min}$;

– множество путей-кандидатов C (в это множество будут добавляться строящиеся пути S_i, которые могут в итоге оказаться минимальными; в конце выполнения алгоритма это множество окажется пустым);

– M_i – множество запрещенных вершин для строящегося пути S_i (согласно определению кратного пути вершины, инцидентные кратным ребрам либо являющиеся общим концом мультиребра, не могут использоваться дважды).

Алгоритм 5 (*поиск минимального кратного пути из x в y*).

0. С помощью алгоритмов 1–3 проверяем существование пути из x в y. Если условие выполнено, устанавливаем $L_{\min} = \infty$ и переходим на шаг 1, иначе выход.

1. Если $x \in R_y^o$, то с помощью алгоритма Дейкстры находим кратчайший путь $S(x, y)$, проходящий только по вершинам из множества R_y^o (этот путь будет содержать только обычные ребра).

Запоминаем $S(x, y)$ как путь S_0 и устанавливаем $L_{\min} = l(S(x, y))$.

Если при этом граф является делимым, найденный путь S_0 является минимальным, выход.

2. Если $x \in R_y^k$, то с помощью алгоритма Дейкстры находим кратчайший путь $S(x, y)$, проходящий только по вершинам из множества R_y^k (этот путь будет содержать только кратные ребра).

Запоминаем $S(x, y)$ как путь S_0 и устанавливаем $L_{\min} = l(S(x, y))$.

3. Если x инцидентен кратным ребрам или является общим концом мультиребра, переходим на шаг 4.

Иначе для каждого мультиребра вида $e_b^m = \{\{b_1, \ldots, b_k\}, b\}$, где все $b_p \in R_x^o$ ($p \in \overline{1, k}$), находим с помощью алгоритма Дейкстры кратчайшие пути $S(x, b_p)$, проходящие только по вершинам из множества R_x^o. Если при этом

$$\sum_{p=1}^{k} l(S(x, b_p)) + l(e_b^m) < L_{\min},$$

то участок кратного пути $S_i = S(x, b_1) \cup \ldots \cup S(x, b_p) \cup e_b^m$ добавляем в множество путей кандидатов C (i – это некоторый уникальный в пределах множества C идентификатор).

Связываем с S_i множество запрещенных вершин $M_i = \varnothing$.

Переходим на шаг 5.

4. Для каждой вершины $a \in R_x^k$, являющейся началом какого-либо мультиребра, находим с помощью алгоритма Дейкстры кратчайший путь $S(x, a)$, проходящий только по вершинам из множества R_x^k (если $a = x$, этот путь будет иметь нулевую длину). Если

$$l(S(x, a)) + l(e_a^m) + k \leq L_{\min},$$

где e_a^m – мультиребро с началом в a, то участок кратного пути $S_i = S(x, a) \cup e_a^m$

добавляем в множество путей-кандидатов C (i – уникальный в пределах множества C идентификатор).

Связываем с S_i множество запрещенных вершин M_i, содержащее все вершины пути $S(x, a)$.

5. Если множество C пусто, то запомненный путь S_0 длины $L_{\min}$ является минимальным, выход.

6. Извлекаем произвольный участок пути S_i из множества C. Если последняя вершина S_i инцидентна кратному ребру или является общим концом мультиребра, переходим на шаг 7.

Иначе переходим на шаг 9.

7. Пусть a – последняя вершина пути S_i. Рассмотрим множество $R_a^k \setminus M_i$.

Если $y \in R_a^k \setminus M_i$, то с помощью алгоритма Дейкстры ищем кратчайший путь $S(a, y)$, проходящий только по вершинам из множества $R_a^k \setminus M_i$. Если такой путь существует и

$$l(S_i) + l(S(a, y)) < L_{\min},$$

то запоминаем кратный путь $S_i \cup S(a, y)$ как S_0 и устанавливаем $L_{\min} = l(S_i) + l(S(a, y))$.

8. Рассмотрим все вершины $b \in R_a^k \setminus M_i$, каждая из которых является началом мультиребра $e_b^m = \{b, \{b_1, \ldots, b_k\}\}$, если при этом в путь S_i ранее не было включено мультиребро $\{c, \{b_1, \ldots, b_k\}\}$, проходимое в том же направлении (от общего конца). Будем с помощью алгоритма Дейкстры искать кратчайший путь $S(a, b)$, проходящий только по вершинам из множества $R_a^k \setminus M_i$. Если такой путь существует и

$$l(S_i) + l(S(a, b)) + l(e_b^m) + k \leq L_{\min},$$

то добавляем в C участок кратного пути $S_j = S_i \cup S(a, b) \cup e_b^m$ (j – уникальный в пределах множества C идентификатор).

Множество M_j состоит из всех вершин множества M_i и всех вершин пути $S(a, b)$.

Переходим на шаг 5.

9. Последними вершинами S_i являются вершины $\{a_1, \ldots, a_k\}$ – k концов некоторого мультиребра.

Если все $a_p \in R_y^o$ ($p \in \overline{1, k}$) (для делимого графа это условие заведомо не будет выполнено, так как в делимом графе $R_{a_q}^o \cap R_{a_s}^o = \varnothing$, если $q \neq s$), то с помощью алгоритма Дейкстры находим кратчайшие пути $S(a_p, y)$, проходящие только по вершинам из множества R_y^o. Если

$$l(S_i) + \sum_{p=1}^{k} l(S(a_p, y)) < L_{\min},$$

то запоминаем получившийся кратный путь $S_i \cup S(a_1, y) \cup \ldots \cup S(a_k, y)$ как S_0 и устанавливаем $L_{\min} = l(S_i) + \sum\limits_{p=1}^{k} l(S(a_p, y))$.

10. Рассмотрим все мультиребра вида $e_b^m = \{\{b_1, \ldots, b_k\}, b\}$ такие, что $b_p \in R_{a_p}^o$ ($p \in \overline{1, k}$) и в путь S_i не было ранее включено мультиребро $\{\{b_1, \ldots, b_k\}, c\}$, проходимое в том же направлении (к общему концу). При этом мультиребра, отличающиеся только порядком следования вершин b_p в перечислении концов, мы считаем одинаковыми и повторно не рассматриваем (эта ситуация возникает, когда

граф не является делимым и $R^o_{b_q} = R^o_{b_s}$, $q \neq s$; тогда одно и то же мультиребро e^m_b можно рассматривать как в виде $\{\{b_1, \ldots, b_q, \ldots, b_s, \ldots, b_k\}, b\}$, так и в виде $\{\{b_1, \ldots, b_s, \ldots, b_q, \ldots, b_k\}, b\}$).

Для каждого мультиребра e^m_b указанного вида найдем с помощью алгоритма Дейкстры кратчайшие пути $S(a_p, b_p)$, проходящие только по вершинам соответствующих множеств $R^o_{a_p}$ ($p \in \overline{1,k}$). Если

$$l(S_i) + \sum_{p=1}^{k} l(S(a_p, b_p)) + l(e^m_b) < L_{\min},$$

то добавляем в C участок кратного пути $S_j = S_i \cup S(a_1, b_1) \cup \ldots \cup S(a_k, b_k) \cup e^m_b$ (j – уникальный в пределах множества C идентификатор).

Устанавливаем $M_j = M_i$.

Переходим на шаг 5.

Отметим, что на шагах 4 и 8 алгоритма 4 строится участок пути, оканчивающийся k различными вершинами-концами мультиребра. Поэтому для достижения вершины y нам заведомо потребуется увеличить длину этого пути не менее чем на $k-1$ для произвольного кратного графа (если y – один из концов мультиребра, а остальные $k-1$ концов мультиребра соединены с y обычными ребрами длины 1) и не менее чем на k для делимого графа (мультиребро минимальной длины). Соответственно, при проверке условия на шагах 4, 8 появляется слагаемое k. Для делимого графа указанное условие можно представить как строгое неравенство.

Каждый из шагов алгоритма 4 полиномиален, однако шаги могут повторяться экспоненциальное количество раз. В худшем случае алгоритм переберет все возможные варианты пути из x в y. Однако следует отметить, что задача 1 скорее всего является NP-полной. Это можно предположить, исходя из того, что NP-полной является задача о нахождении максимального потока в кратной сети (см. [5] – [6]), решение которой получается с помощью последовательного нахождения и насыщения обобщенных путей прорыва, определение которых близко определению кратного пути.

Количество операций в алгоритме 4 можно существенно сократить, если изначально задать какой-то путь S_0 между x и y. Также при обновлении S_0 и значения $L_{\min}$ можно просматривать множество C и исключать оттуда пути S_i, длина которых стала больше $L_{\min}$, если последняя вершина S_i является общим концом мультиребра, или больше $L_{\min} - k + 1$ в противном случае.

Заключение

В данной статье мы ввели понятие кратного графа, который содержит обычные, кратные и мультиребра. При этом для кратных графов мы обобщили ряд понятий, используемых для обычных графов (степень вершины, связность, путь), поставили задачу о минимальном кратном пути между двумя вершинами.

Также мы построили полиномиальный алгоритм проверки связности кратного графа и экспоненциальный алгоритм поиска наименьшего кратного пути, который является обобщением алгоритма Дейкстры.

Список литературы / References

[1] Cormen T. H., Leiserson C. E., Rivest R. L., Stein C., *Introduction to Algorithms*, 3rd ed., The MIT Press, McGraw-Hill Book Company, 2009.

[2] Berge C., *Graphs and Hypergraphs*, North-Holland Publishing Company, 1973.

[3] Basu A., Blanning R. W., "Metagraphs in workflow support systems", *Decision Support Systems*, **25**:3 (1999), 199–208.

[4] Basu A., Blanning R. W., *Metagraphs and Their Applications*, Integrated Series in Information Systems, **15**, Springer US, 2007.

[5] Рублев В. С., Смирнов А. В., "Потоки в кратных сетях", *Ярославский педагогический вестник*, **3**:2 (2011), 60–68; [Rublev V. S., Smirnov A. V., "Flows in Multiple Networks", *Yaroslavsky Pedagogichesky Vestnik*, **3**:2 (2011), 60–68, (in Russian).]

[6] Smirnov A. V., "The Problem of Finding the Maximum Multiple Flow in the Divisible Network and its Special Cases", *Automatic Control and Computer Sciences*, **50**:7 (2016), 527–535.

[7] Ford L. R., Fulkerson D. R., *Flows in Networks*, Princeton University Press, 1962.

[8] Рублев В. С., Смирнов А. В., "Задача целочисленного сбалансирования трехмерной матрицы и алгоритмы ее решения", *Моделирование и анализ информационных систем*, **17**:2 (2010), 72–98; [Roublev V. S., Smirnov A. V., "The Problem of Integer-Valued Balancing of a Three-Dimensional Matrix and Algorithms of Its Solution", *Modeling and Analysis of Information Systems*, **17**:2 (2010), 72–98, (in Russian).]

[9] Smirnov A. V., "Some Solvability Classes for the Problem of Integer Balancing of a Three-Dimensional Matrix with Constraints of the Second Type", *Automatic Control and Computer Sciences*, **48**:7 (2014), 543–553.

[10] Смирнов А. В., "Сетевая модель для задачи целочисленного сбалансирования четырехмерной матрицы", *Моделирование и анализ информационных систем*, **23**:4 (2016), 466–478; [Smirnov A. V., "Network Model for The Problem of Integer Balancing of a Four-dimensional Matrix", *Modeling and Analysis of Information Systems*, **23**:4 (2016), 466–478, (in Russian).]

[11] Корбут А. А., Финкельштейн Ю. Ю., *Дискретное программирование*, Наука, М., 1969; [Korbut A. A., Finkelstein J. J., *Diskretnoe programmirovanie*, Nauka, Moscow, 1969, (in Russian).]

[12] Раскин Л. Г., Кириченко И. О., *Многоиндексные задачи линейного программирования*, Радио и связь, М., 1982; [Raskin L. G., Kirichenko I. O., *Mnogoindeksnye zadachi lineynogo programmirovaniya*, Radio i svyaz, Moscow, 1982, (in Russian).]

[13] Spieksma F. C. R., "Multi index assignment problems: complexity, approximation, applications", *Nonlinear Assignment Problems. Algorithms and Applications*, eds. P. M. Pardalos, L. S. Pitsoulis, Kluwer Academic Publishers, 2000, 1–11.

[14] Рублев В. С., Смирнов А. В., "NP-полнота задачи целочисленного сбалансирования трехмерной матрицы", *Доклады Академии Наук*, **435**:3 (2010), 314–316; English transl.: Roublev V. S., Smirnov A. V., "NP-Completeness of the Integer Balancing Problem for a Three-Dimensional Matrix", *Doklady Mathematics*, **82**:3 (2010), 912–914.

[15] Dijkstra E. W., "A Note on Two Problems in Connexion with Graphs", *Numerische Mathematik*, **1**:1 (1959), 269–271.

Smirnov A. V., "The Shortest Path Problem for a Multiple Graph", *Modeling and Analysis of Information Systems*, **24**:6 (2017), 788–801.

DOI: 10.18255/1818-1015-2017-6-788-801

Abstract. In the article, the definition of an undirected multiple graph of any natural multiplicity $k > 1$ is stated. There are edges of three types: ordinary edges, multiple edges and multi-edges. Each edge of the last two types is the union of k linked edges, which connect 2 or $k+1$ vertices, correspondingly. The linked edges should be used simultaneously. If a vertex is incident to a multiple edge, it can be also incident to other multiple edges and it can be the common ending vertex to k linked edges of some multi-edge. If a vertex is the common end of some multi-edge, it cannot be the common end of any other multi-edge. Also, a class of the divisible multiple graphs is considered. The main peculiarity of them is a possibility to divide the graph into k parts, which are adjusted on the linked edges and which have no common edges. Each part is an ordinary graph. The following terms are generalized: the degree of a vertex, the connectedness of a graph, the path, the cycle, the weight of an edge, and the path length. There is stated the definition of the reachability set for the ordinary and multiple edges. The adjacency property is defined for a pair of reachability sets. It is shown, that we can check the connectedness of some multiple graph with the polynomial algorithm based on the search for the reachability sets and testing their adjacency. There is considered a criterion of the existence of a multiple path between two given vertices. The shortest multiple path problem is stated. Then we suggest an algorithm of finding the shortest path in a multiple graph. It uses Dijkstra's algorithm of finding the shortest paths in subgraphs, which correspond to different reachability sets.

Keywords: multiple graph, divisible graph, reachability set, connectedness, multiple path, shortest path

On the authors:
Alexander V. Smirnov, orcid.org/0000-0002-0980-2507, PhD, Associate Professor
P.G. Demidov Yaroslavl State University,
14 Sovetskaya str., Yaroslavl, 150003 Russia, e-mail: alexander_sm@mail.ru

Acknowledgments:
[1] This work was supported by the Russian Foundation for Basic Research under the Grant No 17-07-00823 A.

Моделирование и анализ информационных систем. Т. 24, №6 (2017), с. 802–810
Modeling and Analysis of Information Systems. Vol. 24, No 6 (2017), pp. 802–810

DOI: 10.18255/1818-1015-2017-6-802-810

УДК 62-503.54

Синтез управления и наблюдателя для слабо нелинейных систем на основе техники псевдолинеаризации

Макаров Д. А.[1]

получена 15 марта 2017

Аннотация. В работе для одного класса слабо нелинейных систем с зависящими от состояния коэффициентами рассматривается подход к построению нелинейного следящего управления по выходу на конечном интервале времени. Предложенный метод синтеза состоит из двух основных этапов. Сначала с помощью ранее предложенного автором алгоритма на основе уравнений Риккати, с зависящими от состояния коэффициентами (State Dependent Riccati Equation – SDRE), находится нелинейный регулятор по состоянию. На втором этапе ставится задача построения наблюдателя полного порядка, которая сводится к задаче дифференциальной игры. Вид её решения получен с помощью принципа гарантированного управления, позволяющего относительно заданного функционала найти наилучшие коэффициенты наблюдателя при наихудшей реализации неопределенностей. Однотипность полученных уравнений позволила для определения матрицы наблюдателя использовать алгоритм решения из первого этапа. Особенностями предложенного подхода являются отсутствие принципа разделения задач синтеза управления и наблюдения, который имеет место в линейных системах, поскольку матрица коэффициентов наблюдателя оказалась зависимой от матрицы коэффициентов обратной связи, и использование численно-аналитических процедур для определения этих матриц, что позволяет значительно снизить вычислительную сложность алгоритма управления.

Ключевые слова: задача слежения, нелинейное управление, уравнения Риккати с зависящими от состояния коэффициентами, гарантированное управление

Для цитирования: Макаров Д. А., "Синтез управления и наблюдателя для слабо нелинейных систем на основе техники псевдолинеаризации", *Моделирование и анализ информационных систем*, **24**:6 (2017), 802–810.

Об авторах:
Макаров Дмитрий Александрович, orcid.org/0000-0001-8930-1288, канд. физ.-мат. наук,
Общество с ограниченной ответственностью «Технологии системного анализа»,
Институт системного анализа Федерального исследовательского центра «Информатика и управление»
Российской академии наук, пр-т 60-летия Октября, 9, г. Москва, 117312 Россия, e-mail: makarov@isa.ru

Благодарности:
[1] Работа выполнена при финансовой поддержке РФФИ (проекты № 16-38-60198-мол_а_дк, № 17-07-00281-а).

Введение

В настоящее время для решения задач управления нелинейными системами применяется множество подходов. Одним из перспективных является техника, осно-

ванная на решении матричных уравнений Риккати для систем с коэффициентами (SDRE), зависящими от состояния (см. обзоры [1, 2]). Её суть состоит в представлении исходной системы в псевдолинейном виде, в котором матрицы системы и управления зависят от состояния. Закон управления имеет стандартный для линейно-квадратичной задачи вид, однако для его реализации в процессе управления необходимо в каждый момент времени численно находить решение соответствующего SDRE, что может наталкиваться на вычислительные ограничения. В работе [3] на основе SDRE-техники был предложен подход для решения задачи слежения на конечном интервале регулирования для слабо нелинейных полностью наблюдаемых систем. Поскольку отыскание точного решения в общем случае представляется достаточно трудоемкой с вычислительной точки зрения задачей, был предложен численно-аналитический алгоритм приближенного синтеза, существенно снижающий вычислительную сложность по сравнению с распространенной SDRE-техникой. Однако для применения подхода [3] необходимо построить соответствующий наблюдатель, позволяющий получить оценку неизвестного состояния системы. Эта задача сводится к задаче синтеза управления при неопределенности, для решения которой используется минимаксный принцип из [4–6].

1. Синтез регулятора

Рассмотрим управляемую нелинейную систему вида

$$\begin{gathered}\dot{x} = A(x,\mu)x + B(x,\mu)u,\ y = Cx,\ x(t_0) = x^0,\\ A(x,\mu) = A_0 + \mu A_1(x),\ B(x,\mu) = B_0 + \mu B_1(x),\\ x \in X \subset \mathbb{R}^n,\ y \in Y \subset \mathbb{R}^m,\ u \in \mathbb{R}^r,\ t \in [t_0, t_1]\,,\ 0 < \mu \le \mu_0,\end{gathered} \tag{1}$$

где x, y и u – векторы состояния, выхода и управления соответственно, μ_0 – некоторое заданное достаточно малое положительное число, A_0, B_0 и C – известные постоянные матрицы, $A_1(x) \in \mathbb{R}^{n\times n}$, $B_1(x) \in \mathbb{R}^{n\times r}$ – известные матрицы с достаточно гладкими и ограниченными по аргументу x элементами, X и Y – некоторые ограниченные множества, для любого непрерывного управления $u(t)$ траектории замкнутой системы (1) существуют, единственны и принадлежат X на $[t_0, t_1]$.

Пусть эталонное поведение системы (1) описывается решением дифференциального уравнения

$$\begin{gathered}\dot{x}_r = A_r(x_r,\mu)x_r,\ y_r = Cx_r,\ x_r(t_0) = x_r^0,\\ A_r(x_r,\mu) = A_{r,0} + A_{r,1}(x_r),\ x_r \in X,\ y_r \in Y,\ t \in [t_0, t_1],\end{gathered}$$

где x_r и y_r – желаемые (эталонные) траектория системы и ее выход, $A_{r,0}$ – известная постоянная матрица, а $A_{r,1}(x_r)$ – известная матрица с достаточно гладкими и ограниченными по аргументу x_r элементами. Начальные состояния x^0 и ${x_r}^0$ в общем случае полагаются неизвестными. Если ${x_r}^0$ известно, то вся эталонная траектория фактически заранее известна. Этому случаю может соответствовать, например, использование специального задающего устройства, начальное состояние которого может выбираться.

Определим следующий функционал качества

$$I(u) = \tfrac{1}{2}e^T(t_1)Fe(t_1) + \tfrac{1}{2}\int_{t_0}^{t_1}\left(e^TQ(y,y_r,\mu)e + u^TRu\right)dt \to \min_u,$$
$$Q(y,y_r,\mu) = Q_0 + \mu Q_1(y,y_r),\ e = y - y_r, \tag{2}$$

где заданные симметрические матрицы $Q(y,y_r,\mu) \geq 0,\ Q_0 > 0,\ R > 0,\ F > 0$ при $y, y_r \in Y,\ 0 < \mu \leq \mu_0$. Здесь и далее знаками >0 (≥ 0) обозначается положительная определенность (полуопределенность) соответствующей матрицы. Необходимо найти такое непрерывное управление по выходу $u(y,\ \mu,\ t)$, которое обеспечивает приближенное решение задачи (1)–(2).

Известно, что исходная задача (1)–(2) может быть представлена [7] в виде

$$\dot{\tilde{x}} = \tilde{A}(\tilde{x},\mu)\tilde{x} + \tilde{B}(x,\mu)u,\ \tilde{x}(t_0) = \tilde{x}^0,$$
$$\tilde{I}(u) = \tfrac{1}{2}\tilde{x}^T(t_1)\tilde{F}\tilde{x}(t_1) + \tfrac{1}{2}\int_{t_0}^{t_1}\left(\tilde{x}^T\tilde{Q}(\tilde{y},\mu)\tilde{x} + u^TRu\right)dt \to \min_u, \tag{3}$$

где $\tilde{x} = \begin{bmatrix} x \\ x_r \end{bmatrix} \in \mathbb{R}^{2n}$, $\tilde{y} = \begin{bmatrix} y \\ y_r \end{bmatrix} \in \mathbb{R}^{2m}$ – расширенные векторы состояния и выхода,

$$\tilde{A}(\tilde{x},\mu) = \begin{bmatrix} A(x,\mu) & 0 \\ 0 & A_r(x_r,\mu) \end{bmatrix} \in \mathbb{R}^{2n\times 2n},\ \tilde{B}(x,\mu) = \begin{bmatrix} B(x,\mu) \\ 0 \end{bmatrix} \in \mathbb{R}^{2n\times r},$$
$$\tilde{Q}(\tilde{y},\mu) = \begin{bmatrix} C^TQ(\tilde{y},\mu)C & -C^TQ(\tilde{y},\mu)C \\ -C^TQ(\tilde{y},\mu)C & C^TQ(\tilde{y},\mu)C \end{bmatrix} \in \mathbb{R}^{2n\times 2n} \geq 0,$$
$$\tilde{F} = \begin{bmatrix} C^TFC & -C^TFC \\ -C^TFC & C^TFC \end{bmatrix} \in \mathbb{R}^{2n\times 2n} \geq 0,$$

нулевые блоки в матрицах $\tilde{A}(\tilde{x},\mu)$ и $\tilde{B}(x,\mu)$ есть матрицы соответствующих размерностей.

Зададим представления

$$\tilde{A}(\tilde{x},\mu) = \tilde{A}_0 + \mu\tilde{A}_1(\tilde{x}),\ \tilde{B}(x,\mu) = \tilde{B}_0 + \mu\tilde{B}_1(x),\ \tilde{Q}(\tilde{y},\mu) = \tilde{Q}_0 + \mu\tilde{Q}_1(\tilde{y}),$$
$$\tilde{A}_0 = \begin{bmatrix} A_0 & 0 \\ 0 & A_{r,0} \end{bmatrix},\ \tilde{A}_1(\tilde{x}) = \begin{bmatrix} A_1(x) & 0 \\ 0 & A_{r,1}(x_r) \end{bmatrix},\ \tilde{B}_0 = \begin{bmatrix} B_0 \\ 0 \end{bmatrix},\ \tilde{B}_1(x) = \begin{bmatrix} B_1(x) \\ 0 \end{bmatrix},$$
$$\tilde{Q}_0 = \begin{bmatrix} C^TQ_0C & -C^TQ_0C \\ -C^TQ_0C & C^TQ_0C \end{bmatrix},\ \tilde{Q}_1(\tilde{y}) = \begin{bmatrix} C^TQ_1(\tilde{y})C & -C^TQ_1(\tilde{y})C \\ -C^TQ_1(\tilde{y})C & C^TQ_1(\tilde{y})C \end{bmatrix}.$$

Введем следующие условия:

I. Траектории замкнутой системы (1) существуют, единственны и принадлежат X на $[t_0,t_1]$ для любого непрерывного управления $u(t)$, где X – некоторое ограниченное множество пространства состояний; элементы матриц $\tilde{A}_1(\tilde{x})$, $\tilde{B}_1(x)$ ограниченные, непрерывные и достаточно гладкие при $x,\ x_r \in X$; $\mu \in (0,\ \mu_0]$.

II. Тройка матриц $\left\{\tilde{A}_0, \tilde{B}_0, H_P\right\}$, где $H_P^TH_P = \tilde{Q}_0$, стабилизируема и наблюдаема.

III. Матрицы системы $\tilde{A}_0, \tilde{A}_1(\tilde{x})$, $\tilde{B}_0, \tilde{B}_1(x)$ и симметрические матрицы критерия $R > 0,\ Q_0 \geq 0,\ Q_1(y) \geq 0,\ F > 0$, а также $\mu_0 > 0$ таковы, что $\tilde{P}_0 + \mu\tilde{P}_1(\tilde{x},t) > 0$ при $x,\ x_r \in X,\ t \in [t_0,t_1),\ \mu \in (0,\ \mu_0]$.

В работе [3] при условиях I–III для полностью наблюдаемой системы (вектор x точно известен) предложено управление, приближенно решающее задачу (3), в виде

$$u(\tilde{x},\mu,t) = -K(\tilde{x},\mu,t)\tilde{x} = u_0(\tilde{x}) + \mu u_1(\tilde{x},t,\mu), \tag{4}$$

где $K(\tilde{x},\mu,t) = R^{-1}(\tilde{B}_0 + \mu\tilde{B}_1(\tilde{x}))^T(\tilde{P}_0 + \mu\tilde{P}_1(\tilde{x},t))$, $u_0(\tilde{x}) = -R^{-1}\tilde{B}_0^T\tilde{P}_0\tilde{x}$ – линейная часть, а нелинейную коррекцию формирует $\mu u_1(\tilde{x},t,\mu) = -\mu R^{-1}\left(\tilde{B}_1^T(\tilde{x})\tilde{P}_0 + (\tilde{B}_0 + \mu\tilde{B}_1(\tilde{x}))^T\tilde{P}_1(\tilde{x},t)\right)\tilde{x}$. Согласно [3] при условиях I–III имеем следующий численно-аналитический алгоритм построения приближенного управления в задаче нелинейного слежения.

1. Находим $u_0(\tilde{x})$ как $u_0(\tilde{x}) = -R^{-1}\tilde{B}_0^T\tilde{P}_0\tilde{x}$, где $\tilde{P}_0$ – положительно определенное решение уравнения $\tilde{P}_0\tilde{A}_0 + \tilde{A}_0^T\tilde{P}_0 - \tilde{P}_0\tilde{B}_0R_0^{-1}\tilde{B}_0^T\tilde{P}_0 + \tilde{Q}_0 = 0$.

2. Находим $\tilde{P}_1(\tilde{x},\mu,t)$ с помощью

$$\tilde{P}_1(\tilde{x},\mu,t) = e^{\tilde{A}^T_{Pcl,0}(t_1-t)}M_P e^{\tilde{A}_{Pcl,0}(t_1-t)} + \int_0^\infty e^{\tilde{A}^T_{Pcl,0}\sigma}D_P(\tilde{x})e^{\tilde{A}_{Pcl,0}\sigma}d\sigma, \tag{5}$$

где $D_P(\tilde{x}) = \tilde{P}_0\left(\tilde{A}_1 - \tilde{B}_1R^{-1}\tilde{B}_0^T\tilde{P}_0\right) + \left(\tilde{A}_1 - \tilde{B}_1R^{-1}\tilde{B}_0^T\tilde{P}_0\right)^T\tilde{P}_0 + \tilde{Q}_1$, $\tilde{A}_{Pcl,0} = \tilde{A}_0 - \tilde{B}_0R^{-1}\tilde{B}_0^T\tilde{P}_0$, а матрица M_P находится как

$$M_P = \frac{1}{\mu}(\tilde{F} - \tilde{P}_0) - \int_0^\infty e^{\tilde{A}^T_{cl,0}\sigma}\, D_P\left(\tilde{x}(t_1)\right)\big|_{x(t_1)=x(t)}\, e^{\tilde{A}_{cl,0}\sigma}d\sigma.$$

3. Определяем итоговое управление (4).

Сделаем несколько замечаний. Поскольку будущая траектория системы является неизвестной, в предложенном алгоритме матрица $D_P\left(\tilde{x}(t_1)\right)$ вычисляется в предположении, что $\tilde{x}(t)$ и $\tilde{x}(t_1)$ слабо отличаются вблизи t_1, т.е. вместо $\tilde{x}(t_1)$ используется $\tilde{x}(t)$. Если эталонная траектория известна заранее, то можно, предполагая близость $x(t_1)$ к $x_r(t_1)$, вычислить D_P, используя $x(t_1) = x_r(t_1)$. «Жесткость» этих двух предположений смягчается тем, что первый член в (5) в силу $Re\lambda\left(\tilde{A}_{Pcl,0}\right) < 0$ существенен лишь в окрестности t_1. Отметим, что алгоритм предлагает аналитическую формулу для неизвестной матрицы $\tilde{P}_1(\tilde{x},t)$, зависящей от состояния системы, что существенно снижает вычислительную сложность алгоритма управления.

Кроме того, отметим, что имеет место выражение

$$\tilde{P}(\tilde{x},t,\mu) = \tilde{P}_0 + \mu\tilde{P}_1(\tilde{x},t,\mu) =$$
$$e^{\tilde{A}^T_{Pcl,0}(t_1-t)}\tilde{F}e^{\tilde{A}_{Pcl,0}(t_1-t)} + \tilde{P}_0 - e^{\tilde{A}^T_{Pcl,0}(t_1-t)}\tilde{P}_0e^{\tilde{A}_{Pcl,0}(t_1-t)} +$$
$$\mu\left(\int_0^\infty e^{\tilde{A}^T_{Pcl,0}\sigma}D(\tilde{x}(t))e^{\tilde{A}_{Pcl,0}\sigma}d\sigma - e^{\tilde{A}^T_{Pcl,0}(t_1-t)}\int_0^\infty e^{\tilde{A}^T_{Pcl,0}\sigma}D\left(\tilde{x}(t)\right)e^{\tilde{A}_{Pcl,0}\sigma}d\sigma e^{\tilde{A}_{Pcl,0}(t_1-t)}\right).$$

Таким образом, поскольку $e^{\tilde{A}^T_{Pcl,0}(t_1-t)}\tilde{F}e^{\tilde{A}_{Pcl,0}(t_1-t)} + \tilde{P}_0 - e^{\tilde{A}^T_{Pcl,0}(t_1-t)}\tilde{P}_0e^{\tilde{A}_{Pcl,0}(t_1-t)} > 0$ при $t \in [t_0,t_1)$, то условие III будет всегда выполнено при достаточно малом μ_0. Кроме того, для выполнения III достаточно, чтобы $D\left(\tilde{x}(t)\right) > 0$ при $x,\ x_r \in X,\ t \in [t_0,t_1)$.

Предложенный алгоритм может быть применен к задаче управления по выходу, если построить соответствующий наблюдатель, определяющий в каждый момент времени оценку неизвестного состояния $\tilde{x}$.

2. Синтез наблюдателя

Составим уравнение наблюдателя состояния полного порядка [8]

$$\dot{\tilde{\chi}} = \tilde{A}(\tilde{\chi},\mu)\tilde{\chi} + \tilde{B}(\chi,\mu)u + \Gamma\tilde{C}(\tilde{x}-\tilde{\chi}), \quad \tilde{\chi}(0) = \tilde{\chi}^0, \tag{6}$$

где $\tilde{\chi} = \begin{bmatrix} \chi \\ \chi_r \end{bmatrix} \in \mathbb{R}^{2n}$ – вектор оценки состояния $\tilde{x}$, т.е. χ – оценка x, а χ_r – оценка x_r, $\chi, \chi_r \in X$, $\tilde{C} = \begin{bmatrix} C & 0 \\ 0 & C \end{bmatrix} \in \mathbb{R}^{2m\times 2n}$, а $\Gamma \in \mathbb{R}^{2n\times 2m}$ – подлежащая определению матрица коэффициентов наблюдателя. Вычитая (6) из первого уравнения системы (3), получаем

$$\dot{\tilde{x}} - \dot{\tilde{\chi}} = \tilde{A}(\tilde{x},\mu)\tilde{x} - \tilde{A}(\tilde{\chi},\mu)\tilde{\chi} + (\tilde{B}(x,\mu) - \tilde{B}(\chi,\mu))u - \Gamma\tilde{C}(\tilde{x}-\tilde{\chi}) =$$
$$(\tilde{A}_0 - \Gamma\tilde{C})(\tilde{x}-\tilde{\chi}) + \mu\left(\tilde{A}_1(\tilde{x})\tilde{x} - \tilde{A}_1(\tilde{\chi})\tilde{\chi} + (\tilde{B}_1(x) - \tilde{B}_1(\chi))u\right).$$

Введем ошибку слежения $e_x = \tilde{x} - \tilde{\chi}$, тогда последнее уравнение можно переписать в виде

$$\dot{e}_x = (\tilde{A}_0 - \Gamma\tilde{C})e_x + \mu\left(\tilde{A}_1(\tilde{x})\tilde{x} - \tilde{A}_1(\tilde{\chi})\tilde{\chi} + (\tilde{B}_1(x) - \tilde{B}_1(\chi))u\right).$$

Подчеркнем, что $\tilde{x}$ и x здесь являются неизвестными. Рассмотрим неоднородность, присутствующую при μ. С учетом того, что управление (4) будет строиться на основе наблюдателя, справедливо $u = -K(\tilde{\chi},\mu,t)\tilde{\chi}$. Тогда имеем

$$\tilde{A}_1(\tilde{x})\tilde{x} - \tilde{A}_1(\tilde{\chi})\tilde{\chi} - (\tilde{B}_1(x) - \tilde{B}_1(\chi))K\tilde{\chi} =$$
$$\tilde{A}_1(\tilde{x})\tilde{x} - (\tilde{A}_1(\tilde{\chi}) + (\tilde{B}_1(x) - \tilde{B}_1(\chi))K)\tilde{\chi} + (\tilde{A}_1(\tilde{\chi}) + (\tilde{B}_1(x) - \tilde{B}_1(\chi))K)\tilde{x} -$$
$$(\tilde{A}_1(\tilde{\chi}) + (\tilde{B}_1(x) - \tilde{B}_1(\chi))K)\tilde{x} =$$
$$(\tilde{A}_1(\tilde{x}) - \tilde{A}_1(\tilde{\chi}) - (\tilde{B}_1(x) - \tilde{B}_1(\chi))K)\tilde{x} + (\tilde{A}_1(\tilde{\chi}) + (\tilde{B}_1(x) - \tilde{B}_1(\chi))K)e_x =$$
$$(\tilde{A}_1(\tilde{\chi}) - \tilde{B}_1(\chi)K)e_x + \tilde{B}_1(x)Ke_x + (\tilde{A}_1(\tilde{x}) - \tilde{A}_1(\tilde{\chi}) - (\tilde{B}_1(x) - \tilde{B}_1(\chi))K)\tilde{x} =$$
$$(\tilde{A}_1(\tilde{\chi}) - \tilde{B}_1(\chi)K)e_x + l(\tilde{x},\tilde{\chi},\mu,t),$$

где $l(\tilde{x},\tilde{\chi},\mu,t) = \tilde{B}_1(x)K(\tilde{\chi},\mu,t)e_x + (\tilde{A}_1(\tilde{x}) - \tilde{A}_1(\tilde{\chi}) - (\tilde{B}_1(x) - \tilde{B}_1(\chi))K(\tilde{\chi},\mu,t))\tilde{x} \in \mathbb{R}^{2n}$ – неизвестный вектор. Представим l в виде $l(\tilde{x},\tilde{\chi},\mu,t) = L(\tilde{\chi},\mu,t)e_x$, где $L(\tilde{\chi},\mu,t) \in \mathbb{R}^{2n\times 2n}$ – неизвестная матрица. Тогда уравнение динамики ошибки имеет вид

$$\dot{e}_x = (\tilde{A}_0 - \Gamma\tilde{C} + \mu(\tilde{A}_1(\tilde{\chi}) - \tilde{B}_1(\chi)K + L))e_x =$$
$$(\tilde{A}_0 - \Gamma\tilde{C} + \mu(\tilde{A}_{cl,1}(\tilde{\chi},\mu,t) + L))e_x = W(\tilde{\chi},\mu,t,L)e_x,$$

где $W(\tilde{\chi},\mu,t,L) = \tilde{A}_0 - \Gamma\tilde{C} + \mu(\tilde{A}_{cl,1}(\tilde{\chi},\mu,t) + L), \tilde{A}_{cl,1}(\tilde{\chi},\mu,t) = \tilde{A}_1(\tilde{\chi}) - \tilde{B}_1(\chi)K(\tilde{\chi},\mu,t)$.

От полученной системы для ошибки перейдем к другой, обладающей при каждом $\tilde{\chi},\mu,t,L$ теми же собственными значениями, а значит, в линейном приближении эквивалентную исходной с точки зрения свойств устойчивости

$$\dot{e}_x = W^T(\tilde{\chi},\mu,t,L)e_x = (\tilde{A}_0 - \Gamma\tilde{C} + \mu(\tilde{A}_{cl,1}(\tilde{\chi},\mu,t) + L))^T e_x =$$
$$(\tilde{A}_0 + \mu\tilde{A}_{cl,1}(\tilde{\chi},\mu,t))^T e_x - \tilde{C}^T\Gamma^T e_x + \mu L^T e_x.$$

Таким образом, приходим к системе

$$\dot{e}_x = \bar{A}^T(\tilde{\chi},\mu,t)e_x - \tilde{C}^T\upsilon + \upsilon, \tag{7}$$

где $\bar{A}^T(\tilde{\chi},\mu,t) = \tilde{A}_0^T + \mu\tilde{A}_{cl,1}^T(\tilde{\chi},\mu,t)$, а $\upsilon(\tilde{x},\tilde{\chi},\mu,t) = \Gamma^T e_x \in \mathbb{R}^{2m}$, $\upsilon(\tilde{x},\tilde{\chi},\mu,t) = \mu L^T e_x \in \mathbb{R}^{2n}$ – новые управление и неизвестное возмущение.

Рассмотрим неопределенность $\upsilon(\tilde{x},\tilde{\chi},\mu,t)$ как управление противодействующего игрока. Такая трактовка приводит к дифференциальной игре. Для её решения применим принцип гарантированного управления [4–6], который приводит к отысканию оптимального $v(\tilde{x},\tilde{\chi},\mu,t)$ при наихудшей реализации υ с точки зрения функционала

$$\begin{gathered} I_e(v,\upsilon) = \\ \tfrac{1}{2}e_x^T(t_1)F_e e_x(t_1) + \tfrac{1}{2}\int_{t_0}^{t_1}\left(e_x^T Q_e(\tilde{\chi},\mu)e_x + v^T R_v v - \upsilon^T R_\upsilon \upsilon\right)dt \to \min_v \max_\upsilon , \\ Q_e(\tilde{\chi},\mu) = Q_{e,0} + \mu Q_{e,1}(\tilde{\chi}), \end{gathered} \tag{8}$$

где $F_e \geq 0,\ Q_{e,0} \geq 0,\ Q_{e,1} \geq 0,\ R_v > 0, R_\upsilon > 0$ – заданные весовые матрицы.

Определим условия

IV. Пара матриц $\left\{\bar{A}^T(\tilde{\chi},\mu,t), \tilde{C}^T\right\}$ управляема при всех $\chi,\chi_r \in X$, $\mu \in (0,\ \mu_0]$, $t \in [t_0,t_1]$.

V. $\tilde{C}^T R_v^{-1}\tilde{C} - R_\upsilon^{-1} > 0$.

По аналогии с работой [6], где интервал регулирования предполагается достаточно большим по сравнению со временем переходного процесса, при выполнении IV–V законы управления и возмущения определяются как $v = R_v^{-1}CN(\tilde{\chi},\mu,t)e_x$, $\upsilon = R_\upsilon^{-1}N(\tilde{\chi},\mu,t)e_x$, где $N(\tilde{\chi},\mu,t)$ – положительно определенное решение уравнения SDRE

$$\dot{N} + N\bar{A}^T + \bar{A}N - N\left(\tilde{C}^T R_v^{-1}\tilde{C} - R_\upsilon^{-1}\right)N + Q_e = 0, \quad N(t_1) = F_e, \tag{9}$$

Из полученных соотношений следует, что

$$\begin{gathered} \Gamma(\tilde{\chi},\mu,t) = N(\tilde{\chi},\mu,t)\tilde{C}^T R_v^{-1}, \\ L(\tilde{\chi},\mu,t) = \tfrac{1}{\mu}N(\tilde{\chi},\mu,t)R_\upsilon^{-1}. \end{gathered} \tag{10}$$

Основную трудность представляет решение дифференциального SDRE (9). Однако можно заметить, что сформулированная выше задача (3) и задача (7)–(8) имеют схожую структуру. Поэтому для приближенного решения (9) в задаче построения наблюдателя можно применить уже изложенный выше алгоритм из [3], использовавшийся для решения схожего SDRE в задаче синтеза управления. Для этого введем условия

VI. Траектории замкнутой системы (7) на $[t_0,t_1]$ существуют, единственны и $\chi,\chi_r \in X$ для любых непрерывных $v(t)$, $\upsilon(t)$; элементы матрицы $\bar{A}(\tilde{\chi},\mu,t)$ ограничены, непрерывны и достаточно гладкие при $\chi,\chi_r \in X, \mu \in (0,\ \mu_0], t \in [t_0,t_1]$.

VII. Тройка матриц $\left\{\tilde{A}_0^T, \tilde{C}^T, H_N\right\}$, где $H_N^T H_N = Q_{e,0}$, стабилизируема и наблюдаема.

VIII. Матрицы системы $\tilde{A}_0, \tilde{A}_{cl,1}(\tilde{\chi}, \mu, t)$, $\tilde{C}$ и симметрические матрицы критерия $R_v > 0, R_v > 0$, $Q_{e,0} \geq 0, Q_{e,1}(\tilde{\chi}) \geq 0$, $F_e \geq 0$, а также $\mu_0 > 0$ таковы, что $N_0 + \mu N_1(\tilde{\chi}, \mu, t) > 0$ при $\chi, \chi_r \in X$, $t \in [t_0, t_1)$, $\mu \in (0, \mu_0]$.

Таким образом, в соответствии с разделом 1 находится регулятор по состоянию. Затем, при выполнении IV–VIII строится наблюдатель по следующей численно-аналитической процедуре.

1. Находим N_0 как положительно определенное решение уравнения

$$N_0\tilde{A}_0^T + \tilde{A}_0 N_0 \ - N_0\left(\tilde{C}^T R_v^{-1}\tilde{C} - R_v^{-1}\right)N_0 + Q_{e,0} = 0.$$

2. Находим $N_1(\tilde{\chi}, \mu, t)$ с помощью

$$N_1(\tilde{\chi}, \mu, t) = e^{\tilde{A}^T_{Ncl,0}(t_1-t)} M_N e^{\tilde{A}_{Ncl,0}(t_1-t)} + \int_0^\infty e^{\tilde{A}^T_{Ncl,0}\sigma} D_N(\tilde{\chi}, \mu, t) e^{\tilde{A}_{Ncl,0}\sigma} d\sigma,$$

где

$$M_N = \tfrac{1}{\mu}(F_e - N_0) - \int_0^\infty e^{\tilde{A}^T_{Ncl,0}\sigma}\ D_N\left(\tilde{\chi}(t_1), \mu, t_1\right)\big|_{\tilde{\chi}(t_1)=\tilde{\chi}(t)} e^{\tilde{A}_{Ncl,0}\sigma} d\sigma,$$
$$D_N(\tilde{\chi}, \mu, t) = N_0\tilde{A}^T_{cl,1}(\tilde{\chi}, \mu, t) + \tilde{A}_{cl,1}(\tilde{\chi}, \mu, t)N_0 + Q_{e,1}(\tilde{\chi}),$$
$$\tilde{A}_{Ncl,0} = \tilde{A}_0^T - \left(\tilde{C}^T R_v^{-1}\tilde{C} - R_v^{-1}\right)N_0.$$

3. Определяем $\Gamma(\tilde{\chi}, \mu, t)$ с помощью (10), считая $N(\tilde{\chi}, \mu, t) = N_0 + \mu N_1(\tilde{\chi}, \mu, t)$.

4. Находим наблюдатель (6), задав начальное состояние $\tilde{\chi}^0$ произвольным образом.

Теперь мы можем применить найденное управление (4), используя в нем вместо неизвестного вектора состояния $\tilde{x}$ его оценку $\tilde{\chi}$.

Замечание 1. В отличие от линейных систем [9] в данном случае принцип разделения задач синтеза управления и наблюдения полностью не выполняется, поскольку матрица коэффициентов наблюдателя Γ зависит от матрицы коэффициентов обратной связи K.

Замечание 2. Если известно начальное состояние x^0 или ${x_r}^0$, тогда при $t \geq t_0$ с помощью наблюдателя можно получить абсолютно точные оценки вектора x или x_r соответственно, задав $\chi^0 = x^0$ или ${\chi_r}^0 = {x_r}^0$. Если эталонная траектория x_r известна заранее (что означает $\chi_r \equiv x_r$ согласно замечанию выше), то можно, предполагая близость $x(t_1)$ к $x_r(t_1)$ и близость $\chi(t_1)$ к $\chi_r(t_1)$, вычислять D_N, используя $\chi(t_1) = \chi_r(t_1) = x_r(t_1)$.

Замечание 3. Как в случае с условием III, условие VIII выполняется при достаточно малом μ_0.

Заключение

В данной статье рассмотрен подход к построению нелинейного управления по выходу для задачи слежения в слабо нелинейной системе. Построение наблюдателя состояния основано на принципе гарантированного управления. Синтез управления и наблюдателя приводит к необходимости рассмотрения подобных уравнений Риккати, с зависящими от состояния коэффициентами, и для их решения применяется один и тот же приближенный подход. Его основное достоинство – использование аналитических выражений, которые существенно снижают вычислительные затраты по сравнению с обычным методом, когда соответствующее уравнение Риккати решается в процессе управления для каждого состояния системы.

Список литературы / References

[1] Cimen T., "Survey of state-dependent Riccati equation in nonlinear optimal feedback control synthesis", *Journal of Guidance, Control, and Dynamics*, **35**:4 (2012), 1025–1047.

[2] Cloutier J. R., "State-Dependent Riccati Equation Techniques: An Overview", *Proc. American Control Conference*, **2** (1997), 932–936.

[3] Макаров Д. А., "Подход к построению нелинейного управления в задаче слежения с коэффициентами, зависящими от состояния. Часть I. Алгоритм", *Информационные технологии и вычислительные системы*, 2017, №3, 10–19; [Makarov D. A., "A nonlinear approach to a feedback control design for a tracking state-dependent problem. Part I. An algorithm", *Information Technologies and Computing Systems*, 2017, №3, 10–19, (in Russian).]

[4] Афанасьев В. Н., *Динамические системы управления с неполной информацией: алгоритмическое конструирование*, УРСС, КомКнига, М., 2007, 214 с.; [Afanasiev V. N., *Dinamicheskie sistemy upravlenija s nepolnoj informaciej: algoritmicheskoe konstruirovanie*, URSS, KomKniga, Moscow, 2007, 214 pp., (in Russian).]

[5] Афанасьев В. Н., "Концепция гарантированного управления неопределенными объектами", *Известия РАН. Теория и системы управления*, 2010, №1, 24–31; English transl.: Afanas'ev V.N., "Guaranteed control concept for uncertain objects", *Journal of Computer and Systems Sciences International*, **49**:1 (2010), 22–29.

[6] Афанасьев В. Н., "Метод расширенной линеаризации в задаче управления неопределенным динамическим нелинейным объектом", *Современные проблемы прикладной математики, информатики, автоматизации и управления*, Материалы 4-го научно-технического семинара, ИПИ РАН, М., 2014, 47–54; [Afanas'ev V. N., "Extended linearization method in the problem of uncertain nonlinear object control", *Sovremennye problemy prikladnoy matematiki, informatiki, avtomatizatsii i upravleniya*, IPI RAN, Moscow, 2014, 47–54, (in Russian).]

[7] *Методы классической и современной теории автоматического управления*, Учебник в 5 т.. Т. 4: *Теория оптимизации систем автоматического управления*, 2-е изд., перераб. и доп., ред. Пупков К. А., Егупов Н. Д., Издательство МГТУ им. Баумана, М., 2004, 742 с.; [*Metody klassicheskoj i sovremennoj teorii avtomaticheskogo upravlenija*, Uchebnik v 5 t.. V. 4: *Teorija optimizacii sistem avtomaticheskogo upravlenija*, 2-e izd., pererab. i dop., eds. Pupkov K. A., Egupov N. D., Izdatelstvo MGTU im. Baumana, Moscow, 2004, 742 pp., (in Russian).]

[8] Kwwakernaak H., Sivan R., *Linear optimal control systems*, Wiley-Interscience, New York, 1972.

[9] Первозванский А. А., *Курс теории автоматического управления*, Наука, М., 1986, 616 с.; [Pervozvanskij A. A., *Kurs teorii avtomaticheskogo upravlenija*, Nauka, Moscow, 1986, 616 pp., (in Russian).]

Makarov D. A., "Synthesis of Control and State Observer for Weakly Nonlinear Systems Based on the Pseudo-Linearization Technique", *Modeling and Analysis of Information Systems*, **24**:6 (2017), 802–810.

DOI: 10.18255/1818-1015-2017-6-802-810

Abstract. In this paper, an approach to the construction of nonlinear output tracking control on a finite time interval for a class of weakly nonlinear systems with state-dependent coefficients is considered. The proposed method of control synthesis consists of two main stages. At the first stage, a nonlinear state feedback regulator is constructed by using a previously proposed control algorithm based on the State Dependent Riccati Equation (SDRE). At the second stage, the problem of full-order observer construction is formulated and then it is reduced to the differential game problem. The form of its solution is obtained with the help of the guaranteed (minimax) control principle, which allows to find the best observer coefficients with respect to a given functional considering the worst-case uncertainty realization. The form of the obtained equations made it possible to use the algorithm from the first stage to determine the observer matrix. The proposed approach is characterized by the nonapplicability of the estimation and control separation principle used for linear systems, since the matrix of observer coefficients turned out to be dependent on the feedback coefficients matrix. The use of numerical-analytical procedures for determination of observer and feedback coefficients matrices significantly reduces the computational complexity of the control algorithm.

Keywords: tracking problem, nonlinear control, state-dependent Riccati equation, minimax control

On the authors:
Dmitry A. Makarov, orcid.org/0000-0001-8930-1288, PhD (Physics and Mathematics),
«Technologies Of System Analysis» Ltd,
Institute for Systems Analysis, Federal Research Center «Computer Science and Control» of Russian Academy of Sciences,
9 pr-t 60-letiya Oktyabrya, Moscow 117312, Russia, e-mail: makarov@isa.ru

Acknowledgments:
[1] This work was supported by RFBR (projects № 16-38-60198-мол_а_дк, № 17-07-00281-а).

Моделирование и анализ информационных систем. Т. 24, № 6 (2017), с. 811–815
Modeling and Analysis of Information Systems. Vol. 24, No 6 (2017), pp. 811–815

DOI: 10.18255/1818-1015-2017-6-811-815
УДК 519.7 + 025.4.06

Поэтология: задачи построения тезауруса и спецификации стихового текста

Бойков В. Н., Каряева М. С.

получена 15 октября 2017

Аннотация. Представлено краткое изложение доклада авторов на семинаре “Моделирование и анализ информационных систем”, Ярославль, 17 мая 2017 г. В нём рассматривается взаимосвязь задач по автоматизации построения тезауруса и спецификации текста стихотворного произведения в поэтологии.

Ключевые слова: стих, поэтология, тезаурус, информационно-аналитическая система

Для цитирования: Бойков В. Н., Каряева М. С., "Поэтология: задачи построения тезауруса и спецификации стихового текста", *Моделирование и анализ информационных систем*, **24**:6 (2017), 811–815.

Об авторах:
Бойков Владимир Николаевич, математик,
Ярославский государственный университет им. П.Г. Демидова,
ул. Советская, 14, г. Ярославль, 150003 Россия, e-mail: boykov_bh@bk.ru

Каряева Мария Сергеевна, orcid.org/0000-0003-4466-1735, аспирант,
Ярославский государственный университет им. П.Г. Демидова,
ул. Советская, 14, г. Ярославль, 150003 Россия,
e-mail: mari.karyaeva@gmail.com

Благодарности:
Работа выполнена при поддержке Российского фонда фундаментальных исследований (РФФИ), проекты № 16-07-01180 и № 16-06-00497.

Под поэтологией понимается группа дисциплин, ориентированная на всестороннее теоретическое и историческое изучение поэзии (как способа организации речи), ее текстов и стиха (как сегмента поэтического текста).

Задачей изучения поэтического текста как объекта поэтологии является его спецификация, т.е., с одной стороны, задание параметров спецификации – определенной матрицы (формуляра) характерных атрибутов (признаков, свойств, особенностей) некоторого класса объектов исследования; с другой стороны, процедура идентификации объекта – выявление специфических значений этих атрибутов для конкретного объекта.

Такого рода изучение предполагает метаописание поэтического текста в терминах тезауруса по поэтологии и, следовательно, создание такого тезауруса. Под тезаурусом понимается совокупность терминов (слов или словосочетаний), представляющих понятия (концепты) данной предметной области с соответствующими

им определениями, а также отношениями и связями между ними, составляющими их спецификацию [3]. Таким образом, объектами исследования и соответственно установления их спецификации (метаописания) в поэтологии являются как поэтические произведения (стиховые тексты), так и сами термины тезауруса по поэтологии. Комплексное решение этих задач представляет поэтологию как информационно-аналитическую систему [1–2].

Тезаурус по поэтологии в ранее проведенных исследованиях задается как инструмент лингво-стиховедческого и историко-литературного метаописания поэзии и поэтического текста [5–9,11–14]. Исходный терминологический словник тезауруса по поэтологии составлен из 1544 слов и словосочетаний. Создан иерархический указатель для верхних уровней иерархии терминов с кластеризацией нижних уровней, т.е. рубрикация терминов тезауруса по поэтологии на подобласти:

1. Стиховедение;
2. Стилистика;
3. Поэтика;
4. Риторика;
5. История литературы;
6. Переводоведение и литературная компаративистика;
7. Текстология;
8. Герменевтика;
9. Теоретические школы и направления;
10. Логика и методология науки.

Не для всех разделов первого уровня тезауруса может применяться автоматическое построение. Большинство терминов тезауруса представляют предметную подобласть Стиховедение, подрубрика которого – первая рубрика второго уровня «Стих» – разбивается на 6 подрубрик третьего уровня:

1.1.1 Метрика;
1.1.2. Явления начала и конца стихотворной строки;
1.1.3. Ритмика;
1.1.4. Строфика;
1.1.5. Рифмика;
1.1.6. Лингвистика стиха.

Основные атрибуты этих подрубрик могут поддаваться автоматическому анализу и задавать соответствующую спецификацию стихового текста.

В широком смысле предназначением рассматриваемой информационно-аналитической системы является установление трех уровней спецификации текста:

С1. Метрико-строфическая спецификация конкретного поэтического произведения (стихового текста);

С2. Поэтико-стилистическая спецификация поэтического творчества конкретного автора на основе метрико-строфической спецификации его произведений;

С3. Историко-литературная спецификация направлений, школ и периодов на основе статистической близости поэтико-стилистической спецификации различных авторов.

В процессе анализа необходимо различать поэтическое произведение как эстетическую целостность и стиховой текст произведения как объект изучения в Стиховедении. В связи с этим следует различать лингво-стиховедческую разметку стиха и

спецификацию стихового текста поэтического произведения. Разметка стиха отражает в основном числовые (количественные и порядковые) характеристики стихового текста, а спецификация представляет собой его метаописание на основе понятий тезауруса по поэтологии.

Разметка стиха (строки) позволяет получить формально-языковое слоговое представление стиха [4,10]:

$$b^{r(1)}A_1d^{s(1)}b^{r(2)}A_2d^{s(2)}...b^{r(k)}A_kd^{s(k)},$$

где в цепочке символов A означает икты (метрически ударные слоги), k – количество иктов, b и d означают безударные соответственно препозитивные и постпозитивные слоги между иктами, r и s длину соответствующих подцепочек безударных слогов. Подцепочки $b^{r(1)}$ (анакруза), $d^{s(k)}$ (клаузула), $d^{s(j)}b^{r(j+1)}$ (междуиктовый интервал), $j = 1, 2, ...(k-1)$, вместе с числом иктов k и числом слогов в строке $(k + \sum(r(j) + s(j+1))_{j=1,2,...(k-1)})$ представляют собой регулятивы, определяющие в различных сочетаниях их значений репертуар метро-ритмических форм стиха. Полный репертуар метрических схем должен включать во всех формах всевозможные пропуски схемных ударений и сверхсхемные (внеметрические) ударения. Определение метро-ритмической формы стиха является важнейшей характеристикой его спецификации.

С течением времени появляются новые авторы поэтических текстов, подчас привносящие новшества в стихосложение, а в стиховедении появляются новые исследователи, открывающие новые явления в поэтологии, что вносит как дополнения терминов в тезаурус, так и изменения и дополнения спецификации его терминов. Это, в свою очередь, вносит коррективы в матрицу спецификации поэтического текста. Таким образом, поэтология как информационно-аналитическая система не может, да и не должна иметь завершения.

Список литературы / References

[1] Захаров В.Е., Бойков В.Н., Вигурский К.В., Пильщиков И.А., *Русская поэзия: проблемы консолидации и анализа в электронном формате*, Москва, 2004; [Zakharov V.E., Boykov V.N., Vigurskiy K.V., Pil'shchikov I.A., *Russkaja poezija: problemy konsolidacii i analiza v elektronnom formate*, Moskva, 2004., (in Russian).]

[2] Бойков В.Н., Захаров В.Е., Пильщиков И.А., Сысоев Т.М., "Тезаурус как инструмент поэтологии", *Модел. и анализ информ. систем*, **17**:1 (2010), 5–23; [Boykov V.N., Zakharov V.E., Pil'shchikov I.A., Sysoev T.M., "Thesaurus as a poetological tool", *Model. Anal. Inform. Sist.*, **17**:1 (2010), 5–23, (in Russian).]

[3] Лукашевич Н.В., *Тезаурусы в задачах информационного поиска*, Издательство Московского университета, М., 2011; [Lukashevich N.V., *Tezaurusy v zadachah informacionnogo poiska*, Izdatelstvo Moskovskogo universiteta, Moskow, 2011, (in Russian).]

[4] Бойков В.Н., "Контекстно-свободная грамматика одной ритмической модели русского стиха", *Модел. и анализ информ. систем*, **19**:4 (2012), 154–167; [Boykov V.N., "A Context-Free Grammar of One Rhythmic Model of Russian Verse", *Model. Anal. Inform. Sist.*, **19**:4 (2012), 154–167, (in Russian).]

[5] Бойков В.Н., Пильщиков И.А., "Семантическая модель "Тезауруса по поэтологии" в составе информационно-аналитической системы", *Интернет и современное общество*, Труды XVI Всероссийской объединенной конференции IMS-2013, Санкт-Петербург, 2013, 273–279; [Boykov V.N., Pil'shchikov I.A., "Semanticheskaya model

"Tezaurusa po poehtologii" v sostave informacionno-analiticheskoj sistemy", *Internet i sovremennoe obshchestvo*, Trudy XVI Vserossiyskoy konferencii IMS-2013, Sankt-Peterburg, 2013, 273–279, (in Russian).]

[6] Бойков В.Н., Захаров В.Е., Каряева М.С., Соколов В.А., "Тезаурус по поэтологии как инструмент для информационного поиска и коллекции знаний", *Модел. и анализ информ. систем*, **20**:4 (2013), 125–135; [Boykov V.N., Zakharov V.E., Karyaeva M.S., Sokolov V.A., "Domain-Specific Thesaurus as a Tool for Information Retrieval and Collection of Knowledge", *Model. Anal. Inform. Sist.*, **20**:4 (2013), 125–135, (in Russian).]

[7] Бойков В.Н., Захаров В.Е., Каряева М.С., Соколов В.А., "Предметно-ориентированный тезаурус в открытой информационно-аналитической системе", *Электронные библиотеки: перспективы, методы и технологии, электронные коллекции*, RCDL'2013, 2013, 70–76; [Boykov V.N., Zakharov V.E., Karyaeva M.S., Sokolov V.A., "Predmetno-orientirovannyj tezaurus v otkrytoj informacionno-analiticheskoj sisteme", *Ehlektronnye biblioteki: perspektivy, metody i tekhnologii, ehlektronnye kollekcii*, RCDL'2013, 2013, 70–76, (in Russian).]

[8] Бойков В.Н., Захаров В.Е., Каряева М.С., Соколов В.А., "Об автоматической рубрикации терминов тезауруса открытой информационно-аналитической системы", *RCDL-2014*, Дубна, 2014; [Boykov V.N., Zakharov V.E., Karyaeva M.S., Sokolov V.A., "Ob avtomaticheskoj rubrikacii terminov tezaurusa otkrytoj informacionno-analiticheskoj sistemy", *RCDL-2014*, Dubna, 2014, (in Russian).]

[9] Бойков В.Н., Захаров В.Е., Каряева М.С., Соколов В.А., "Информационно-аналитический блок тезауруса как инструмент обучения", *Образование, наука и экономика в вузах и школах. Интеграция в международное образовательное пространство*, Труды международной научной конференции, Цахкадзор, 2014, 205–209, http://nuclphys.sinp.msu.ru/math/conf/tsaghkadzor_conference_part1.pdf; [Boykov V.N., Zakharov V.E., Karyaeva M.S., Sokolov V.A., "Informacionno-analiticheskij blok tezaurusa kak instrument obuchenija", *Education, science and economics at universities and schools*, Publications International Scientific Conference Integration to international educational area, Tsaghkadzor, 2014, 205–209, http://nuclphys.sinp.msu.ru/math/conf/tsaghkadzor_conference_part1.pdf, (in Russian).]

[10] Бойков В.Н., Каряева М.С., Соколов В.А., Пильщиков И.А., "Об автоматической спецификации стиха в информационно-аналитической системе", *DAMDID/RCDL-2015*, Обнинск, 2015; [Boykov V.N, Karyaeva M.S., Sokolov V.A., Pil'shchikov I.A., "Ob avtomaticheskoj specifikacii stiha v informacionno-analiticheskoj sisteme", *DAMDID/RCDL-2015*, Obninsk, 2015, (in Russian).]

[11] Каряева М.С., "Лингвостатистический анализ терминологии для построения тезауруса предметной области", *Модел. и анализ информ. систем*, **22**:6 (2015), 834–851; [Karyaeva M.S., "Linguistic and statistical analysis of the terminology for constructing the thesaurus of a specified field", *Model. Anal. Inform. Sist.*, **22**:6 (2015), 834–851, (in Russian).]

[12] Каряева М.С., Соколов В.А., "Обзор методов извлечения гипо/гиперонимов из коллекций документов", *Заметки по информатике и математике*, Сборник научных статей, **7**, ЯрГУ им. П.Г. Демидова, Ярославль, 2015, 53–58, http://www.lib.uniyar.ac.ru/edocs/iuni/20150406.pdf; [Karyaeva M.S., Sokolov V.A., "Obzor metodov izvlechenija gipo/giperonimov iz kollekcij dokumentov", *Zametki po informatike i matematike*, Sbornik nauchnyh statej, **7**, YarGU im. P.G. Demidova, Jaroslavl, 2015, 53–58, http://www.lib.uniyar.ac.ru/edocs/iuni/20150406.pdf, (in Russian).]

[13] Каряева М.С., Соколов В.А., "Об информационных технологиях решения задачи извлечения терминов предметной области", *Математика, физика, информатика и их приложения в науке и образовании*, Сборник тезисов докладов Международной школы-конференции молодых ученых, МИРЭА, М., 2016, 143–145, https://elibrary.ru/item.asp?id=28373830; [Karyaeva M.S., Sokolov V.A., "Ob informacionnyh tehnologijah reshenija zadachi izvlechenija terminov predmetnoj oblasti", *Mathematics, physics, informatics and their applications in science and education*, Moskva, 2016, 143–145, https://elibrary.ru/item.asp?id=28373830, (in Russian).]

[14] Karyaeva M. S., Sokolov V.A., “On the Problem of Multi-word Term Extraction from a Domain-specific Document Collection”, *Data Analytics and Management in Data Intensive Domains (DAMDID/RCDL’2017)*, Proceedings of the XIX International Conference, Moscow, 218–221.

Boykov V. N., Karyaeva M. S. , "Poetology: Problems of Constructing a Thesaurus and Verse Text Specification", *Modeling and Analysis of Information Systems*, **24**:6 (2017), 811–815.

DOI: 10.18255/1818-1015-2017-6-811-815

Abstract. It is a brief version of the report made by the authors at the seminar “Modeling and Analysis of Information Systems”, Yaroslavl, May 17, 2017. The interrelation of problems of computer-aided constructing the thesaurus and specification of verse texts in poetology is considered.

Keywords: verse, poetology, thesaurus, information-analytical system

On the authors:
Vladimir N. Boykov, mathematician,
P.G. Demidov Yaroslavl State University,
14 Sovetskaya str., Yaroslavl 150003,
Russia, e-mail: boykov_bh@bk.ru

Maria S. Karyaeva, orcid.org/0000-0003-4466-1735, graduate student,
P.G. Demidov Yaroslavl State University,
14 Sovetskaya str., Yaroslavl 150003, Russia,
e-mail: mari.karyaeva@gmail.com

Acknowledgments:
This work was supported by RFBR, projects № 16-07-01180, № 16-06-00497.