ТЕОРЕТИЧЕСКИЙ И ПРИКЛАДНОЙ НАУЧНО-ТЕХНИЧЕСКИЙ ЖУРНАЛ

Программная инженерия

ПрИн

Том 10
№ 9-10
2019

Учредитель: Издательство "НОВЫЕ ТЕХНОЛОГИИ"

Издается с сентября 2010 г. DOI 10.17587/issn.2220-3397 ISSN 2220-3397

Редакционный совет
Садовничий В.А., акад. РАН (*председатель*)
Бетелин В.Б., акад. РАН
Васильев В.Н., чл.-корр. РАН
Жижченко А.Б., акад. РАН
Макаров В.Л., акад. РАН
Панченко В.Я., акад. РАН
Стемпковский А.Л., акад. РАН
Ухлинов Л.М., д.т.н.
Федоров И.Б., акад. РАН
Четверушкин Б.Н., акад. РАН

Главный редактор
Васенин В.А., д.ф.-м.н., проф.

Редколлегия
Антонов Б.И.
Афонин С.А., к.ф.-м.н.
Бурдонов И.Б., д.ф.-м.н., проф.
Борзовс Ю., проф. (Латвия)
Гаврилов А.В., к.т.н.
Галатенко А.В., к.ф.-м.н.
Корнеев В.В., д.т.н., проф.
Костюхин К.А., к.ф.-м.н.
Махортов С.Д., д.ф.-м.н., доц.
Манцивода А.В., д.ф.-м.н., доц.
Назиров Р.Р., д.т.н., проф.
Нечаев В.В., д.т.н., проф.
Новиков Б.А., д.ф.-м.н., проф.
Павлов В.Л. (США)
Пальчунов Д.Е., д.ф.-м.н., доц.
Петренко А.К., д.ф.-м.н., проф.
Позднеев Б.М., д.т.н., проф.
Позин Б.А., д.т.н., проф.
Серебряков В.А., д.ф.-м.н., проф.
Сорокин А.В., к.т.н., доц.
Терехов А.Н., д.ф.-м.н., проф.
Филимонов Н.Б., д.т.н., проф.
Шапченко К.А., к.ф.-м.н.
Шундеев А.С., к.ф.-м.н.
Щур Л.Н., д.ф.-м.н., проф.
Язов Ю.К., д.т.н., проф.
Якобсон И., проф. (Швейцария)

Редакция
Лысенко А.В., Чугунова А.В.

Журнал издается при поддержке Отделения математических наук РАН, Отделения нанотехнологий и информационных технологий РАН, МГУ имени М.В. Ломоносова, МГТУ имени Н.Э. Баумана

СОДЕРЖАНИЕ

Галатенко В. А., Костюхин К. А. Автоматический ремонт программ: базовые понятия и подходы 355

Пономарев В. А. Имитационное моделирование показателей функционирования твердотельной системы хранения данных 367

Гвоздев В. Е., Черняховская Л. Р., Насырова Р. А. Анализ надежности информационных сервисов с учетом их объективных характеристик и субъективных оценок пользователей 377

Грузенкин Д. В., Михалев А. С. Определение метрики диверсифицированности мультиверсионного программного обеспечения на уровне языков программирования 384

Трофимов И. В. Морфологический анализ русского языка: обзор прикладного характера 391

Соколов А. П., Макаренков В. М., Першин А. Ю., Лаишевский И. А. Разработка программного обеспечения генерации кода на основе шаблонов при создании систем инженерного анализа 400

Журнал зарегистрирован в Федеральной службе по надзору в сфере связи, информационных технологий и массовых коммуникаций. Свидетельство о регистрации ПИ № ФС77-38590 от 24 декабря 2009 г.

Журнал распространяется по подписке, которую можно оформить в любом почтовом отделении (индекс по Объединенному каталогу "Пресса России" — **22765**) или непосредственно в редакции.
Тел.: (499) 269-53-97. Факс: (499) 269-55-10.
Http://novtex.ru/prin/rus E-mail: prin@novtex.ru
Журнал включен в систему Российского индекса научного цитирования и базу данных RSCI на платформе Web of Science.
Журнал входит в Перечень научных журналов, в которых по рекомендации ВАК РФ должны быть опубликованы научные результаты диссертаций на соискание ученой степени доктора и кандидата наук.

© Издательство "Новые технологии", "Программная инженерия", 2019

THEORETICAL AND APPLIED SCIENTIFIC AND TECHNICAL JOURNAL

SOFTWARE ENGINEERING

PROGRAMMNAYA INGENERIA

Vol. 10
N 9-10
2019

Published since September 2010 DOI 10.17587/issn.2220-3397 ISSN 2220-3397

Editorial Council:
SADOVNICHY V. A., Dr. Sci. (Phys.-Math.), Acad. RAS (*Head*)
BETELIN V. B., Dr. Sci. (Phys.-Math.), Acad. RAS
VASIL'EV V. N., Dr. Sci. (Tech.), Cor.-Mem. RAS
ZHIZHCHENKO A. B., Dr. Sci. (Phys.-Math.), Acad. RAS
MAKAROV V. L., Dr. Sci. (Phys.-Math.), Acad. RAS
PANCHENKO V. YA., Dr. Sci. (Phys.-Math.), Acad. RAS
STEMPKOVSKY A. L., Dr. Sci. (Tech.), Acad. RAS
UKHLINOV L. M., Dr. Sci. (Tech.)
FEDOROV I. B., Dr. Sci. (Tech.), Acad. RAS
CHETVERTUSHKIN B. N., Dr. Sci. (Phys.-Math.), Acad. RAS

Editor-in-Chief:
VASENIN V. A., Dr. Sci. (Phys.-Math.)

Editorial Board:
ANTONOV B.I.
AFONIN S.A., Cand. Sci. (Phys.-Math)
BURDONOV I.B., Dr. Sci. (Phys.-Math)
BORZOVS JURIS, Dr. Sci. (Comp. Sci), Latvia
GALATENKO A.V., Cand. Sci. (Phys.-Math)
GAVRILOV A.V., Cand. Sci. (Tech)
JACOBSON IVAR, Dr. Sci. (Philos., Comp. Sci.), Switzerland
KORNEEV V.V., Dr. Sci. (Tech)
KOSTYUKHIN K.A., Cand. Sci. (Phys.-Math)
MAKHORTOV S.D., Dr. Sci. (Phys.-Math)
MANCIVODA A.V., Dr. Sci. (Phys.-Math)
NAZIROV R.R. , Dr. Sci. (Tech)
NECHAEV V.V., Cand. Sci. (Tech)
NOVIKOV B.A., Dr. Sci. (Phys.-Math)
PAVLOV V.L., USA
PAL'CHUNOV D.E., Dr. Sci. (Phys.-Math)
PETRENKO A.K., Dr. Sci. (Phys.-Math)
POZDNEEV B.M., Dr. Sci. (Tech)
POZIN B.A., Dr. Sci. (Tech)
SEREBRJAKOV V.A., Dr. Sci. (Phys.-Math)
SOROKIN A.V., Cand. Sci. (Tech)
TEREKHOV A.N., Dr. Sci. (Phys.-Math)
FILIMONOV N.B., Dr. Sci. (Tech)
SHAPCHENKO K.A., Cand. Sci. (Phys.-Math)
SHUNDEEV A.S., Cand. Sci. (Phys.-Math)
SHCHUR L.N., Dr. Sci. (Phys.-Math)
YAZOV Yu. K., Dr. Sci. (Tech)

Editors: LYSENKO A.V., CHUGUNOVA A.V.

CONTENTS

Galatenko V. A., Kostyukhin K. A. Automated Program Repair: Basic Concepts and Approaches 355

Ponomarev V. A. Simulation Modeling Performance Indicators for Solid State Storage Systems 367

Gvozdev V. E., Chernyakhovskaya L. R., Nasyrova R. A. Analysis of the Reliability of Information Services, Taking into Account their Objective Characteristics and Subjective Assessments of Users 377

Gruzenkin D. V., Mikhalev A. S. N-Version Software Diversity Metric Definition at the Programming Languages Level 384

Trofimov I. V. Automatic Morphological Analysis for Russian: Application-Oriented Survey 391

Sokolov A. P., Makarenkov V. M., Pershin A. Yu., Laishevskiy I. A. Development of Template-Based Code Generation Software for Development of Computer-Aided Engineering System 400

Information about the jornal is available online at:
http://novtex.ru/prin/eng e-mail:prin@novtex.ru

УДК 004.4'233 DOI: 10.17587/prin.10.355-366

В. А. Галатенко, д-р физ.-мат. наук, зав. сектором, e-mail: galat@niisi.ras.ru,
К. А. Костюхин, канд. физ.-мат. наук, ст. науч. сотр., e-mail: kost@niisi.ras.ru,
Научно-исследовательский институт системных исследований Российской академии наук (НИИСИ РАН), Москва

Автоматический ремонт программ: базовые понятия и подходы*

Сформулированы базовые понятия, относящиеся к автоматическому ремонту программ, рассмотрены основные идеи и подходы. Тема эта весьма актуальна, поскольку современная технология программирования не позволяет делать крупные, сложные системы свободными от ошибок, а проявление этих ошибок при эксплуатации способно привести к тяжелым последствиям. К тому же по объективным причинам число ошибок в программах больше, чем разработчики могут исправить.

В настоящей статье представлена первая часть результатов исследований, посвященных автоматическому ремонту программ. В ней авторами предпринята попытка выделить перспективные идеи и подходы, которые уже сформированы на этом направлении, а также обозначить перспективы дальнейших исследований и прикладных разработок.

***Ключевые слова:** отладка, ремонт программ, восстановление программ, воспроизведение выполнения, самолечение, восстановление данных*

Особенности программ как объекта обработки

Программа — неудобный объект для исследования и обработки в целях ее тестирования. Теорема Райса [1] гласит, что любое нетривиальное семантическое свойство программ алгоритмически неразрешимо. Более точно: не существует эффективного универсального алгоритма проверки того, что вычислимая функция обладает заданным нетривиальным свойством.

Помимо алгоритмической неразрешимости, проблемой является сложность программ, как по числу составляющих их сущностей, так и по количеству связей между ними. Для больших программ переборные алгоритмы могут оказаться неприемлемыми в силу комбинаторного взрыва и, как следствие, потребуют большого времени для работы.

Пространство программ дискретно. Для программы не определено понятие малых изменений. Любое изменение влияет на корректность и на нефункциональные свойства, например, на время выполнения. Близость абстрактных синтаксических деревьев программ не означает близости поведения (трасс выполнения).

Перечисленные выше проблемные вопросы свидетельствуют о том, что для каждой программы доказательство правильности будет уникальным, как и сама программа. Следовательно, оно должно порождаться и поддерживаться параллельно с разработкой и сопровождением программы.

Программы имеют двоякую природу. Они не только служат инструкциями для ЭВМ, но и фиксируют знания в определенных предметных областях. Поэтому, чтобы обрабатывать и, в частности, преобразовывать и/или корректировать программы, нужно иметь как программистские, так и предметные знания.

Таким образом, с готовой программой, ввиду ее сложности и алгоритмической неразрешимости присущих ей проблем, извне ничего сделать нельзя — остается лишь ее выполнять, доверяя этой программе (точнее — ее авторам). Только сама программа знает свою семантику и может "позаботиться" о себе.

Постановка задачи ремонта программ

Существует широкий спектр постановок задачи ремонта программ — от реалистичных до модельных.

Реалистичная постановка задачи состоит в том, чтобы любая программа удовлетворяла спецификациям — явным и/или неявным. Если во время эксплуатации аппаратно-программной системы выявляется отклонение от спецификаций или опасность подобного отклонения, необходимо восстановить нормальный (безопасный) режим ее функционирования. Параллельно необходимо подготовить и

* Результаты исследований, представленные в публикации, выполнены в рамках государственного задания по проведению фундаментальных научных исследований по теме (проекту) "38. Проблемы создания глобальных и интегрированных информационно-телекоммуникационных систем и сетей, развитие технологий и стандартов GRID. Исследование и реализация программной платформы для перспективных многоядерных процессоров (0065-2019-0002)".

отправить в организацию, отвечающую за сопровождение системы, доклад о выявленных отклонениях. Отметим, что аварийное завершение — крайняя, но не единственно возможная форма отклонения.

Если в эксплуатирующей организации понимают, как в будущем избежать новых отклонений (или если программа сама знает, как себя вылечить), это, разумеется, нужно сделать. В свою очередь, специалисты, отвечающие за сопровождение, получив доклад об отклонениях, должны определить первопричину проблемы и устранить ее. Для этого им в первую очередь необходимо воспроизвести нештатную ситуацию. Сделать это может быть непросто, так как система, возможно, эксплуатируется на специфической аппаратной платформе в нестандартной конфигурации, а ее поведение может быть недетерминированным и/или зависеть от предыстории. Если проблемную ситуацию удалось воспроизвести, то соответствующий тест следует упростить, чтобы локализовать и найти ошибку. Когда ошибка найдена, разработчики должны ее исправить и убедиться в корректности исправлений, что также может оказаться непросто, особенно для платформно-зависимых, распределенных систем. Корректные исправления посылаются в эксплуатирующую организацию (с инструкцией по их реализации), где их должны внести сначала в тестовом режиме, а затем (при положительных результатах тестирования) в производственном. Такова реалистичная постановка задачи. Полная автоматизация процессов ее решения пока не рассматривается.

Модельная постановка задачи выглядит иначе. В качестве спецификаций для ее решения рассматривают тестовый набор. Имеется тестируемая программа и группа тестов, дающих отрицательные результаты, свидетельствующие о наличии ошибок. Следует исправить программу так, чтобы все тесты из используемого набора завершались успешно. Именно такая постановка рассматривается в большинстве публикаций по автоматическому ремонту программ.

Используемая терминология

Вслед за обзором [2] определим основные понятия предметной области "ремонт программ".

Неисправность (*failure*) — наблюдаемое неприемлемое поведение.

Отклонение (*error*) — нарастание некорректного состояния до неисправности (остающееся до времени незамеченным).

Дефект (*fault*) — первопричина ошибки (в частности, некорректный программный код).

Ошибка (*bug*) — собирательный термин для неисправности, отклонения и дефекта, расхождение между ожидаемым и наблюдаемым поведением программы. Отметим, что здесь в роли наблюдателя может выступать как человек, так и некая аппаратно-программная сущность.

Класс ошибок (*bug class*) — абстрактное понятие, обозначающее семейство ошибок, имеющих какие-либо общие характеристики (например, одинаковые проявления, одинаковые первопричины или одинаковые способы исправления).

Ремонт (*repair*) — преобразование неприемлемого поведения программы в приемлемое, соответствующее спецификациям, осуществляемое с помощью оператора вручную или автоматически с использованием специальных программ. В последнем случае говорят об **автоматическом ремонте** (*automatic repair*).

Ремонт поведения (*behavioral repair*) — изменение кода ремонтируемой программы и тем самым изменение ее последующего поведения. Может изменяться исходный или исполняемый код, эти изменения могут проводиться как в оперативном, так и в автономном режимах.

Программная коррекция, заплата (*patch*) — исправление, вносимое в программу.

Переподгонка (*overfitting*) — обеспечение положительного результата всех тестов из заданного тестового набора, возможно, в ущерб другим тестам или неявным спецификациям.

Ремонт состояния (*state repair*) — изменение состояния ремонтируемой программы и/или ее окружения. Ремонт состояния проводится в оперативном режиме, во время выполнения программы и может заключаться в перезагрузке, откатке, изменении конфигурации и т. п.

Оснащение программы (*program equipment*) — модификация исходного или бинарного кода программы для обеспечения возможности воздействия на нее внешних аппаратно-программных средств.

Постановка задачи автоматического ремонта поведения как задачи поиска

Вероятно, первой печатной работой, где автоматический ремонт программ на предмет исследования их поведения (далее — автоматический ремонт поведения) рассматривался как задача поиска, был доклад [3]. Однако следует подчеркнуть, что он остался практически незамеченным. Реальной отправной точкой исследований в этом направлении стали доклады [4, 5], а наиболее полное, систематическое изложение подхода содержится в работе [6].

Постановка задачи поиска выглядит следующим образом. Пусть имеется программа P, не удовлетворяющая спецификациям, и набор допустимых операций изменения программы (например, удаление и вставка операторов). Требуется найти последовательность операций, в результате применения которых программа P преобразуется в программу P', удовлетворяющую спецификациям.

В качестве очевидного отметим требования полноты набора операций изменения: для любой пары программ ($P1$, $P2$) должна существовать последовательность операций, переводящая $P1$ в $P2$.

Обычные методы поиска состоят в том, чтобы пробовать разные операции и выбирать те из них, которые приближают к цели. К числу таких методов принадлежат метод восхождения к вершине и генетическое программирование.

Поскольку целевая программа P' неизвестна, нет возможности напрямую оценить степень близости к ней. В качестве косвенной оценки в работах по автоматическому ремонту поведения используют

так называемую функцию годности (*fitness function*), которая характеризует степень некорректности текущего варианта программы, т. е. степень ее отклонения от спецификаций. Например, если в роли спецификаций выступает набор тестов, то степень некорректности может измеряться долей тестов с отрицательным результатом. Если очередное изменение эту долю уменьшает, значит оно является кандидатом на включение в искомую последовательность.

К сожалению, представленное выше предположение неверно. Из того, что последовательность изменений переводит неправильную программу в правильную, не следует, что по мере продвижения по последовательности имеет место приближение к цели. Последовательность изменений — единая транзакция, частичное выполнение которой может дать непредсказуемый результат. В частности, программа может стать неработоспособной. В пространстве программ нет возможности действовать методом последовательных приближений.

Попытки количественно измерять степень некорректности программы выглядят странно. Например, в работе [6] для программ, оперирующих целыми числами, предлагается в качестве такой меры использовать абсолютную величину разности между ожидаемым и реальным результатами. Это предложение означает, что если программа вычисления суммы чисел для выражения 1 + 1 дает результат 3, то это лучше, чем 100, полагает автор статьи [6].

Из систем автоматического ремонта поведения, решающих задачу поиска, наиболее известной и цитируемой является система GenProg [7, 8], использующая методы генетического программирования. Приведенные выше рассуждения показывают, что применение этих методов для автоматического ремонта программ не имеет обоснования. Иногда случайно может быть получен положительный результат, если последовательность изменений обладает свойством монотонности в смысле относительной корректности [9], однако в общем случае рассчитывать на это нельзя.

Теоретическая необоснованность поискового подхода к автоматическому ремонту поведения усугубляется недостоверностью практических результатов, представленных разработчиками системы GenProg в работе [8] применительно к программам на языке Си. Как показано в диссертации [10] (см. также работу [11]), на самом деле правдоподобных исправлений, т. е. исправлений, при которых все тесты из набора завершаются успешно, лишь 18. Вместе с тем первопричина недостоверности в неаккуратном программировании процесса тестирования, когда проверялся только код завершения, но не совпадение полученных результатов с ожидаемыми. Вследствие этого правдоподобным исправлением считалась программа из одного выполняемого оператора return 0.

Тестовый набор — слабая спецификация, поэтому правдоподобные исправления не обязательно являются корректными. Как показал "ручной" анализ результатов работы системы GenProg, корректных исправлений всего два и относятся они к "игрушечным" примерам менее чем из 30 строк.

Анализ систем автоматического ремонта поведения программ на языке Java дал аналогичные результаты [12]. Из 224 дефектов только 27 получили правдоподобные исправления системой jGenProg, лишь пять из которых оказались корректными.

Трактовка задачи автоматического ремонта поведения как задачи поиска представляется тупиковым направлением.

Определение места дефектов

Ремонт поведения программ начинается с определения места дефектов. В большинстве систем автоматического ремонта используется так называемая спектральная локализация дефектов [13]. Суть этого метода состоит в следующем. Имеется программа и набор тестов, часть из которых дает отрицательный результат. Программа разбивается на блоки, оснащается, и для каждого теста определяется, выполняется или нет каждый из блоков. Строка из нулей и единиц называется спектром теста, а совокупность строк — спектром тестового набора, отсюда и название метода. Для каждого блока также строится бинарный вектор, показывающий, в каких тестах блок выполнялся. Помимо этого строится бинарный вектор ошибок, единицы в котором соответствуют тестам с отрицательным результатом. Затем векторы блоков ранжируются по степени схожести с вектором ошибок. Максимальная схожесть соответствует максимальной вероятности наличия дефекта в этом блоке.

В работе [13] приведены эмпирические результаты, показывающие, что наилучшей из используемых для автоматического ремонта поведения мер сходства является коэффициент Охаи (Ochiai), заимствованный из биологии, применение которого позволяет исключить из числа подозрительных примерно 80 % операторов программы. Чтобы определить коэффициент Охаи, введем обозначения с двумя бинарными индексами, первый из которых показывает, какие блоки для подмножества тестов рассматриваются: те, что выполнялись (1), или те, что не выполнялись (0), а второй индекс — какое подмножество тестов имеется в виду: с отрицательным (1) или с положительным (0) результатом. Например, $a11(k)$ — это число тестов с отрицательным результатом, для которых блок k выполнялся. В этих обозначениях коэффициент Охаи вычисляется по формуле

$$s(k) = a11(k)/\mathrm{sqrt}((a11(k) + a01(k))(a11(k) + a10(k))).$$

Отметим, что для любого $k(a11(k) + a01(k))$ — это общее число тестов с отрицательным результатом, а $(a11(k) + a10(k))$ — число тестов, в которых блок k выполнялся.

Спектральная локализация дефектов вызывает много вопросов. Далее отметим некоторые из них.

Набор тестов изначально является слабой спецификацией, а тут еще теряется основная часть информации — результаты тестов. Остается только код завершения (успех/неудача). При ручном поиске места дефекта характер расхождения ожидаемых и полученных результатов теста является одним из

главных ориентиров. Например, если отклонение в поведении программы является следствием нефиксируемого целочисленного переполнения, то отличий в спектре выполняемых блоков может и не оказаться, просто будут получаться неверные результаты.

Поскольку не принимается во внимание какая-либо модель программы, тестирование ведется методом "черного ящика", поэтому точность определения места дефекта изначально ограничена.

Дефект разыскивается только среди выполняемых операторов, однако ошибочным может быть и описание: тип объекта или его инициализация, например, использование спецификатора short для целочисленной переменной, когда на самом деле требуется long.

Спектральная локализация предназначена для поиска места одиночных дефектов. Когда дефектов несколько, спектр оказывается размытым.

Даже для одиночных дефектов подозрительными могут оказаться до 40 % операторов программы [14]. Это, конечно, слишком много.

Разумеется, спектральная локализация применима только к детерминированным, последовательным программам.

Резюмируя перечисленные выше моменты, можно сделать вывод, что спектральная локализация является неадекватным методом определения места дефектов для систем автоматического ремонта поведения программ. Здесь еще раз целесообразно отметить, что без понимания смысла программы с ней в плане тестирования нельзя сделать ничего содержательного.

Пространство программных коррекций

В большинстве систем автоматического ремонта поведения программ применяется подход "порождай и проверяй".

В работе [15] рассмотрены ключевые характеристики пространства возможных программных коррекций для двух систем — SPR [16] и Prophet [17], реализующих этот подход. Рассмотрение ведется на примере ремонта восьми больших (от 62 тысяч до почти 3 млн строк) проектов с открытыми исходными текстами, содержащих в сумме 69 дефектов.

Системы SPR и Prophet, располагая ремонтируемой программой и тестовым набором, начинают свою работу с определения мест дефектов. Для этого применяется спектральная локализация с добавлением метки времени. Из операторов с одинаковой подозрительностью более приоритетным для исправления считается оператор с более поздней меткой времени.

Уже на этом первом этапе возникают проблемные вопросы. Реальный дефект может оказаться на 1926-м месте в списке подозрительных операторов, но если очень повезет, то "всего лишь" на 136-м. Это означает, что значительное время уходит на порождение и проверку заведомо ненужных исправлений.

После определения места дефекта (точнее, после ранжирования операторов по подозрительности) начинается порождение и проверка программных коррекций. У каждой системы автоматического ремонта поведения своя дисциплина порождения. Например, система GenProg поступает в соответствии с "гипотезой пластической хирургии" [18, 19], согласно которой для каждого неверного оператора в исходном тексте программы имеется правильный "двойник", который требуется лишь поставить на нужное место (возможно, сделав замену имен переменных). Читателю предлагается взглянуть на свои программы под этим углом зрения и вынести собственное суждение о справедливости гипотезы пластической хирургии. Авторам настоящей статьи гипотеза представляется сомнительной, применимой только к типовым операторам вроде "return 0" и к плохо структурированным программам. Конечно, возможность использования исходных текстов других проектов является очевидным обобщением гипотезы пластической хирургии.

Системы SPR и Prophet пытаются применять к дефектным местам параметризованные шаблоны преобразований. Например, к условию может быть добавлен (или из условия удален) конъюнкт или дизъюнкт, перед оператором может быть вставлено условие его выполнения (например, проверка указателя на неравенство NULL), в выражении может быть заменена переменная или константа и т. п. Новые переменные и константы берут из окружения места дефекта, а константы для сравнения на равенство/неравенство — из множества значений, которые принимает сравниваемая переменная при выполнении оснащенной версии программы.

Заранее заданное число (например, 100, 200 и т. д.) наиболее подозрительных мест и правила синтеза заплат определяют полное пространство программных коррекций. Если правильное исправление находится вне этого пространства, система автоматического ремонта поведения никогда его не найдет. Для систем SPR и Prophet в базовой конфигурации лишь для 19 дефектов из 69 правильная заплата попала в полное пространство. При этом для трех проектов, имеющих в сумме 22 дефекта, в полном пространстве не оказалось ни одной правильной заплаты.

Если увеличить число рассматриваемых подозрительных мест (например, до 2000) и расширить набор преобразований, можно добавить к 19 еще пять правильных заплат. Отметим, что для системы GenProg только одна заплата из 19 оказалась в полном пространстве (и была найдена).

Полное пространство слишком велико для полного перебора. Обычно для систем автоматического ремонта поведения устанавливают ограничение на время работы. Это сужает полное пространство программных коррекций до подпространства рассматриваемых кандидатов. Если правильная заплата окажется вне этого подпространства, система его не найдет, хотя в полном пространстве заплата присутствует.

Важной характеристикой исследуемой программы является отношение числа правдоподобных (успешно проходящих все тесты) и правильных заплат. Обычно в полном пространстве для каждого дефекта имеется не более одной правильной заплаты, в то время как правдоподобных — сотни и даже тысячи. Если набор тестов очень велик, число правдоподобных заплат,

приходящихся на одну правильную, измеряется десятками. Если при переборе заплат правдоподобная заплата попадется раньше правильной, она и будет выдана в качестве (некорректного) результата ремонта, заблокировав тем самым правильную. Это называется проблемой переподгонки, а приведенные данные характеризуют ее остроту.

Из этого факта следует несколько важных, перечисленных далее выводов.

Во-первых, если расширять полное пространство программных коррекций (например, путем увеличения числа рассматриваемых подозрительных мест и/или числа способов синтеза заплат), то в это расширенное пространство может попасть больше правдоподобных заплат, но число найденных правильных заплат может уменьшиться, поскольку больше времени уйдет на перебор "неправдоподобных" исправлений, а также в силу того, что правильная заплата окажется заблокированной правдоподобной (которых в расширенном пространстве тоже стало больше). Авторы работы [15] обратили внимание на этот парадокс.

Во-вторых, системы автоматического ремонта поведения, случайным образом перебирающие элементы пространства программных коррекций, такие как RSRepair [20, 21], несмотря на оптимистические заявления их авторов, не имеют практических шансов на успех. Причина в том, что многочисленные правдоподобные исправления заблокируют правильные.

В-третьих, желательно иметь дополнительные способы фильтрации правдоподобных заплат помимо набора тестов, чтобы бороться с проблемой переподгонки. Эвристики системы Prophet иногда справляются с этим, помещая правильные заплаты раньше прочих правдоподобных в списке рассматриваемых исправлений. Можно представить себе полуавтоматический ремонт поведения, когда система выдает несколько правдоподобных заплат, а разработчик использует их для отладки. К сожалению, как показывает исследование [22], некачественные программные коррекции — это хуже, чем их отсутствие: они скорее мешают, чем помогают, снижая производительность труда разработчиков.

В-четвертых, имеет смысл вместо универсальных систем ремонта поведения, пытающихся исправлять любые дефекты, развивать специализированные системы, ориентированные на определенный класс дефектов и/или исправлений. В полном пространстве программных коррекций специализированных систем меньше элементов и, следовательно, больше шансов найти и выдать правильное исправление.

В целом анализ пространства программных коррекций для систем автоматического ремонта поведения показывает, что практические шансы на успешный ремонт невелики, даже если дефект один и ликвидируется однострочным исправлением.

Синтез программных коррекций

Копирование заплат из других мест программы или из других проектов — не единственно возможный способ ремонта поведения. Можно попытаться, действуя в духе автоматического синтеза программ, вывести спецификации для сомнительного участка кода, а затем синтезировать по этим спецификациям программную коррекцию. Такой подход называется семантическим ремонтом. Семантический ремонт Java-программ реализован в системе Nopol [23].

Система Nopol предназначена для исправления ошибочных условий в условных операторах и для вставки предусловий (таких, например, как проверка на отличие указателя от NULL) перед операторами, которые не всегда должны выполняться.

Исходными данными для системы Nopol являются ремонтируемая программа и набор тестов, по крайней мере один из которых заканчивается с отрицательным результатом. На основе тестового набора проводится спектральная локализация дефекта, и операторы ранжируются по степени подозрительности, определяющей порядок их рассмотрения. Если очередной рассматриваемый оператор является условным, делается попытка исправить его условие. Если оператор не является оператором ветвления, делается попытка снабдить его предусловием. Дальнейшие действия подразделяются на отмеченные ниже три этапа.

Этап 1 — подбор результата проверки условия, который ведет к успешному завершению тестов. Вообще говоря, при выполнении одного теста рассматриваемый оператор может выполняться несколько раз. При этом фигурирующее в нем условие может давать различные результаты. Однако в системе Nopol, чтобы избежать комбинаторного взрыва, налагается дополнительное ограничение: отыскиваются только коррекции, дающие при всех выполнениях заданного теста один и тот же результат. Если для рассматриваемого оператора это невозможно, Nopol переходит к рассмотрению следующего кандидата на ремонт. Если нужный результат подобрать удается, он называется ангельским. Действие по искусственному изменению состояния программы и установке ангельского значения называется ангельской локализацией коррекции. Для этого действия используются только тесты с отрицательным результатом. Отметим, что для разных тестов ангельские значения могут быть разными. Отметим также, что при ремонте условия рассматриваются два возможных ангельских значения — ИСТИНА и ЛОЖЬ, а для отсутствующего предусловия — только ЛОЖЬ (подозрительный оператор не должен выполняться).

Этап 2 — сбор ожидаемой трассировочной информации для синтеза коррекции рассматриваемого условия. Для этого используются оснащенные варианты всех тестов. Для тестов с отрицательным исходом ожидаемым является ангельское значение условия, для тестов с положительным исходом — реальный результат выполнения. Для синтеза отсутствующего предусловия от тестов с отрицательным исходом в этом месте ожидается ЛОЖЬ, для тестов с положительным исходом — ИСТИНА. Кроме этого, собираются значения локальных переменных, параметров метода и полей класса базовых типов, доступных в рассматриваемом месте. Здесь, в отличие от поиска ангельского значения условия, учитываются все

случаи выполнения оператора для каждого теста. Для указателей фиксируется отличие от NULL, для объектов — результаты методов опроса их состояния. Отметим, что эти методы не должны иметь аргументов и побочных эффектов и вводятся в систему Nopol вручную. Для последующего синтеза коррекций к собранным данным добавляются статические значения и константы 0, 1 и −1.

Этап 3 — синтез программной коррекции. Требуется построить выражение, вычисление которого при всех вариантах собранных значений дает ожидаемый результат. Для этого используется покомпонентный синтез с применением SMT-решателя. Если решение SMT-задачи найти удается, оно транслируется в выражение условия. В процессе синтеза могут использоваться операции сравнения, арифметические и логические операции. Например, константа 2 может быть синтезирована как 1 + 1. Nopol пытается действовать от простого к сложному, используя сначала только операции сравнения, и только в случае необходимости — прочие операции.

Система Nopol была протестирована с использованием реальных программ, содержащих в сумме 22 дефекта. Для 17 из них были синтезированы коррекции, три из которых оказались эквивалентны исправлениям, сделанным разработчиками. Конечно, требуется более обширное экспериментальное исследование. В то же время очевидны перечисленные далее ограничения системы Nopol, наложенные для того, чтобы обеспечивалась приемлемая эффективность работы.

Во-первых, требование неизменности ангельского значения в процессе выполнения одного теста делает невозможным ремонт условий в заголовках циклов. Но даже для условных операторов оно может оказаться слишком обременительным. Именно по этой причине не удалось исправить один из дефектов.

Во-вторых, поиск ангельского значения — небезопасное действие. Принудительная установка значения условного выражения может привести к зацикливанию программы и невозможности исправить дефект.

В-третьих, требование отсутствия аргументов у методов, используемых при конструировании условий, не позволяет употреблять даже обычные математические функции, такие, например, как abs(x). Вследствие этого не удалось исправить еще один дефект.

Можно сделать вывод, что ограничения системы Nopol значительно снижают ее ценность.

Система Angelix [24] осуществляет семантический ремонт программ на языке C (имеется адаптация Angelix для языка Java, получившая наименование JFIX [25]). Angelix обобщает Nopol по следующим направлениям:

- могут исправляться не только условные, но и целочисленные арифметические выражения;
- могут одновременно исправляться несколько выражений;
- ангельское значение не обязано быть постоянным для теста.

Наряду с ангельскими значениями вводится понятие ангельского маршрута, приводящего тест к успешному исходу. На маршруте могут присутствовать несколько вычислений одного выражения с разными результатами. Совокупность ангельских маршрутов составляет ангельский лес, являющийся спецификацией для синтеза программных коррекций.

Важной особенностью системы Angelix, способствующей ее масштабируемости, является контролируемое символьное выполнение, сочетающее символьное выполнение для рассматриваемых выражений и конкретное — для прочих частей программы.

Для синтеза коррекций используются инструментальные средства частичной максимальной выполнимости формул в теориях (*Partial Maximum Satisfiability Modulo Theories*, *Partial MaxSMT*). Ограничения подразделяются на жесткие и мягкие. Решение должно обеспечивать выполнение всех жестких и максимального числа мягких ограничений. Мягкие ограничения направлены на сохранение синтаксической близости с исходной программой, что должно способствовать борьбе с переподгонкой и с конструированием больших и сложных заплат.

К сожалению, как показано в диссертации [26], семантический ремонт подвержен тем же проблемам, что и подход "порождай и проверяй". Имеются в виду неточное определение места дефекта и переподгонка.

Система Angelix, будучи примененной к пяти группам маленьких программ из начального курса программирования, для двух из них не смогла синтезировать ни одной заплаты в силу неточного определения места дефекта. Там, где заплаты синтезировать удалось, уровень переподгонки в среднем составил около 80 %. Иногда выражения заменялись на конкретные исходные данные тестов (small = 6), иногда синтезировались условия с константным исходом ($n1 > n1$) и т. п. В целом работу системы Angelix следует оценить как неудовлетворительную.

Чтобы справиться с трудностями переподгонки, в работе [27] предложено сочетание семантических и синтаксических методов. Последние направлены на синтез простых заплат, максимально близких к исходному тексту. Возможно, этот подход окажется эффективнее, чем мягкие ограничения Angelix.

В работе [28] предлагается в качестве спецификации использовать не набор тестов, а эталонную реализацию. Подобная реализация может быть доступна "бесплатно", если ремонтируется программы, удовлетворяющие общепринятым стандартам. К их числу, например, относятся утилиты ОС Unix, следующие стандарту POSIX и имеющие многочисленные реализации. Кроме того, эталонная реализация может быть простой, неоптимизированной, например, прямолинейная программа сортировки. Тем не менее в большинстве реальных ситуаций эталонная реализация отсутствует, а ее создание является излишне трудоемким.

Более реалистичной и перспективной представляется идея параметризации тестов, когда в тесте фигурируют не конкретные исходные данные, а множества данных, удовлетворяющие определенным ограничениям. Символьное выполнение способно поддержать интерпретацию подобных тестов при синтезе коррекций. Можно надеяться, что параметризованные тесты снизят степень переподгонки.

Результативность средств автоматического ремонта существенно зависит от набора ремонтируемых программ. Применительно к набору Mockito [29] система Nopol, выбранная как представитель семантического подхода, смогла исправить лишь один дефект из 38.

На листинге 1 показана одна из неверных (наложенных не в том месте) заплат, синтезированная системой Nopol.

```
   if (!((actual != null) && (actual != null))) { //
Заплата от Nopol
       return actual == null;
```

Листинг 1. Пример заплаты, синтезированной системой Nopol

Представляется, что ту же мысль можно было бы выразить проще.

Справедливости ради отметим, что системы SimFix [30] и CapGen [31], реализующие подход "порождай и проверяй", не смогли исправить ни одного дефекта из набора Mockito.

В обзоре [32] можно найти краткие описания множества других систем автоматического ремонта поведения.

В работе [33] приведено аккуратное сравнение 14 систем автоматического ремонта поведения программ на языке Java и анализ того, как влияет (не) точность определения места дефектов на (не)результативность работы этих систем. Нельзя исправить то, что не удается найти.

Обработка данных, ориентированная на приемлемость

Поведение программной системы может быть некорректным (не соответствовать спецификациям), но приемлемым для пользователей. Например, сгенерированные изображения могут содержать дефекты, которых пользователь не замечает. Если к системе предъявляются высокие требования по доступности, то целесообразно не прерывать выполнение и не пытаться искать причины некорректности, а продолжать работу, оставаясь в рамках приемлемости. Такова основная идея обработки данных, ориентированной на приемлемость (*Acceptability-Oriented Computing*) [34, 35]).

Помимо спецификаций корректности для программной системы могут быть заданы спецификации приемлемости. Очевидно, корректные состояния являются подмножеством приемлемых. Одно из предположений состоит в том, что приемлемую программу написать проще, чем корректную. Значит, можно сэкономить время и ресурсы, чтобы уделить больше внимания критически важным компонентам программной системы, требующим корректности.

Приемлемость поведения программной системы необходимо контролировать. Следовательно, нужен компонент мониторинга. Если мониторинг обнаруживает угрожающие отклонения в поведении программы, ее нужно вернуть в рамки приемлемости. Следовательно, нужен компонент "принуждения к приемлемости". Кроме того, нужен компонент протоколирования, фиксирующий угрожающие события и принятые меры к исправлению ситуации. Протокол может быть полезен для последующего поиска ошибок и их исправления, если и когда это будет признано целесообразным.

Принуждение к приемлемости может принимать различные формы. Это может быть безопасное завершение или ремонт состояния, позволяющий продолжить выполнение. Примером средства для ремонта состояния может служить утилита fsck ОС семейства Unix, устраняющая неисправности в файловых системах. Для принуждения к приемлемости также могут использоваться фильтры входных и/или выходных данных. Если известно, что какие-то исходные данные (например, слишком длинная текстовая строка) делают поведение программы неприемлемым, такие данные можно отбросить или заменить. Аналогично можно поступить с неприемлемым выводом. Механизм конвейеров в ОС семейства Unix позволяет реализовать подобные действия путем встраивания новых фильтров и/или преобразователей.

Практическим примером применения обработки данных, ориентированной на приемлемость, может служить работа [36]. Web-приложения, управляемые событиями, — это стандартный современный стиль программирования. Обработчики системных и пользовательских событий выполняются асинхронно, что может порождать в параллельных программах неисправности "нарушение порядка". Например, возможны попытки использовать объекты до их создания и инициализации. Искать и исправлять подобные дефекты параллелизма сложно, можно привнести новые ошибки. Проще и надежнее ограничить недетерминизм, планируя запуск обработчиков событий таким образом, чтобы нарушения порядка не проявлялись.

В работе [36] рассматривается система EventRace Commander, предназначенная для "принуждения к приемлемости" web-приложений, созданных с использованием сценариев на JavaScript. Код JavaScript оснащается, после чего события перехватываются контроллером, а обработка некоторых из них в соответствии с установленной политикой планирования, откладывается или отменяется. Оказывается, для исправления большинства ошибок нарушения порядка в данном случае достаточно универсальной политики планирования, в соответствии с которой:

- пользовательские и системные события откладываются до завершения загрузки всех статически определенных сценариев;
- пользовательские события откладываются до завершения асинхронной (динамической) инициализации;
- в промежутке между порождением асинхронного события (например, запроса к серверу) и получением ответа все пользовательские события отбрасываются;
- ответы на запросы к серверу обслуживаются в порядке очереди (т. е. в том же порядке, в котором проводились запросы).

Перечисленные правила отражают неявные предположения разработчиков об атомарности и очередности событий, которые в общем случае неверны.

В работе [36] проанализированы web-приложения с сайтов 20 крупнейших компаний из списка Fortune 500. В них было обнаружено 117 потенциальных нарушений порядка. С помощью универсальных политик планирования система EventRaceCommander смогла предотвратить 94 из них, сохранив удобство пользовательского интерфейса на приемлемом уровне. Это очень хороший результат, достигнутый простыми средствами.

Несмотря на успехи, представленные в работе [36], идея обработки данных, ориентированной на приемлемость, вызывает много вопросов. Спецификации приемлемости могут оказаться фрагментарными, неполными, т. е. при эксплуатации могут возникать новые проблемные вопросы. Эти спецификации зависят не только от вызываемой, но и от вызывающей системы, т. е. реализация мониторинга и принуждения к приемлемости для одной вызываемой, но разных вызывающих систем могут различаться, что влечет дополнительные сложности и трудозатраты. Если меняется версия вызываемой системы, могут меняться и спецификации приемлемости, что также приводит к необходимости обновления версий вызывающих систем. Средства мониторинга и принуждения к приемлемости могут быть сложными (содержащими ошибки) и даже в предположении безошибочности не дают гарантированных результатов.

Обработка данных со сглаживанием неисправностей

Обработка данных со сглаживанием неисправностей (*Failure-oblivious Computing*) может трактоваться как частный случай ориентации на приемлемость. Однако это очень важный, содержательный частный случай со своей системой понятий, подходами и готовыми решениями.

Чаще всего сглаживанию подвергается выход за границу массива. Если вне допустимого диапазона оказывается индекс отдельного элемента, то запись по такому индексу можно просто отбросить, а в качестве результата чтения выдать какой-либо элемент массива (например, один из крайних) или некое стандартное значение [37]. Более сложное решение состоит в сохранении записываемого значения в хэш-таблице и его выдаче при последующем обращении по тому же индексу на чтение.

Если выход за границу массива происходит при групповых операциях (например, с текстовыми строками), это называется переполнением буфера (оно возможно как на запись, так и на чтение) и сглаживается путем отсечения "лишних" элементов. Применительно к языку C этот подход детально исследован в работе [38].

Основная идея состоит в том, чтобы путем оснащения кода иметь возможность во время выполнения выяснять размеры массивов (буферов). Функция _size_right () принимает в качестве аргумента указатель и выдает в качестве результата число доступных байт справа от указуемого места (или 0, если таковых нет).

Для функций из библиотеки libc реализуются обертки, выполняющие отсечение. Пример подобной обертки для функции `memcpy ()` приведен на листинге 2. Макрос `ORIGINAL ()` возвращает указатель на стандартный вариант своего аргумента.

```
void *memcpy (void *dest, const void *src, size _ t n)
{
  ssize _ t dstsz = _ size _ right (dest);
  ssize _ t srcsz = _ size _ right (src);
  size _ t len = n;

  if (dstsz < len)
    len = dstsz;
  if (srcsz < len)
    len = srcsz;
  return ORIGINAL (memcpy) (dest, src, len);
}
```

Листинг 2. Пример сглаживающей обертки для функции `memcpy ()`

Функция `_ size _ right ()` может быть реализована несколькими способами, например, на основе AddressSanitizer в LLVM (так называемый санитайзер адресов — модуль компилятора LLVM, выявляющий потенциальные ошибки работы с памятью) или Pointer Bounds Checker в GCC (аналогичный модуль компилятора GCC). При этом для некоторых аппаратных платформ приведенные программные средства могут использовать встроенные средства контроля памяти, в частности, для платформы Intel это технология Intel Memory Protection Extensions (MPX). Накладные расходы в последнем случае невелики (около нескольких процентов).

Эксперименты с известными уязвимостями показали, что в четырех из пяти случаев сглаживание неисправностей позволило продолжить выполнение без неприемлемых последствий. Лишь в одном случае (см. листинг 3) реализованных средств оказалось недостаточно.

```
for (p = image->directory; *p != \0; p + +) {
  q = p;
  while ((*q! = \n) && (*q != \0))
    q++;
  (void) strncpy (image _ info->filename, p, q – p);
  image _ info->filename [q-p] = \0;
  p = q;
  ...
}
```

Листинг 3. Пример неисправности, сгладить которую не удалось

Причиной неисправности здесь может стать слишком длинное имя каталога. Обертка для `strncpy ()` с этим справится, но запись завершающего нулевого байта описанным подходом не сглаживается, что лишний раз подчеркивает важность концептуальной целостности.

В работе [39] исследованы различные способы сглаживания неисправности — обращения по пустому указателю. Предложены следующие варианты:

- подмена пустого указателя указателем на существующий или вновь созданный объект;

- пропуск оператора, осуществляющего доступ по пустому указателю;
- пропуск остатка метода и возврат пустого указателя или указателя на существующие или вновь созданные объекты.

Не все рассматриваемые способы делают продолжение выполнения приемлемым, их успех зависит от контекста. Если неисправностей несколько, то, вообще говоря, требуется перебор возможных последовательностей сглаживающих действий. Конечно, пространство поиска здесь существенно меньше, чем при генерации программных коррекций.

Система RCV [40] ориентирована на сглаживание таких неисправностей, как деление на 0 и обращение по пустому указателю, проявляющихся при выполнении на аппаратно-программной платформе x86/Linux. Основная идея состоит в том, чтобы перехватывать и обрабатывать сигналы SIGFPE и SIGSEGV, возвращать нулевой результат и/или пропускать запись или вызов по адресу, близкому к нулю, после чего присоединиться к сглаживаемому процессу и продолжить его выполнение, не допуская, чтобы сфабрикованные результаты повлияли на хранимые данные. Например, системные вызовы со сфабрикованными аргументами пропускаются. Когда сфабрикованные результаты перестанут существовать (например, ввиду ликвидации содержащих их объектов), происходит отсоединение от процесса, который далее выполняется в обычном режиме.

Система RCV была проверена на 18 неисправностях в семи реальных приложениях, например, таких как LibreOffice. В 17 случаях было обеспечено продолжение выполнения с приемлемыми результатами. Еще в одном случае выполнение аварийно завершалось вследствие присутствовавших в программе проверок (*assertions*). Если их отключить, то и это, последнее приложение продолжило бы приемлемое выполнение. В 13 из 18 случаев сфабрикованные результаты в конце концов "испарялись", что позволяло отсоединиться от сглаживаемого процесса. Последующий ручной анализ исходных текстов показал, что в 11 из 18 случаев RCV и программные коррекции разработчиков обеспечивали эквивалентные результаты для всех исходных данных.

Поскольку до присоединения и после отсоединения система RCV не влияет на выполнение сглаживаемого приложения, накладные расходы имеют место только во время сопровождения сфабрикованных результатов. Эти результаты обычно "испаряются" за несколько десятков секунд, так что в целом RCV не оказывает неприемлемого влияния на выполнение приложения.

Приведенные результаты (которые, несомненно, следует считать положительными, хотя и не очень представительными), вероятно, объясняются тем обстоятельством, что сглаживаемые приложения обрабатывали исходные данные порциями (например, такими как сетевые пакеты), слабо зависящими друг от друга, поэтому неисправность при обработке одной порции не влияла на обработку следующих порций. Кроме того, проявление неисправностей свидетельствовало о некорректности исходных данных, поэтому заплаты от разработчиков сводились к "санитарным" проверкам и пропуску обработки порции. Сопровождение сфабрикованных результатов в RCV — это сложный способ сделать то же самое. Систему RCV можно оценить как удачное специализированное средство сглаживания неисправностей.

Система Ares [41] предназначена для сглаживания неисправностей — непредвиденных необрабатываемых исключительных ситуаций в Java-программах. Ares встраивается в виртуальную Java-машину, чтобы перехватывать и обрабатывать исключительные ситуации. Если такие ситуации возникают, Ares анализирует текущий стек вызовов и применяет следующие стратегии сглаживания:

- использование присутствующих в стеке обработчиков других исключительных ситуаций;
- игнорирование исключительной ситуации и возврат из метода с каким-либо предопределенным результатом (например, 0 для арифметических типов или null для ссылочных).

Исследуется не только верхний кадр стека, но весь контекст. Это означает, в частности, что может выполниться возврат сразу из нескольких методов с соответствующим сокращением стека. Механизмы Ares направлены на поиск в стеке вызовов метода с типом void, чтобы избежать трудностей с выбором сфабрикованного результата.

Важной качественной характеристикой системы Ares является проведенная апробация нескольких возможных способов сглаживания, которое проводится в виртуальной среде (песочнице) средствами Java PathFinder. Отметим, что выбранное с использованием набора эвристик сглаживание не оформляется в виде постоянной заплаты. Каждый раз ее синтез проводится заново, поскольку может измениться контекст.

Механизмы Ares прошли испытания на 43 традиционных приложениях (удалось обеспечить приемлемое продолжение выполнения для 31 из них) и 9 мобильных (на платформе Android, восемь случаев приемлемого продолжения). Успех здесь, как и для системы RCV, во многом определяется характером приложений, их требованиями приемлемости.

Самолечение программ

Самолечение программ — обязательный элемент автономных вычислений в случае невозможности получить помощь извне, тогда программа должна сама обеспечить приемлемое продолжение процесса функционирования, возможно, ценой усечения функциональных возможностей. В подобном положении оказываются мобильные приложения, когда пользователю нужно немедленно что-то сделать (например, позвонить, несмотря на проявившуюся неисправность). Этой теме посвящена работа [42]. Рассматриваются мобильные приложения на платформе Android. Неисправности в таких приложениях могут быть временными или постоянными. Примеры временных неисправностей — нехватка ресурсов или потеря сетевого соединения. Примеры постоянных неисправностей — исключительная ситуация как следствие ошибки в программе или отсутствие прав доступа. В фоновом режиме функционирует система мониторинга, которая анализирует регистрационную информацию и выявляет неисправности.

Мобильные приложения предоставляют пользователю последовательность экранов графического интерфейса. Статический анализ байт-кода приложения позволяет построить граф переходов между этими экранами.

Если при выполнении какого-либо метода фиксируется неисправность, то в первую очередь выясняется ее тип — временная или постоянная. Если неисправность временная, делается попытка продолжить выполнение с места неисправности в надежде, что сетевая связность восстановилась или что операционная система освободила требуемые ресурсы и т. п. Если неисправность постоянная, то метод, в котором она проявилась, "запечатывается": его код помещается в блок try, а в блоке catch записывается возврат из метода. Эти действия требуют перезаписи байт-кода приложения. После этого происходит возврат к состоянию (экрану графического интерфейса), признанному безопасной точкой отката, и выполнение приложения возобновляется. Отметим, что усечение функциональных возможностей (ранний возврат из запечатанного метода) будет иметь место, только если неисправность снова проявится. На нормальную работу программная коррекция не повлияет.

Идея самолечения программ представляется весьма перспективной. Она может учитываться при разработке приложений в рамках концепции контролируемого выполнения [43]. Это требует от разработчика дополнительных усилий, однако дает новое качество, которое со временем может стать обязательным.

Заключение

Рассмотрены базовые идеи и подходы, лежащие в основе исследований, посвященных автоматическому ремонту программ. Эти идеи и подходы оказываются далеко не равноценными.

Существующие подходы имеют весьма ограниченный спектр применимости, что свидетельствует о необходимости дальнейших исследований. В качестве таковых хотелось бы выделить самолечение программ. Автономные вычисления становятся все более важными и распространенными, и с этой точки зрения у самолечения нет альтернативы.

Список литературы

1. **Rice H. G.** Classes of recursively enumerable sets and their decision problems // Transactions of the American Mathematical Society. 1953. Vol. 74, No. 2. P. 358—366.
2. **Monperrus M.** Automatic Software Repair: A Bibliography // ACM Computing Surveys. 2018. Vol. 51, Issue 1. Article 17. 24 p.
3. **Arcuri A.** On the Automation of Fixing Software Bugs // Proc. of ICSE'08. May 10—18, 2008, Leipzig, Germany, 2008. P. 1003—1006.
4. **Weimer W., Nguyen T. V., Le Goues C., Forrest S.** Automatically Finding Patches Using Genetic Programming // Proc. of ICSE'09. May 16—24, 2009, Vancouver, Canada, 2009. P. 364—374.
5. **Forrest S., Nguyen T. V., Weimer W., Le Goues C.** A Genetic Programming Approach to Automated Software Repair // Proc. of GECCO'09. July 8—12, 2009, Montral Qubec, Canada, 2009. P. 947—954.
6. **Arcuri A.** Evolutionary repair of faulty software // Applied Soft Computing. 2011. Vol. 11. P. 3494—3514.
7. **Le Goues C., Nguyen T. V., Forrest S., Weimer W.** GenProg: A Generic Method for Automatic Software Repair // IEEE Transactions on Software Engineering. 2012. Vol. 38, No. 1. P. 54—72.
8. **Le Goues C., Dewey-vogt M., Forrest S., Weimer W.** A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each // Proc. of ICSE 2012. Zurich, Switzerland. 2012. P. 3—13.
9. **Khaireddine B., Zakharchenko A., Mili A.** A Generic Algorithm for Program Repair // Proc. of 2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE), 2017. P. 65—71.
10. **Long F.** Automatic Patch Generation via Learning from Successful Human Patches. Massachusetts Institute of Technology, 2018. 296 p.
11. **Qi Z., Long F., Achour S., Rinard M.** An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems // Proc. of ISSTA'15, July 13—17, 2015. Baltimore, MD, USA. 2015. P. 24—36.
12. **Martinez M., Durieux T., Sommerard R.** et al. Automatic Repair of Real Bugs in Java: A Large-Scale Experiment on the Defects4J Dataset // Empirical Software Engineering, Springer Verlag. 2017. Vol. 22, Issue 4. P. 1936—1964.
13. **Abreu R., Zoeteweij P., van Gemund A. J. C.** On the Accuracy of Spectrum-based Fault Localization // Proc. of Testing, Academic and Industrial Conference Practice And Research Techniques. 10—14 Sept. 2007, Windsor, UK. 2007. P. 89—98.
14. **Abreu R., Zoeteweij P., van Gemund A. J. C.** An Evaluation of Similarity Coefficients for Software Fault Localization // Proc. of 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06). 18—20 Dec. 2006, Riverside, CA, USA. 2006. P. 39—46.
15. **Long F., Rinard M.** An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems // Proc. of IEEE/ACM 38th IEEE International Conference on Software Engineering ICSE'16. May 14—22, 2016, Austin, TX, USA, 2016. P. 702—713.
16. **Long F., Rinard M.** Staged Program Repair with Condition Synthesis // Proc. of ESEC/FSE'15, August 30—September 4, 2015, Bergamo, Italy, 2015. P. 166—178.
17. **Long F., Rinard M.** Automatic Patch Generation by Learning Correct Code: Proc. of POPL'16, January 20—22, 2016, St. Petersburg, FL, USA, P. 298—312.
18. **Barr E. T., Brun Y., Devanbu P., Mark Harman, Sarro F.** The Plastic Surgery Hypothesis // Proc. of FSE'14. November 16—21, 2014, Hong Kong, China, 2014. P. 306—317.
19. **Martinez M., Weimer W., Monperrus M.** Do the Fix Ingredients Already Exist? An Empirical Inquiry into the Redundancy Assumptions of Program Repair Approaches // Proc. of ICSE Companion'14. May 31—June 7, 2014, Hyderabad, India, 2014. P. 492—495.
20. **Yuhua Qi, Xiaoguang Mao, Yan Lei** et al. Does Genetic Programming Work Well on Automated Program Repair // Proc. of 2013 International Conference on Computational and Information Sciences, 2013. P. 1875—1878.
21. **Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, Chengsong Wang.** The Strength of Random Search on Automated Program Repair // Proc. of ICSE'14. May 31—June 7, 2014, Hyderabad, India. 2014. P. 254—265.
22. **Yida Tao, Jindae Kim, Sunghun Kim, Chang Xu.** Automatically Generated Patches as Debugging Aids: A Human Study // Proc. of FSE'14, November 16—21, 2014, Hong Kong, China, 2014. P. 64—74.
23. **Xuan J., Martinez M., DeMarco F.** et al. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs // IEEE Transactions On Software Engineering, 2017. Vol. 43, No. 1. P. 34—55.
24. **Mechtaev S., Yi J., Roychoudhury A.** Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis // Proc. of 2016 IEEE/ACM 38th IEEE International Conference on Software Engineering ICSE'16. May 14—22, 2016, Austin, TX, USA. 2016. P. 691—701.
25. **Le X.-B. D., Chu D.-H., Lo D.** et al. JFIX: Semantics-Based Repair of Java Programs via Symbolic PathFinder // Proc. of 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA'17, July 10—14, 2017 Santa Barbara, CA, USA, 2017. P. 376—379.
26. **Le X.-B. D.** Overfitting in Automated Program Repair: Challenges and Solutions. School of Information Systems, Singapore Management University, 2018. 151 p.
27. **Le X.-B. D., Chu D.-H., Lo D.** et al. S3: Syntax- and Semantic-Guided Repair Synthesis via Programming by Examples // Proc. of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'17. September 4—8, 2017, Paderborn, Germany. 2017, P. 593—604.
28. **Mechtaev S., Nguyen M.-D., Noller Y.** et al. 2018. Semantic Program Repair Using a Reference Implementation // Proc. of 40th International Conference on Software Engineering, ICSE'18. May 27—June 3, 2018 Gothenburg, Sweden. 2018. P. 129—139.
29. **Wang S., Wen M., Mao X., Yang D.** Attention Please: Consider Mockito when Evaluating Newly Proposed Automated Program Repair Techniques. URL: https://arxiv.org/abs/1812.05370 (дата обращения 24.06.2019).
30. **Jiang J., Xiong Y., Zhang H.** et al. Shaping Program Repair Space with Existing Patches and Similar Code // Proc. of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18), New York, NY, USA. 2018. P. 298—209.

31. **Wen M., Chen J., Wu Rn.** et al. Context-Aware Patch Generation for Better Automated Program Repair // Proc. of 40th International Conference on Software Engineering. May 27—June 3, 2018, Gothenburg, Sweden, 2018. P. 1—11.
32. **Gazzola L., Micucci D., Mariani L.** Automatic Software Repair: A Survey // IEEE Transactions On Software Engineering. 2019. Vol. 45, No. 1. P. 34—67.
33. **Liu K., Koyuncu A., Bissyand T. F.** et al. You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems. URL: https://arxiv.org/pdf/1812.07283.pdf (дата обращения 24.06.2019).
34. **Rinard M.** Acceptability-Oriented Computing // Proc. of OOPSLA'03. October 26—30, 2003, Anaheim, California, USA. 2003. P. 57—75.
35. **Rinard M., Cadar C., Nguyen H. H.** Exploring the Acceptability Envelope // Proc. of OOPSLA'05. Oct. 16—20, 2005, San Diego, CA, USA, 2005. P. 21—29.
36. **Adamsen C. Q. Moller A., Karim R.** et al. Repairing Event Race Errors by Controlling Nondeterminism // Proc. of 2017 IEEE/ACM 39th International Conference on Software Engineering. 2017. P. 289—299.
37. **Rinard M., Cadar C., Dumitran D.** et al. Enhancing Server Availability and Security Through Failure-Oblivious Computing: Proc. of OSDI'04: 6th Symposium on Operating Systems Design and Implementation, P. 303—316.
38. **Rigger M., Pekarek D., Mssenbock H.** Context-Aware Failure-Oblivious Computing as a Means of Preventing Buffer Overflows // Proc. of Network and System Security, 12th International Conference, NSS 2018, August 27—29, 2018. Hong Kong, China. 208. P. 376—390.
39. **Durieux T., Hamadi Y., Yu Z.** et al. Exhaustive Exploration of the Failure-oblivious Computing Search Space // Proc. of 11th International Conference on Software Testing, Verification and Validation, Vasteras, Sweden, April 9—13, 2018. P. 139—149.
40. **Long F., Sidiroglou-Douskos S., Rinard M.** Automatic Runtime Error Repair and Containment via Recovery Shepherding // Proc. of PLDI'14, June 9—11, 2014. Edinburgh, United Kingdom, 2014. P. 227—238.
41. **Gu T., Sun Ch., Ma X.** et al. Automatic Runtime Recovery via Error Handler Synthesis // Proc. of ASE'16. September 3—7, 2016, Singapore, 2016. P. 684—695.
42. **Azim T., Neamtiu I., Marvel L.** Towards Self-healing Smartphone Software via Automated Patching // Proc. of ASE'14. September 15—19, 2014, Vasteras, Sweden, 2014. P. 623—628.
43. **Галатенко В. А., Костюхин К. А., Шмырев Н. В.** Контролируемое выполнение/Под ред. акад. РАН В. Б. Бетелина. М.: ФГУ ФНЦ НИИСИ РАН, 2012. 160 с.

Automated Program Repair: Basic Concepts and Approaches

V. A. Galatenko, galat@niisi.ras.ru, **K. A. Kostyukhin**, kost@niisi.ras.ru,
Federal State Institution "Scientific Research Institute for System Analysis of the Russian Academy of Sciences", Moscow, 117218, Russian Federation

Corresponding author:

Kostyukhin Konstantin A., Senior Researcher, Federal State Institution "Scientific Research Institute for System Analysis of the Russian Academy of Sciences", Moscow, 117218, Russian Federation
E-mail: kost@niisi.ras.ru

Received on July 09, 2019
Accepted on August 05, 2019

The article formulates the basic concepts related to the automated program repair. It also discusses the main ideas and approaches. This topic is very relevant, because modern programming technology does not allow to make large, complex systems free from errors. The occurring of these errors during program execution can lead to serious consequences. In addition, the number of errors in programs is greater than the developers can fix. The Article is the first part of the work devoted to the automated program repair and is mainly theoretical.

Unfortunately, a program is a very inconvenient object for processing, and this creates fundamental difficulties for the implementation of automated repair. A number of the proposed approaches have no theoretical basis, and the practical results of their application are unsatisfactory. There is no effective universal algorithm to verify that a computable function has a given nontrivial property.

In addition to algorithmic insolubility, the problem is the complexity of programs, both in the number of their constituent entities, and the number of relationships between them. For large programs, overdriven algorithms may not be acceptable because of the combinatorial explosion and, as a consequence, too much time.

The program space is discrete. The concept of small changes is not defined for the program. Any change affects correctness and non-functional properties, such as runtime. The proximity of programs abstract syntax trees does not mean the proximity of program behavior (execution traces).

This means that for each program, the proof of correctness will be as unique as the program itself. It must therefore be generated and maintained in parallel with the development and maintenance of the program.

Programs have a dual nature. They not only serve as instructions for computers, but also record knowledge in certain subject areas. Therefore, in order to process and, in particular, transform and/or correct programs, it is necessary to have both programming and subject knowledge.

Thus, with the deployed program, because of its complexity and the algorithmic insolubility of its inherent problems, nothing can be done from the outside — it remains only to perform, trusting it (or rather — its authors). Only the program itself knows its semantics and can take care of itself.

The article attempts to highlight promising ideas and approaches, as well as directions for further research and development.

***Keywords**: debugging, automated program repair, program recovery, program re-execution, program self-treatment, data recovery*

For citation:

Galatenko V. A., Kostyukhin K. A. Automated Program Repair: Basic Concepts and Approaches, *Programmnaya Ingeneria*, 2019, vol. 10, no. 9—10, pp. 355—366.

DOI: 10.17587/prin.10.355-366

References

1. **Rice H. G.** Classes of recursively enumerable sets and their decision problems, *Transactions of the American Mathematical Society*, 1953, vol. 74, no. 2, pp. 358—366.
2. **Monperrus M.** Automatic Software Repair: A Bibliography, *ACM Computing Surveys,* 2018, vol. 51, issue 1, Article 17, 24 p.
3. **Arcuri A.** On the Automation of Fixing Software Bugs, *Proc. of ICSE'08*, May 10—18, 2008, Leipzig, Germany, pp. 1003—1006.
4. **Weimer W., Nguyen T. V., Le Goues C., Forrest S.** Automatically Finding Patches Using Genetic Programming, *Proc. of ICSE'09,* May 16—24, 2009, Vancouver, Canada, 2009, pp. 364—374.
5. **Forrest S., Nguyen T. V., Weimer W., Le Goues C.** A Genetic Programming Approach to Automated Software Repair, *Proc. of GECCO'09,* July 8—12, 2009, Montral Qubec, Canada, 2009. pp. 947—954.
6. **Arcuri A.** Evolutionary repair of faulty software, *Applied Soft Computing,* 2011, vol. 11, pp. 3494—3514.
7. **Goues C., Nguyen T. V., Forrest S., Weimer W.** GenProg: A Generic Method for Automatic Software Repair, *IEEE Transactions on Software Engineering,* 2012, vol. 38, no. 1, pp. 54—72.
8. **Goues C., Dewey-Vogt M., Forrest S., Weimer W.** A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each, *Proc. of ICSE 2012*, Zurich, Switzerland, 2012, pp. 3—13.
9. **Khaireddine B., Zakharchenko A., Mili A.** A Generic Algorithm for Program Repair, *Proc. of 2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE)*, 2017, pp. 65—71.
10. **Long F.** *Automatic Patch Generation via Learning from Successful Human Patches,* Massachusetts Institute of Technology, 2018, 296 p.
11. **Qi Z., Long F., Achour S., Rinard M.** An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems, *Proc. of ISSTA'15,* July 13—17, 2015, Baltimore, MD, USA, 2015, pp. 24—36.
12. **Martinez M., Durieux T., Sommerard R., Xuan J., Monperrus M.** Automatic Repair of Real Bugs in Java: A Large-Scale Experiment on the Defects4J Dataset, *Empirical Software Engineering*, 2017, vol. 22, issue 4, pp. 1936—1964.
13. **Abreu R., Zoeteweij P., van Gemund A. J. C.** On the Accuracy of Spectrum-based Fault Localization, *Proc. of Testing, Academic and Industrial Conference Practice And Research Techniques*, 10—14 Sept. 2007, Windsor, UK, 2007, pp. 89—98.
14. **Abreu R., Zoeteweij P., van Gemund A. J. C.** An Evaluation of Similarity Coefficients for Software Fault Localization, *Proc. of 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, 18—20 Dec., 2006, Riverside, CA, USA, 2006, pp. 39—46.
15. **Long F., Rinard M.** An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems, *Proc. of IEEE/ACM 38th IEEE International Conference on Software Engineering ICSE'16*, May 14—22, 2016, Austin, TX, USA, 2016, pp. 702—713.
16. **Long F., Rinard M.** Staged Program Repair with Condition Synthesis, *Proc. of ESEC/FSE'15,* August 30 — September 4, 2015, Bergamo, Italy, 2015, pp. 166—178.
17. **Long F., Rinard M.** Automatic Patch Generation by Learning Correct Code, *Proc. of POPL'16,* January 20—22, 2016, St. Petersburg, FL, USA, 2016, pp. 298—312.
18. **Barr E. T., Brun Y., Devanbu P., Harman M., Sarro F.** The Plastic Surgery Hypothesis, *Proc. of FSE'14*, November 16—21, 2014, Hong Kong, China, 2014, pp. 306—317.
19. **Martinez M., Weimer W., Monperrus M.** Do the Fix Ingredients Already Exist? An Empirical Inquiry into the Redundancy Assumptions of Program Repair Approaches, *Proc. of ICSE Companion'14*, May 31—June 7, 2014, Hyderabad, India, 2014, pp. 492—495.
20. **Qi Y., Mao X., Lei Y., Dai Z., Wang C.** Does Genetic Programming Work Well on Automated Program Repair, *Proc. of 2013 International Conference on Computational and Information Sciences,* 2013, pp. 1875—1878.
21. **Qi Y., Mao X., Lei Y., Dai Z., Wang C.** The Strength of Random Search on Automated Program Repair, *Proc. of ICSE'14,* May 31—June 7, 2014, Hyderabad, India, 2014, pp. 254—265.
22. **Tao Y., Kim J., Kim S., Xu C.** Automatically Generated Patches as Debugging Aids: A Human Study, *Proc. of FSE'14*, November 16—21, 2014, Hong Kong, China, 2014, pp. 64—74.
23. **Xuan J., Martinez M., DeMarco F., Clement M., Marcote L. S., Durieux T., Le Berre D., Monperrus M.** Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs, *IEEE Transactions On Software Engineering*, 2017, vol. 43, no. 1, pp. 34—55.
24. **Mechtaev S., Yi J., Roychoudhury A.** Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis, *Proc. of 2016 IEEE/ACM 38th IEEE International Conference on Software Engineering, ICSE'16*, May 14—22, 2016, Austin, TX, USA, 2016, pp. 691—701.
25. **Le X.-B. D., Chu D.-H., Lo D., Le Goues C., Visser W.** JFIX: Semantics-Based Repair of Java Programs via Symbolic PathFinder, *Proc. of 26th ACM SIGSOFT International Symposium on Software Testing and Analysis,* (ISSTA'17), July 10—14, 2017, Santa Barbara, CA, USA, 2017. P. 376—379.
26. **Le X.-B. D.** *Overfitting in Automated Program Repair: Challenges and Solutions,* School of Information Systems, Singapore Management University, 2018, 151 p.
27. **Le X.-B. D., Chu D.-H., Lo D., Le Goues C., Visser W.** S3: Syntax- and Semantic-Guided Repair Synthesis via Programming by Examples, *Proc. of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'17)*, September 4—8, 2017, Paderborn, Germany, 2017, pp. 593—604.
28. **Mechtaev S., Nguyen M.-D., Noller Y., Grunske L., Roychoudhury A.** 2018. Semantic Program Repair Using a Reference Implementation, *Proc. of 40th International Conference on Software Engineering (ICSE'18),* May 27—June 3, 2018, Gothenburg, Sweden, 2018, pp. 129—139.
29. **Wang S., Wen M., Mao X., Yang D.** Attention Please: Consider Mockito when Evaluating Newly Proposed Automated Program Repair Techniques, available at: https://arxiv.org/abs/1812.05370 (accessed 24.06.2019).
30. **Jiang J., Xiong Y., Zhang H., Gao Q., Chen X.** Shaping Program Repair Space with Existing Patches and Similar Code, *Proc. of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18),* New York, NY, USA, 2018, pp. 298—209.
31. **Wen M., Chen J., Wu R., Hao D., Cheung S.-C.** Context-Aware Patch Generation for Better Automated Program Repair, *Proc. of 40th International Conference on Software Engineering,* May 27—June 3, 2018, Gothenburg, Sweden, 2018, pp. 1—11.
32. **Gazzola L., Micucci D., Mariani L.** Automatic Software Repair: A Survey, *IEEE Transactions On Software Engineering*, 2019, vol. 45, no. 1, pp. 34—67.
33. **Liu K., Koyuncu A., Bissyand T. F., Kim D., Klein J., Traon Y. Le.** You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems, available at: https://arxiv.org/pdf/1812.07283.pdf (accessed 24.06.2019).
34. **Rinard M.** Acceptability-Oriented Computing, Proc. of OOPSLA'03, October 26—30, 2003, Anaheim, California, USA, P. 57—75.
35. **Rinard M., Cadar C., Nguyen H. H.** Exploring the Acceptability Envelope, *Proc. of OOPSLA'05*, Oct. 16—20 2005, San Diego, CA, USA, 2005, pp. 21—29.
36. **Adamsen C. Q., Moller A., Karim R., Sridharan M., Tip F., Sen K.** Repairing Event Race Errors by Controlling Nondeterminism, *Proc. of 2017 IEEE/ACM 39th International Conference on Software Engineering*, pp. 289—299.
37. **Rinard M., Cadar C., Dumitran D., Roy D. M., Leu T., Beebee W. S. Jr.** Enhancing Server Availability and Security Through Failure-Oblivious Computing, *Proc. of OSDI'04: 6th Symposium on Operating Systems Design and Implementation*, 2004, pp. 303—316.
38. **Rigger M., Pekarek D., Mssenbock H.** Context-Aware Failure-Oblivious Computing as a Means of Preventing Buffer Overflows, *Proc. of Network and System Security, 12th International Conference, NSS 2018*, August 27—29, 2018, Hong Kong, China, 2018, pp. 376—390.
39. **Durieux T., Hamadi Y., Yu Z., Baudry B., Monperrus M.** Exhaustive Exploration of the Failure-oblivious Computing Search Space, *Proc. of 11th International Conference on Software Testing, Verification and Validation*, April 9—13, 2018, Vasteras, Sweden, 2018, pp. 139—149.
40. **Long F., Sidiroglou-Douskos S., Rinard M.** Automatic Runtime Error Repair and Containment via Recovery Shepherding, *Proc. of PLDI'14,* June 9—11, 2014, Edinburgh, United Kingdom, 2014, pp. 227—238.
41. **Gu T., Sun C., Ma X., Lu J., Su Z.** Automatic Runtime Recovery via Error Handler Synthesis, *Proc. of ASE'16*, September 3—7, 2016, Singapore, Singapore, 2016, pp. 684—695.
42. **Azim T., Neamtiu I., Marvel L.** Towards Self-healing Smartphone Software via Automated Patching, *Proc. of ASE'14,* September 15—19, 2014, Vasteras, Sweden, 2014, pp. 623—628.
43. **Galatenko V. A., Kostiukhin K. A., Shmyrev N. V.** *Controlled execution*, Moscow, FGU FNC NIISI RAN, 2012, 160 p. (in Russian).

УДК 004.942+004.087 DOI: 10.17587/prin.10.367-376

В. А. Пономарев, ст. препод., vadim@cs.petrsu.ru,
Петрозаводский государственный университет

Имитационное моделирование показателей функционирования твердотельной системы хранения данных*

На основе разработанных оригинальных алгоритмов приведены результаты имитационного моделирования основных показателей производительности, надежности и стоимости функционирования системы хранения дедуплицированных данных на твердотельных носителях. Анализируется влияние входных управляющих параметров системы на выходные показатели функционирования, что позволяет предложить при проектировании оптимальные параметры и структуру системы хранения данных.

Ключевые слова: *имитационное моделирование, вычислительный эксперимент, твердотельные накопители, дедупликация, системы хранения данных, оптимизация*

Введение

Рост спроса на услуги по хранению данных отмечается многими аналитиками в этой области [1, 2]. В частности, в IV квартале 2018 г. российский рынок внешних системы хранения данных (СХД) вырос на 25,6 % по сравнению с показателями прошлого года [3].

Основными драйверами роста рынка СХД считаются финансовый и телекоммуникационный секторы, а также государственные органы власти [3]. Одновременно с ростом объемов данных растут требования к производительности систем хранения данных и их энергоэффективности [4].

Существенно снизить время доступа к данным и энергопотребление оказывается возможным с применением твердотельных накопителей, а также СХД на их основе. Твердотельные СХД актуальны, например, для корпоративных "облаков" с произвольным профилем нагрузки, который невозможно спрогнозировать, при соблюдении жестких требований к числу операций ввода-вывода и времени отклика [5]. Бесспорным преимуществом систем хранения данных на твердотельных накопителях (*solid-state drive*, SSD) является их высокая производительность [6]. Спрос на такое оборудование постоянно растет, а стоимость, наоборот, снижается, что делает твердотельные системы хранения данных все более доступными для потребителей [7, 8].

Основными параметрами качества функционирования СХД являются производительность, надежность, стоимость и длительность хранения, объем данных [9]. При этом недостатком твердотельных СХД является их ограниченный ресурс вследствие происходящего при операциях записи износа флеш-памяти. Компенсировать этот недостаток призвана технология дедупликации [9], исключающая при записи дублирующие копии повторяющихся данных.

Проведение натурных экспериментов, например, с помощью популярных инструментальных средств fio, IOmeter и им подобных [10, 11] является наиболее точным способом определения производительности системы и подбора правильных параметров [12]. Этот подход, однако, крайне затратен и не позволяет оценить надежность. В таких условиях естественно опираться при проектировании твердотельной СХД с механизмом дедупликации на ее математические и имитационные модели [9].

1. Родственные исследования

В работе [13] получены модели оценки скорости передачи данных для RAID-массивов HDD-дисков уровней 0, 1 и 5. Математические и имитационные модели надежности современных распределенных отказоустойчивых систем хранения данных представлены в работах [14, 15]. В них не учтены особенности твердотельных накопителей, в частности, ограничение ресурса по записи вследствие износа. Анализ надежности SSD RAID является сложной задачей, так как частота ошибок твердотельных накопителей зависит от времени.

В работах [16, 17] поступление ошибок моделируется как пуассоновский процесс с постоянной скоростью. Однако твердотельные накопители имеют увеличивающуюся скорость поступления ошибок, поскольку они изнашиваются с большим числом операций стирания. В работе [18] представлена имитационная модель производительности твердотельных накопителей, в которой для повышения

* Работа выполнена при поддержке Министерства науки и высшего образования РФ (соглашение № 14.580.21.0009, уникальный идентификатор RFMEFI58017X0009).

надежности применялась репликация данных. В работе [19] предложена аналитическая модель на основе "неоднородной" непрерывной модели цепи Маркова (CTMC model) для количественной оценки динамики надежности RAID-массивов на твердотельных накопителях. Проанализирован вопрос изнашивания твердотельных накопителей от битовых ошибок, частота которых зависит от времени и увеличивается по мере износа твердотельной памяти. Описано имитационное моделирование СХД на примере Diff-RAID и RAID-5.

В монографии [20] проведен анализ самых последних разработок в области моделирования работы твердотельных накопителей. Кроме параметризации твердотельных накопителей и свойств NAND флеш-памяти рассмотрена виртуальная платформа имитации SSDExplorer. Проблемы повышения производительности и надежности систем хранения, основанных на твердотельных накопителях, путем разработки RAID-массива с эластичным чередованием и повсеместной четностью были рассмотрены в работе [21].

Перечисленные выше работы, рассмотренные в них математические модели и программные механизмы имитации не в полной мере решают задачу моделирования процессов функционирования СХД на базе твердотельных накопителей. Они не охватывают в комплексе показатели производительности, надежности и стоимости хранения, а также не ориентированы на механизмы дедупликации, которые в настоящее время востребованы при формировании предложений на рынке СХД [22].

Отдельной проблемой, возникшей в связи с распространением высокоскоростных твердотельных накопителей, стало отставание разработки алгоритмов оптимизации времени выполнения запроса баз данных от разработки аппаратного обеспечения (быстродействующих твердотельных накопителей). В настоящее время в большинстве распространенных баз данных применяют алгоритмы, разработанные еще для механических накопителей. Они не учитывают специфику функционирования высокоскоростных твердотельных накопителей, в частности, считается, что операции ввода-вывода выполняются намного медленнее, чем обращения к памяти и вычисления [23].

С учетом изложенного выше актуальной является задача разработки инструментария, который бы позволил спроектировать систему хранения данных заданного объема и длительности хранения с минимальной себестоимостью хранения для потребителя при заданных показателях производительности и надежности.

2. Концептуальная модель функционирования системы хранения данных

В основу решения задачи положена разработанная автором концептуальная модель функционирования системы хранения данных на основе твердотельных накопителей с технологией дедупликации VDO [9], эффективность применения которой для твердотельных накопителей обоснована в работе [24]. Согласно ей структурно-функциональная модель СХД состоит из перечисленных далее четырех последовательно соединенных компонентов, с которыми взаимодействует по сети пользовательское приложение:

- сетевой сервис;
- система дедупликации на основе технологии VDO;
- система управления записью, программный RAID;
- массив из твердотельных накопителей.

Соответствующие этим компонентам математические модели производительности, надежности и стоимости функционирования системы хранения дедуплицированных данных на твердотельных накопителях разработаны автором (соответствующая статья на момент написания настоящей работы готовится к печати). На основе созданных математических моделей с использованием элементов теории массового обслуживания была разработана имитационная модель функционирования СХД.

На рис. 1 приведена структурная схема системы имитационного моделирования. Система объединяет в себе имитационную модель функционирования СХД и алгоритм поиска наиболее эффективных настроек СХД. Такой алгоритм предназначен для решения обратной задачи: проектирования системы, оптимальной с точки зрения сочетания факторов надежности, скорости и стоимости хранения данных. Поток запросов, генерируемый приложениями, обозначен $\{q_i\}$, а задержки, вносимые на каждом из этапов выполнения запроса, обозначены Δt_i^{NS}, Δt_i^{DED}, Δt_i^{RAID} и Δt_i^{DISK} для сетевого приложения, системы дедупликации, программного RAID и твердотельного накопителя соответственно.

В работе [9] при описании концептуальной модели производительности СХД в силу функциональной сложности работы системы дедупликации ее модель предлагалось описывать с помощью алгоритмического оператора A^{DED}, который в самой работе представлен не был.

Данный алгоритмический оператор A^{DED} переводил входные параметры модели, а именно поток запросов $\{q_i\}$ в выходные параметры — множество времен обслуживания $\{\Delta t_i^{DED}\}$:

$$A^{DED}\colon \{q_i\} \to \{\Delta t_i^{DED}\}.$$

3. Имитационная модель системы дедупликации

Важной составной частью твердотельной СХД является система дедупликации. Для некоторых видов рабочих нагрузок (системы резервного копирования, хранение образов виртуальных машин) наличие системы дедупликации позволяет существенно уменьшить требования к объему твердотельных накопителей, частично компенсировать высокую стоимость хранения данных, уменьшить износ накопителей.

Автором был проведен сравнительный анализ нескольких доступных систем дедупликации (VDO,

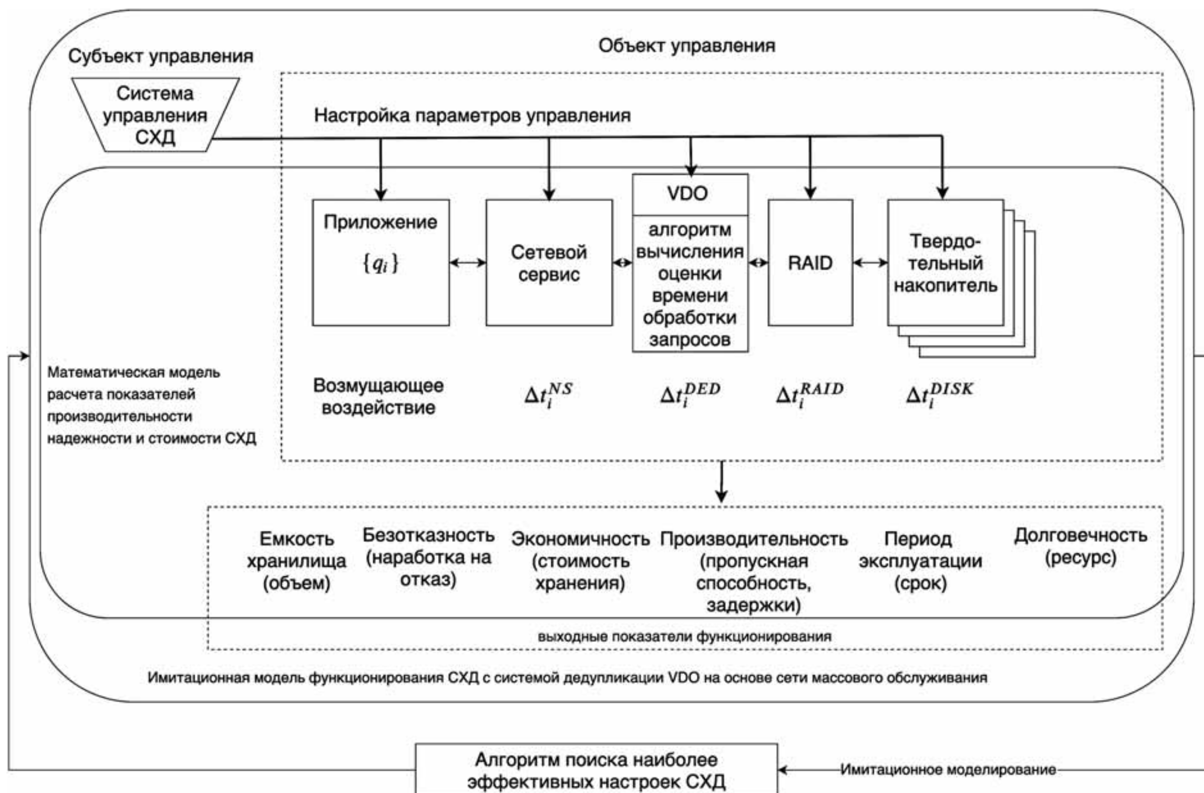


Рис. 1. Система имитационного моделирования СХД

SDFS, а также файловая система zfs), на основании которого была выбрана система дедупликации VDO (*Virtual Disk Optimizer*). Она реализована для ОС Linux и предоставляет дедупликацию и сжатие на блочном уровне в рамках Linux Device Mapper (механизм ядра ОС Linux, позволяющий организовать многослойную иерархию блочных устройств) [25].

Предложенная схема имитационного моделирования работы системы VDO была разработана автором на основании документации RedHat (гл. 29 "Интеграция VDO" из "Руководства администратора хранилищ данных" [26]).

В модели учитываются такие особенности твердотельных накопителей, как различная скорость и задержка при чтении и записи данных, а также износ накопителя в результате записи.

Процесс дедупликации является достаточно сложным алгоритмом, поэтому его целесообразно представить с помощью сети массового обслуживания, схема которой представлена на рис. 2. Названия серверов обслуживания соответствуют принятым в системе дедупликации VDO именам потоков выполнения ядра. Число серверов обслуживания (N_{log}, N_{CPU}, N_{phys}, N_{ack}, N_{bio}) соответствует числу ядер процессора, выделенных для соответствующей задачи на этапе создания или модификации тома VDO.

Одна из основных структур данных, используемых в системе VDO, — таблица отображения (*block map*) логических номеров блоков устройства с дедупликацией (*logical block number*, LBN) на физические номера блоков (*physical block number*, PBN). Логические номера блоков идентифицируют блоки данных до дедупликации. Физические номера блоков указываются на носителе информации, который используется системой VDO для хранения дедуплицированных данных (нижележащее устройство в иерархии *device mapper*, им может быть RAID или одиночный носитель информации).

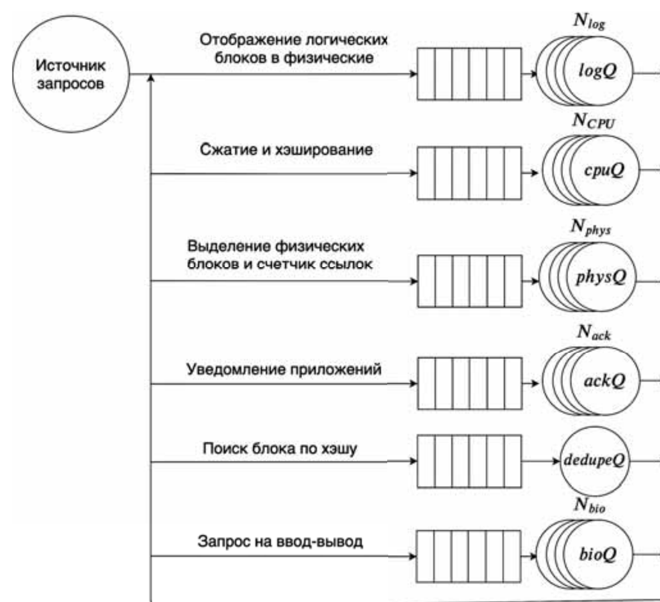


Рис. 2. Обобщенная схема сети массового обслуживания, использовавшейся при моделировании системы дедупликации VDO

Для операций обновления этой таблицы система VDO использует отдельную очередь (далее обозначена *logQ* по аналогии с именами процессов ядра *kvdo : logQ* модуля *kvdo*), при создании устройства с дедупликацией указывается число ядер процессора, которые занимаются обслуживанием этой очереди (N_{Log}).

Для операций выделения блоков на нижележащем устройстве также используется отдельный центр обслуживания (*physQ*), в котором используются N_{Phys} ядер процессора. Для передачи запросов на ввод-вывод нижележащему устройству используется центр обслуживания *bioQ* (N_{Bio} ядер процессора). Для вычислительных операций (сжатие, вычисление хэш-функции) используется центр обслуживания *cpuQ* (N_{CPU} ядер процессора), для отправки уведомлений приложениям о завершении операции ввода-вывода используется центр обслуживания *ackQ* (N_{ack} ядер процессора).

При создании устройства с дедупликацией перечисленные выше центры обслуживания позволяют указывать произвольное число ядер процессора (в зависимости от нагрузки) [27].

При получении модулем ядра *kvdo* запроса на запись в первую очередь проверяется, не является ли запрос DISCARD или TRIM (уведомление накопителя о том, что конкретный блок данных более не нужен). Проверяется также, не состоит ли блок данных целиком из нулевых байтов. В обоих этих случаях сохранения данных на носителе не происходит, обновляется только таблица отображения логических блоков в физические (центр обслуживания *logQ*).

Далее при операции записи в синхронном режиме блок данных немедленно записывается на нижележащий носитель. Для этого требуется сначала выделить блок на нижележащем носителе (центр обслуживания *physQ*), после чего передать запрос на ввод-вывод нижележащему устройству (центр обслуживания *bioQ*). После успешной записи приложение уведомляется о завершении операции ввода-вывода (центр обслуживания *ackQ*).

На следующем этапе выполняется попытка дедупликации. Вычисляется хэш-функция (алгоритм MurmurHash-3) для записанного блока данных (центр обслуживания *cpuQ*). Полученное значение передается подсистеме поиска физического номера блока по значению хэш-функции (модуль ядра *uds*, центр обслуживания *dedupeQ*). В отличие от всех очередей, упомянутых ранее, *dedupeQ* может использовать только одно ядро процессора.

Если найден блок с таким же значением хэш-функции, то он читается с носителя данных (очередь *bioQ*) и проводится побайтовое сравнение нового и уже существующего блоков. Если блоки полностью совпадают, то в таблицу отображения добавляется ссылка на уже существующий блок (очередь *logQ*), а временно использованный физический блок освобождается (очередь *physQ*).

Если же существующего блока с таким же значением хэш-функции не найдено, или при побайтовом сравнении обнаружены различия, то происходит обновление таблицы отображения (центр обслуживания *logQ*) и временный блок на физическом носителе становится постоянным.

Описанный процесс представляет возможность формализовать алгоритмический оператор A^{DED} в виде алгоритма вычисления оценки времени обработки запросов системой дедупликации. Блок-схема этого процесса приведена на рис. 3. Номера блоков на нижележащем носителе, соответствующие новой и уже существовавшей копиям данных, обозначены $PBN_{нов}$ и $PBN_{стар}$ соответственно.

Разработанный алгоритм позволяет численно оценить время задержки запроса на запись в системе дедупликации на основе многосерверной модели с неограниченным размером очереди. Данный алгоритм реализован на языках R и Python с использованием специализированных библиотек имитационного моделирования.

Формальное описание данного алгоритма завершает построение имитационной модели производительности твердотельной СХД с системой дедупликации.

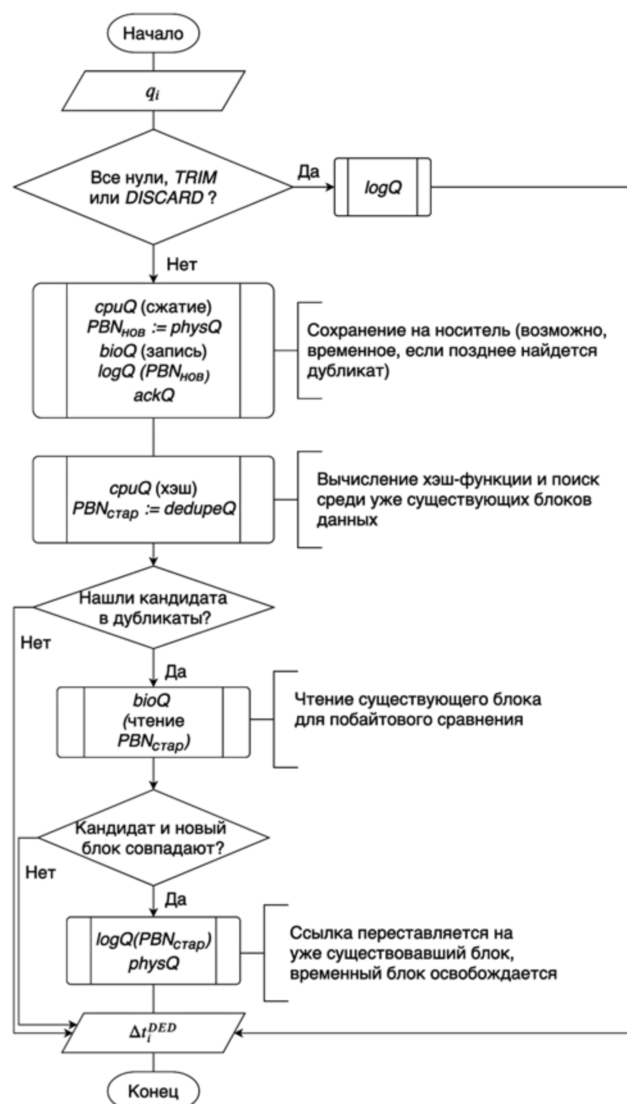


Рис. 3. Блок-схема алгоритма вычисления оценки времени обработки запросов на запись системой дедупликации

4. Поиск оптимальных управляющих параметров СХД

Для реализации системы имитационного моделирования к данной модели необходимо добавить алгоритм, позволяющий решать обратную задачу проектирования: поиск оптимальных управляющих параметров СХД, или другими словами, наиболее эффективных настроек СХД. На рис. 4 приведена блок-схема алгоритма поиска наиболее эффективных настроек СХД, использующего разработанную имитационную модель. Алгоритм поиска оптимальных параметров включает в себя три основных уровня, соответствующих уровням иерархии блочных устройств СХД.

Самый низкий уровень — одиночные накопители (DISK). В алгоритме предполагается, что возможность изменения их параметров отсутствует и возможен лишь выбор конкретной модели накопителя (с какими-либо показателями производительности, стоимости, надежности) из доступных на рынке. Также возможно изменение числа накопителей от одного до 24 (максимальное число дисковых отсеков в корпусе СХД). Принимается допущение, что устанавливаются накопители одной модели.

Следующий уровень в иерархии блочных устройств — программный RAID. На этом уровне, согласно ранее выработанной концепции [9], рассматриваются три уровня RAID (RAID5, RAID6, RAID10). Для реализации Linux md RAID10 возможен также выбор числа копий данных (параметр *layout* утилиты *mdadm*).

Самый верхний уровень в иерархии блочных устройств — система дедупликации и сжатия VDO.

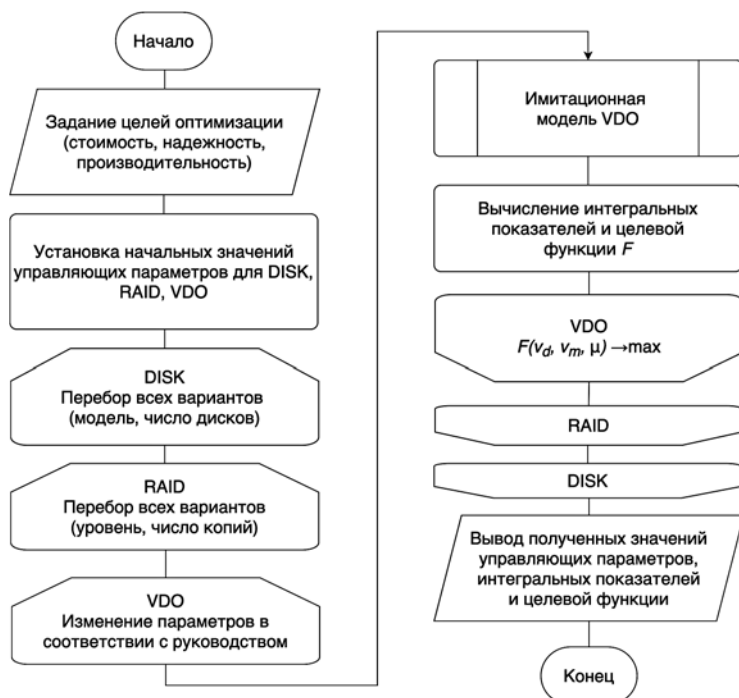


Рис. 4. Блок-схема алгоритма поиска наиболее эффективных настроек СХД

При создании тома VDO указывается желаемый размер тома, например, возможно создать том объемом 2 Тбайт на носителе объемом 1 Тбайт, если предполагается, что соотношение объемов данных до и после дедупликации (*deduplication ratio*) будет не менее 2. Также указывается число ядер процессора, выделяемых для различных задач (N_{log} ядер для обработки обновлений таблицы отображения блоков, N_{CPU} ядер для сжатия данных и вычисления хэш-сумм и т. д. для каждого центра обслуживания, см. схему сети на рис. 2), отмечается объем оперативной памяти, выделяемый для кэширования таблицы отображения блоков.

Таким образом, в начале работы алгоритм поиска наиболее эффективных настроек СХД получает на входе следующие параметры:

- характеристики твердотельных накопителей DISK, включая стоимость, объем, производительность (число операций в секунду для чтения и записи, пропускная способность в Мбайт/с для чтения и записи), надежность (ресурс накопителя на запись, среднее время до отказа), число дисков;
- характеристики RAID — уровень RAID-массива и число копий в случае RAID10;
- характеристики системы дедупликации VDO — размер тома, число ядер процессора для каждого центра обслуживания (N_{log}, N_{CPU}, N_{phys}, N_{ack}, N_{bio}), объем ОЗУ для кэширования;
- желаемые целевые показатели функционирования СХД, а именно желаемое время хранения данных, объем, надежность, производительность, стоимость.

Входные характеристики DISK, RAID, VDO задают выходной вектор проектных параметров $\mu \in \Omega_\mu$, где Ω_μ — множество допустимых значений проектных параметров. Целевые показатели функционирования СХД задаются вектором желаемых значений показателей функционирования v_d. На основе вектора μ с помощью имитационного моделирования рассчитывается вектор модельных значений показателей функционирования $v_m(\mu)$. На основе сравнения желаемых и модельных значений рассчитывается целевая функция $F(v_d, v_m(\mu))$.

Задается критерий оптимизации — минимум или максимум целевой функции. Для примера, в качестве целевой функции положим такую характеристику производительности СХД, как скорость передачи данных. Тогда критерием оптимизации будет максимум целевой функции $F(v_d, v_m(\mu)) \rightarrow \max$.

Постановка задачи оптимизации в данном примере сводится к поиску такого вектора оптимальных значений выходных параметров алгоритма μ*, который бы обеспечил максимум целевой функции:

$$\mu^* = \arg\max_{\mu \in \Omega_\mu} \left\{ F\left(v_d,\, v_m(\mu)\right) \right\}.$$

Процесс оптимизации для данного алгоритма (рис. 4) заключается в реализации перечисленных далее шагов.

Шаг 1. Осуществляется перебор различных вариантов накопителей (цикл DISK на рис. 4).

Шаг 2. Для каждого зафиксированного варианта модели накопителя и числа накопителей осуществляется перебор различных вариантов RAID (цикл RAID на рис. 4).

Шаг 3. Для каждого варианта RAID выполняется подбор параметров настройки системы дедупликации VDO с использованием имитационной модели VDO (цикл VDO на рис. 4) и алгоритма руководства администратора хранилищ данных RedHat по настройке параметров VDO [27].

Шаг 4. От значений этих параметров, которые образуют вектор μ, вычисляется целевая функция $F\left(v_d, v_m(\mu)\right)$.

Шаг 5. Полученное значение целевой функции $F\left(v_d, v_m(\mu)\right)$ сравнивается с найденным ранее лучшим значением $F\left(v_d, v_m(\mu^*)\right)$, полученным на предыдущем шаге.

Шаг 6. Если найденное значение больше предыдущего $F\left(v_d, v_m(\mu)\right) > F\left(v_d, v_m(\mu^*)\right)$, то оно становится оптимальным: $F\left(v_d, v_m(\mu^*)\right) = F\left(v_d, v_m(\mu)\right)$, и вектор оптимальных выходных параметров μ*, при которых оно достигается, также сохраняется.

Шаг 7. Алгоритм снова возвращается на шаг 1 к изменению вектора μ. Так продолжается, пока не будут перебраны все возможные варианты сочетаний допустимых значений накопителей DISK, конфигураций RAID и настроек VDO.

В данном примере результатом работы алгоритма будет являться вектор μ* эффективных настроек СХД, который включает в себя оптимальные управляющие параметры системы дедупликации VDO, уровень RAID, модель и число дисков, при которых достигается наилучшее (наибольшее) значение целевой функции $F\left(v_d, v_m(\mu^*)\right)$ — производительности СХД с учетом заданных целевых показателей оптимизации.

Предложенный алгоритм поиска наиболее эффективных настроек СХД является основой функционирования системы имитационного моделирования при решении задачи проектирования СХД согласно требованиям заказчика.

5. Результаты и обсуждение

Разработанная система имитационного моделирования реализована на языках R и Python с использованием специализированных библиотек для моделирования систем массового обслуживания.

Вывод системы имитационного моделирования был реализован по аналогии с выводом утилиты fio, которая широко используется для тестирования производительности подсистемы ввода-вывода в операционной системы Linux. Указанная утилита позволяет получить стандартные интегральные показатели функционирования (пропускную способность для чтения и записи, число операций в секунду для чтения и записи, средние значения этих параметров), а также гистограмму распределения времени обслуживания для операций ввода-вывода.

С точки зрения решаемых СХД задач, все приложения условно подразделяются на приложения со случайным доступом к данным (транзакционные системы) и приложения с потоковым доступом к данным (потоковые приложения) [12].

Ниже представлены некоторые результаты работы системы имитационного моделирования. Имитировалась работа подсистемы хранения, состоящей из сетевого сервиса, системы дедупликации VDO, программного RAID и переменного числа накопителей данных (от 4 до 24).

Моделировалась работа СХД, в которой на вход сетевого сервиса подавались заявки на обслуживание двух различных типов, а именно достаточно большие файлы и короткие запросы системы обработки транзакций (базы данных или аналогичные рабочие нагрузки). Коэффициент дублирования (соотношение повторяющихся и уникальных данных) для большинства экспериментов был принят равным 0,3.

На рис. 5 представлены гистограммы запросов, генерируемых приложением пользователя, для двух видов нагрузки.

Система имитационного моделирования вычисляла для каждой заявки время обслуживания, остальные параметры вычислялись на основе размера заявки и времени ее обслуживания. Выборка последовательности времени обслуживания заявок приведена на рис. 6.

Эти данные хорошо иллюстрируются в виде гистограммы, в связи с тем, что такой формат представления данных позволяет легче заметить характерные особенности распределения времени обслуживания заявок (рис. 7).

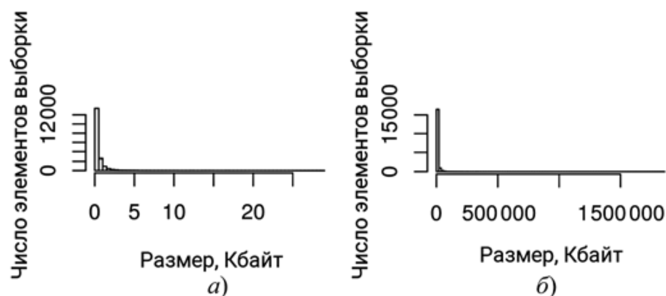


Рис. 5. Гистограмма распределения размеров запросов к СХД: *а* — транзакционная нагрузка; *б* — потоковая нагрузка

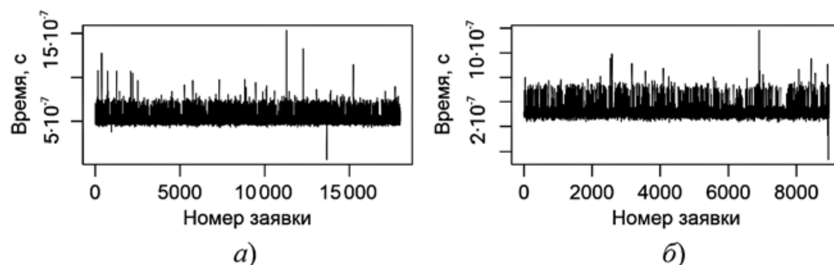


Рис. 6. Снимок случайного процесса общего времени обслуживания: *а* — транзакционная нагрузка; *б* — потоковая нагрузка

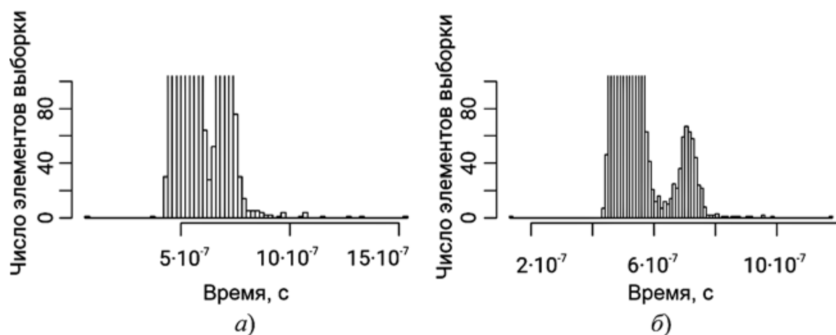


Рис. 7. Гистограмма распределения общего времени обслуживания:
а — транзакционная нагрузка; *б* — потоковая нагрузка

Отдельно рассматривались составляющие компоненты итогового времени обслуживания: задержка сетевого сервиса; задержка системы дедупликации; задержка, которую вносит программная реализация RAID; задержка, создаваемая накопителем. При этом в эксперименте использовался усредненный профиль нагрузки по размеру запроса.

При анализе распределений времени обслуживания (рис. 8—11) видно, что система дедупликации VDO вносит существенный вклад во время обслуживания, поэтому далее эта технология была рассмотрена более подробно.

В целях изучения влияния соотношения уникальных и повторяющихся данных на задержку, вносимую системой дедупликации, были проведены дополнительные эксперименты.

Для фиксированной конфигурации (RAID10, четыре накопителя) коэффициент дублирования принимали равным 0,001, 0,3 и 0,6 — "повторяющиеся данные практически отсутствуют", "30 % данных повторяется" и "60 % данных повторяется" соответственно. На рис. 12 приведены гистограммы распределения общего времени записи СХД при различных коэффициентах дублирования.

Также был проведен эксперимент для изучения влияния числа дисков, используемых для хранения данных, на показатели функционирования СХД. Для этого для трех вариантов конфигу-

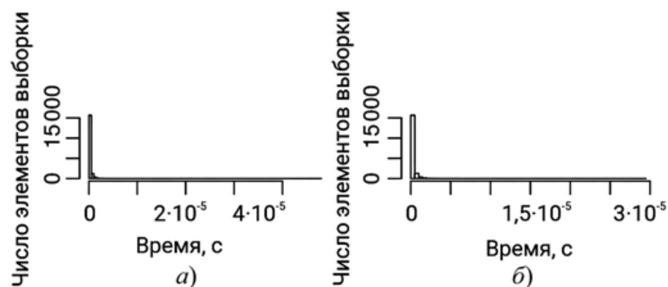


Рис. 8. Гистограмма распределения времени обслуживания для сетевого сервиса:
а — время чтения; *б* — время записи

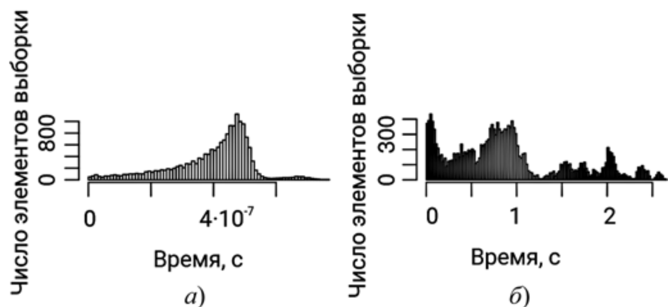


Рис. 9. Гистограмма распределения времени обслуживания для системы дедупликации VDO:
а — время чтения; *б* — время записи

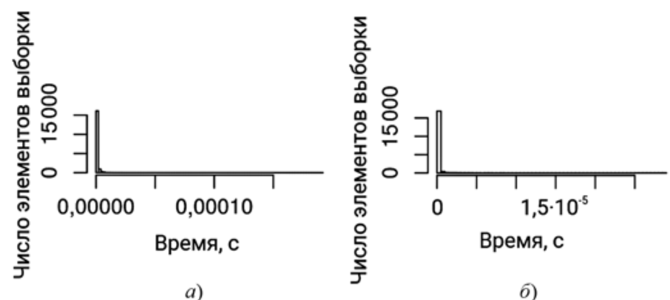


Рис. 10. Гистограмма распределения времени обслуживания для программного RAID:
а — время чтения; *б* — время записи

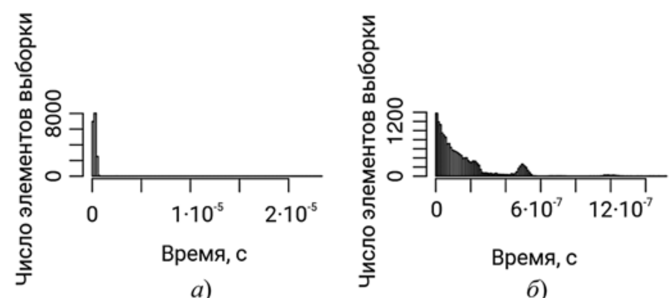


Рис. 11. Гистограмма распределения времени обслуживания для твердотельного накопителя:
а — время чтения; *б* — время записи

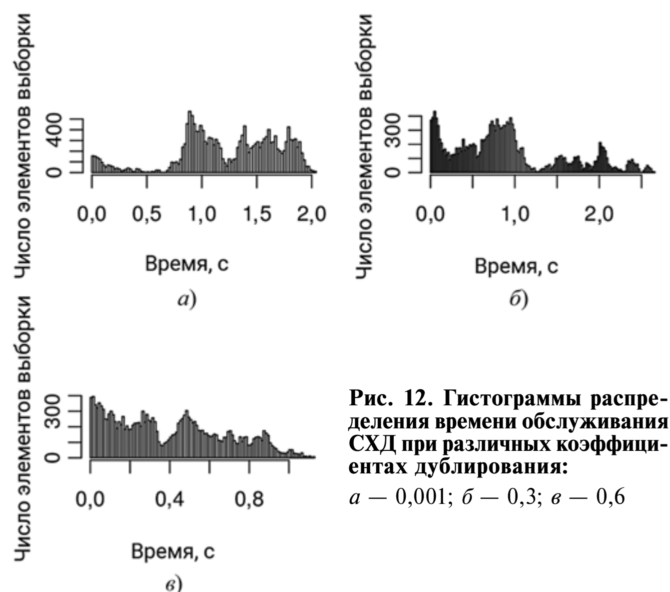


Рис. 12. Гистограммы распределения времени обслуживания СХД при различных коэффициентах дублирования:
а — 0,001; *б* — 0,3; *в* — 0,6

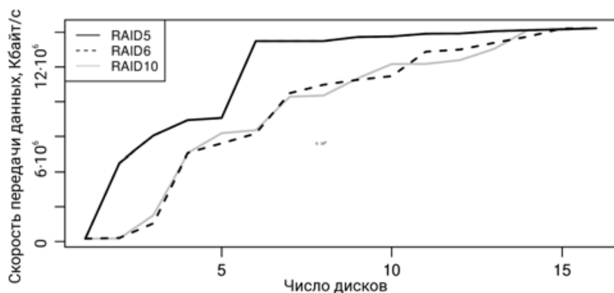


Рис. 13. Скорость передачи данных СХД (режим чтения)

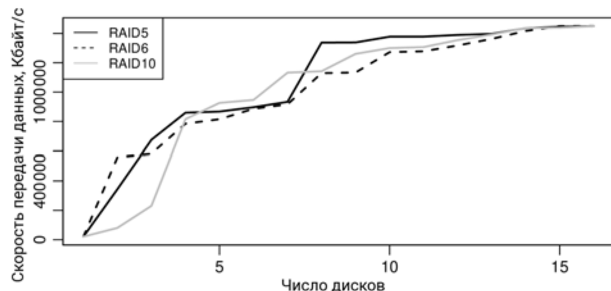


Рис. 14. Скорость передачи данных СХД (режим записи)

раций RAID (RAID5, RAID6, RAID10) число дисков в RAID менялось с 4 до 24 (больше 24 дисков не позволяет установить аппаратная платформа СХД).

На рис. 13 и 14 показаны зависимости пропускной способности твердотельной системы хранения данных от числа накопителей в RAID-массивах.

Заключение

В результате проведенного исследования была разработана и создана система имитационного моделирования, реализованная на языках R и Python и основанная на предложенной имитационной модели и алгоритмах.

Имитационная модель системы массового обслуживания реализована в виде многосерверной модели обработки запросов с неограниченными очередями. Она позволяет имитировать процессы чтения и записи информации в системе хранения данных и оценивать такие параметры производительности, как скорость передачи данных в режимах чтения и записи, а также время отклика системы.

Особенностью имитационной модели является специально разработанный алгоритм вычисления оценки времени обработки запросов системой дедупликации VDO, который впервые позволяет численно оценить эффект от применения данной технологии. Результаты имитационного моделирования подтвердили эффективность использования технологии дедупликации VDO для твердотельных накопителей в целях улучшения показателей производительности СХД.

Для решения обратной задачи проектирования был разработан алгоритм поиска наиболее эффективных настроек СХД, положенный в основу работы системы имитационного моделирования. Предложенный алгоритм позволяет получать решения, оптимальные с точки зрения сочетания факторов надежности, скорости и стоимости хранения данных.

Полученные в итоге имитационного моделирования результаты соответствуют данным натурных экспериментов, что подтверждает адекватность построенной модели.

Разработанный инструментарий позволяет спроектировать систему хранения данных необходимого объема и длительности хранения с минимальной себестоимостью хранения для потребителя при заданных показателях производительности и надежности.

Список литературы

1. **Новый** сегмент рынка — склады хранения данных. CRE. URL: https://www.cre.ru/analytics/73491.

2. **Балашова А.** Большим данным стало мало места. Информационное агентство "РБК". URL: https://www.rbc.ru/newspaper/2018/11/06/5bdc45019a79472ab0ecdbc2.

3. **Бахур В.** Рынок СХД в России резко пошел вверх // Интернет-издание о высоких технологиях, 26.03.2019. URL: http://www.cnews.ru/news/top/2019-03-26_kvartalnye_prodazhi_shd_v_rossii_prevysili_150

4. **Силин А.** Производительность СХД: как определить, какая система подходит бизнесу? // Журнал сетевых решений/LAN. 2017. № 06. URL: https://www.osp.ru/lan/2017/06/13052244/.

5. **Вислоцкий И.** Переходим на SSD: как строили систему хранения данных в виртуализированной среде // Хакер. 2016. URL: https://xakep.ru/2016/11/02/ssd-migration.

6. **Как** SSD влияют на производительность систем хранения данных // Tom's Hardware Guide Russia — интернет-издание, посвященное компьютерным технологиям. URL: www.thg.ru/storage/ssd_performance_storage_systems/index.html.

7. **Исследование:** рынок HDD сократится на треть к 2021 году. ИТ-ГРАД. URL: https://habr.com/ru/company/it-grad/blog/435182/.

8. **O'Reilly J.** Перспективы рынка хранения данных. URL: https://habr.com/ru/company/cbs/blog/302300/.

9. **Пономарев В. А., Питухин Е. А.** Концептуальная модель функционирования системы хранения данных на основе твердотельных накопителей с технологией дедупликации // Инженерный вестник Дона. 2019. № 5. URL: https://www.ivdon.ru/ru/magazine/archive/N5y2019/5905.

10. **Jens Axboe** fio — Flexible I/O tester rev. 3.15. URL: https://fio.readthedocs.io/en/latest/fio_doc.html.

11. **Оценка** производительности блочного хранилища. URL: https://www.8host.com/blog/ocenka-proizvoditelnosti-blochnogo-xranilishha/.

12. **Трофимова П. В.** СХД для потоковых данных: проблемы и решения / Storage News. 2011. № 2 (46). С. 23—25. URL: https://storagenews.ru/46/AvroRAID_46.pdf.

13. **Фадеев А. Ю.** Применение концепции IP-storage для создания распределенных систем хранения данных высокой степени готовности // Электронный журнал "Исследовано в России". 2002. С. 1226—1236. URL: https://cyberleninka.ru/article/v/primenenie-kontseptsii-ip-storage-dlya-sozdaniya-raspredelennyh-sistem-hraneniya-dannyh-vysokoy-stepeni-gotovnosti

14. **Ivanichkina L. V., Neporada A. P.** The reliability model of a distributed data storage in case of explicit and latent disk faults // Journal of Engineering and Applied Sciences, 2015. Vol. 10, N. 20. P. 9713—9724.

15. **Ivanichkina L. V., Neporada A. P.** Computer Simulator of Failures in Super Large Data Storage // Contemporary Engineering Sciences. 2015. Vol. 8, N. 28. P. 1679—1691.

16. **Muntz R. R., Lui J. C. S.** Performance Analysis of Disk Arrays under Failure // In Proc. of VLDB, Aug 1990. P. 162—173.

17. **Schulze M., Gibson G., Katz R., Patterson D. A.** How reliable is a RAID? // Digest of Papers. COMPCON Spring 89. Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage, San Francisco, CA, USA, 1989. P. 118—123.

18. **Rumyantsev A., Ivashko E., Chernov I., Kositsyn D., Shabaev A., Ponomarev V.** Latency/Wearout in a Flash-Based Storage System with Replication on Write // 24th Conference of

Open Innovations Association (FRUCT), Moscow, Russia. 2019. P. 360—366. DOI: 10.23919/FRUCT.2019.8711984.
19. **Li Y., Lee P. P. C., Lui J. C. S.** Stochastic Analysis on RAID Reliability for Solid-State Drives // IEEE 32nd International Symposium on Reliable Distributed Systems, Braga, 2013. DOI: 10.1109/SRDS.2013.16.
20. **Solid-State** Drives (SSDs) Modeling. Simulation Tools & Strategies / Eds. Micheloni R. Springer Series in Advanced Microelectronics. Vol. 58. Springer International Publishing, 2017. 170 p. DOI: 10.1007/978-3-319-51735-3.
21. **Kim J., Lee E., Choi J., Lee D., Noh S.** Chip-level RAID with flexible stripe size and parity placement for enhanced SSD reliability // IEEE Transactions on Computers. 2016. Vol. 65, Issue 4. P. 1116—1130. DOI: 10.1109/TC.2014.2375179.
22. **Почему** RAID6? URL: www.avroraid.ru/technology/-raid6-.
23. **Кузнецов С. Д.** Новые устройства хранения данных и их влияние на технологию баз данных // Программная инженерия. 2018, Т. 9, № 4. С. 147—155. DOI: 10.17587/prin.9.147-155.
24. **Chernov I., Ivashko E., Kositsyn D., Ponomarev V., Rumyantsev A., Shabaev A.** Flash-Based Storage Deduplication Techniques: A Survey // International Journal of Embedded and Real-Time Communication Systems (IJERTCS). 2019. Vol. 10, Issue 3. P. 32—48. 10.4018/IJERTCS.2019070103.
25. **Levine S.** The device mapper. Red Hat. Products & Services. Product Documentation. Red Hat. Enterprise Linux. 7. Logical Volume Manager Administration. URL: access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/logical_volume_manager_administration/device_mapper.
26. **Walsh A.** VDO Integration. Red Hat. Products & Services. Product Documentation. Red Hat. Enterprise Linux. 7. Storage Administration Guide. URL: http://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/storage_administration_guide/vdo-integration
27. **Walsh A.** Tuning VDO. Red Hat. Products & Services. Product Documentation. Red Hat. Enterprise Linux. 7. Storage Administration Guide. URL: access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/storage_administration_guide/vdo-ig-tuning-vdo

Simulation Modeling Performance Indicators for Solid State Storage Systems

V. A. Ponomarev, vadim@cs.petrsu.ru, Petrozavodsk State University, Petrozavodsk, 185910, Karelia, Russiam Federation

Corresponding author:

Ponomarev Vadim A., Senior Lecturer, vadim@cs.petrsu.ru, Petrozavodsk State University, Petrozavodsk, 185910, Karelia, Russian Federation
E-mail: vadim@cs.petrsu.ru

Received on August 02, 2019
Accepted on August 09, 2019

Based on newly developed algorithms, the article presents results of simulation modeling of main indicators of productivity, reliability and operation costs of a storage system for deduplicated data on solid-state media.

The influence of input control parameters of the system on the output performance indicators is analyzed, which allows proposing optimal parameters and structure of a data storage system during its design. As a result of the study, a simulation system was developed and implemented using R and Python programming languages, based on the proposed simulation model and algorithms.

The simulation model of a queuing system is implemented as a multi-server model for processing requests with unlimited queues. It allows simulating the processes of reading and writing information in a data storage system and evaluating performance indicators, such as data transfer speed in read and write modes, as well as the system response time.

A distinctive feature of the simulation model is a newly developed algorithm for calculating the estimated requests processing time by the deduplication system, which allows for the first time to quantify the effect of applying this technology. The simulation results confirmed the effectiveness of using the deduplication technology for storage performance of a data storage system.

To solve the inverse design problem, an algorithm was developed for finding the most effective storage settings. The proposed algorithm allows finding solutions that are optimal in terms of a combination of factors — reliability, speed and cost of data storage.

The results obtained during simulation modeling correspond to results of practice experiments, which confirms the adequacy of the model.

The developed toolkit allows designing a data storage system of required volume and storage duration with minimum storage costs with given performance and reliability indicators.

Keywords: *simulation modeling, solid state media, deduplication, data storage system, algorithm, performance indicators.*

Acknowledgements:

This work is supported by the Ministry of Education and Science of Russian Federation (project no. 14.580.21.0009, unique identifier RFMEFI58017X0009).

For citation:

Ponomarev V. A. Simulation Modeling Performance Indicators for Solid State Storage Systems, *Programmnaya Ingeneria,* 2019, vol. 10, no. 9—10, pp. 367—376.

DOI: 10.17587/prin.10.367-376

References

1. **New** market segment as a data warehouse, available at: https://www.cre.ru/analytics/73491 (in Russian).
2. **Balashova A.** Big data is running out of space, Informacionnoe agentstvo "RBK", available at: https://www.rbc.ru/newspaper/2018/11/06/5bdc45019a79472ab0ecdbc2 (in Russian).
3. **Bahur V.** DSS market went up sharply in Russia, *Internet-izdanie o vysokih tehnologijah*, available at: http://www.cnews.ru/news/top/2019-03-26_kvartalnye_prodazhi_20shd_v_rossii_prevysili_150 (in Russian).
4. **Silin A.** DSS performance: how to determine which system is right for the business? *Zhurnal setevyh reshenij/LAN*, 2017, no. 06. available at: https://www.osp.ru/lan/2017/06/13052244/ (in Russian).
5. **Vislotskiy I.** Switch to SSD: how storage System was built in a virtualized environment, *Xakep*, 2016, available at: https://xakep.ru/2016/11/02/ssd-migration/ (in Russian).
6. **How** SSD Affects Storage Performance*, Tom's Hardware Guide Russia* — available at: http://www.thg.ru/storage/ssd_performance_storage_systems/index.html (in Russian).
7. **Research:** HDD market will shrink by one third by 2021, available at: https://habr.com/ru/company/it-grad/blog/435182/ (in Russian).
8. **O'Reilly J.** Prospects for the data storage market, available at: https: //habr.com/ru/company/cbs/blog/302300/ (in Russian).
9. **Ponomarev V. A., Pituhin E. A.** A conceptual model of the functioning of a data storage system based on solid-state drives with deduplication technology. *Inzhenernyj vestnik Dona*, 2019, no. 5, available at: https://www.ivdon.ru/ru/magazine/archive/N5y2019/5905 (in Russian).
10. **Jens Axboe** fio — Flexible I/O tester rev. 3.15, available at: https://fio.readthedocs.io/en/latest/fio_doc.html.
11. **Block** Storage Performance Assessment, available at: http://www.8host.com/blog/ocenka-proizvoditelnosti-blochnogo-xranilishha/. (in Russian).
12. **Trofimova P. V.** DSS for streaming data: problems and solutions, *Storage News*, 2011, no. 2 (46), pp. 23—25, available at: http://storagenews.ru/46/AvroRAID_46.pdf (in Russian).
13. **Fadeev A. Ju.** Applying the concept of IP storage to create high availability distributed storage systems, *Electronic journal "Issledovano v Rossii"*. Moscow, Russia, 2002. pp. 1226—1236, available at: https://cyberleninka.ru/article/v/primenenie-kontseptsii-ip-storage-dlya-sozdaniya-raspredelennyh-sistem-hraneniya-dannyh-vysokoy-stepeni-gotovnosti (in Russian).
14. **Ivanichkina L. V., Neporada A. P.** The reliability model of a distributed data storage in case of explicit and latent disk faults, *Journal of Engineering and Applied Sciences,* 2015, vol. 10, no. 20, pp. 9713—9724.
15. **Ivanichkina L. V., Neporada A. P.** Computer Simulator of Failures in Super Large Data Storage, *Contemporary Engineering Sciences,* 2015, vol. 8, no. 28, pp. 1679—1691.
16. **Muntz R. R., Lui J. C. S.** Performance Analysis of Disk Arrays under Failure, *In Proc. of VLDB*, Aug 1990, pp. 162—173.
17. **Schulze M., Gibson G., Katz R., Patterson D. A.** How reliable is a RAID?, *Digest of Papers. COMPCON Spring 89. Thirty-Fourth IEEE Computer Society International Conference*, Intellectual Leverage, San Francisco, CA, USA, 1989, pp. 118—123.
18. **Rumyantsev A., Ivashko E., Chernov I., Kositsyn D., Shabaev A., Ponomarev V.** Latency/Wearout in a Flash-Based Storage System with Replication on Write, *24th Conference of Open Innovations Association (FRUCT)*, Moscow, 2019, pp. 360—366, DOI: 10.23919/FRUCT.2019.8711984.
19. **Li Y., Lee P. P. C., Lui J. C. S.** Stochastic Analysis on RAID Reliability for Solid-State Drives, 2013, *IEEE 32nd International Symposium on Reliable Distributed Systems*, Braga, 2013, DOI: 10.1109/SRDS.2013.16.
20. **Solid-State** Drives (SSDs) Modeling. Simulation Tools & Strategies / Eds. Micheloni R., *Springer Series in Advanced Microelectronics*, vol. 58, Springer International Publishing, 2017. 170 p., DOI 10.1007/978-3-319-51735-3.
21. **Kim J., Lee E., Choi J., Lee D., Noh S.** Chip-level raid with exible stripe size and parity placement for enhanced SSD reliability, *IEEE Transactions on Computers*, 2014, vol. 65, issue 4, pp. 1116—1130, DOI: 10.1109/TC.2014.2375179.
22. **Why** RAID6? Avroraid, available at: http://www.avroraid.ru/technology/-raid6- (in Russian).
23. **Kuznetsov S. D.** New Storage Devices and their Impact on Database Technology, *Programmnaya Ingeneria*, 2018, vol. 9, no. 4, pp. 147—155. DOI: 10.17587/prin. 9.147—155.
24. **Chernov I., Ivashko E., Kositsyn D., Ponomarev V., Rumyantsev A., Shabaev A.** Flash-Based Storage Deduplication Techniques: A Survey, *International Journal of Embedded and Real-Time Communication Systems (IJERTCS),* 2019, vol. 10, issue 3, pp. 32—48. DOI: 10.4018/IJERTCS.2019070103.
25. **Levine S.** VDO Integration. Red Hat. Products & Services. Product Documentation. Red Hat. Enterprise Linux. 7. Storage Administration Guide, available at: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/storage_administration_guide/vdo-integration.
26. **Walsh A.** The device mapper. Red Hat. Products & Services. Product Documentation. Red Hat. Enterprise Linux. 7. Logical Volume Manager Administration, available at: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/logical_volume_manager_administration/device_mapper
27. **Walsh A.** Tuning VDO. Red Hat. Products & Services. Product Documentation. Red Hat. Enterprise Linux. 7. Storage Administration Guide, available at: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/ storage_administration_guide/vdo-ig-tuning-vdo

УДК 004.75 DOI: 10.17587/prin.10.377-383

В. Е. Гвоздев, д-р техн. наук, проф., e-mail: wega55@mail.ru,
Л. Р. Черняховская, д-р техн. наук, проф., e-mail: lrchern@yandex.ru,
Р. А. Насырова, магистрант, e-mail: nasyrova.rima@yandex.ru,
Уфимский государственный авиационный технический университет

Анализ надежности информационных сервисов с учетом их объективных характеристик и субъективных оценок пользователей

Ключевая роль цифровой экосреды в деятельности гибко перестраиваемого и обучаемого предприятия обусловливает качественное изменение требований к надежности информационных систем. В настоящей работе рассмотрены вопросы анализа надежности информационных сервисов на основе одновременного учета объективных характеристик их нефункциональных свойств и субъективных оценок пользователями их функциональных возможностей.

Ключевые слова: *умное предприятие, функциональная безопасность, информационный сервис, профиль информационных сервисов, надежность информационных сервисов, цифровая экосреда*

Введение

Ключевым фактором реализации положений доктрины Industry 4.0 является формирование цифровой экосреды "умного предприятия". Назначением цифровой экосреды является, во-первых, информационное обеспечение гибкого реконфигурирования бизнес-приложений предприятия в соответствии с изменениями требований и предпочтений потребителей продукции. Во-вторых, оперативное распространение среди субъектов, участвующих в деятельности предприятий (далее — акторов), новых сведений, получаемых в результате практической деятельности. Обязательным требованием к свойствам цифровой экосреды является представление различным акторам данных и информации в форме, адекватной особенностям решаемых ими бизнес-задач [1, 2]. Исходя из представленных соображений можно сделать заключение, что основу функционирования "умного предприятия" составляет своевременное предоставление акторам надежных информационных сервисов, создающих условия для эффективного решения ими бизнес-задач. Следовательно, управление надежностью информационных сервисов является одной из основных задач формирования цифровой среды "умного предприятия".

В работах [3, 4] представлены соображения о целесообразности научно обоснованной адаптации решений, связанных с управлением качеством технических систем, в область программной инженерии. Теория надежности технических систем получила развитие с начала 50-х годов прошлого столетия. К настоящему времени в рамках этих теоретических исследований разработаны различные математические модели и на их основе реализованы методики для решения задач, связанных с управлением надежностью технических систем. Однако простой механический перенос этих моделей в область управления надежностью информационных сервисов не представляется возможным. Это обусловлено тем, что основу функционирования технических систем составляют объективные законы природы [5], в то время как программные системы по своей сути являются субъекто-центрическими. Некоторые из проблем управления надежностью программных продуктов, обусловленные их субъекто-центричностью, обсуждаются, например, в работах [6—8]. Тем не менее адаптация ряда решений из теории надежности технических систем в область управления надежностью информационных сервисов становится возможной, если разработать методы количественной оценки показателей надежности информационных сервисов. Главным предметом рассмотрения в настоящей работе является метод количественной оценки одного из базовых показателей надежности — вероятности безотказной работы информационных сервисов — на основе учета как их объективных свойств (характеризуемых вероятностью нарушения ограничений на ресурсы, выделяемые для предоставления сервисов), так и субъективного восприятия свойств сервисов разными акторами.

Анализ подходов к управлению надежностью программных систем

Важной задачей, непосредственно связанной с управлением надежностью информационных сервисов, является задача управления надежностью программных систем, реализующих сервисы. Анализ литературы, посвященной управлению надежностью программных систем, позволяет предложить следующую классификацию подходов. Во-первых, это подходы, направленные на устранение дефектов в конструкции программных продуктов. Во-вторых, это решения, ориентированные на оценку соответствия потребительских свойств программных систем требованиям (функциональным и нефункциональным) пользователей. Следует подчеркнуть, что эти подходы преследуют одну цель и ориентированы на достижение того уровня качества, которое определено в техническом задании на разработку программной системы.

Управление надежностью программных систем в рамках упомянутых подходов основано на использовании дескриптивных математических моделей. В качестве примеров моделей, ориентированных на устранение дефектов в конструкции программных продуктов, можно привести модели роста надежности, а также модели плотности дефектов (в англоязычной литературе именуемые "*software reliability growth models*" и "*detect density models*"). Содержанием моделей, относящихся к группе моделей роста надежности, является построение эмпирических зависимостей частоты проявления дефектов в программных продуктах от затрат ресурсов на испытание и отладку программ. Содержанием моделей, которые относятся к группе моделей плотности дефектов, является построение эмпирических зависимостей числа дефектов от конструктивных особенностей программных продуктов (числа строк кода, показателя сложности алгоритма) [9, 10].

Ограничениями упомянутых моделей являются:

- ориентация на поиск и устранение дефектов, являющихся непосредственной причиной отказов программных продуктов без стремления установить коренные причины дефектов;
- эмпирический характер моделей; отсутствие у них свойства общности; низкая потенциальность моделей.

В силу того, что формальную основу построения упомянутых дескриптивных моделей составляют методы математической статистики, областью применения моделей является область получения исходных данных [11]. Иными словами, нельзя в массовом порядке применять эмпирические зависимости, полученные при реализации одного программного проекта, для реализации других программных проектов.

В качестве примера моделей, ориентированных на оценку надежности программных систем с точки зрения оценки соответствия их потребительских свойств требованиям пользователей можно привести профили программных систем. Под профилем понимается полное множество альтернатив (например, множество альтернативных категорий пользователей, функций и т. п.), для каждой из которых существует вероятность появления [12]. Впервые концепция профилей, содержание которой сводится к стремлению взглянуть на программную систему глазами пользователей, описана в работе [13].

Целью построения профилей является создание основы для распределения ресурсов программного проекта по видам работ, с учетом различия в значимости разных свойств системы с точки зрения пользователей. Примером может служить описанная в работе [14] задача распределения ресурсов на тестирование и отладку программных модулей, обеспечивающих реализацию разных прикладных задач, с учетом вероятностей обращения пользователей к разным задачам. Формально профиль представляет собой дерево, каждому ребру которого ставится в соответствие весовая характеристика, характеризующая вероятность перехода по нему. Подходы к определению весовых характеристик обсуждены, например, в работе [15].

Особенностью анализа свойств программных систем на основе концепции профилей является, во-первых, предположение о том, что разные пользователи, относящиеся к одной целевой группе, одинаково воспринимают одни и те же потребительские свойства (в том числе надежность) систем. Такое предположение создает основу для оценивания весовых характеристик ребер на основе сравнения числа реализаций разных путей в графе. При этом игнорируется то обстоятельство, что различные пользователи могут по-разному оценивать качество одних и тех же информационных продуктов и услуг.

Во-вторых, характеристики надежности программных систем задаются в той форме, которая принята при анализе надежности технических систем, т. е. в числовом виде. При этом игнорируется тот факт, что в технических системах значения характеристических параметров могут быть измерены инструментальными средствами. Измерить же инструментальными средствами соответствие функциональных возможностей программных систем ожиданиям пользователей не представляется возможным. Отсутствуют механизмы преобразования качественных оценок свойств программных систем, даваемых пользователями, в количественные значения характеристик надежности.

На основе проведенного анализа литературных источников можно сделать заключение, что к настоящему времени разработан ряд методов анализа надежности программных систем на стадиях проектирования высокого уровня, детального проектирования, кодирования. Истоками разработанных методов являются методы анализа надежности технических систем, ориентированных на обработку измерительных данных.

Вместе с тем создание цифровой экосреды "умного предприятия" в качестве обязательного выдвигает требование предоставления акторам информационных сервисов с учетом особенностей их профессиональной деятельности. Из этого следует необходи-

мость развития созданного к настоящему времени аппарата анализа надежности программных систем с учетом особенностей субъективного восприятия свойств информационных сервисов разными пользователями.

Профиль информационных сервисов

В настоящее время в литературе приведены различные определения IT-сервисов [16—18]. В настоящей работе IT-сервис рассматривается как услуга, предоставляемая цифровой экосредой "умного предприятия" потребителю и представляющая для него ценность как создающая возможность повышения результативности решения его бизнес-задач.

Основная идея подхода, ориентированного на управление надежностью информационных сервисов, — максимальный учет восприятия свойств сервисов потребителями с тем, чтобы определить направления их совершенствования.

Особенностью предлагаемого подхода является учет субъективизма в оценивании надежности одних и тех же сервисов разными потребителями результатов сервисов. Во-первых, это обусловлено различным восприятием результатов сервисов различными субъектами, во-вторых, тем, что одни и те же сервисы оказываются в разной степени полезными при решении различных бизнес-задач.

Основной целью сервис-менеджмента является предоставление потребителям надежных стабильных услуг, которые полностью удовлетворяют их потребности в заданной области [16]. Целью построения профиля информационных сервисов является определение приоритетных направлений совершенствования цифровой экосреды. Без четкого понимания таких направлений предприятие не будет обладать базовыми признаками "умного предприятия", а именно оперативной адаптируемостью к изменяющимся требованиям и предпочтениям потребителей и постоянным самообучением акторов, ассоциируемых с предприятием [2]. К числу основных задач, связанных с достижением этой цели, относятся: оценка степени покрытия бизнес-процессов информационными сервисами; оценка надежности информационных сервисов с учетом как их объективных характеристик, так и субъективных оценок свойств информационных сервисов разными акторами.

Важным фактором, который учитывается при построении профиля информационных сервисов, является учет множественности режимов функционирования "умного предприятия" (штатного, предпосылки возникновения нештатного режима; нештатного). При функционировании предприятия в разных режимах, во-первых, реализуются несовпадающие множества бизнес-приложений, что обусловливает обращение к разным информационным сервисам. Во-вторых, изменяется содержание информационных потребностей пользователей сервисов, что влечет за собой изменение оценок свойств одних и тех же сервисов в разных режимах. Например, в режиме штатного функционирования предприятия предъявляются высокие требования к точности определения зон загрязнения выбросами промышленного предприятия. При возникновении же аварийных ситуаций, связанных с выбросом в атмосферу точно не известного на начальной стадии ликвидации аварийной ситуации количества загрязняющих веществ, большую ценность имеют даже весьма приближенные оценки зон поражения как основы выделения сил и средств.

В качестве существенных факторов, также учитываемых при построении профилей информационных сервисов, выступали перечисленные далее.

- Информационные сервисы ассоциируются с бизнес-задачами, которые реализуются в составе бизнес-приложений, обеспечивающих функционирование "умного предприятия".
- Каждой бизнес-задаче можно поставить в соответствие множество возможных схем ее решения, при этом каждой из схем можно поставить в соответствие свой информационный сервис.
- Реализация бизнес-задачи осуществляется на одном рабочем месте. При этом в каждый момент времени на одном рабочем месте может использоваться лишь один из информационных сервисов, ассоциируемых с бизнес-задачей.

В рамках сделанных допущений профиль информационных сервисов приобретает вид, представленный на рис. 1.

Приведенная модель представляет собой дерево, в котором бизнес-процессы — корень, а потребители П — листья. При движении от корня к листьям каждому узлу графа в каждом слое ставятся в соответствие исходящие ребра. Каждому ребру ставится в соответствие весовая характеристика (вероятность перехода). Полагаем, что у каждого узла возможен переход только по одному из ребер. При этом переход обязательно происходит; выбор направления перехода заранее неизвестен; число ребер, исходящих из узла, конечно. Тогда каждому узлу можно поставить в соответствие соотношение

$$\sum_j P_{i,j}^{(l)} = 1,$$

где $P_{i,j}^{(l)}$ — вероятность перехода по j-му ребру исходящему из i-го узла графа;

l — признак слоя;

i — признак узла в слое;

j — признак ребра, ассоциированного с узлом.

На рис. 1 сплошными линиями изображены приложения, бизнес-задачи и информационные сервисы, соответствующие текущему состоянию "умного предприятия". Штриховыми линиями отмечены те, что будут востребованы в будущем с учетом прогнозных оценок изменения предпочтений потребителей, в работе [16] такие сервисы именуют *Service Pipeline.* Сплошными и штриховыми линиями выделены сервисы, востребованные как в настоящее время, так и в будущем, возможно, с учетом опыта использования сервисов, требующих доработки.

Построение профиля сервисов позволяет дать количественную оценку степени покрытия бизнес-задач информационными сервисами. Так, для i-й

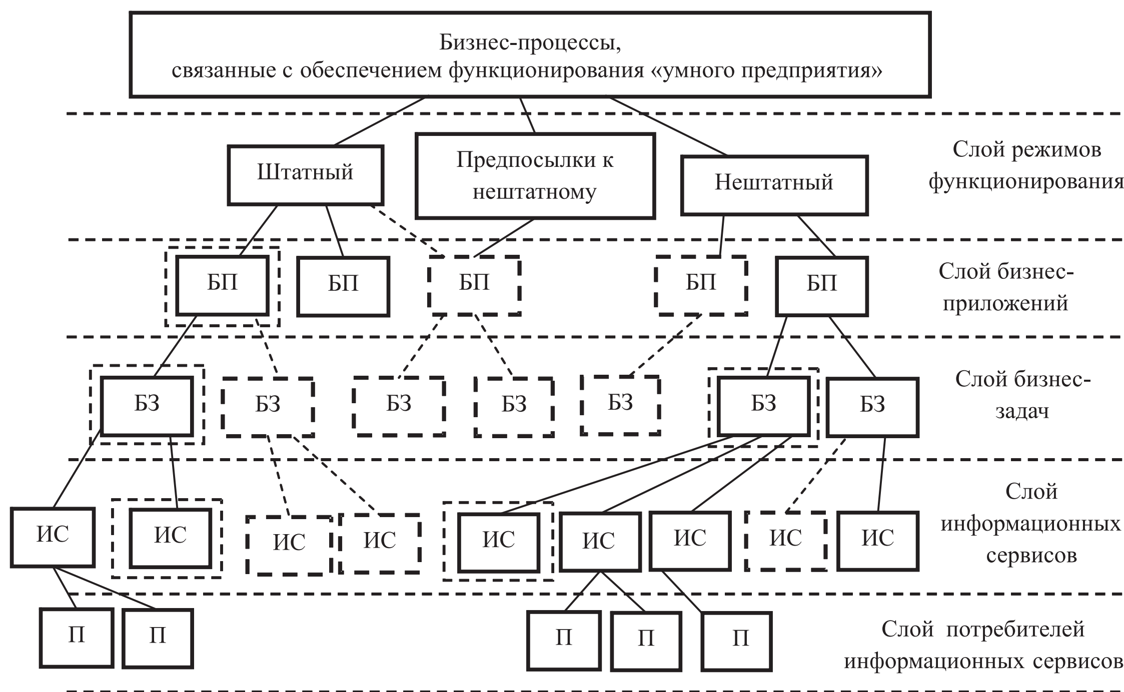


Рис. 1. Профиль информационных сервисов:

БП — бизнес-приложения, в совокупности обеспечивающие функционирование "умного предприятия" в разных режимах; БЗ — бизнес-задачи, решаемые на рабочих местах; ИС — информационные сервисы; П — акторы — потребители результатов информационных сервисов

задачи слоя бизнес-задач степень покрытия определяется как

$$\text{СП}_i^{(\text{БЗ})} = \frac{\text{ИС}_i^{(\text{Р})}}{\text{ИС}_i^{(\text{Р})} + \text{ИС}_i^{(\text{ПР})}},$$

где $\text{ИС}_i^{(\text{Р})}$ — число информационных сервисов, обозначенных сплошными/сплошными и штриховыми линиями (реализованные сервисы); $\text{ИС}_i^{(\text{ПР})}$ — число информационных сервисов, обозначенных штриховыми линиями (предполагаемые к реализации сервисы).

Аналогичные оценки могут быть получены для разных режимов функционирования "умного предприятия". Степень покрытия бизнес-задач информационными сервисами является одной из характеристик, которая учитывается при принятии решений по распределению ресурсов на разработку новых и совершенствование существующих информационных сервисов.

Профиль сервисов может рассматриваться как один из информационных компонентов стратегии перевода предприятия из текущего состояния ("как есть") в будущее ("как будет"). Реализация базовых свойств "умного предприятия", а именно адаптируемость к условиям внешней среды и самообучаемость, диктует необходимость постоянного расширения и обновления портфеля информационных сервисов. При построении дорожной карты разработки новых сервисов и совершенствования существующих в состав критериев для сравнения вариантов дорожных карт в числе прочих следует включить показатель, характеризующий динамику изменения степени покрытия бизнес-задач информационными сервисами по мере реализации этапов дорожной карты.

Оценивание характеристик надежности информационных сервисов

Различным вариантам профилей сервисов можно поставить в соответствие системную модель вида

$$\langle E, Pr, Rel \rangle,$$

где E — множество возможных событий (простых путей от корня к листьям в графе, представленном на рис. 1); Pr — множество значений вероятности (неявных профилей), соответствующих каждому из событий (неявный профиль определяется как произведение весов дуг, входящих в простой путь [12]), Rel — множество оценок надежности ИС.

В работе [15] подчеркнуто, что при оценивании надежности информационных систем необходимо учитывать удовлетворенность пользователей как функциональными возможностями сервиса, так и его нефункциональными свойствами. Оценивание характеристик надежности программных продуктов с точки зрения реализации нефункциональных требований осуществляется на основе измерительных данных [3]. Оценивание характеристик надежности предоставляемых пользователям услуг осуществляется на основе субъективных оценок пользователей [17, 18].

Набор информационных сервисов, необходимых "умному предприятию", индивидуален и в значительной степени зависит от отрасли, размеров предприятия, уровня его зрелости, стратегии развития и т. п. Отмеченные особенности определяют набор функциональных и нефункциональных характеристик сервисов, на основе которых проводится оценка их надежности.

В литературе, посвященной управлению надежностью аппаратно-программных комплексов [5, 19], четко разграничиваются задачи, связанные с получением данных о характеристиках надежности и с обработкой данных. Представленные далее материалы посвящены вопросам обработки данных, в качестве которых выступают субъективные оценки пользователями информационных сервисов и объективные характеристики нефункциональных свойств сервисов. Вопросы получения таких данных являются темой отдельных исследований, и в настоящей работе они не рассматривается. Некоторые из задач, посвященных таким исследованиям, рассматриваются, например, в работах [20—22].

Постулируя положение о независимости свойств "функциональные возможности" и "нефункциональные возможности" сервисов, результирующую оценку надежности $R_{\text{С}}$ можно представить в виде

$$R_{\text{С}} = R_{\text{Ф}} R_{\text{НФ}},$$

где $R_{\text{Ф}}$ — характеристика надежности, соответствующая функциональным возможностям;

$R_{\text{НФ}}$ — характеристика надежности, соответствующая нефункциональным возможностям.

В терминах работы [16] составляющая $R_{\text{Ф}}$ соответствует понятию "полезность", а $R_{\text{НФ}}$ — понятию "гарантия".

Для оценивания характеристик надежности на основе субъективных оценок пользователей предлагается использовать известный подход [23]. В рамках этого подхода исходными данными для оценивания характеристик надежности информационных сервисов являются значения лингвистической шкалы (например, "совершенно ненадежен", "относительно надежен", "почти надежен", "надежен"). Каждому значению шкалы соответствует своя нечеткая функция принадлежности μ, определенная на оси вероятности безотказной работы информационного сервиса. Положение максимума функции принадлежности l-го лингвистического значения называется его опорным значением r_l.

В терминах лингвистической шкалы j-й $\left(j = \overline{1;n}\right)$ пользователь выражает свое мнение относительно надежности сервиса, а также указывает уверенность в выбранном значении, выражаемую числом $\mu_j \in [0, 1]$. Совокупная оценка надежности определяется на основе соотношения

$$R_{\text{Ф}} = \frac{\sum_{j=1}^{n} (\mu_l r_l)}{\sum_{j=1}^{n} \mu_l},$$

где l — признак пользователя; n — общее число пользователей.

В качестве лингвистического аналога совокупной оценки $R_{\text{Ф}}$ признака принимается значение лингвистической переменной, опорное значение которой r_l ближе всего к $R_{\text{Ф}}$. Степень уверенности в выбранном значении лингвистической переменной определяется значением функции принадлежности, соответствующей точке на оси вероятности безотказной работы. Если все значения r_l в числителе приведенной выше формулы одинаковы, то в этом особенном случае степень уверенности полагается равной наименьшей из μ_l.

Пример лингвистической шкалы с функциями принадлежности представлен на рис. 2.

В качестве примера допустим, что три разных пользователя дали следующие оценки надежности i-го информационного сервиса:

1) почти надежен (степень уверенности — 0,9);
2) относительно надежен (степень уверенности — 0,3);
3) почти надежен (степень уверенности — 0,6).

Опорное значение для лингвистической переменной "почти надежен" — 0,7; опорное значение для лингвистической переменной "относительно надежен" — 0,35. Значение оценки $R_{\text{Ф}}$ составляет:

$$R_{\text{Ф}} = \frac{0,7 \cdot 0,9 + 0,35 \cdot 0,3 + 0,7 \cdot 0,6}{0,9 + 0,3 + 0,6} \approx 0,64.$$

Ближайшее к рассчитанному значению $R_{\text{Ф}}$ опорное значение $r_l = 0,7$ и соответствует лингвистической переменной "почти надежен". Значение функции принадлежности, соответствующее $R_{\text{Ф}} = 0,64$, составляет 0,7. Таким образом, по совокупности оценок пользователей i-й информационный сервис является "почти надежным", причем степень уверенности в такой оценке составляет 0,7.

Для оценивания $R_{\text{Нф}}$ в форме вероятности безотказной работы на основе измерительных данных (например, времени оказания услуги), могут использоваться известные математико-статистические методы анализа надежности, в том числе по выборкам малого объема, описанные в многочисленных литературных источниках, например, в работах [5, 19, 24, 25].

Предлагаемый подход к оцениванию надежности информационных сервисов основан на ранее не описанном в литературе сочетании известных подходов к обработке измерительных данных и субъективных оценок. Представление в сопоставляемой форме — вероятности безотказной работы — характеристик надежности, определяемых на основе обработки измерительных данных и субъективных оценок потребителей, делает возможным формирование результирующей

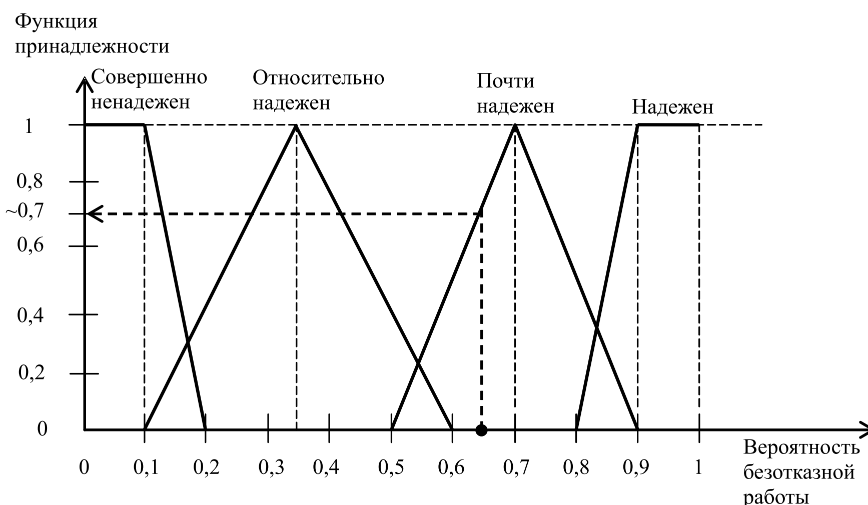


Рис. 2. Функции принадлежности лингвистической шкалы

метрической оценки надежности. Это, в свою очередь, создает предпосылки научно-обоснованной адаптации методов анализа надежности технических систем в область программной инженерии.

Заключение

Реализация положений доктрины Industry 4.0 предполагает предоставление сотрудникам "умного предприятия" (акторам) данных и информации в виде, адаптированном к особенностям решаемых ими бизнес-задач. Реализация этого положения обусловливает необходимость учета при анализе информационных сервисов не только их объективных свойств (характеризуемых вероятностью нарушения ограничений на ресурсы, например, время, выделяемое для предоставления услуг), но и субъективного восприятия надежности сервисов разными акторами.

Предложен метод, основанный на новом сочетании методов обработки измерительных данных и субъективных оценок. Основу метода составляет преобразование различных по форме исходных данных к единому виду — вероятности безотказной работы. Такое преобразование позволяет получать метрические оценки надежности информационных сервисов на основе совокупного использования как измерительных данных, так и субъективных оценок акторов. Предлагаемый подход создает предпосылки для теоретически обоснованной адаптации методов анализа надежности технических систем в область программной инженерии.

Работа поддержана грантом РФФИ 19-08-00177 "Методологические, теоретические и модельные основы управления функциональной безопасностью аппаратно-программных комплексов в составе распределенных сложных технических систем".

Список литературы

1. **Шваб К.** Четвертая промышленная революция: монография. М.: Издательство "Э", 2017. 208 с.
2. **Schuh G., Anderl R., Gausemeier J., ten Hompel M., Wahlster W.** Industrie 4.0. Maturity Index. Managing the Digital Transformation of Companies. Acatech_STUDY. URL: https://www.acatech.de/wp-content/uploads/2018/03/acatech_STUDIE_Maturity_Index_eng_WEB.pdf (дата обращения 05.04.2019).
3. **Липаев В. В.** Функциональная безопасность программных средств. М.: СИНТЕГ, 2004. 348 с.
4. **Батоврин В. К., Позин Б. А.** Инженерия требований на современном промышленном предприятии // Программная инженерия. 2019. Т. 10, № 3. С. 114—124.
5. **Дружинин Г. В.** Надежность автоматизированных систем. М.: Энергия, 1977. 536 с.
6. **Fuqun Huang.** Human Error Analysis in Software Engineering, Theory and Application on Cognitive Factors and Risk Management — New Trends and Procedures, IntechOpen, 2017. URL: https://www.intechopen.com/books/theory-and-application-on-cognitive-factors-and-risk-management-new-trends-and-procedures/human-error-analysis-in-software-engineering/ (дата обращения 05.04.2019).
7. **Anu V., Hu W., Carver J. C., Walia G. S., Bradshaw G.** Development of a Human Error Taxonomy for Software Requirements: A Systematic Literature Review // Information and Software Technology. 2018. Vol. 103. P. 112—124.
8. **Huang F., Liu B.** Software defect prevention based on human error theories // Chinese Journal of Aeronautics. 2017. Vol. 30, No. 3. P. 1054—1070.
9. **Гвоздев В. Е., Бежаева О. Я., Субхангулова А. С.** Оценивание количества дефектов в программных компонентах на основе показателя сложности // Материалы 3-й Международной конференции "Информационные технологии интеллектуальной поддержки принятия решений" ITIDS'2015, 18—21 мая, 2015. Уфа, Россия. Уфимский государственный авиационный технический университет. 2015. Т. 2. С. 55—60.
10. **ESA PSS-05-10.** Guide to Software verification and validation. European space agency, 1995.
11. **Вентцель Е. С.** Теория вероятностей: учеб. для вузов. 6-е изд. стер. — М.: Высш. шк., 1999. 576 с.
12. **Мороз Г. Б., Коваль Г. И., Коротун Т. М.** Концепция профилей в инженерии надежности программных систем // Математические машины и системы. 2004. № 1. С. 166—182.
13. **Cheung R.** A User-oriented Software Reliability Model // IEEE Trans. Soft. Eng. 1980. SE-6, N 2. P. 11—125.
14. **Lyu M. R.** Handbook of Software Reliability Engineering, Volume 1. Front Cover. IEEE Computer Society Press, 1996. 850 p.
15. **Singh L. K., Tripathi A. K., Vinod G.** Software Reliability Early Prediction in Architectural Design Phase: Overview and Limitations // Journal of Software Engineering and Applications. 2011. Vol. 4, No. 3. P. 181—186.
16. **Скрипник Д. А.** ITIL. IT Service Management по стандартам V.3.1. М.: НОУ "ИНТУИТ", 2016. 374 с.
17. **Дубова Н.** От информационных технологий к информационным сервисам [интервью с генеральным директором компании IT Expert Потоцким М.] // ITMF, 2013. URL: https://www.osp.ru/itsm/2013/02/13033935.html (дата обращения 02.05.2019).
18. **Ван Бон Я., Кеммерлинг Г., Пондман Д.** Введение в ИТ Сервис-менеджмент. itSMF, 2003. 237 с.
19. **Черкесов Г. Н.** Надежность аппаратно-программных комплексов. СПб.: Питер, 2005. 147 с.
20. **Wallace D., Coleman C.** Application and Improvement of Software Reliability Models. NASA Software Assurance Technology Center, 2001. 38 p.
21. **Ashwini Kumar Srivastava, Vijay Kumar.** Analysis of Software Reliability Data using Exponential Power Model // International Journal of Advanced Computer Science and Applications (IJACSA). 2011. Vol. 2, No. 2. P. 38—45.
22. **Lyu M. R.** Software Reliability Theory. Encyclopedia of Software Engineering. The Chinese University of Hong Kong, 2002. 43 p.
23. **Pelaez C. E., Bowles J. B.** Using fuzzy cognitive maps as a system model for failure modes and effects analysis // Information Sciences. 1996. Vol. 88, No. 1—4. P. 177—199.
24. **Гаскаров Д. В., Шаповалов В. И.** Малая выборка. М.: Статистика, 1978. 248 с.
25. **Гузаиров М. Б., Гвоздев В. Е., Ильясов Б. Г., Колоденкова А. Е.** Статистическое исследование территориальных систем. М.: Машиностроение, 2008. 187 с.

Analysis of the Reliability of Information Services, Taking into Account their Objective Characteristics and Subjective Assessments of Users

V. E. Gvozdev, wega55@mail.ru, **L. R. Chernyakhovskaya**, lrchern@yandex.ru,
R. A. Nasyrova, nasyrova.rima@yandex.ru, Ufa State Aviation Technical University, Ufa, 450008, Russian Federation

Corresponding author:

Gvozdev Vladimir E., Professor, Ufa State Aviation Technical University, Ufa, 450008, Russian Federation
E-mail: wega55@mail.ru

Received on May 05, 2019
Accepted on July 22, 2019

One of the basic tenets of the doctrine of Industry 4.0 is the creation of "Smart factories" whose environmental information is a backbone component. The information ecology environment is designed, firstly, to provide a comprehensive service for the information needs of enterprise personnel. Secondly, to ensure the possibility of their continuous learning through the identification and dissemination of new knowledge. The key role of the information environment in the activities of a flexible and learning enterprise leads to a qualitative change in the requirements for the reliability of the information support system.

A distinctive feature of information services is their human-oriented nature. This circumstance should be taken into account when assessing the reliability of information services, which are an integral part of the information ecological environment. This Article addresses the issues of analyzing the reliability of information services based on the simultaneous consideration of objective characteristics of their non-functional properties and subjective assessments by users of their functionality.

The development of methods for quantifying the reliability of information services, taking into account both their objective properties and subjective assessments of users, creates the basis for scientifically based adaptation of methods for managing the reliability of technical systems in the field of managing the reliability of the digital environment.

Keywords: *smart factory; functional security; information service; information services profile; reliability of information services; digital ecological environment*

Acknowledgements:
This work was supported by the Russian Foundation for Basic Research, project nos. 19-08-00177.

For citation:

Gvozdev V. E., Chernyakhovskaya L. R., Nasyrova R. A. Analysis of the Reliability of Information Services, Taking into Account their Objective Characteristics and Subjective Assessments of Users, *Programmnaya Ingeneria*, 2019, vol. 10, no. 9—10, pp. 377—383.

DOI: 10.17587/prin.10.377-383

References

1. **Schwab K.** *Fourth Industrial Revolution*, Moscow, publishing house "E", 2017, 208 p. (in Russian).
2. **Schuh G., Anderl R., Gausemeier J., ten Hompel M., Wahlster W.** Industrie 4.0. Maturity Index. Managing the Digital Transformation of Companies. Acatech_STUDY, available at: https://www.acatech.de/wp-content/uploads/2018/03/acatech_STUDIE_Maturity_Index_eng_WEB.pdf.
3. **Lipaev V. V.** *Functional security software,* Moscow, SINTEG, 2004, 348 p. (in Russian).
4. **Batovrin V. K., Pozin B. A.** Requirements Engineering at a Modern Industrial Enterprise, *Programmnaya inzheneriya*, 2019, vol. 10, no. 3, pp. 114—124 (in Russian).
5. **Druzhinin G. V.** *The reliability of automated systems*, Moscow, Energiya, 1977, 536 p. (in Russian).
6. **Fuqun Huang.** *Human Error Analysis in Software Engineering, Theory and Application on Cognitive Factors and Risk Management — New Trends and Procedures*, IntechOpen, 2017, available at: https://www.intechopen.com/books/theory-and-application-on-cognitive-factors-and-risk-management-new-trends-and-procedures/human-error-analysis-in-software-engineering.
7. **Anu V., Hu W., Carver J. C., Walia G. S., Bradshaw G.** Development of a Human Error Taxonomy for Software Requirements: A Systematic Literature Review, *Information and Software Technology*, 2008, vol. 103, pp. 112—124.
8. **Huang F., Liu B.** Software defect prevention based on human error theories, *Chinese Journal of Aeronautics*, 2017, vol. 30, no. 3, pp. 1054—1070.
9. **Gvozdev V. E., Bezhaeva O. Y., Subhangulova A. S.** Determining the number of defects in software components based on the complexity indicator, *Proceedings of the 3rd International Conference "Information Technologies for Intellectual Decision Support"* ITIDS'2015, Ufa, Russia, 2015, pp. 55—60 (in Russian).
10. **ESA PSS-05-10.** Guide to Software verification and validation. European space agency, 1995.
11. **Ventcel E. S.** *Probability Theory,* Ucheb. dlya universitetov, Moscow, Vysshaya shkola, 1999, 576 p. (in Russian).
12. **Moroz G. B., Koval' G. I., Korotun T. M.** The concept of profiles in the reliability engineering of software systems, *Matematicheskiye mashiny i sistemy*, 2004, no. 1, pp. 166—182 (in Russian).
13. **Cheung R.** A User-oriented Software Reliability Model, *IEEE Trans. Soft. Eng.* 1980, SE-6, no. 2, pp. 11—125.
14. **Lyu M. R.** *Handbook of Software Reliability Engineering*, Volume 1. Front Cover. IEEE Computer Society Press, 1996, 850 p.
15. **Singh L. K., Tripathi A. K., Vinod G.** Software Reliability Early Prediction in Architectural Design Phase: Overview and Limitations, *Journal of Software Engineering and Applications*, 2011, vol. 4, no. 3, pp. 181—186.
16. **Skripnik D. A.** *ITIL. IT Service Management standards V.3.1.*, Moscow, INTUIT, 2016, 374 p. (in Russian).
17. **Dubova N.** From information technology to information services [interview with the CEO IT Expert Potocki M.], *ITMF*, 2013, available at: https://www.osp.ru/itsm/2013/02/13033935.html (in Russian).
18. **Van Bon Ya., Kemmerling G., Pondman D.** Introduction to IT Service Management, itSMF, 2003, 237 p. (in Russian).
19. **Cherkesov G. N.** *Reliability of hardware and software systems*, St. Petersburg, Piter, 2005, 147 p. (in Russian).
20. **Wallace D., Coleman C.** Application and Improvement of Software Reliability Models. NASA Software Assurance Technology Center, 2001, 38 p.
21. **Ashwini Kumar Srivastava, Vijay Kumar.** Analysis of Software Reliability Data using Exponential Power Model, *International Journal of Advanced Computer Science and Applications (IJACSA)*, 2011, vol. 2, no. 2, pp. 38—45.
22. **Lyu M. R.** *Software Reliability Theory. Encyclopedia of Software Engineering*, The Chinese University of Hong Kong, 2002, 43 p.
23. **Pelaez C. E., Bowles J. B.** Using fuzzy cognitive maps as a system model for failure modes and effects analysis, *Information Sciences*, 1996, vol. 88, no. 1—4, pp. 177—199.
24. **Gaskarov D. V., Shapovalov V. I.** *Small sample,* Moscow, Statistika, 1978, 248 p. (in Russian).
25. **Guzairov M. B., Gvozdev V. Ye., Il'yasov B. G., Kolodenkova A. Ye.** *Statistical research of territorial systems,* Moscow, Mashinostroyeniye, 2008, 187 p. (in Russian).

УДК 004.052.2 DOI: 10.17587/prin.10.384-390

Д. В. Грузенкин, ст. препод., e-mail: gruzenkin.denis@good-look.su,
А. С. Михалев, ст. препод., e-mail: asmikhalev@yandex.ru,
Сибирский федеральный университет, г. Красноярск

Определение метрики диверсифицированности мультиверсионного программного обеспечения на уровне языков программирования

Рассмотрена проблема оценки качества мультиверсионного программного обеспечения. Поскольку концепция мультиверсионного программирования заключается в обеспечении независимости сбоев различных версий каждого конкретного модуля для повышения надежности всей системы в целом, надежность конкретного модуля тем выше, чем больше различий между его версиями. Введена и обоснована численная метрика, определяющая значение различия версий одного программного модуля на уровне языков программирования.

Ключевые слова: *мультиверсионное программирование, мультиверсионное программное обеспечение, мера различия программного обеспечения, метрика диверсифицированности программного обеспечения на уровне алгоритмов, качество программного обеспечения, надежность программного обеспечения, диверсификация мультиверсионного программного обеспечения*

Введение

В настоящее время существует большое число систем управления различного рода, которые могут быть применены практически во всех сферах жизнедеятельности человека. Однако существуют такие направления науки и техники, в которых надежность систем управления важна критически. К ним могут быть отнесены атомная энергетика, авиация, космические исследования, химическая промышленность и др. Это те сферы, в которых эффект от полезной работы программного обеспечения (ПО) систем управления и обработки информации может оказаться несоизмеримо мал по сравнению с негативными последствиями работы ПО, возникшими в результате его сбоев.

Для повышения надежности ПО существует большое число методов и подходов, в том числе основанных на различных видах избыточности. Применение методологии мультиверсионного программирования, основанной на идее программной избыточности, является одним из эффективных способов повышения надежности программных систем управления и обработки информации.

Каждый программный модуль мультиверсионной программной системы состоит из версий. Каждая версия модуля должна быть диверсифицирована, но при этом функционально эквивалентна остальным версиям данного модуля [1]. Для обеспечения высокого уровня надежности версии модулей должны быть диверсифицированы. Диверсификация версий каждого модуля мультиверсионного программного обеспечения должна осуществляется по меньшей мере на одном из следующих уровней:

1) уровне алгоритмов;
2) уровне языков программирования;
3) уровне средств разработки;
4) уровне средств тестирования [2].

В общем смысле диверсификация — это процесс формирования многообразия состояний системы за счет перераспределения системообразующего качества между разнообразными элементами, образующими эту систему [3]. В контексте настоящей работы под диверсификацией понимается процесс формирования многообразия реализаций (версий) конкретного модуля мультиверсионной программной системы за счет использования различных алгоритмов, или различных языков программирования, или различных средств разработки, или различных средств тестирования при их разработке. В свою очередь, под диверсифицированностью понимается наличие различий между реализациями (версиями) одного модуля, по крайней мере по одному из выше упомянутых признаков, благодаря применению диверсификации на этапе разработки системы.

Применение методологии мультиверсионного программирования позволяет обеспечить независимость сбоев версий модулей, что в свою очередь позволяет избежать выхода из строя конкретного модуля и программной системы в целом. Поскольку даже если одна версия выдаст ошибочный результат или завершит свою работу с ошибкой, не выдав никакого результата, то это обстоятельство не повлияет на работу остальных версий, что в целом позволяет

Таблица 1

Парадигмы языков программирования

Парадигма	Язык программирования						
	C	C++	C#	Java	Python	Ruby	Delphi
Императивная	+	+	+	+	+	+	+
Объектно-ориентированная	–/+	+	+	+	+	+	+
Функциональная	+/–	–/+	+/–	–/+	+	+	–/+
Рефлексивная	–	–/+	–/+	–/+	+	+	–/+
Обобщенное программирование	+	–/+	+	+	+	+	+
Логическая	–	–	–	–	–	–	–
Декларативная	–	–	–/+	–	+	+	–
Распределенная	+/–	+/–	–/+	+	–/+	–/+	–

обеспечить устойчивую работу программной системы [4].

Зачастую версии считаются диверсифицированными, если они разрабатывались различными командами разработчиков, изолированными друг от друга, по единой спецификации. Однако в рамках оценки качества ПО этот вопрос остается открытым. Мультиверсионное ПО может быть оценено с помощью ряда известных метрик для оценки качества ПО. Существует также метрика диверсифицированности мультиверсионного программного обеспечения на уровне алгоритмов [5]. Однако до настоящего времени не была введена метрика диверсифицированности мультиверсионного программного обеспечения на уровне языков программирования. С учетом этого обстоятельства авторами настоящей работы предлагается такая метрика, определяющая степень различия между версиями одного программного модуля мультверсионного программного обеспечения на уровне языков программирования.

Концептуальная идея исследования

Определение меры различия версий на уровне языков программирования может осуществляться путем сравнения языков программирования по заданному набору признаков. Однако следует отметить, что в настоящее время не существует единой общепринятой таксономии языков программирования. Все множество языков программирования можно разделить на классы по определенному набору общих признаков.

В данной работе для классификации и исследования были отобраны следующие признаки языков программирования: парадигма, типизация, управление памятью, управление потоком вычислений. Именно эти признаки определяют наиболее значимые различия между программным кодом, написанным на разных языках программирования [6]. Краткое описание каждого из перечисленных признаков приведено далее на примере нескольких наиболее популярных в настоящее время языков программирования.

Парадигма программирования

Парадигма программирования — это совокупность идей и понятий, определяющих стиль написания программного кода [7]. Существует много парадигм программирования, однако среди них принято выделять несколько основных: императивная, функциональная, логическая, объектно-ориентированная, рефлексивная, обобщенное программирование, распределенная, декларативная [7].

Парадигма программирования зачастую не может быть однозначно определена языком программирования, поскольку большинство современных языков программирования так или иначе допускают использование различных парадигм [7]. В то же время существуют и языки, ориентированные на реализацию лишь одной парадигмы [8].

В табл. 1 представлен список поддерживаемых парадигм некоторых наиболее популярных языков программирования. В табл. 1 и далее во всех остальных таблицах используются следующие обозначения:

- "+" — указанное значение признака присутствует у данного языка;
- "–" — указанное значение признака отсутствует у языка;
- "+/–" — указанное значение признака поддерживается не полностью данным языком программирования;
- "–/+" — указанное значение признака имеет значительные ограничения использования в данном языке программирования;
- "N/A" — указанное значение признака не поддерживается данным языком программирования;
- "?" — авторам достоверно неизвестно, поддерживается ли указанное значение признака данным языком.

Типизация

Типизация — свойство языка программирования различать типы данных [9]. Языки программирования по признаку типизации могут быть разделены на следующие группы: со статической и динамической типизацией, с явной и неявной типизацией [9].

Таблица 2

Варианты типизации в языках программирования

Типизация	Язык программирования						
	C	C++	C#	Java	Python	Ruby	Delphi
Статическая типизация	+	+	+	+	–	–	+
Динамическая типизация	–	–	+	–	+	+	–/+
Явная типизация	+	+	+	+	+/–	–	+
Неявная типизация	–	–/+	–/+	–	+	+	–
Неявное приведение типов без потери данных	+	+	+	+	+	+	+
Неявное приведение типов с потерей данных	+	+	–	–	–	–	+
Неявное приведение типов в неоднозначных ситуациях	+	+	+	–	–	–	–
Алиасы типов	+	+	+	–	N/A	N/A	+
Вывод типов переменных из инициализатора	–	+/–	+	–	N/A	N/A	–
Вывод типов переменных из использования	–	+/–	–	–	N/A	N/A	–
Вывод типов-аргументов при вызове метода	–	+	+	+	N/A	N/A	–
Вывод сигнатуры для локальных функций	–	+/–	–	–	N/A	N/A	–
Параметрический полиморфизм	N/A	–	+	+	N/A	N/A	–
Параметрический полиморфизм с ковариантностью	N/A	–	+/–	+	N/A	N/A	–
Параметрический полиморфизм высших порядков	N/A	–	–	–	N/A	N/A	–
Информация о типах в runtime	–	–/+	+	+	+	+	+
Информация о типах-параметрах в runtime	–	–/+	+	–	+	+	+

В языках со статической типизацией переменная связывается с типом в момент своего объявления и впоследствии не может изменить свой тип. В языках же с динамической типизацией тип переменной определяется в момент ее инициализации (а не объявления). Как следствие, переменная в любой момент во время следующей инициализации может изменить свой тип [9].

Основным отличием явно типизированных языков программирования является необходимость однозначного (явного) задания типов новых переменных/функций/аргументов функций. При этом языки с неявной типизацией перекладывают решение данной задачи с программиста на транслятор [9].

Следует отметить, что трансляторы некоторых языков программирования поддерживают возможность самостоятельного логического вывода типов значений выражений. Вывод типа может происходить при инициализации переменных, в момент установки значений по умолчанию, а также во время определения типов возвращаемых функциями значений [9].

Список возможных вариантов типизации для некоторых популярных языков программирования приведен в табл. 2.

Управление памятью

Задача управления памятью является одной из наиболее фундаментальных в программировании [6]. Часто разработчикам, которые пишут свой код на популярных скриптовых языках, не приходится заботиться об управлении памятью, так как за это отвечает "движок" языка [9]. Однако такие возможности не делают проблему управления памятью менее важной. В целях эффективной разработки критически важны знания возможностей и ограничений используемого менеджера памяти.

Большинство системных языков, в свою очередь, не поддерживает автоматического управления памятью. Для создания объекта в динамической памяти необходимо явно вызывать команду выделения памяти. Для доступа к объекту программа обращается по указателю к выделенной области памяти. После окончания работы с объектом необходимо вызвать команду освобождения памяти.

В табл. 3 представлены поддерживаемые виды управления памятью в некоторых популярных в настоящее время языках программирования.

Управление потоком вычислений

Еще одной важной задачей в программировании является управление потоком вычислений. Процесс выполнения последовательности операторов может происходить как непрерывно, так и прерываясь при определенных условиях. Прерывание происходит в том случае, если в потоке вычислений будут обнаружены соответствующие операторы. В случае прерывания управление будет передаваться в другое место [9].

Нормальное выполнение оператора может быть прервано также при возникновении исключительных ситуаций. Возбуждение исключительной ситуации прерывает нормальное выполнение оператора.

Таблица 3

Виды управления памятью в языках программирования

Стратегия управления памятью	Язык программирования						
	C	C++	C#	Java	Python	Ruby	Delphi
Создание объектов на стеке	+	+	+	–	–	–	–/+
Неуправляемые указатели	+	+	+	–	–	–	+
Ручное управление памятью	+	+	+	–	–	–	+
Сборка мусора	–	–/+	+	+	+	+	–

Таблица 4

Варианты управления потоком вычислений в языках программирования

Средство управления потоком вычислений	Язык программирования						
	C	C++	C#	Java	Python	Ruby	Delphi
Инструкция goto	+	+	+	–	–	–/+	+
Инструкция break без метки	+	+	+	+	+	+	+
Инструкция break с меткой	–	–	–	+	–	+	–
Поддержка try/catch	–	+	+	+	+	+	+
Блок finally	–	–	+	+	+	+	+
Блок else	–	–	–	+	+	+	+
Перезапуски	–	?	–	?	?	+	?
Ленивые вычисления	–	–/+	–/+	–	+	–/+	–
Continuations	–/+	?	+	?	–	+	?
Легковесные процессы (coroutines)	–	–	–	+/–	+/–	+	–

В некоторых языках программирования поддерживается стратегия вычисления, согласно которой вычисления следует откладывать до тех пор, пока не понадобится их результат [10].

Список возможных вариантов управления потоком вычислений для некоторых популярных языков программирования представлен в табл. 4.

Модель расчета степени диверсифицированности модулей мультиверсионного программного обеспечения

Как уже отмечалось ранее, многие языки имеют много значений одного признака, т. е. относятся сразу к нескольким классам. Так, например, если в качестве признака взять парадигму программирования, а за описываемый язык — Python, то можно сказать, что на этом языке могут быть написаны программы в функциональном, декларативном, объектно-ориентированном и других стилях программирования.

Таким образом, каждому признаку любого языка программирования может быть поставлено в соответствие некоторое непустое множество значений. Это обстоятельство по своей сути является формализацией задачи сравнения языков программирования между собой по некоторому набору признаков. Таким образом, путем нахождения мощности множества, которое образовалось на пересечении двух множеств значений определенного признака двух конкретных языков программирования, может быть получено число сходных значений данного признака. При делении этого значения на общее число значений данного признака для конкретного языка может быть найдено процентное соотношение схожести одного языка с другим.

С учетом изложенного выше степень схожести двух языков программирования по k-му признаку определяется как

$$S_{ij}^k = \frac{\left|set_i(k) \cap set_j(k)\right|}{\left|set_j(k)\right|}, \quad i = \overline{1,2},\ j = \overline{1,2},\ i \neq j, \qquad (1)$$

где i и j — индексы языков программирования; k — номер признака из всего множества признаков; $set_i(k)$ — множество значений k-го признака i-го языка программирования; $set_j(k)$ — множество значений k-го признака j-го языка программирования.

Для расчета степени схожести двух языков программирования с учетом всех признаков используется следующая формула:

$$S_{ij} = \frac{\sum_{k=1}^{m} S_{ij}^k}{m}, \qquad (2)$$

где i и j — индексы языков программирования; k — номер признака из всего множества признаков; m — число признаков; S_{ij}^k — степень схожести двух языков программирования по k-му признаку.

Однако на основании формул (1) и (2) еще нельзя ввести полноценную метрику диверсифицированности языков программирования. Причина в том, что не выполняется одна из аксиом метрического пространства, а именно: $\rho(a, b) = \rho(b, a)$ для любых двух элементов a и b из множества X — "аксиома симметрии", где ρ — расстояние между элементами (точками) в метрическом пространстве [11].

Приведенная выше аксиома не выполняется, когда мощности множеств возможных значений у одного признака различны. В таких случаях при сравнении одних и тех же языков программирования знаменатель в формуле (1) будет различаться в зависимости от того, для какого языка будет находиться процентное соотношение схожести с другим — для первого или для второго.

По этой причине для ввода полноценной метрики диверсифицированности языков программирования необходима начальная точка, с которой бы сравнивались все остальные значения. Это означает, что при расчетах меры схожести конкретного языка с ней в метрическом пространстве определялось бы расстояние от этой точки до точки, соответствующей сравниваемому с ней языку программирования.

Такой начальной точкой, с которой будут сравниваться все остальные языки программирования, станет абстрактный несуществующий язык программирования, который имеет все признаки сравниваемых языков программирования, а каждый его признак может принимать все значения этого признака, содержащиеся во всех сравниваемых при решении конкретной задачи языках программирования. В контексте данной работы назовем такой язык "идеальным".

Таким образом, для определения метрики диверсифицированности двух мультиверсий одного модуля мультиверсионного программного обеспечения на уровне языков программирования необходимо сравнить языки, на которых они были написаны, с "идеальным" языком, т. е. найти их удаленность от него в метрическом пространстве. После выполнения такого шага с помощью вычитания одного значения из другого может быть получена мера схожести мультиверсий в метрическом пространстве. С учетом данного заключения формула (2) принимает следующий вид:

$$S_{ij} = \frac{\sum_{k=1}^{m}\left(S_{ip}^k - S_{jp}^k\right)}{m}, \qquad (3)$$

где i и j — индексы языков программирования; k — номер признака из всего множества признаков; m — число признаков; S_{ip}^k и S_{jp}^k — степени схожести i-го и j-го языков программирования, соответственно, по k-му признаку с "идеальным" языком программирования, который является началом координат в данном метрическом пространстве.

В итоге метрика диверсифицированности двух мультиверсий одного модуля мультиверсионного программного обеспечения на уровне языков программирования M_d может быть рассчитана по формуле

$$M_d = 1 - S_{ij}. \qquad (4)$$

Программа и результаты эксперимента

Для проведения эксперимента остановимся на четырех языках программирования, а именно C, Java, Python, Delphi.

По каждому признаку были рассчитаны степени схожести между рассматриваемыми языками программирования и "идеальным" языком программирования. Полученные результаты представлены в табл. 5 и 6.

Затем по формуле (3) были рассчитаны меры схожести каждого исследуемого языка с каждым из числа других, обозначенных в табл. 5, 6. Результаты этих расчетов приведены в табл. 7.

Далее по формуле (4) был осуществлен расчет самой метрики диверсифицированности версий одного программного модуля мультиверсионного программного обеспечения на уровне языков программирования для каждого языка программирования из

Таблица 5

Степень схожести анализируемых языков программирования с "идеальным" языком программирования по парадигме и типизации

Язык программирования	Парадигма программирования				Типизация			
	Язык программирования							
	C	Java	Python	Delphi	C	Java	Python	Delphi
C	—	0	0,25	0,125	—	0,059	0	0,059
Java	0	—	0,25	0,125	0,059	—	0,059	0
Python	0,25	0,25	—	0,375	0	0,059	—	0,059
Delphi	0,125	0,125	0,375	—	0,059	0	0,059	—

Таблица 6

Степень схожести анализируемых языков программирования с "идеальным" языком программирования по управлению памятью и управлению потоком вычислений

Язык программирования	Управление памятью				Управление потоком вычислений			
	Язык программирования							
	C	Java	Python	Delphi	C	Java	Python	Delphi
C	—	0,5	0,5	0,25	—	0,4	0,4	0,3
Java	0,5	—	0	0,25	0,4	—	0	0,1
Python	0,5	0	—	0,25	0,4	0	—	0,1
Delphi	0,25	0,25	0,25	—	0,3	0,1	0,1	—

рассматриваемых в этом эксперименте. Результаты данного расчета приведены в табл. 8.

На основе анализа представленных выше результатов эксперимента можно сделать вывод, что все сравниваемые языки имеют небольшие различия между собой при сравнении их по определенным в данной работе критериям. Такие результаты вполне объяснимы, принимая во внимание, что все сравниваемые языки являются современными высокоуровневыми языками программирования и каждый из них подходит в качестве инструментального средства для решения широкого круга задач.

Метрика диверсифицированности, предложенная в данной работе, по своей сути является неким правилом сравнения языков программирования по заданным критериям. Сравнивая языки между собой, можно установить как их степень схожести, так и степень различия.

Таблица 7

Меры схожести исследуемых языков между собой

Язык программирования	C	Java	Python	Delphi
C	—	0,76025	0,7125	0,8165
Java	0,76025	—	0,92275	0,88125
Python	0,7125	0,92275	—	0,804
Delphi	0,8165	0,88125	0,804	—

Таблица 8

Результат эксперимента — мера диверсифицированности анализируемых языков программирования

Язык программирования	C	Java	Python	Delphi
C	—	0,23975	0,2875	0,1835
Java	0,23975	—	0,07725	0,11875
Python	0,2875	0,07725	—	0,196
Delphi	0,1835	0,11875	0,196	—

Данная метрика выражает степень различия двух языков программирования в вероятностном соотношении. При значении, близком к нулевому, сравниваемые языки будут полностью совпадать по заданным критериям. В свою очередь, при значении, стремящемся к единице, рассматриваемые языки будут максимально различаться. Значение различия будет максимальным только в случае использования тех возможностей языка, которые отсутствуют в сравниваемом с ним языке.

На основе получаемого значения различия языков программирования можно делать вывод и о степени различия мультиверсий, реализованных с их использованием. Значение данной метрики эквивалентно вероятности диверсифицированности версий, реализованных на сравниваемых языках программирования.

Заключение

Описан процесс ввода и обоснования метрики диверсифицированности версий одного программного модуля мультиверсионного программного обеспечения на уровне языков программирования. Данная метрика может быть применена при оценке качества мультиверсионного программного обеспечения как показатель, отражающий его надежность. Как отмечено в работе, чем версии одного модуля более диверсифицированы, тем менее вероятно в них возникновение совместных сбоев, что в целом свидетельствует о более высоком уровне надежности системы.

Список литературы

1. **Грузенкин Д. В., Михалев А. С., Царев Р. Ю., Суханова А. В., Новиков О. С.** Применение технологии блокчейн для повышения надежности мультиверсионного программного обеспечения // Современные наукоемкие технологии. 2018. № 2. С. 42—46.

2. **Kovalev I. V.** System of Multi-Version Development of Spacecrafts Control Software. Berlin: Pro Universitate Verlag Sinzheim, 2001. 80 p.

3. **Юдина И. С., Антонов А. П.** К вопросу об определении термина "Диверсификация" // Инновационная наука. 2016. № 6—1. С. 303—308.

4. **Михалев А. С., Исаев А. Н., Носарев К. А.** Современные технологии реализации мультиверсионного программного обеспечения // Новая наука: теоретический и практический взгляд. 2016. № 12. С. 129—132.

5. **Грузенкин Д. В., Якимов И. А., Кузнецов А. С., Царев Р. Ю.** Определение метрики диверсифицированности мультиверсионного программного обеспечения на уровне алгоритмов // Фундаментальные исследования. 2017. № 6. С. 36—40.
6. **Керниган Б. В., Ричи Д. М.** Язык программирования С. 2-е издание. М.: Вильямс, 2012. 229 с.
7. **Неклюдова С. А., Балса А. Р.** Парадигмы программирования как инструменты разработчика программных систем // Информационные технологии и системы: управление, экономика, транспорт, право. 2014. № 1. С. 27—34.
8. **Душкин Р.** Функциональное программирование на языке Haskell. М.: ДМК Пресс, 2016. 608 с.
9. **Ахо А., Лам М. С., Сети Р., Ульман Д. Д.** Компиляторы: принципы, технологии и инструментарий. М.: Вильямс, 2008. 1184 с.
10. **Опалева Э. А., Самойленко В. П.** Языки программирования и методы трансляции. СПб.: БХВ-Петербург, 2005. 576 с.
11. **Васильев Н.** Метрические пространства // Квант. 1990. № 1. С. 17—23. URL: http://kvant.mccme.ru/1990/01/metricheskie_prostranstva.htm (дата обращения 02.08.2019).

N-Version Software Diversity Metric Definition at the Programming Languages Level

D. V. Gruzenkin, gruzenkin.denis@good-look.su, **A. S. Mikhalev,** asmikhalev@yandex.ru,
Siberian Federal University, Krasnoyarsk, 660041, Russian Federation

Corresponding author:

Gruzenkin Denis V., Senior Teacher, Siberian Federal University, 660041, Krasnoyarsk, Russian Federation
E-mail: gruzenkin.denis@good-look.su

Received on May 14, 2019
Accepted on July 31, 2019

N-version programming is one of the most effective approaches to improve software reliability. It is based on the program redundancy, so one program module consists of several versions. They aim to solve identical tasks, but have differences in their constructions. For this reason, faults and errors are independent. As a result the higher one module versions diversity, the higher N-version software reliability. In view of this, it is necessary to define diversity metric for that purpose. The authors suggest comparing versions by comparing their programming languages to find versions measure of diversity, which is diversity metric. Comparison is based on a set of criteria, which allow one to figure out the difference between current programming language and a "perfect" one. The notion (collocation) of perfect programming language means an abstract programming language, which contains the values of all criteria for every compared programming language. The more values of some criterion for current programming language match the values of the same criterion for "perfect" programming language, the higher level of similarity is determined between them on the basis of this criterion. To figure out a full similarity level between current and "perfect" programming languages it is necessary to divide sum of similarity levels for all criteria by number of criteria. So diversity metric for two N-version software versions is presented as difference between one (it is equal to 100 %) and full similarity level difference between those versions languages. The metric can be used for program quality estimation.

Keywords: *N-version programming, N-version software, measure of software diversity, diversity metric on programming languages level, program quality estimation, software quality*

For citation:

Gruzenkin D. V., Mikhalev A. S. N-Version Software Diversity Metric Definition at the Programming Languages Level, *Programmnaya Ingeneria,* 2019, vol. 10, no. 9—10, pp. 384—390.

DOI: 10.17587/prin.10.384-390

References

1. **Gruzenkin D. V., Mikhalev A. S., Tsarev R. Yu., Sukhanova A. V., Novikokv O. S.** Application of the blockchain technology to increase n-version software reliability, *Sovremennye naukoemkie tekhnologii*, 2018, no. 2, pp. 42—46 (in Russian).
2. **Kovalev I. V.** *System of Multi-Version Development of Spacecrafts Control Software,* Berlin, Pro Universitate Verlag Sinzheim, 2001, 80 p.
3. **Yudina I. S., Antonov A. P.** About "Diversity" term definition, *Innovacionnaya nauka*, 2016, no. 6—1, pp. 303—308 (in Russian).
4. **Mikhalev A. S., Isaev A. N., Nosarev K. A.** Modern technologies of multi-version software development, *Novaya nauka: teoreticheskij i prakticheskij vzglyad,* 2016, no. 12, pp. 129—132 (in Russian).
5. **Gruzenkin D. V., Yakimov I. A., Kuznetsov A. S., Tsarev R. Yu.** The diversification metric of multiversional software determination at the algorithm level, *Fundamental research*, 2017, no. 6, pp. 36—40 (in Russian).
6. **Kernighan B. W., Ritchie D. M.** *The C Programming Language*, 2-nd edition, Prentice Hall, 1988, 274 p.
7. **Neklyudova S. A., Balsa A. R.** Paradigmy programmirovaniya kak instrumenty razrabotchika programmnyh sistem, *Informacionnye tekhnologii i sistemy: upravlenie, ekonomika, transport, pravo*, 2014, no. 1, pp. 27—34 (in Russian).
8. **Dushkin R.** *Functional programming in Haskell*, Moscow, DMK Press, 2016, 608 p. (in Russian).
9. **Aho A. V., Lam M. S., Sethi R., Ullman J. D.** *Compilers: principles, techniques and tools*, 2-nd edition, Addison Wesley, 2007, 1009 p.
10. **Opaleva E. A., Samojlenko V. P.** *Programming languages and translation methods*, 2005, 476 p. (in Russian).
11. **Vasilev N.** Metric spaces, *Kvant*, 1990, no. 1, pp. 17—23, available at: http://kvant.mccme.ru/1990/01/metricheskie_prostranstva.htm (in Russian).

УДК 81'322.2:004.912 DOI: 10.17587/prin.10.391-399

И. В. Трофимов, ст. науч. сотр., e-mail: itrofimov@gmail.com,
Институт программных систем им. А. К. Айламазяна РАН, г. Переславль-Залесский

Морфологический анализ русского языка: обзор прикладного характера

Прошедшее в 2017 г. соревнование MorphoRuEval позволило получить представление об эффективности современных алгоритмов морфологического анализа. В то же время открытым остался вопрос качества морфологического анализа в общедоступных комплексных аналитических конвейерах для русского языка. В настоящей работе представлены результаты оценки двух таких систем.

Ключевые слова: *обработка естественного языка, морфологический анализ, MorphoRuEval, TreeTagger, UDPipe, rnnmorph, морфологическая нотация, лемматизация*

Введение

Исследователи, занимающиеся высокоуровневыми задачами анализа естественного языка, а также разработчики прикладных систем обработки текста, как правило, используют готовые технические решения для низкоуровневых подзадач: токенизации, определения границ предложений, морфологического и синтаксического анализа. Многие используют аналитические конвейеры "из коробки", охватывающие весь нижний уровень и не требующие усилий по настройке (обучению моделей, конфигурированию). Для русского языка, в частности, известны конвейеры Шарова—Нивре [1] и UDPipe [2, 3].

Появление более совершенных подходов в низкоуровневом анализе ставит вопрос возможности и целесообразности встраивания этих новых решений в уже используемый аналитический конвейер. Сложность этого вопроса для *морфологического* уровня обусловлена, в первую очередь, разнообразием используемых морфологических описаний (тегсетов) и подходов к нормализации[1] словоформ. Алгоритмы, решающие задачи верхнего уровня, так или иначе опираются на какую-то определенную морфологическую нотацию. Поэтому замена аналитического модуля на более современный может быть сопряжена с необходимостью его адаптации к стандарту конвейера. Такая адаптация может свести на нет преимущества нового модуля или даже привести к ухудшению работы высокоуровневой системы в целом. Кроме того, прежде чем предпринимать усилия по встраиванию более совершенного алгоритма в конвейер, целесообразно оценить эффективность конвейера в условиях, максимально приближенных к тем, в которых испытывался новый алгоритм. Может оказаться, что при такой постановке задачи морфологический анализатор в составе конвейера продемонстрирует результаты, сопоставимые с результатами потенциально более совершенного алгоритма.

Результаты прошедшего в 2017 г. мероприятия по оценке морфологических анализаторов для русского языка MorphoRuEval [4] показали, что алгоритм на базе рекуррентной нейронной сети [5, 6] *существенно* превосходит другие подходы. В то же время технический уровень упомянутых выше готовых конвейеров в рамках этих соревнований не оценивали. В настоящей работе приведены результаты сопоставления эффективностей современной реализации нейросетевого алгоритма-лидера (rnnmorph) и морфологических анализаторов из конвейеров Шарова—Нивре и UDPipe. Для исследования использовалась оценочная инфраструктура, созданная в рамках MorphoRuEval—2017. Таким образом, конвейеры исследуются в условиях, приближенных к тем, в которых был оценен победитель соревнований.

Дальнейший материал структурирован следующим образом. Сначала приведено краткое описание исследуемых морфологических анализаторов с акцентом на различиях в используемых системах признаков. Затем изложены методология сопоставления анализаторов и результаты оценки. В заключение дана авторская интерпретация результатов, а также замечания о встраивании rnnmorph в готовые конвейеры.

Объекты исследования

Дадим краткую характеристику каждой из сравниваемых систем.

Конвейер Шарова—Нивре был представлен в 2011 г. в работе [1]. Впоследствии конвейер продолжал развиваться; современная его версия доступна онлайн[2]. В данном исследовании оценивалась версия конца 2018 г.

Конвейер состоит из простого модуля разбиения текста на предложения и слова, а также

[1] В литературе также используют термин *лемматизация*.

[2] http://corpus.leeds.ac.uk/mocky/

морфологического и синтаксического анализаторов. Для морфологического анализа предлагается использовать одно из трех инструментальных средств: TreeTagger [7], TnT [8] или SVMTool [9]. Первые два средства методологически опираются на скрытую марковскую модель второго порядка, последнее средство — на SVM-классификатор. Для исследования, результаты которого представлены в статье, был выбран TreeTagger.

В TreeTagger множество состояний марковской модели состоит из всех возможных *тегов* (допустимых наборов граммем) используемого тегсета. Например, если в используемом тегсете существительные могут характеризоваться только падежом, числом и родом, то множество *тегов* включает четверки <существ., им., ед., муж.>, <существ., род., ед., муж.> и т. д. Для других частей речи строятся собственные аналогичные кортежи. Все такие кортежи формируют множество состояний марковской модели.

Алфавитом скрытой марковской модели служат словоформы. Задача морфологического анализа ставится следующим образом. Даны марковская модель и последовательность словоформ отдельного предложения текста: w_1, ..., w_N, где N — число слов в предложении. Требуется отыскать такую последовательность тегов t_1, ..., t_N, которая наилучшим образом соответствует наблюдаемой последовательности словоформ. Формально для марковской модели второго порядка имеем

$$\arg\max_{t_1 \dots t_N} \left[\prod_{i=1}^{N} P\left(t_i \middle| t_{i-1}, t_{i-2}\right) P\left(w_i \middle| t_i\right) \right],$$

где $P(t_i|t_{i-1}, t_{i-2})$ — вероятность перехода в состояние t_i; $P(w_i|t_i)$ — вероятность словоформы w_i при условии, что ей ставится в соответствие тег t_i. Наиболее вероятная последовательность тегов вычисляется алгоритмом Витерби.

Опустим частные вопросы расчета и представления вероятностей, а также вопрос моделирования вероятностей словоформ, не вошедших в обучающее множество. Отметим ключевые характеристики метода, реализованного в TreeTagger.

- При анализе отыскивается наилучший вариант морфологического разбора для *всего* предложения.
- Контекст для принятия локального решения — текущая словоформа и два предшествующих тега.
- Используется единственная марковская модель (для "прочтения" предложения слева направо).

Сведений о том, каким образом TreeTagger осуществляет построение нормальной формы, обнаружить в литературе не удалось. Приводимые в данной работе оценки качества нормализации характеризуют именно TreeTagger и построенную для него русскоязычную модель, хотя авторы конвейера рекомендуют дополнительно использовать CST lemmatiser[3]. В рамках настоящего исследования оценивалась готовая модель, доступная пользователю онлайн[4]. Модель обучена на данных Национального корпуса русского языка [11, 12]. Результирующая морфологическая нотация — MULTEXT-East (ru) [13].

Предварительные эксперименты показали, что использование данной модели часто приводит к некорректным результатам при обработке слов, содержащих дефис, в том числе довольно частотных (*из-за*, *какой-то* и др.). Имеется системная ошибка с видом глагола (инверсия граммем "совершенный"/"несовершенный"). Для значительной доли словоформ не предсказывается нормальная форма.

Конвейер UDPipe создавался усилиями Милана Страки и Яны Страковой. Он включает в себя развитый модуль разбиения текста на предложения и слова, морфологический и синтаксический анализаторы. Актуальная версия конвейера доступна онлайн[5]; в настоящем исследовании использовалась версия начала 2019 г.

Реализация морфологического анализа в UDPipe уходит корнями к работам Ратнапаркхи [14] и Коллинза [15]. Общая идея подхода состоит в следующем. Вводится понятие *контекста i*-й словоформы предложения (у Коллинза это кортеж $c_i = \langle i, t_{i-1}, t_{i-2}, w_1, ..., w_N \rangle$). Контекст служит для определения признаков, потенциально полезных для предсказания тега. Формально признаки представляются в виде бинарных функций, аргументами которых являются тег-кандидат для рассматриваемой словоформы и контекст. Например, функция-признак

$$\phi(t_i, c_i) = \begin{cases} 1, \text{ если окончание } (w_{i-1}) = \text{'-ый'} \\ \text{и } t_i = \langle \text{сущ., им., ед., муж.} \rangle \\ 0, \text{ в противном случае} \end{cases}$$

принимает значение 1, когда предыдущая словоформа заканчивается на *-ый*, а в качестве тега-кандидата для рассматриваемой (*i*-й) словоформы выбран кортеж <сущ., им., ед., муж.>. Также функции-признаки могут апеллировать к порядку следования тегов. Например, так:

$$\phi(t_i, c_i) = \begin{cases} 1, \text{ если } t_{i-2} = \langle \text{предлог} \rangle, \\ t_{i-1} = \langle \text{прилаг., дат., мн.} \rangle, \\ t_i = \langle \text{сущ., дат., мн., муж.} \rangle \\ 0, \text{ в противном случае.} \end{cases}$$

В общем случае функция-признак может оперировать произвольным набором информации, представленной в контексте, и обязательно содержит ограничение, затрагивающее t_i.

Аппарат функций-признаков используется для формулирования задачи морфологического анализа предложения в следующем виде:

$$\arg\max_{t_1 \dots t_N} \left[\sum_{i=1}^{N} \sum_{f \in F} a_f \phi_f\left(t_i, c_i\right) \right],$$

где F — множество признаков; a_f — значимость (вес) признака f. Наиболее вероятная последовательность

[3] https://github.com/kuhumcst/cstlemma, https://cst.dk/download/cstlemma/russian/ [10].

[4] http://corpus.leeds.ac.uk/mocky/russian.par.gz

[5] http://ufal.mff.cuni.cz/udpipe

тегов вычисляется алгоритмом Витерби. Вопрос оценки коэффициентов a_f на базе обучающего множества описан в работе [15]; в настоящей статье он не рассматривается.

Успешность морфологического анализа в такой формулировке определяется выбранной системой признаков. В UDPipe множество признаков автоматически порождается по обучающему множеству с помощью шаблонов, разработанных экспертами. Например, шаблон $[w_{i+1} = X \ \& \ t_i = T]$ по каждому токену аннотированного текста порождает функцию-признак вида

$$\phi(t_i, c_i) = \begin{cases} 1, \text{ если } w_{i+1} = X \text{ и } t_i = T \\ 0, \text{ в противном случае,} \end{cases}$$

где X — следующая словоформа; T — тег, приписанный текущей словоформе. Таким образом, число признаков довольно велико: для чешского языка на корпусе объемом 1,5 млн токенов с помощью 63 шаблонов извлекаются 8,4 млн признаков [16].

В идеале система шаблонов должна учитывать специфику конкретного языка. В UDPipe используется система шаблонов, созданная для чешского языка, в предположении, что она достаточно универсальна и пригодна для всех поддерживаемых конвейером языков.

Описанная выше общая методология анализа в UDPipe была дополнена еще одним важным элементом — алгоритмом порождения гипотез (*morphological guesser*). Его задача — для рассматриваемой словоформы (опираясь на ее окончание длиной до четырех символов) породить *небольшое* число гипотез относительно тега, что повышает скорость анализа. Алгоритм порождения гипотез опирается на статистику, извлекаемую из аннотированного корпуса.

Нормализация в UDPipe опирается, в первую очередь, на словарь, построенный по обучающему множеству. Для слов, не вошедших в словарь, вновь используется алгоритм порождения гипотез, в котором окончаниям словоформ ставятся в соответствие теги и правила порождения нормальной формы[6]. Нормализация осуществляется независимо от процесса тегирования.

Отметим ключевые характеристики метода, реализованного в UDPipe.

- При анализе отыскивается наилучший вариант морфологического разбора для *всего* предложения.
- Контекст для принятия локального решения — два предшествующих тега, текущая словоформа, две предыдущие и две последующие словоформы, префиксы и суффиксы текущей словоформы, признаки наличия заглавных букв, дефиса, цифр и др.
- Однонаправленный поиск пути Витерби (слева направо).

В настоящем исследовании использовалась готовая модель версии 181115, доступная онлайн [17]. Модель обучена на корпусе СинТагРус [18, 19], приведенном к нотации UD [20]. Результирующая морфологическая нотация модели — UniversalDependencies v 2.3 [21].

Предварительные исследования показали, что модель часто допускает ошибки при нормализации глаголов в результате некорректных операций с префиксами и суффиксами. Системные недоработки имеются с глаголами повелительного наклонения.

Морфологический анализатор **rnnmorph** описан в работах [5, 6]. Эта система не решает самостоятельно задачи токенизации и определения границ предложения (в качестве входных данных ожидает последовательность токенов предложения). В настоящей работе исследовалась версия rnnmorph, извлеченная из репозитория[7] в мае 2019 г.

Задача морфологического анализа в rnnmorph рассматривается как задача пословной классификации (тег соответствует классу). Алгоритм опирается на двунаправленный LSTM-классификатор [22]. Перед подачей на вход классификатора слова предложения представляются в виде вектора, являющегося конкатенацией перечисленных далее составляющих.

- Вектор граммем для данной словоформы. Элементы этого вектора соответствуют граммемам тегсета. Значение элемента определяется как вероятность данной словоформы быть охарактеризованной данной граммемой. Вероятности вычисляются на обучающем множестве.
- Вектор тегов для данной словоформы. Его элементы соответствуют тегам тегсета. Значение элемента — вероятность данной словоформы быть охарактеризованной данным тегом.
- Вектор пунктуационных признаков в окрестности словоформы. Элементы вектора кодируют наличие определенных знаков препинания в определенных позициях относительно словоформы. Значения бинарны.
- Вектор графематических признаков словоформы. Элементы кодируют типовые комбинации регистра символов (нижний, верхний, первая заглавная). Значения бинарны.
- Вектор суффиксных признаков для данной словоформы. Элементы кодируют наиболее частотные суффиксы длиной до трех символов (около 3 тыс. суффиксов). Значения бинарны (имеется ли у словоформы данный суффикс).
- Эмбеддинг словоформы. Обычно эмбеддинг моделирует степень общности контекстов у представленных в таком виде словоформ. Иными словами, слова, встречающиеся в схожих контекстах, будут иметь схожие эмбеддинги. В работе [5] эмбеддинги служили лишь для представления самих словоформ (были инициализированы случайными числами); всего использовалось 25 тыс. 250-мерных эмбеддингов для наиболее частотных словоформ. В дальнейшем [6] они были заменены на символьные эмбеддинги (*character-level representation*), используемые системой так, чтобы в конечном счете моделировать общность контекстов, а кроме того, обеспечить устойчивость к опечаткам и возможность работы с несловарными словоформами.

[6] В терминах "удалить определенный суффикс или префикс", "добавить суффикс/префикс".

[7] https://github.com/IlyaGusev/rnnmorph/

К ключевым характеристикам метода, реализованного в rnnmorph, отнесем следующее.

◇ Классификатор учитывает весь предшествующий контекст и принятые решения в рамках предложения (в силу архитектуры LSTM).

◇ Контекст для принятия локального решения — текущая словоформа (в виде эмбеддинга), суффиксы текущей словоформы, ее графематические характеристики и пунктуационное окружение, оценки вероятности тегов и отдельных граммем для текущей словоформы.

◇ Двунаправленная оптимизация в рамках предложения.

Нейросеть обучена авторами алгоритма на данных[8] из корпуса ГИКРЯ [23], подготовленных для участников MorphoRuEval.

Таким образом, среди сопоставляемых систем TreeTagger опирается на наиболее бедную систему признаков, но обучен на самом объемном обучающем множестве. Теоретическим преимуществом rnnmorph над UDPipe являются двунаправленная оптимизация и возможность учитывать при классификации признаки, находящиеся за пределами ближайшего контекста анализируемого слова (за счет LSTM). Вместе с тем UDPipe может учитывать левый и правый лексические контексты непосредственно при принятии локального решения.

Метод исследования

Для сопоставления морфологических анализаторов были использованы оценочный инструментарий и дейтасеты[9], разработанные в рамках MorphoRuEval—2017[10]. В основе методологии оценки лежат следующие положения. Исходными данными служат предложения, записанные в форме последовательности токенов (границы предложений и токенов известны). Задача оцениваемого анализатора — приписать каждому токену единственный вариант морфологического разбора (тег и нормальную форму). Ответы анализатора затем сопоставляются с эталонной разметкой, выполненной (или проверенной) экспертами-людьми, для вычисления количественной меры качества анализа (точности). Подробнее процедура тестирования изложена в работе [4].

Оригинальная утилита вычисления показателей точности морфологического анализа реализует стратегию выборочной оценки. Она охватывает семь частей речи: существительные, глаголы, прилагательные, местоимения, наречия, числительные и детерминанты[11]. Для каждой части речи определен собственный набор грамматических категорий, подлежащих оценке. В интересах исследования оригинальный инструментарий был доработан так, чтобы имелась возможность получать оценки по отдельным частям речи, а также по нормализации словоформ независимо от успешности определения граммем.

Чтобы убедиться в устойчивости демонстрируемых системами результатов, в качестве тестовых множеств были задействованы несколько дейтасетов. Несмотря на то, что некоторые из этих множеств использовались при обучении моделей (такие результаты далее будут помечены звездочкой), их включение в исследование дает более полную картину в парных сравнениях. В итоге были выбраны следующие.

— Тестовое множество MorphoRuEval—2017. Оно состоит из трех частей, выделенных на жанровой основе (новостные сообщения, художественная литература, сообщения из соцсетей). Их объединение условимся в дальнейшем называть *3-in-1*. Множество содержит чуть более 1,3 тыс. предложений. Разметка тщательно проверена организаторами соревнований.

— Дейтасеты gikrya_train и syntagrus_train. Эти аннотированные корпуса выступали в качестве обучающих множеств MorphoRuEval. Они значительно больше по объему — около 62 тыс. и 37 тыс. предложений соответственно.

Все дейтасеты размечены в единой морфологической нотации, выработанной для MorphoRuEval. Из трех систем, включенных в исследование, лишь rnnmorph выдает результат в данной нотации. Для оценки остальных систем потребовалась разработка средств конвертации.

Для приведения морфологической нотации исследуемой русскоязычной модели для TreeTagger к нотации тестовых множеств выполнялись следующие действия.

Шаг 1. Техническое преобразование формата MULTEXT в UD.

Шаг 2. Использование атрибута "синтаксическая роль" местоимений для разделения MULTEXT-местоимений на детерминанты, наречия и местоимения в нотации MorphoRuEval.

Шаг 3. Приведение порядковых числительных к прилагательным.

Шаг 4. Представление в явном виде положительной степени сравнения для наречий.

Шаг 5. Приведение причастий к прилагательным. Для получения нормальной формы применялся поиск в морфологическом словаре проекта АОТ[12] [24].

Шаг 6. Приведение превосходной степени сравнения прилагательных к положительной.

Шаг 7. Немногочисленные "точечные" эвристические преобразования. В качестве примеров можно привести переквалификацию местоимений *сам/самый* из детерминантов в прилагательные, нормализацию местоимений *она/оно/они* к форме *он* и т. п.

Кроме того, выполнялась временная замена *ё* на *е*, различных форм кавычек на прямые, различных форм тире на дефис в целях устранения известных ошибок модели, связанных с этими факторами.

---

[8] https://github.com/dialogue-evaluation/morphoRuEval—2017/blob/master/GIKRYA_texts_new.zip

[9] Дейтасет представляет собой множество аннотированных предложений, где каждому слову приписаны морфологический тег и нормальная форма. Такие множества служат для обучения и тестирования алгоритмов. Морфологическая информация в дейтасетах MorphoRuEval порождена высокоточными инструментальными средствами и верифицирована людьми (полностью или в значительной степени).

[10] https://github.com/dialogue-evaluation/morphoRuEval—2017, https://drive.google.com/drive/folders/0B600DBw1ZmZASDFRVkJVd0pqNXM

[11] Закрытое множество слов, состоящее из определительных местоимений.

[12] http://aot.ru

Для адаптации выходных данных UDPipe к требованиям MorphoRuEval потребовались лишь шаги 5—7 из вышеприведенного списка.

Результаты

В рамках MorphoRuEval—2017 оценивались следующие два показателя:

— доля слов (%) с корректно определенными морфологическими атрибутами;

— доля предложений, не содержащих ошибок морфологического разбора.

Ограничимся рассмотрением только первого из них. В качестве результата будем приводить пару величин: точность (*accuracy*) определения граммем и точность полного разбора. В последнем случае оценивается корректность не только набора граммем, но и нормальной формы.

В табл. 1 приведена оценка систем оригинальной утилитой, использовавшейся в ходе MorphoRuEval. Полученные результаты показывают весомое превосходство rnnmorph; в дальнейшем уточним, из чего оно складывается.

Интересно отметить следующее наблюдение. Для оценки современного уровня (*baseline*) морфологического анализа организаторы MorphoRuEval использовали TreeTagger. Модели для него строились, в том числе, на обучающих множествах gikrya_train и syntagrus_train [4], т. е. не требовали адаптации тегсета при тестировании. В зависимости от выбираемых обучающих и тестовых данных TreeTagger демонстрировал точность определения граммем в диапазоне 72,10...79,49 %, что значительно ниже полученных в ходе настоящего исследования результатов с готовой моделью и конверсией тегсета. В качестве причин такого расхождения можно предположить недостаточность данных для обучения алгоритма TreeTagger на указанных обучающих множествах, неудачный выбор параметров обучения авторами эксперимента. Согласно работе [25] TreeTagger может достигать даже более высокой планки — 92,56 %.

Точность определения частей речи (из числа подлежащих оценке по условиям MorphoRuEval) приведена в табл. 2.

Более детальный анализ ошибок показал следующее. На дейтасете 3-in-1 системы UDPipe и TreeTagger недобирают точности преимущественно по причине различий в теоретической трактовке. В частности, это проявляется в выборе между наречием и частицей (*только*, *тоже* и т. п.), кратким прилагательным и наречием[13] (*нужно*, *тихо* и т. п.). Результат rnnmorph на syntagrus_train ниже чем на 3-in-1 также по интерпретационной причине. В целом можно отметить:

— довольно высокий общий уровень в определении части речи у всех трех систем;

— отсутствие возможности точного сопоставления по этому показателю[14].

Далее приведем детализацию оценок по отдельным частям речи. В корпусе 3-in-1 подлежащие оценке части речи представлены в следующих пропорциях (см. рисунок).

Результаты оценки для существительных приведены в табл. 3, 4.

Таблица 1

Сравнение систем по правилам MorphoRuEval (точность определения граммем/точность полного разбора, %)

Система	Дейтасет		
	3-in-1	gikrya_train	syntagrus_train
TreeTagger	89,88/85,18	89,69/85,12	88,97/82,98
UDPipe	89,01/84,28	88,90/83,83	94,68/92,36*
rnnmorph	96,28/92,30	98,08/94,99*	93,17/89,71

* Система тестировалась на обучающем множестве или его фрагменте (здесь и далее)

Таблица 2

Точность определения части речи, %

Система	Дейтасет		
	3-in-1	gikrya_train	syntagrus_train
TreeTagger	96,15	95,83	96,08
UDPipe	95,81	95,74	97,08*
rnnmorph	98,78	99,41*	96,87

Таблица 3

Точность обнаружения существительных, %

Система	Дейтасет		
	3-in-1	gikrya_train	syntagrus_train
TreeTagger	98,85	98,92	98,9
UDPipe	97,93	98,23	99,77*
rnnmorph	99,04	99,48*	98,79

Таблица 4

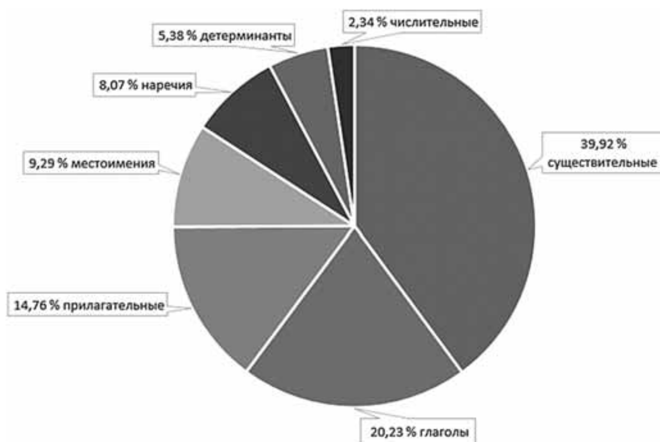
Точность определения граммем/точность полного разбора существительных, %

Система	Дейтасет		
	3-in-1	gikrya_train	syntagrus_train
TreeTagger	90,07/85,97	89,87/85,55	89,55/83,43
UDPipe	88,90/87,04	88,49/86,06	97,29/96,61*
rnnmorph	95,69/91,82	97,56/96,35*	93,04/91,34

---

[13] Особенности MorphoRuEval: предикативы, омонимичные кратким прилагательным, размечаются как краткие прилагательные; омонимия кратких прилагательных и наречий разрешена следующим образом: прилагательное обязано быть частью сказуемого.

[14] Последующие оценки для прилагательных и наречий также будут неточны.



Соотношение подлежащих оценке частей речи в корпусе 3-in-1

Таблица 5

Число типичных ошибок определения граммем для существительных на дейтасете 3-in-1

Эталон	Ошибка	Система		
		UDpipe	TreeTagger	rnnmorph
вин., ед., муж.	им.	35	54	16
им., ед., муж.	вин.	22	45	13
вин., ед., ср.	им.	13	28	4
им., ед., ср.	вин.	6	14	3
им., мн., жен.	род., ед.	6	12	5
им., мн., жен.	вин.	11	9	3
им., ед., жен.	вин.	8	8	5
им., мн., ср.	род., ед.	6	6	7

По качеству обнаружения существительных (см. табл. 3) системы демонстрируют сопоставимые результаты. Снижение точности rnnmorph на syntagrus_train связано, в частности, с регулярной ошибкой при обработке инициалов[15] (распознаются как знак пунктуации). В определении граммем (см. табл. 4) rnnmorph лучше других; наиболее распространенные ошибки для каждой из систем сведены в табл. 5 (в столбце "ошибка" указаны только ошибочно определенные граммемы из набора в столбце "эталон").

Результаты оценки для глаголов приведены в табл. 6, 7.

Здесь можно отметить существенное отставание UDPipe и TreeTagger от rnnmorph по точности построения нормальных форм.

Для остальных частей речи приведем оценку только на дейтасете 3-in-1 (табл. 8).

Кроме того, в ходе исследования были выполнены оценка точности восстановления нормальных форм, независимо от корректности определения граммем (табл. 9), и измерение скорости анализа (табл. 10). Скорость замерялась на наиболее крупном дейтасете gikrya_train, чтобы снизить влияние фактора загрузки модели на оценку скорости обработки.

Исследованная реализация rnnmorph уступила другим системам по скорости анализа. В табл. 10 приведено время, затрачиваемое rnnmorph при обработке дейтасета пакетами по 100 предложений с tensorflow 1.12, оптимизированным под CPU (с поддержкой AVX2). Можно ожидать, что применение GPU позволит добиться лучшей производительности.

[15] Акцентируем на этом внимание, так как в прикладных системах это может быть весьма существенным.

Таблица 6

Точность обнаружения глаголов, %

Система	Дейтасет		
	3-in-1	gikrya_train	syntagrus_train
TreeTagger	98,79	99,09	98,92
UDPipe	98,83	98,95	99,86*
rnnmorph	99,72	99,89*	99,9

Таблица 7

Точность определения граммем/точность полного разбора глаголов, %

Система	Дейтасет		
	3-in-1	gikrya_train	syntagrus_train
TreeTagger	98,02/92,72	98,36/93,29	98,29/93,97
UDPipe	98,22/84,50	98,23/84,58	99,80/91,13*
rnnmorph	99,15/99,03	99,44/98,95*	99,71/99,09

Таблица 8

Точность определения граммем/точность полного разбора на дейтасете 3-in-1, %

Часть речи	Система		
	TreeTagger	UDPipe	rnnmorph
Прилагательное	82,09/71,55	88,02/83,19	95,62/91,63
Местоимение	89,78/89,78	82,47/82,11	95,42/89,60
Наречение	84,60/81,61	87,70/83,68	98,28/90,69
Детерминант	85,24/84,47	71,84/69,56	91,48/78,54
Числительное	92,68/86,76	86,76/86,41	94,08/93,73

Таблица 9

Точность восстановления нормальной формы (независимо от граммем), %

Система	Дейтасет		
	3-in-1	gikrya_train	syntagrus_train
TreeTagger	92,56	92,22	90,68
UDPipe	93,36	93,03	96,65*
rnnmorph	95,28	96,46*	94,79

Таблица 10

Время анализа дейтасета gikrya_train

Система	Время анализа gikrya_train
TreeTagger	1 мин 44 с
UDPipe	2 мин 39 с
rnnmorph	10 мин 40 с

Таблица 11

Точность восстановления нормальной формы (независимо от граммем) — TreeTagger + AOT, %

Система	Дейтасет		
	3-in-1	gikrya_train	syntagrus_train
TreeTagger + AOT	97,19	96,95	95,35
rnnmorph	95,28	96,46*	94,79

Анализ результатов

Результаты эмпирической оценки показали, что при условии следования морфологической нотации MorphoRuEval анализатор rnnmorph имеет бесспорное преимущество над анализаторами из конвейеров Шарова—Нивре и UDPipe по следующим показателям:

— точность определения граммем существительных и местоимений;

— точность определения нормальной формы.

По точности для числительных и глаголов rnnmorph имеет небольшое преимущество. Для прилагательных, наречий и детерминантов точное сравнение затруднительно в силу различий морфологических нотаций, которые не были нивелированы конвертированием. Тем не менее эксперименты показали, что rnnmorph с высокой точностью определяет граммемы и для этих частей речи.

В то же время с прикладной точки зрения необходимо отметить следующее. Во-первых, время, затрачиваемое rnnmorph на анализ, существенно больше, чем у современных конвейерных решений. Во-вторых, неоцененными остались способности систем в выявлении других практически важных атрибутов, таких как одушевленность, вид, залог, принадлежность имени собственному. За кадром также оказались служебные части речи, успешное распознавание которых важно по крайней мере для синтаксического анализа. В-третьих, неисследованным осталось поведение rnnmorph в "нелабораторных" условиях, когда входными данными являются *автоматически* выявленные предложения и токены.

Отметим также, что высокая точность определения нормальных форм может быть достигнута менее затратным (по времени анализа) путем. В рамках проведенного исследования было создано простое гибридное решение на базе TreeTagger и словарей AOT, задачами которого было:

— устранить ошибки TreeTagger при обработке слов, содержащих дефис;

— использовать словарный вариант нормализации (если он единственный).

Дефект с дефисами устранялся эвристически с учетом специфики русского языка. Например, существительные (*вирус-вымогатель*, *красавица-дочь*) разбивались на составляющие и выполнялась словарная нормализация каждой из частей с последующим объединением через дефис. Для ряда слов (таких как *из-за*, *все-таки*) составлялся словарь. За счет таких приемов удалось добиться точности нормализации, превышающей показатели rnnmorph (табл. 11). При этом время анализа на дейтасете gikrya_train составило 3 мин.

В качестве краткого резюме отметим следующее. Встраивание rnnmorph в исследованные конвейеры оправдано при условии, что "скоростные" характеристики rnnmorph удовлетворяют условиям прикладной задачи. С учетом всех факторов наиболее безопасным (без дополнительных исследований) видится следующий сценарий встраивания: параллельный запуск конвейерного морфологического анализатора и rnnmorph с последующим уточнением результатов конвейерного разбора. Нецелесообразно пересматривать результаты конвейерного анализатора в определении частей речи, так как на них опираются синтаксический и более высокие уровни анализа. При совпадении предсказанных конвейерным анализатором и rnnmorph частей речи (из списка тех, что были верифицированы в рамках данного исследования) следует предпочесть граммемы и нормальную форму, выданную rnnmorph. Таким образом, не будут пересмотрены подходы к аннотированию причастий и порядковых числительных, а также сохранится морфологическая информация, не вошедшая в стандарт MorphoRuEval (одушевленность, залог и др.).

В будущем представляется целесообразным провести следующие исследования:

— обучить rnnmorph на дейтасетах, использовавшихся при обучении конвейерных морфологических анализаторов, а затем оценить его эффективность (методом кроссвалидации) и влияние на точность синтаксического разбора;

— количественно исследовать влияние на синтаксический разбор предложенной схемы встраивания rnnmorph в конвейеры.

Отметим также, что все три анализатора допускают значительное число ошибок при обработке многозначных слов (*уже*, *банках*, *судами*, *гвоздики* и т. п.), что обусловлено, вероятно, не методологическими недостатками, а неполнотой обучающих множеств. Было бы интересно оценить возможности систем отдельно по задаче разрешения неоднозначности.

Заключение

В ходе описанного исследования был верифицирован и подтвержден технический уровень модуля морфологического анализа для русского языка rnnmorph. Реабилитирован модуль TreeTagger, что не-

маловажно для тех, кто уже использует его в своих программных продуктах. Получена оценка точности системы UDPipe на наиболее современном дейтасете для русскоязычных морфологических анализаторов. Разработка средств конвертации в целях приведения результатов сравниваемых модулей к единому тегсету позволила осуществить более точное сопоставление, а детальный анализ показал, в какой мере rnnmorph превосходит конвейерные анализаторы по отдельным частям речи и в подзадаче нормализации словоформ.

Полученные результаты и приведенные аналитические выкладки помогут прикладным разработчикам принять решение о встраивании морфологического анализатора rnnmorph в свои программные системы.

Исследование выполнено при финансовой поддержке РФФИ в рамках научного проекта № 19-07-00779.

Автор выражает искреннюю признательность Н. А. Власовой, Ю. П. Сердюку и Е. А. Сулеймановой за помощь в проведении исследования и работе над статьей.

Список литературы

1. **Sharoff S., Nivre J.** The proper place of men and machines in language technology: Processing Russian without any linguistic knowledge // Computational Linguistics and Intellectual Technologies: Proceedings of the International Conference "Dialog 2011". M.: RGGU, 2011. Issue 10. P. 591—605.

2. **Straka M., Hajič J., Straková J.** UDPipe: Trainable Pipeline for Processing CoNLL-U Files Performing Tokenization, Morphological Analysis, POS Tagging and Parsing // Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016). European Language Resources Association (ELRA), 2016. P. 4290—4297.

3. **Straka M., Straková J.** Tokenizing, POS Tagging, Lemmatizing and Parsing UD 2.0 with UDPipe // Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies. Association for Computational Linguistics, 2017. P. 88—99.

4. **Sorokin A., Shavrina T., Lyashevskaya O.** et al. MorphoRuEval-2017: an Evaluation Track for the Automatic Morphological Analysis Methods for Russian // Computational Linguistics and Intellectual Technologies. Proceedings of the International Conference "Dialogue 2017". M.: RGGU, 2017. Vol. 1, Issue 16. P. 297—313.

5. **Anastasyev D. G., Andrianov A. I., Indenbom E. M.** Part-of-speech tagging with rich language description // Computational Linguistics and Intellectual Technologies. Proceedings of the International Conference "Dialogue 2017". M.: RGGU, 2017. Vol. 1, Issue 16. P. 2—13.

6. **Anastasyev D., Gusev I., Indenbom E.** Improving part-of-speech tagging via multi-task learning and character-level word representations // Computational Linguistics and Intellectual Technologies. Proceedings of the International Conference "Dialogue 2018". M.: RGGU, 2018. Issue 17. P. 14—27.

7. **Schmid H.** Probabilistic Part-of-Speech Tagging Using Decision Trees // Proceedings of International Conference on New Methods in Language Processing. 1994. Vol. 12. P. 44—49.

8. **Brants T.** TnT: a statistical part-of-speech tagger // Proceedings of the sixth conference on Applied natural language processing. Association for Computational Linguistics, 2000. P. 224—231.

9. **Giménez J., Màrquez L.** SVMTool: A General POS Tagger Generator Based on Support Vector Machines // Proceedings of the Fourth International Conference on Language Resources and Evaluation (LREC'04). European Language Resources Association (ELRA), 2004. P. 43—46.

10. **Jongejan B., Dalianis H.** Automatic training of lemmatization rules that handle morphological changes in pre-, in- and suffixes alike // *Proceedings of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*. Association for Computational Linguistics, 2009. P. 145—153.

11. **Плунгян В. А.** Зачем нужен Национальный корпус русского языка? Неформальное введение // Национальный корпус русского языка: 2003—2005. Результаты и перспективы. М.: Индрик, 2005. С. 6—20.

12. **Ляшевская О. Н., Плунгян В. А., Сичинава Д. В.** О морфологическом стандарте Национального корпуса русского языка // Национальный корпус русского языка: 2003—2005. Результаты и перспективы. М.: Индрик, 2005. С. 111—135.

13. **Sharoff S., Kopotev M., Erjavec T., Feldman A., Divjak D.** Designing and evaluating Russian tagsets // *Proceedings of the 6th International Conference on Language Resources and Evaluation* (LREC 2008). 2008. P. 279—285.

14. **Ratnaparkhi A.** A Maximum Entropy Model for Part-Of-Speech Tagging // Proceedings of the Conference on Empirical Methods in Natural Language Processing. 1996. P. 133—142.

15. **Collins M.** Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms // Proceedings of the Conference on Empirical Methods in Natural Language Processing. 2002. P. 1—8.

16. **Spoustová D., Hajič J., Raab J., Spousta M.** Semi-Supervised Training for the Averaged Perceptron POS Tagger // Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009). Association for Computational Linguistics, 2009. P. 763—771.

17. **Straka M., Straková J.** Universal Dependencies 2.3 Models for UDPipe (2018-11-15), LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University. 2018. URL: http://hdl.handle.net/11234/1-2898.

18. **Апресян Ю. Д., Богуславский И. М., Иомдин Б. Л.** и др. Синтаксически и семантически аннотированный корпус русского языка: современное состояние и перспективы // Национальный корпус русского языка: 2003—2005. Результаты и перспективы. М.:Индрик, 2005. С. 193—214.

19. **Boguslavsky I.** SynTagRus — a Deeply Annotated Corpus of Russian // Les émotions dans le discours — Emotions in Discourse/ Eds. P. Blumenthal, I. Novakova, D. Siepmann. Germany, Frankfurt am Mine: Peter Lang, 2014. P. 367—380.

20. **Droganova K., Lyashevskaya O., Zeman D.** Data Conversion and Consistency of Monolingual Corpora: Russian UD Treebanks. // Proceedings of the 17th International Workshop on Treebanks and Linguistic Theories (TLT 2018). Linköping University Electronic Press, 2018. No. 155. P. 52—65.

21. **Nivre J., Abrams M., Agić Ž.** et al. Universal Dependencies 2.3, LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University. 2018. URL: http://hdl.handle.net/11234/1-2895.

22. **Hochreiter S., Schmidhuber J.** Long Short-Term Memory // Neural Computation. 1997. Vol. 9, Issue 8. P. 1735—1780.

23. **Belikov V., Kopylov N., Piperski A., Selegey V., Sharoff S.** Corpus as language: from scalability to register variation. // Computational Linguistics and Intellectual Technologies. Proceedings of the International Conference "Dialogue 2013". M.: RGGU, 2013. Issue 12. P. 83—95.

24. **Сокирко А. В.** Морфологические модули на сайте www.aot.ru // Труды межд. конф. "Диалог-2004". М.: Наука, 2004. С. 559—564.

25. **Dereza O., Kayutenko D., Fenogenova A.** Automatic morphological analysis for Russian: A comparative study // Proceedings of the International Conference Dialogue 2016. Computational linguistics and intellectual technologies. Student session (online publication). 2016. URL: http://www.dialog-21.ru/media/3473/dereza.pdf

Automatic Morphological Analysis for Russian: Application-Oriented Survey

I. V. Trofimov, itrofimov@gmail.com, Ailamazyan Program Systems Institute of RAS, Pereslavl-Zalessky, 152020, Russian Federation

Corresponding author:

Trofimov Igor' V., Senior Researcher, Ailamazyan Program Systems Institute of RAS, Pereslavl-Zalessky, 152020, Russian Federation

E-mail: itrofimov@gmail.com

Received on July 22, 2019
Accepted on August 06, 2019

Researchers who focus on higher-level NLP tasks, and NLP application developers often rely on off-the-shelf solutions for lower-level subtasks like tokenization, sentence segmentation, lemmatizing, morphological tagging, and dependency parsing.

The paper presents an accuracy evaluation of two morphological modules for the Russian language: the one used within the Sharoff&Nivre's pipeline, and UDPipe. Their performance is compared against rnnmorph neural algorithm that showed the best results at the MorphoRuEval-2017 competition. For evaluation purposes we used its implementation as of May 2019.

The study uses the datasets from MorphoRuEval and follows its evaluation framework. The experiments have revealed in which respects and to what extent rnnmorph outperforms the state-of-the-art pipeline solutions. Specifically, rnnmorph proves to be highly accurate (> .95) in identifying grammemes of nouns and pronouns, which is relevant for syntactic analysis of Russian. It is worth mentioning that rnnmorph was trained using five times less training data than TreeTagger, the morphological analyzer in the Sharoff's and Nivre's pipeline. At the same time, rnnmorph is fairly slow, and the trained model at hand fails to generate a number of key morphological features.

The comparative study data and supporting analyses presented in the paper will be of help for software designers challenged with the choice of a morphological analyzer to build into their applications.

***Keywords:** natural language processing, morphological analysis, MorphoRuEval, TreeTagger, UDPipe, rnnmorph, morphological tagsets, lemmatization, Russian language*

Acknowledgements: *The reported study was funded by RFBR according to the research project No. 19-07-00779.*

For citation:

Trofimov I. V. Automatic Morphological Analysis for Russian: Application-Oriented Survey, *Programmnaya Ingeneria,* 2019, vol. 10, no. 9—10, pp. 391—399.

DOI: 10.17587/prin.10.391-399

References

1. **Sharoff S., Nivre J.** The proper place of men and machines in language technology: Processing Russian without any linguistic knowledge, *Computational Linguistics and Intellectual Technologies: Proc. International Conference "Dialog 2011"*, Moscow, RGGU, 2011, issue 10, pp. 591—605.
2. **Straka M., Hajič J., Straková J.** UDPipe: Trainable Pipeline for Processing CoNLL-U Files Performing Tokenization, Morphological Analysis, POS Tagging and Parsing, *Proc. Tenth International Conference on Language Resources and Evaluation (LREC 2016)*, European Language Resources Association (ELRA), 2016, pp. 4290—4297.
3. **Straka M., Straková J.** Tokenizing, POS Tagging, Lemmatizing and Parsing UD 2.0 with UDPipe, *Proc. CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, Association for Computational Linguistics, 2017, pp. 88—99.
4. **Sorokin A., Shavrina T., Lyashevskaya O., Bocharov V., Alexeeva S., Droganova K., Fenogenova A., Granovsky D.** MorphoRuEval-2017: an Evaluation Track for the Automatic Morphological Analysis Methods for Russian, *Computational Linguistics and Intellectual Technologies. Proc. International Conference "Dialogue 2017"*, Moscow, RGGU, 2017, vol. 1, issue 16, pp. 297—313.
5. **Anastasyev D. G., Andrianov A. I., Indenbom E. M.** Part-of-speech tagging with rich language description, *Computational Linguistics and Intellectual Technologies. Proc. International Conference "Dialogue 2017"*, Moscow, RGGU, 2017, vol. 1, issue 16, pp. 2—13.
6. **Anastasyev D., Gusev I., Indenbom E.** Improving part-of-speech tagging via multi-task learning and character-level word representations, *Computational Linguistics and Intellectual Technologies. Proc. International Conference "Dialogue 2018"*, Moscow, RGGU, 2018, issue 17, pp. 14—27.
7. **Schmid H.** Probabilistic Part-of-Speech Tagging Using Decision Trees, *Proc. International Conference on New Methods in Language Processing*, 1994, vol. 12, pp. 44—49.
8. **Brants T.** TnT: a statistical part-of-speech tagger, *Proc. Sixth conference on applied natural language processing*, Association for Computational Linguistics, 2000, pp. 224—231.
9. **Giménez J., Màrquez L.** SVMTool: A General POS Tagger Generator Based on Support Vector Machines, *Proc. Fourth International Conference on Language Resources and Evaluation (LREC'04)*, European Language Resources Association (ELRA), 2004, pp. 43—46.
10. **Jongejan B., Dalianis H.** Automatic training of lemmatization rules that handle morphological changes in pre-, in- and suffixes alike, *Proc. 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, Association for Computational Linguistics, 2009, pp. 145—153.
11. **Plungyan V. A.** What do we need Russian National Corpus for? An informal introduction, *Natsionalnyi korpus russkogo yazyka: 2003—2005. Rezul'taty i perspektivy*, Moscow, Indrik, pp. 6—20 (in Russian).
12. **Lyashevskaya O. N., Plungyan V. A., Sichinava D. V.** Morphological standard of the Russian National Corpus, *Natsionalnyi korpus russkogo yazyka: 2003—2005. Rezul'taty i perspektivy,* Moscow, Indrik, pp. 111—135 (in Russian).
13. **Sharoff S., Kopotev M., Erjavec T., Feldman A., Divjak D.** Designing and evaluating Russian tagsets, *Proc. 6th International Conference on Language Resources and Evaluation (LREC 2008)*, 2008, pp. 279—285.
14. **Ratnaparkhi A.** A Maximum Entropy Model for Part-Of-Speech Tagging, *Proc. Conference on Empirical Methods in Natural Language Processing*, 1996, pp. 133—142.
15. **Collins M.** Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms, *Proc. Conference on Empirical Methods in Natural Language Processing*, 2002. pp. 1—8.
16. **Spoustová D., Hajič J., Raab J., Spousta M.** Semi-Supervised Training for the Averaged Perceptron POS Tagger, *Proc. 12th Conference of the European Chapter of the ACL (EACL 2009)*, Association for Computational Linguistics, 2009, pp. 763—771.
17. **Straka M., Straková J.** Universal Dependencies 2.3 Models for UDPipe (2018-11-15), *LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University*, 2018. available at: http://hdl.handle.net/11234/1-2898.
18. **Apresyan Yu. D., Boguslavsky I. M., Iomdin B. L.** et al. Syntactically and Semantically Annotated Corpus of Russian: State-of-the-Art and Prospects, *Natsionalnyi korpus russkogo yazyka: 2003—2005. Rezul'taty i perspektivy,* Moscow, Indrik, 2005, pp. 193—214 (in Russian).
19. **Boguslavsky I.** SynTagRus — a Deeply Annotated Corpus of Russian, *Les émotions dans le discours — Emotions in Discourse* / Eds P. Blumenthal, I. Novakova, D. Siepmann, Germany, Frankfurt am Mine, Peter Lang, 2014, pp. 367—380.
20. **Droganova K., Lyashevskaya O., Zeman D.** Data Conversion and Consistency of Monolingual Corpora: Russian UD Treebanks, *Proc. of the 17th International Workshop on Treebanks and Linguistic Theories (TLT 2018)*, Linköping University Electronic Press, 2018, no. 155, pp. 52—65.
21. **Nivre J., Abrams M., Agić Ž.** et al. Universal Dependencies 2.3, *LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University*, 2018, available at: http://hdl.handle.net/11234/1-2895.
22. **Hochreiter S., Schmidhuber J.** Long Short-Term Memory, *Neural Computation*, 1997, vol. 9, issue 8, pp. 1735—1780.
23. **Belikov V., Kopylov N., Piperski A., Selegey V., Sharoff S.** Corpus as language: from scalability to register variation. *Computational Linguistics and Intellectual Technologies. Proc. International Conference "Dialogue 2013"*, Moscow, RGGU, 2013, Issue 12, pp. 83—95.
24. **Sokirko A. V.** Morphological Modules on www.aot.ru Website, *Computational Linguistics and Intellectual Technologies. Proc. International Conference (Dialogue'2004)*, Moscow, 2004, pp. 559—564 (in Russian).
25. **Dereza O., Kayutenko D., Fenogenova A.** Automatic morphological analysis for Russian: A comparative study, *Proc. International Conference Dialogue 2016. Computational linguistics and intellectual technologies. Student session (online publication)*, 2016. available at: http://www.dialog-21.ru/media/3473/dereza.pdf

УДК 004.4'22 DOI: 10.17587/prin.10.400-416

А. П. Соколов, канд. физ.-мат. наук, доц., e-mail: alsokolo@bmstu.ru,
В. М. Макаренков, студент, **А. Ю. Першин**, ассистент, e-mail: tony.pershin@gmail.com,
И. А. Лаишевский, бакалавр, e-mail: ivanlaish@yandex.ru, Московский Государственный технический университет имени Н. Э. Баумана

Разработка программного обеспечения генерации кода на основе шаблонов при создании систем инженерного анализа

Генерация кода на основе шаблонов предполагает автоматизированное формирование исходного кода программ или некоторого текстового результата, соответствующего заранее определенному формату, называемому шаблоном. Представлен программный инструментарий автоматизированного прототипирования программных модулей и документов различного типа и назначения. В основе созданных технических решений лежит применение разработанных авторами специализированного формата представления шаблонов, формата aINI файлов исходных данных и библиотеки функций, позволяющей интерпретировать шаблоны и создавать на их основе объекты различных типов. Представлены алгоритм генерации и метод автоматической интеграции созданного программного инструментария в рамки программного комплекса "Распределенная вычислительная система GCD". Приведены примеры практически значимых шаблонов с описанием их применения. В их числе — шаблоны файлов исходных данных; шаблон документации на программную реализацию сложного вычислительного метода; шаблон отчета о проведенных патентных исследованиях согласно ГОСТ Р 15.011—96.

Ключевые слова: *автоматизация процессов разработки; быстрое прототипирование программного обеспечения и документации; template-based code generation; model-driven engineering; model-to-text transformation; aINI; генерация кода; автоматизация программирования; интерпретация текстовых данных; программные процессоры; технологии разработки систем инженерного анализа*

Введение

Процесс ведения разработки современного прикладного программного обеспечения (ПО) предполагает, помимо прочего, необходимость написания существенного объема исходных кодов программ, что может быть реализовано только при слаженной работе команды разработчиков. При разработке больших программных систем всегда встает задача согласованного повторного использования исходного кода. При любом дублировании исходного кода могут возникать копии ошибок, а с увеличением масштаба разрабатываемого ПО число таких ошибок возрастает многократно. Сокращение числа ошибок в конечном ПО обычно обеспечивается следующими способами:

— регламентацией и стандартизацией процессов разработки;

— созданием таких условий для разработки, при которых с увеличением числа функциональных возможностей системы сохраняется линейный рост сложности внесения изменений в каждую отдельную функцию (реализуется глубоко продуманной архитектурой разрабатываемого ПО, возможностями гибкого расширения функциональных возможностей и пр.);

— применением специализированных программных средств автоматизированной поддержки процессов разработки (CASE-инструментариев, *Computer-aided Software Engineering Tools*).

Применение средств автоматизации процессов разработки ПО является признаком высокого уровня зрелости команды разработки, и в особенности оно необходимо при создании крупных расширяемых программных комплексов [1]. Среди прочих[1] в состав CASE-инструментариев входят программные средства генерации кода различных типов [2].

Применение методов автоматизированной генерации исходного кода программных модулей и до-

[1] Известные CASE-инструментарии — системы управления конфигурациями: контроля версий (SVN, GIT, Mercurial и пр.), построения инсталляторов, непрерывной интеграции (CI); средства управления требованиями; средства планирования (например, Redmine, Jira, MS SharePoint и др.); средства анализа программ (анализаторы программ); средства тестирования, отладки, документирования; системы автоматизации построения графических пользовательских интерфейсов; системы автоматизации генерации кода и др.

кументации началось еще в 50-х годах прошлого века при создании первых компиляторов [3—4]. Известны следующие подходы к генерации кода:

1) преобразование графического представления программного объекта в исходный код (*model-to-text transformations*, M2T) [5];

2) генерация кода на основе шаблонов (*template-based code generation*, TBCG) [4].

Использование первого подхода сопряжено с применением парадигмы модель-ориентированной разработки программного обеспечения (*model-driven engineering*, MDE) [5]. В англоязычных публикациях в контексте MDE используют термин "модель объекта", под которым понимают структуру классов, реляционную модель данных и пр. В рамках MDE с использованием некоторого языка моделирования (например, UML, IDEF0, ARIS [5, 6]) осуществляется построение "модели объекта", ее интерпретация и построение заготовок исходного кода программного обеспечения. Примеры прикладного применения генераторов кода на основе графического представления элементов программных систем (иерархий классов, реляционных моделей данных и пр.) представлены во многих работах, например [7, 8]. В настоящей работе эти подходы, несмотря на их широкое использование, не рассматриваются.

Предметом настоящей работы являются методы генерации кода на основе шаблонов, обеспечивающие автоматизированное формирование некоторого текстового результата (объект генерации), соответствующего заранее определенному формату, называемому шаблоном (рис. 1). Программные средства, реализующие соответствующие методы, называют "генераторами кода на основе шаблонов".

Применение генераторов кода на основе шаблонов обеспечивает прототипирование программного обеспечения, а именно — автоматизирует процедуры создания следующих заготовок (прототипов): исходного кода отдельных программных структур данных; программных модулей на различных языках программирования (например, высокоуровневых языках C++, Java и др.); исходного кода программной документации, разрабатываемой, например, с использованием языка LaTeX; сценариев на языках систем автоматизации сборки (например, CMake) и т. д. Очевидно, такие возможности могут существенно ускорить процессы разработки программного обеспечения.

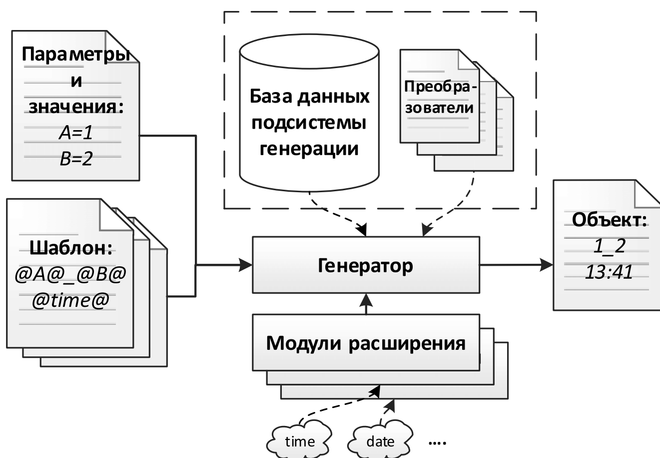


Рис. 1. Традиционная схема процесса генерации кода на основе шаблонов

Процедуры генерации принято различать в зависимости от типа применяемых шаблонов [4]:

1) на основе предварительно определенных шаблонов (*predefined*) или "зашитых" в генерирующую программу;

2) основанные на формате генерируемого выходного объекта (*output-based*);

3) основанные на множестве формализованных правил (*rule-based*).

Традиционно шаблоны состоят из двух частей: статической (постоянной при генерации) и динамической (изменяемой при генерации). Предварительно определенные шаблоны (тип 1) представляют собой такой класс шаблонов, для которых статическая часть может модифицироваться, а результат динамической части сильно зависит от применяемого генератора. Шаблоны, основанные на формате генерируемого выходного объекта (тип 2), представляют собой такие текстовые данные, которые синтаксически соответствуют требуемому на выходе результату генерации. Такой класс шаблонов наиболее часто применяется на практике. Наконец, шаблоны, основанные на правилах (тип 3), предполагают возможности определения формальных правил генерирования непосредственно в динамической части тела шаблона, указывая на необходимость применения специальных алгоритмов вычисления значений выходных параметров. Генераторы, поддерживающие такие шаблоны, обычно более сложные, так как должны обеспечивать интерпретацию соответствующих правил, включенных в состав шаблона. Правила в свою очередь могут быть определены независимо.

Среди известных программных средств и технологий, применяемых для генерации кода на основе шаблонов различных типов, следует отметить (в порядке снижения популярности согласно работе [4]): Acceleo, Xpand, XSLT, Velocity, JET, Fujaba, Simulink TLC, EGL, Rational, String Template, MOF Script, Free Marker, Rhapsody, Xtend и др.

Отметим, что большинство из представленных программных средств предполагают оплату лицензии для их использования [9], а существующие бесплатные разработки либо имеют ограниченные функциональные возможности, либо узкоспециализированы. В обзоре [4] и в докладе [10] представлено подробное сравнение известных систем генерации кода на основе шаблонов.

Генераторы кода, в том числе на основе шаблонов, широко применяют при создании ПО различного назначения: системы, использующие базы данных [11, 12]; встраиваемые системы реального времени [13—15]; web-ориентированные системы [16]; инженерные программные системы [17] и пр. Отдельный интерес представляют генераторы кода, использующие шаблоны, разрабатываемые на специализированном языке [18, 19], либо формирующие код на основе предварительного определения многоуровневого набора правил и без формирования шаблонов вовсе [20].

Информации об отечественных программных средствах генерации кода, доступной в открытой печати, достаточно мало. Например, в работе [21] кратко представлена система автоматизации процесса разработки AT Studio 2015, которую разрабатывает группа компаний "ИВС".

Многообразие существующих решений, в том числе и вновь разрабатываемых [11, 12, 14—18, 20, 21], по мнению авторов, связано с принципиальной важностью развития собственных CASE-инструментариев при создании коммерческого программного обеспечения. Использование в коммерческих разработках сторонних CASE-инструментариев приводит к серьезной внешней зависимости организации от последних.

Несмотря на длительную историю, в области генерации кода до сих пор существуют требующие решения актуальные задачи [4], связанные, например:

— со снижением трудоемкости процессов разработки и обновления шаблонов с учетом изменяющихся требований к конечным объектам генерации[2];

— с применением методов генерации кода при разработке наукоемкого программного обеспечения, в частности, при создании систем математического моделирования и инженерного анализа[3];

— с анализом эффективности процессов генерации кода, в том числе на основе шаблонов, при создании программного обеспечения различного назначения и масштаба.

В рамках проводимых научно-исследовательских и опытно-конструкторских работ авторы ведут разработку прикладного программного обеспечения инженерного анализа. В частности, идет разработка расширяемого программного комплекса "Распределенная вычислительная система GCD" (PBC GCD) [22]. В целях повышения эффективности и качества процесса разработки в целом, уменьшения числа ошибок при создании новых подсистем и программных модулей, автоматизации процессов разработки документации, а также в целях обеспечения независимости ядра и других компонентов системы от стороннего программного обеспечения был создан представляемый в настоящей работе программный инструментарий [23, 24].

Представленные в работе решения основаны на принципах и подходах, активно применяемых при разработке и внедрении промышленных информационных систем. Созданный и представленный в работе программный инструментарий стал основой для разработки CASE-инструментария PBC GCD [24].

[2] Например, если шаблоном определен выходной язык, то возможно получение ошибочно формируемого кода при синтаксически несоответствующих подстановках вычисленных значений параметров, используемых в шаблоне. Такое поведение возможно в случае модификации функции вычисления значения параметра либо при неверном изменении значения параметра.

[3] Разработка наукоемкого ПО представляет самостоятельную сложность, а с усложнением применяемых математических моделей, вычислительных методов и развитием вычислительных систем актуальность применения новых прогрессивных методов создания ПО будет лишь расти.

Актуальность разработки обусловлена существенной трудоемкостью, а часто и невозможностью создания расширяемого и сопровождаемого программного обеспечения без применения вспомогательных автоматизирующих процессы разработки программных средств.

Целью публикации является представление отмеченного выше программного инструментария, позволяющего автоматизировать процессы разработки программного обеспечения и документации, включая каркасы программных компонентов, проекты библиотек, модулей расширения, файлов исходных данных для произвольных функций системы и прочие объекты. Ключевой составляющей инструментария стала библиотека функций генерации кода на основе шаблонов.

1. Проектирование программного инструментария

К разработанному программному инструментарию были предъявлены перечисленные далее функциональные требования.

1. Формат шаблона должен быть текстовым.
2. Должна быть реализована поддержка шаблонов, зависящих от других шаблонов.
3. Должны быть реализованы функции регистрации шаблонов и их атрибутов, включая расположение шаблона, в специальной базе данных.
4. Функции генерации не должны зависеть от выбранного шаблона и его параметров.
5. Должна быть реализована поддержка параметров различных типов: а) скалярные и векторные параметры; б) статические (терминальные и нетерминальные) и динамические (функциональные) параметры; в) общие (шаблон-независимые) и дополнительные (шаблон-зависимые) параметры.
6. Список параметров определенного шаблона не должен быть фиксирован, чтобы оставалась возможность его расширения.
7. Должна быть обеспечена поддержка возможностей загрузки значений параметров из различный источников: файла входных данных; файла дополнительных параметров, специфичных выбранному шаблону; базы данных; объекта специального класса (с целью возможной интеграции с другими системами).
8. Должна быть реализована возможность преобразования получаемых из внешнего источника данных в произвольный текстовый вид в зависимости от синтаксиса определения параметра в шаблоне.
9. В зависимости от типов используемых в шаблоне параметров должны применяться различные алгоритмы вычисления значений параметров.
10. Подготовка списка дополнительных параметров и их значений при создании нового шаблона или редактировании существующего шаблона должна быть доступной для неподготовленного специалиста, не владеющего навыками программирования.
11. Непосредственный ввод значений параметров объекта при генерации должен быть обеспечен с и без использования графического пользовательского интерфейса (GUI).

12. Процесс построения GUI должен осуществляться во время выполнения программы при запросе значений параметров для выбранного пользователем шаблона до генерации.

13. Программный инструментарий должен иметь возможность встраивания в web-ориентированные приложения, приложения для мобильных платформ и приложения для операционных систем семейств Windows, Linux.

В следующем подразделе представлены определения понятий, которые стали основой для разработки архитектуры программного инструментария генерации кода на основе шаблонов. Далее для краткости будем использовать термины "генерация на основе шаблонов" или просто "генерация".

1.1. Базовые понятия

В табл. 1 представлен перечень специальных символов и обозначений. При конкретной программной реализации указанные символы могут быть заменены на иные с сохранением семантики.

Далее представлены введенные вспомогательные определения.

Определение 1. Скалярным параметром шаблона будем называть строковую константу @ParamName@, слева и справа ограниченную символом '@'.

Имя параметра ParamName рекомендуется определять с использованием нотации CamelCase (табл. 1).

Определение 2. Векторным параметром шаблона будем называть текстовое выражение, соответствующее представленному далее синтаксису.

В этом выражении:

@DataSource@ — скалярный параметр, значение которого должно быть либо именем таблицы базы данных, либо именем файла в формате XDBT (XML-формат для описания таблицы базы данных и ее контента после выгрузки, применяемый в PBC GCD);

<filter> — необязательный параметр, определяющий одно или несколько условий отбора данных из указанного источника @DataSource@, где <attr _ i _ name> – имя некоторого *i*-го атрибута источника;

<value> — значение атрибута, задаваемое либо явно в виде строковой константы <string _ value>, либо неявно ссылкой на некоторый скалярный параметр @ScalarParamName@, либо неявно ссылкой на скалярный параметр @DBTablePKValue@.<attr _ j _ name>, вычисляемый автоматически в предположении, что @DBTablePKValue@ — имя параметра, на основе которого могут быть определены идентификатор некоторого другого источника векторных данных (таблицы базы данных) и значение атрибутов, идентифицирующих единственный объект этого источника, тогда как <attr _ j _ name> определяет имя атрибута источника, задаваемого на основе @DBTablePKValue@;

<db _ table _ name> — имя таблицы базы данных, доступ к которой предполагается обеспеченным;

<xdbt _ table _ name> — имя файла в формате XDBT, представляющего либо таблицу базы данных, либо результат запроса к базе данных в виде пред-

```
%#@DataSource@{<filter>}:@OutputFormat@#%
@DataSource@ = {<db _ table _ name>|<xdbt _ table _ name>}
<filter> = [<attr _ i _ name> = <value> [{AND|OR}...]]
<value> = {<string _ value>|@ScalarParamName@|@DBTablePKValue@.<attr _ j _ name>}
@OutputFormat@ = <name _ of _ output _ format>
```

Таблица 1

Применяемые специальные обозначения

Обозначение	Описание
'@'	Используется для отделения скалярного параметра от остального текста в составе определения шаблона
"скобки" '%#...#%'	Используются для отделения векторного параметра от остального текста в составе определения шаблона
~	Используется в контексте *A* ~ *B*, что будет обозначать, что *B* есть тип объекта *A*
char(N)	Обозначение строкового типа, объекты которого должны состоять ровно из *N* символов
varchar(N)	Обозначение строкового типа, объекты которого должны состоять не более чем из *N* символов
[...]	Квадратные скобки используются для определения фильтров при вычислении значений векторных параметров
{...}	Фигурные скобки используются для обозначения необязательных данных, при этом при подстановке значений эти скобки следует удалить
lc	Указание на использование нижнего регистра (lower case)
uc	Указание на использование верхнего регистра (UPPER CASE)
CamelCase	Указание на использование нотации CamelCase (в одно слово английскими буквами, каждое слово с большой буквы, остальные строчные)

варительно выгруженного XML-файла, доступ к которому предполагается обеспеченным;

<name _ of _ output _ format> — имя формата представления результата, зарегистрированного в базе данных системы, которому ставится в соответствие конкретный метод преобразования, основанный, например, на XSL-преобразовании.

Определение 3. Шаблоном генератора кода будем называть непустой каталог в файловой системе, который может включать в свой состав как файлы, так и каталоги с файлами, имеющий следующие свойства: имена файлов и каталогов могут включать определения скалярных параметров; содержание текстовых файлов может включать определение как скалярных так и векторных параметров.

Определение 4. Шаблон *A* будем называть зависящим от шаблона *B*, если среди файлов, определяющих шаблон *A*, присутствует файл с именем `B.ref`.

Скалярные параметры шаблонов подразделяют на статические и динамические в зависимости от метода вычисления значения параметра.

Определение 5. Скалярный параметр будем называть *динамическим*, если его значение может быть вычислено исключительно путем запуска специальной программной функции.

Определение 6. Скалярный параметр будем называть *статическим*, если его значение задано явно в исходных данных (файлах исходных данных или специальных таблицах базы данных) и/или может зависеть от значений других скалярных параметров статических и/или динамических.

Далее приведем пример значений статических параметров.

```
@Param1@ = value1
@Param2@ = @Param1@value2
@Param3@ = value3@Param2@value2
@Param4@ = @DynamicParam@
```

В представленном примере:

`@Param1@` — скалярный статический терминальный параметр с постоянным явно заданным значением;

`@Param2@` и `@Param3@` — скалярные статические нетерминальные параметры с постоянными явно заданными значениями;

`@DynamicParam@` — скалярный динамический параметр, значение которого вычисляется в зависимости от реализации функций генерации;

`@Param4@` — скалярный статический параметр с неизвестным значением, определяемым значением скалярного динамического параметра `@DynamicParam@`.

При разработке шаблона используемые параметры следует разделять на шаблон-независимые и шаблонзависимые.

Определение 7. Параметр будем называть *шаблоннезависимым*, если он может использоваться более чем в одном шаблоне.

Определение 8. Параметр будем называть *шаблонзависимым*, если его использование возможно только в одном шаблоне.

Определение 9. Шаблон будем называть *гибридным*, если в составе его признаков можно выделить признаки шаблонов на основе форматов выходных данных и шаблонов, основанных на правилах.

В контексте представляемой разработки все используемые шаблоны гибридные, поэтому здесь и далее под шаблонами подразумеваются гибридные шаблоны.

1.2. Использование параметров шаблонов векторного типа

При использовании параметров векторного типа источник данных, как и формат представления результата, могут быть указаны в шаблоне явно, однако это формирует зависимость шаблона от конкретных источника данных и/или формата представления результата соответственно.

Для устранения указанных зависимостей рекомендуется использовать скалярные параметры шаблонов в составе векторных параметров. Представим сказанное на примере (листинг 1).

```
...
%#@DataSource@[attr1=@value@]:@OutputFormat@#%
...
```

Листинг 1. Фрагмент кода некоторого шаблона с неявным определением векторного параметра, зависящего от значений скалярных параметров: `@DataSource@` — источник данных; `attr1` — идентификатор существующего атрибута источника данных `@DataSource@`; `@value@` — требуемое значение атрибута; `@OutputFormat@` — идентификатор выходного формата

1.3. Особенности фильтрации значений параметров шаблонов векторного типа

В дополнение к представленному на листинге 1 примеру следует заметить, что значение `@value@` должно быть каким-то образом определено, а именно, либо введено разработчиком шаблона явно, либо задано статически среди входных параметров шаблона (например, в коде).

В рамках PBC GCD для определения значений входных параметров применяется текстовый формат aINI (листинг 2). Подробное его описание дано в работе [25].

```
1 aini _ file = [sections]. (*aINI-файл состоит из секций*)
2 sections = section ["\n" sections]. (*Секций может быть произвольное число*)
3 section = [comment "\n"] "[" ID "]" [comment] "\n" parameters. (*До и после секции воз-
  можны комментарии. Секции состоят из параметров*)
4 comment = "//" TEXT.
5 parameters = [[("-"|"*")] parameter "\n"] [parameters].
6 parameter = (sub _ parameter _ id | ID) " = " value ["//" comment].
7 sub _ parameter _ id = set _ id "$" ID. (*Идентификатор субпараметра*)
```

```
 8 value = (text _ value | bool _ value | file _ ref | dbtable _ ref | dim _ value | array _
   value | interval _ value | combobox _ value).
 9 text _ value = TEXT. (*Текстовое значение*)
10 bool _ value = "[" bool _ val "](0|1)". (*Логическое значение*)
11 bool _ val = ("0" | "1").
12 file _ ref = "[" file _ name "]". (*Ссылка на файл (возможно известного формата)*)
13 file _ name = ID "." ext. (*Имя файла*)
14 ext = ID. (*Расширение файла (определяет формат)*)
15 dbtable _ ref = "[" rec _ id "]$" schema "." tablename. (*Ссылка на запись в таблице БД*)
16 schema = ID. (*Схема данных*)
17 tablename = ID. (*Имя таблицы*)
18 rec _ id = pk _ f [";" rec _ id]. (*Значения атрибутов первичного ключа таблицы, опре-
   деляющих одну запись*)
19 dim _ value = ID "[[" ID "]]". (*Значение с размерностью*)
20 array _ value = "(" array ")". (*Значение массива*)
21 array = single _ value [";" array].
22 single _ value = (TEXT | set _ id).
23 set _ id = ID.
24 interval _ value = "[" current ";" min " = " max ";" step "]". (*Значение, задаваемое
   диапазоном*)
25 current = TEXT. (*Текущее значение*)
26 min = TEXT. (*Минимальное значение*)
27 max = TEXT. (*Максимальное значение*)
28 step = TEXT. (*Предпочтительный шаг*)
29 combobox _ value = "[" current _ id "](" ID [";" ID] ")". (*Множество с текущим выбран-
   ным элементом*)
30 current _ id = ID. (*Текущий выбранный идентификатор*)
```

Листинг 2. Синтаксис языка aINI в расширенной форме Бекуса—Наура с комментариями, где для краткости опущены определения нетерминальных символов ID (строковый идентификатор) и TEXT (строковая константа)

Пусть требуется вывести массив данных из одного источника (источник `@DataSource1@`), фильтруемых при известных значениях некоторых атрибутов другого источника (источник `@DataSource2@`), связанного с первым, например, внешним ключом. В таком случае указанный в листинге 1 синтаксис может быть обобщен (листинг 3).

```
...
%#@DataSource1@[attr1 = @DataSource2.attr1@]:@OutputFormat@#%
...
```

Листинг 3. Фрагмент кода шаблона с неявным определением векторного параметра, зависящего от значений скалярных параметров: `@DataSource1@`, `@DataSource2@` — источники данных

В случае, если определение значений параметров `@DataSource1@` и `@DataSource2@` осуществляется с использованием файла исходных данных и/или файла дополнительных параметров в формате aINI (см. листинг 2), то возможно существенно упростить процедуру определения значения `@DataSource2.attr1@` за счет применения параметров типа "ссылка на запись в таблице базы данных" (см. листинг 2, строка 15).

Пусть определен aINI-файл исходных данных (листинг 4) постановки задачи генерации.

```
...
Template = [PAT]$sys.tmpls // Идентификатор шаблона
...
```

Листинг 4. Фрагмент aINI-файла исходных данных генератора кода

Пусть определен aINI-файл дополнительных параметров (листинг 5), определяющий значения параметров, специфичных выбранному шаблону.

```
...
DataSource1 = sld.matpr // Параметры материалов
DataSource2 = [03X16H15M3]$sld.matrl // Идентификатор материала
OutputFormat = [TXT]$gen.frmts // Выходной формат
...
```

Листинг 5. Фрагмент aINI-файла дополнительных параметров шаблона

Параметр `@DataSource2@` (листинг 5) имеет aINI-тип "ссылка на запись в таблице базы данных" [25], значением которого является идентификатор материала `03X16H15M3`, определяющий значения атрибута(ов) первичного ключа источника данных `sld.matrl`.

В связи с тем, что значение первичного ключа определено, то формально при запросе данных из источника `sld.matrl`, соответствующих этому значению ключа, будет возвращена единственная запись, в составе которой могут присутствовать также и значения неключевых атрибутов. В таком случае очевидной является возможность динамического дополнения списка скалярных параметров генератора во время выполнения новыми параметрами. Названия таких параметров должны соответствовать именам неключевых атрибутов, а значения — значениям этих атрибутов.

Для реализации возможностей последующего уникального выбора значений параметров из

конечного списка, имена таких параметров при их дополнении должны быть уникальными. Обеспечение подобной уникальности реализуется алгоритмом формирования имен автоматически добавляемых параметров, имеющих очевидный синтаксис: `@db _ record _ ref@.[attr]`, где `@db _ record _ ref@` — имя параметра, имеющего aINI-тип "ссылка на запись в таблице базы данных" и присутствующего среди входных параметров задания на генерацию `[attr]` — идентификаторы атрибутов анализируемой таблицы, загружаемые при запросе данных.

Например, пусть таблицы `sld.matrl` и `sld.matpr` определены согласно листингам 6, 7. Ключевые атрибуты подчеркнуты.

```
CREATE TABLE sld.matrl
(
  matna sld.matna NOT NULL DEFAULT sys.get _ next _ user _ sid('MAT'),
  dscra com.dscra,
  rsobj srg.rsobj,
  CONSTRAINT matrl _ pkey PRIMARY KEY (matna),
  CONSTRAINT matrl _ rsobj _ fkey FOREIGN KEY (rsobj)
  REFERENCES srg.rsobs (rsobj)
)
```

Листинг 6. Пример определения таблицы sld.matrl (DataSource2)

```
CREATE TABLE sld.matpr
(
  matna sld.matna NOT NULL,
  ioprm com.ioprm NOT NULL,
  detmt com.detmt NOT NULL,
  unitt com.unitt NOT NULL,
  sllnk com.dscra,
  CONSTRAINT matpr _ pkey PRIMARY KEY (matna, ioprm, detmt, unitt),
  CONSTRAINT fk _ sld _ matpr _ matna FOREIGN KEY (matna)
  REFERENCES sld.matrl (matna)
)
```

Листинг 7. Пример определения таблицы sld.matpr (DataSource1)

Пусть в выбранном шаблоне PAT встречается синтаксическая конструкция согласно листингу 8.

```
...
В работе проводилось исследование эффективных термоупругопрочностных характеристик (ЭТУПХ) композиционного материала @DataSource2.matna@ (@DataSource2.dscra@).
Значения полученных ЭТУПХ представлены в таблице ниже.
%#@DataSource1@[matna = @DataSource2.matna@]:@OutputFormat@#%
...
```

Листинг 8. Фрагмент кода шаблона PAT

Таким образом, при использовании шаблона PAT список параметров генератора кода должен быть динамически расширен новыми параметрами согласно листингу 9.

```
...
DataSource2.matna = 03X16H15M3
DataSource2.dscra = <Описание материала 03X16H15M3>
...
```

Листинг 9. Дополнительные параметры, формируемые автоматически, включая определение их значений

Аналогичное дополнение новыми параметрами обеспечивается для любого параметра, имеющего aINI-тип "ссылка на запись в таблице базы данных". Операция дополнения должна осуществляться при формировании объекта на основе выбранного шаблона.

1.4. Архитектура программного инструментария

Первая версия инструментария была разработана А. П. Соколовым в 2007 г. при создании программного комплекса GCAD v.2., предназначенного для автоматизированного прогнозирования свойств композиционных материалов [26]. В рамках разработки РВС GCD (GCAD v.3) [22] А. П. Соколовым, А. Ю. Першиным и В. М. Макаренковым была создана вторая версия инструментария, архитектура которой представляется в настоящей работе.

Схема разработанного программного инструментария генерации кода в целом соответствует традиционной (рис. 1). Однако она имеет и некоторые особенности, которые отображены на рис. 2. В частности, в состав инструментария вошла база данных, которая позволила сохранять данные о доступных шаблонах и их местоположениях; настройки применяемых синтаксических анализаторов; данные об общедоступных параметрах, их типах и функциях их вычисления; информацию о поддерживаемых преобразователях и их типах.

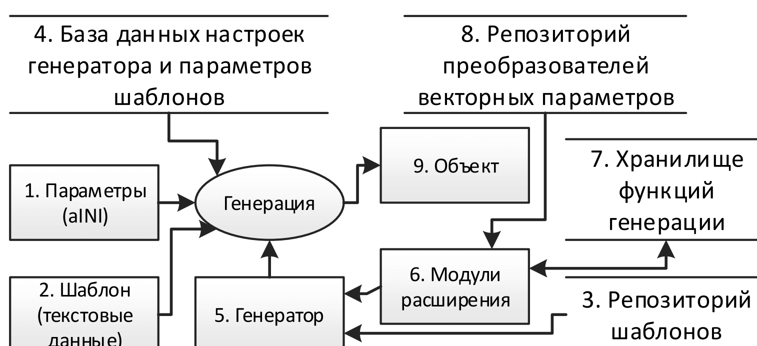


Рис. 2. Диаграмма потоков данных процесса генерации кода на основе шаблонов с использованием разработанного программного инструментария

1.4.1. Особенности программной реализации алгоритма генерации

В процессе программной реализации алгоритма генерации кода третьей версии инструментария применялся графоориентированный подход [27]. В рамках данного подхода предполагается, что алгоритм программы представляется в форме ориентированного графа, для которого узлы являются состояниями данных, а с ребрами связаны функции перехода, преобразующие данные из одного состояния в другое. Тройка объектов — ориентированный граф, множество состояний данных и множество функций-обработчиков позволяет ввести понятие "графовая модель метода обработки данных". Все указанные понятия формально введены в работе [28], детальное описание выходит за рамки настоящей работы.

На рис. 3 представлена графовая модель верхнего уровня абстракции, представляющая алгоритм работы генератора, где использованы обозначения для функций-обработчиков f_{pij} (табл. 2), функций-предикатов p_{ij} состояний данных S_k, а также not (p_{ij}) — обозначение оператора отрицания.

Применение графоориентированного подхода позволяет гибко заменять и/или обновлять любые функции-обработчики, а также естественным образом формировать библиотеки функций программного инструментария в целом.

1.4.2. Формат файла исходных данных задания на генерацию

Для определения файла исходных данных задания на генерацию также использовался формат aINI (листинг 10).

Таблица 2

Описания функций-обработчиков графовой модели алгоритма генерации кода на основе шаблонов

Обозначение	Функция-обработчик	Комментарий
f_{12}	Подготовка массива подстановок на основе скалярных параметров	Источники параметров: а) файлы исходных данных и дополнительные параметры; б) база данных подсистемы генерации; в) неявно определяемые параметры на основе параметров aINI-типа «ссылка на запись в таблице базы данных» (см. подразд. 1.3)
f_{23}	Подготовка преобразуемого контента	Копирование шаблона во временное местоположение
f_{33}	Замена имен файлов и каталогов	Рекурсивная замена во временном местоположении объекта
f_{44}	Преобразование контента на основе скалярных параметров	Рекурсивное преобразование всех скалярных параметров во временном местоположении объекта
f_{55}	Анализ и преобразование контента на основе векторных параметров	Выявление запросов значений векторных параметров, вычисление значений и преобразование векторных параметров с использованием зарегистрированных в системе XSL-преобразователей
f_{56}	Подготовка преобразованного контента	Перенос объекта в конечное местоположение или архивирование и отправка
f_{34}, f_{45}	Вспомогательные функции	—

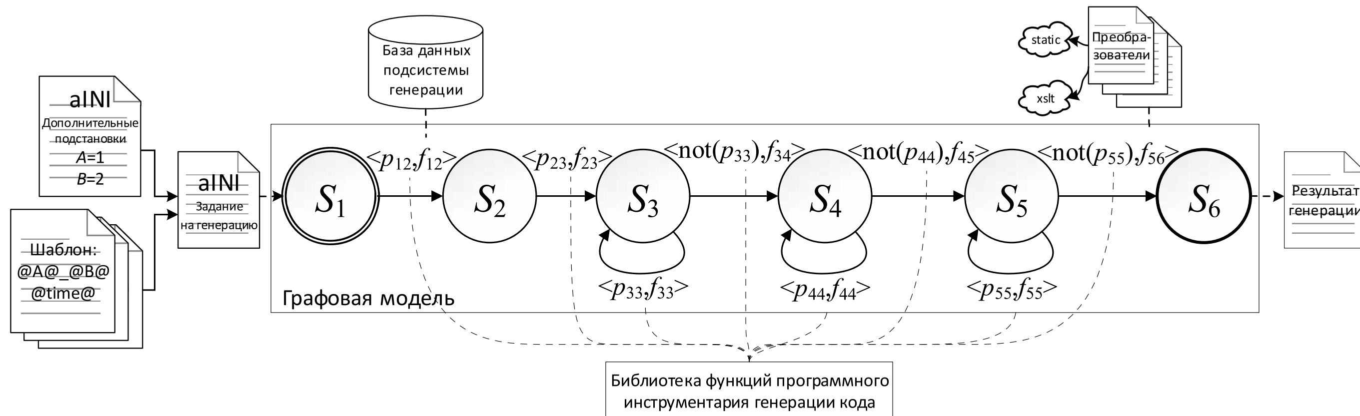


Рис. 3. Схема процесса генерации кода на основе шаблонов в рамках PBC GCD с представлением алгоритма генерации в виде его графовой модели

```
1 [Author]//Идентификация автора разработки
2 AuthorName=[AuthorName]$sys.users//Автор разработки
3 -AuthorSID=@AuthorName.useri@//SID автора
4
5 [Generator parameters]//Параметры генерации
6 *CopyObjectToRep=[0]{0|1}//Перенести объект генерации в репозиторий
7 *RepPath=@RepPath@//Целевой репозитории (URL)
8 *TemplatesPath=@TemplatesPath@//Репозиторий шаблонов (URL)
9 *TemporaryPath=@TemporaryPath@//Каталог временного хранения (URL)
10
11 [Object parameters]//Параметры генерируемого объекта
12 *CodeObjectName=@CodeObjectName@//Наименование объекта(varchar(25))
13 *Description=@Description@//Описание
14 ParametersFile=[ParametersFile.imp]//Дополнительные параметры
15 *TemplateSID=[@TemplateSID@]$gen.tmpls//Идентификатор шаблона
16
17 [Project data]//Идентификация объекта
18 *ComplexSID=[ComplexSID]$sys.cmplx//Идентификатор комплекса
19 *SolutionSID=[SolutionSID]$sys.solun//Идентификатор решения
20 *ProjectSID=[ProjectSID]$sys.prjct//Идентификатор проекта
```

Листинг 10. Шаблон GEN задания для генерации кода в формате aINI

Указанные параметры являются статическими и задаваемыми для каждой новой процедуры генерации. Параметры, отмеченные звездочкой, обязательны для заполнения, а отмеченные символом минус — скрытые.

Параметры `CopyObjectToRep` и `TemporaryPath` используются для указания необходимости размещения результата генерации во временном расположении, что зачастую обусловлено необходимостью отладки результата генерации и шаблона.

Параметр `ParametersFile`, имеющий aINI-тип "имя типизированного файла", применяется для определения списка дополнительных параметров, который также разрабатывается с использованием синтаксиса aINI. Шаблон представлен листингом 11, где `[Additional parameters]` — стандартная секция для размещения списка дополнительных шаблон-зависимых параметров (число параметров произвольно); `@paramname@` — имя параметра; `@param _ value@` — значение параметра; `@param _ description@` — описание параметра.

```
[Additional parameters]//
Дополнительные шаблон-зависимые параметры
@paramname@=@param _ value@//
@param _ description@
...
```

Листинг 11. Шаблон IMP файла дополнительных параметров в формате aINI

1.4.3. Репозиторий и база данных шаблонов

Среди исходных данных задания на генерацию представлен параметр `TemplateSID`, имеющий aINI-тип "ссылка на запись в таблице базы данных" и определяющий используемый шаблон. Местоположение выбранного шаблона определяется на основе информации, размещенной в таблице базы данных `gen.tmpls`. Реляционная модель данных схематично

представлена на рис. 4, *б* (см. вторую сторону обложки). Управление файловой библиотекой шаблонов было реализовано с использованием систем контроля версий (SVN, GIT).

На рис. 4, *а* (см. вторую сторону обложки) представлена базовая структура каталогов для хранения шаблонов. Регистрация шаблонов и их атрибутов, включая информацию о местоположении, а также хранение данных о преобразователях векторных и статических параметров, включая методы их замены, и прочая информация хранятся в таблицах схемы данных `gen`, входящей в состав базы данных PBC GCD подсистемы генерации кода (рис. 4, *б*).

Шаблоны сгруппированы по категориям, которые регистрируются в таблице `tmcat` (рис. 4, *б*). Регистрация отлаженных шаблонов осуществляется в таблице `tmpls`. Базовые статические параметры, которые могут быть использованы в любых шаблонах, регистрируются в таблице `subst`. Форматы выходных данных (см. листинг 1) при преобразовании векторных параметров регистрируются в таблице `frmts`. Каждому зарегистрированному формату соответствует конкретный преобразователь. Для определения преобразователей была использована технология XSLT. Применение конкретного XSL-преобразователя осуществляется автоматически при обработке каждого векторного параметра шаблона. Местоположение доступных XSL-преобразователей определяется на уровне ядра PBC GCD и выходит за рамки тематики настоящей работы.

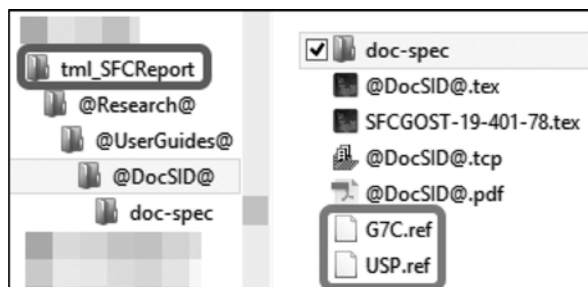
Параметр `CodeObjectName` используется для создания уникального имени формируемого объекта в совокупности с используемой структурой каталогов хранения и значениями параметров `ComplexSID`, `SolutionSID`, `ProjectSID`. Подробнее процесс регистрации объектов генерации в базе данных и алгоритм формирования их идентификаторов представлены далее.

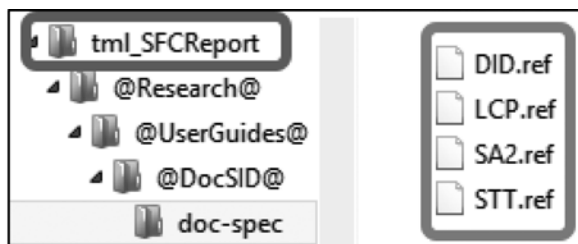
1.4.4. Вложенные (зависимые) шаблоны

На практике часто возникает необходимость повторного использования части кода шаблона в составе двух и более других шаблонов. В такой ситуации в рамках представляемого инструментария была реализована поддержка зависимых шаблонов согласно определению 4. На рис. 5 представлены два шаблона: SFC и SFD, которые зависят от шести других шаблонов: G7C, USP, DID, LCP, SA2, STT (рис. 6). Реализация такой возможности позволила исключить дублирование кода при построении новых шаблонов. Число уровней вложенности произвольно. Обработка исключительной ситуации потенциального формирования зацикливания при обработке шаблона не была реализована и является предметом будущих доработок созданного инструментария.

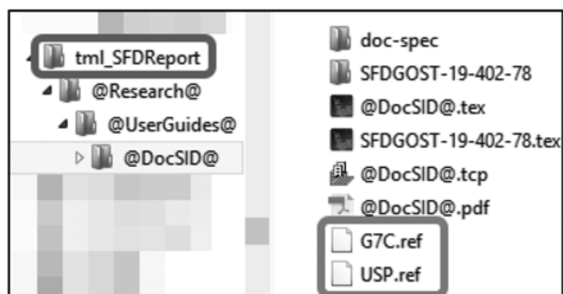
1.4.5. Регистрация объектов, формируемых на основе шаблонов

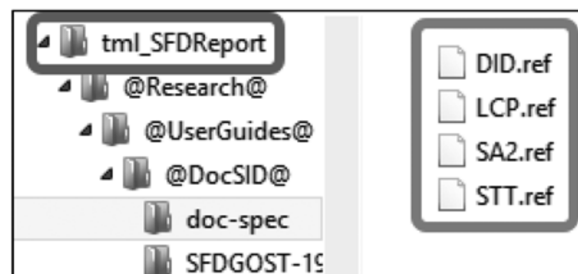
Создание объектов на основе шаблонов может приводить к формированию как корректных, так и некорректных результатов, что связано с существенным числом входных параметров различных





а)





б)

Рис. 5. Примеры двух шаблонов, определения которых зависят от других шаблонов:
а — SFC — текст программы (ГОСТ 19.401—78), зависит от G7C, USP, DID, LCP, SA2, STT; *б* — SFD — описание программы (ГОСТ 19.402—78), зависит от G7C, USP, DID, LCP, SA2, STT

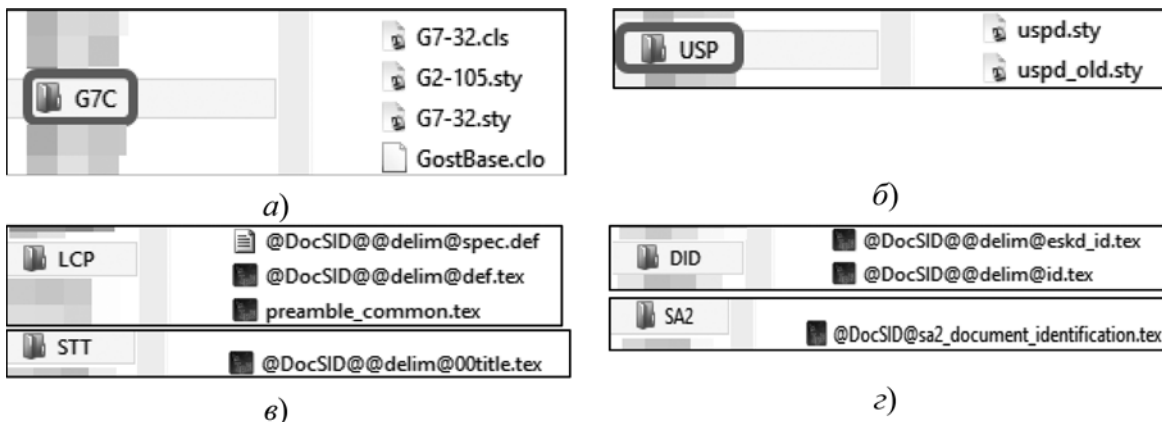


Рис. 6. Шаблоны, использующиеся в других шаблонах:
а — G7C — класс стиля ГОСТ 7.32—2001; *б* — USP — класс стиля "Единая система программной документации" (ЕСПД); *в* — LCP, STT; *г* — DID, SA2

типов, значения которых определяются явно или неявно[4].

Для отладки шаблонов и возможностей корректировки результатов генерации, помимо прочих, был предусмотрен параметр, определяющий путь к временному размещению результатов генерации (см. листинг 10, строка 9). После серии отладочных попыток генерации конечный удовлетворительный результат размещается в целевом каталоге хранения (см. листинг 10, строка 7). Следует заметить, что число формируемых объектов, размещаемых в одних и тех же целевых каталогах, может быть достаточно велико. Это обстоятельство приводит к риску создания файла и/или каталога с именем, совпадающим с уже существующим объектом в целевом каталоге. Исключение такого риска может быть реализовано:

— предварительным сканированием целевых каталогов на предмет существования файлов и/или каталогов с определенными именами;

— присвоением каждому генерируемому объекту на основе шаблона уникального имени (идентификатора) и размещением результата в целевом каталоге с присвоением имени, включающим этот идентификатор.

В рамках представленного подхода выбран второй вариант, это было связано со сложностью алгоритма генерации нового объекта и необходимостью размещения его в общем репозитории, независимо от числа формируемых объектов.

Для идентификации генерируемых объектов было предложено использовать строковые идентификаторы согласно синтаксису, представленному листингом 12.

```
1 FullCodeObjectName = cpxSID slnSID " _ " tmlSID " _ " CodeObjectName.
2 cpxSID = ID3lc. (*Строковый идентификатор комплекса (системы)*)
3 slnSID = ID3lc. (*Строковый идентификатор решения (подсистемы)*)
4 tmlSID = ID3lc. (*Строковый идентификатор шаблона*)
5 CodeObjectName = ID15CamelCase. (*Наименование генерируемого объекта*)
```

Листинг 12. Синтаксис формата имени объекта, генерируемого на основе шаблона, в расширенной форме Бекуса—Наура с комментариями[5]

Такой синтаксис позволяет формировать идентификаторы, включающие семантику, что важно при обзоре сформированных материалов неподготовленным специалистом. Однако иногда для удобства поиска конкретного документа или объекта требуется информация о его коротком числовом идентификаторе. В рамках созданного программного инструментария такой идентификатор формируется только в случае сохранения результата генерации в едином репозитории. При этом сам идентификатор сохраняется вместе с соответствующим строковым идентификатором в специальную базу данных для удобства его последующего использования.

1.5. Статическая и динамическая части структуры каталогов применяемых шаблонов

Одним из важных приложений генератора кода является возможность заранее определить с помощью одного шаблона базовую структуру каталогов, а с помощью серии других шаблонов — структуры объектов, которые требуется размещать в этой заранее определенной базовой структуре. Отмеченные функциональные возможности реализуются путем разделения структур каталогов на статическую и динамическую как в шаблоне базовой структуры, так и в шаблонах структур размещаемых объектов.

---

[4] Например, шаблон PAT, разработанный с использованием созданного программного инструментария и предназначенный для формирования отчета о патентных исследованиях согласно ГОСТ Р 15.011—96, требует определения 37 скалярных дополнительных параметров, среди которых много тех, чьи значения определяются во время выполнения.

[5] Для краткости опущены определения нетерминальных символов ID3lc (строковый идентификатор длины 3, определенный в нотации lc), ID15CamelCase (строковый идентификатор длины 15, определенный в нотации CamelCase), см. обозначения в табл. 1.

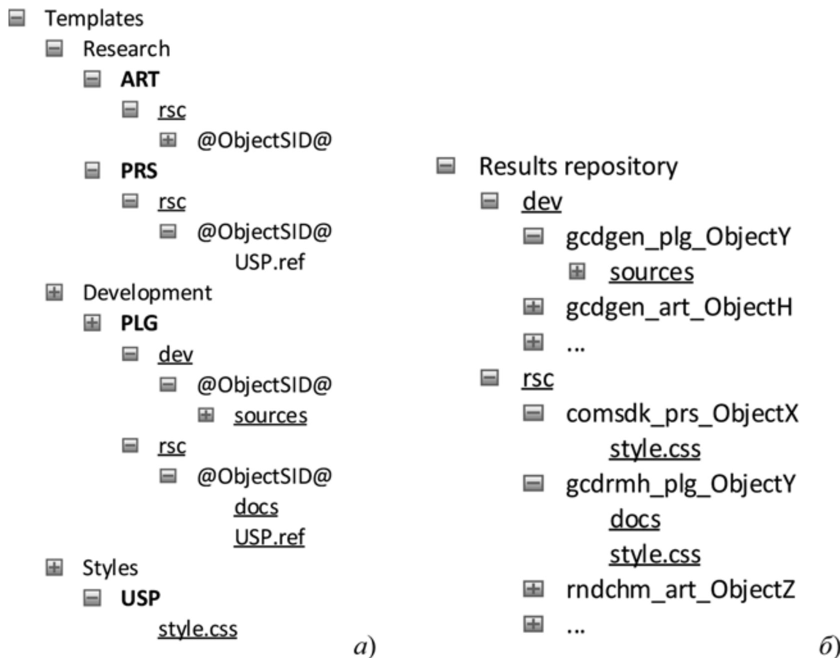


Рис. 7. Пример структуры репозитория шаблонов (*а*) и соответствующего ему автоматически формирующегося репозитория результатов генерации (*б*). Статические части шаблонов и соответствующая им статическая часть формирующегося репозитория подчеркнуты

На рис. 7, *а* представлена структура хранилища шаблонов, включающая четыре шаблона ART, PRS, PLG, USP, сгруппированные по трем категориям Research, Development и Styles. Шаблоны PRS и PLG зависят от USP. На рис. 7, *б* представлена структура каталогов, которая будет автоматически формироваться в результате генерации объектов на основе представленных шаблонов. Идентификаторы шаблонов выделены полужирным шрифтом, названия категорий подчеркнуты.

Другими словами, статическая часть может быть сформирована заранее с помощью предварительно подготовленного шаблона базовой структуры. Динамическая часть может формироваться коллективом исследователей постепенно путем последовательной разработки отдельных документов и прочих материалов с использованием генератора кода.

2. Программная реализация

Программная реализация инструментария осуществлена на языке C++ и интегрирована в состав PBC GCD (рис. 8).

Исходные данные для генерации (рис. 8, *а*) представлены в файле в формате aINI (см. листинг 10), что позволило автоматизировать создание графических меню пользователя для ввода исходных данных [25]. Для сохранения графовой модели (рис. 8, *б*) применялся язык описания графов aDOT [28]. Библиотеку функций генерации (рис. 8, *в*) определили функции, представленные в табл. 2. Программная реализация функций генерации кода осуществлена с использованием языка программирования C++. Числа, связанные с ребрами схемы, показывают последовательность обработки данных.

Согласно представленной ранее архитектуре программный инструментарий генерации кода может быть достаточно гибко дополнен новыми функциональными возможностями.

Применение aINI-формата в качестве формата для файлов входных данных позволяет дополнять списки шаблон-зависимых параметров без изменения исходного кода генератора. Возможность хранения шаблон-независимых параметров в базе данных позволяет также дополнять их без внесения изменений в исходный код генератора.

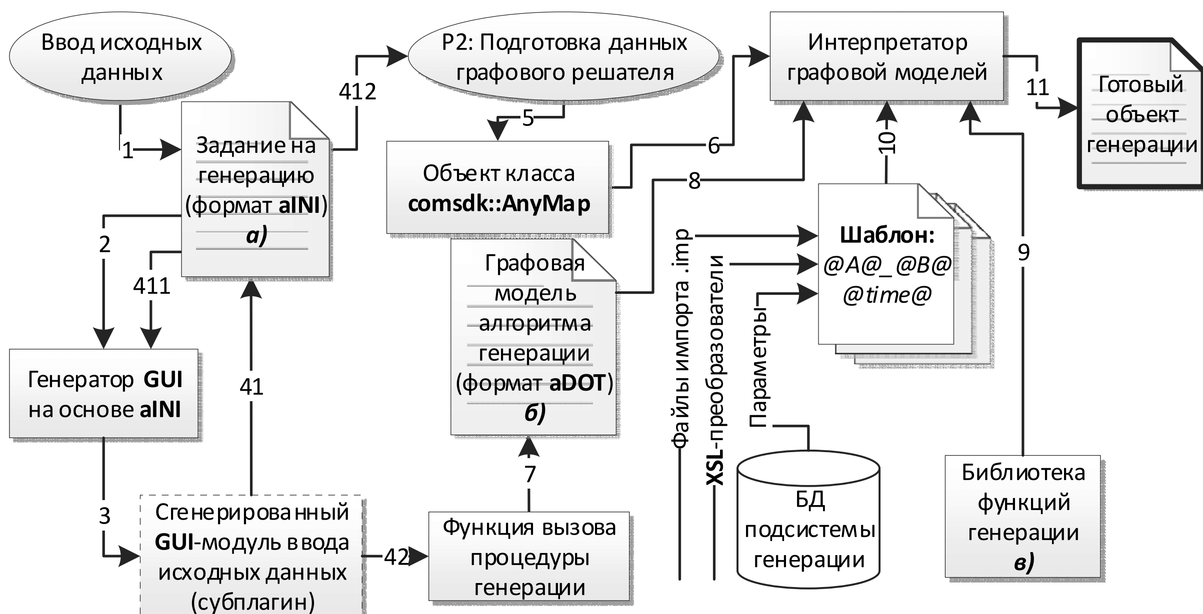


Рис. 8. Принципиальная схема использования программных средств PBC GCD для интеграции созданного программного инструментария и его функционирования

3. Примеры применения

За время эксплуатации представляемого программного инструментария были реализованы десятки шаблонов. В их числе шаблоны файлов исходных данных для различных функций решения инженерных задач в рамках PBC GCD; шаблоны документов, форма представления которых должна соответствовать тем или иным нормативным документам или стандартам (в том числе государственным); шаблоны программных проектов и др. Далее представлены несколько характерных примеров.

3.1. Шаблоны файлов исходных данных

Известным современным принципом разработки программного обеспечения является применение паттерна проектирования "модель—представление—контроллер" (*model—view—controller*, MVC). Данный принцип используется в рамках применения объектно-ориентированного подхода и предполагает жесткую типизацию классов, создаваемых при проектировании и разработке ПО. Для обеспечения независимости классов "представлений" от классов "моделей" может применяться файл в едином формате, который будет использоваться одновременно и теми, и другими, однако по-разному. "Представления" должны интерпретировать файлы в этом формате и визуализировать данные, тогда как "модели" должны использовать эти файлы для загрузки данных и их обработки.

Предметом разработки в инженерном ПО, как правило, являются функции, необходимые для решения задач, реализующих те или иные вычислительные операции. Каждая из таких функций требует определения своего множества входных параметров (исходных данных), которые, в свою очередь, должны быть формализованы. Формализация исходных данных может быть обеспечена, например, применением шаблонизируемых текстовых форматов данных (рис. 9).

В PBC GCD в качестве такого формата данных используется aINI [25]. Среди известных и широко используемых форматов для решения аналогичных задач можно выделить JSON, XML.

Примеры шаблонов, подготовленных в формате aINI, представлены листингами 13, 14, 15. Формат, представленный листингом 13, использовался для определения исходных данных в задаче построения геометрической модели представительного элемента объема (ПЭО) монодисперсного композиционного материала (КМ) на базе сферических включений в рамках численного анализа эффективных упругих характеристик дисперсно-армированных КМ [29]. Форматы, представленные листингами 14, 15, использовались в рамках решения задачи оптимизации топливных затрат ТЭЦ [30].

```
[Input data] // Исходные данные
*OutputGeoName=@OutputGeoName@ // Наименование формируемой геометрической модели
-OutputGEOFile=@GeneratedDate@ _ @OutputGeoName@.geo // Имя файла выходной геометрии
*GenerateMesh=[1]{0|1} // Генерировать расчетную сетку
MeshGranularity = [@MeshGranularity@]{verycoarse|veryfine|moderatde|fine|moderate|coarse} //
Качество расчетной сетки
-OutputAneuFile=@GeneratedDate@ _ @OutputGeoName@ _ @MeshAttrs@.aneu //Имя файла расчетной сетки
// Абсолютные размеры ячейки периодичности
-WidthX=@WidthX@ [[mm]] // Ширина
-HeightY=@HeightY@ [[mm]] // Высота
-LengthZ=@LengthZ@ [[mm]] // Длина
*Concentration=@Concentration@ // Концентрация наполнителя
*Radius=[@CurrentR@;@MinR@:@MaxR@;@Step@] // Внешний радиус включения
```

Листинг 13. Шаблон файла @DocSID@.fbr исходных данных в формате aINI для генерации ПЭО дисперсно-армированных КМ

```
[Input data]# Исходные данные для построения интерполирующей зависимости
InputFile=[@InputFile@.csv] // Файл с опорными точками (исходные данные)
ExpIntData=[@ExpIntData@] // Файл для записи коэффициентов интерполяции
smoothing=[1]{0|1} // Параметр сглаживания
```

Листинг 14. Шаблон файла @DocSID@.int исходных данных для программного модуля аппроксимации расходной характеристики одного энергоблока ТЭЦ

```
[Input data]// Вычисление значения суммарного расхода топлива на энергоблоке
ExpIntData=[@ExpIntData@]//Интерполированная расходная характеристика блока
InputFile=[@InputFile@.csv]// Файл с опорными точками (исходные данные)
Ne=@Ne@ [[MWt]]//Электрическая нагрузка блока
Qt=@Qt@ [[GkalH]]//Отпуск теплоты блоком посредством ПСГ, Гкал/ч
```

Листинг 15. Шаблон файла @DocSID@.gvl исходных данных для программного модуля вычисления значения расхода топлива при произвольных электрической нагрузке и отпуску тепла на основе ранее построенной расходной характеристики для одного энергоблока ТЭЦ

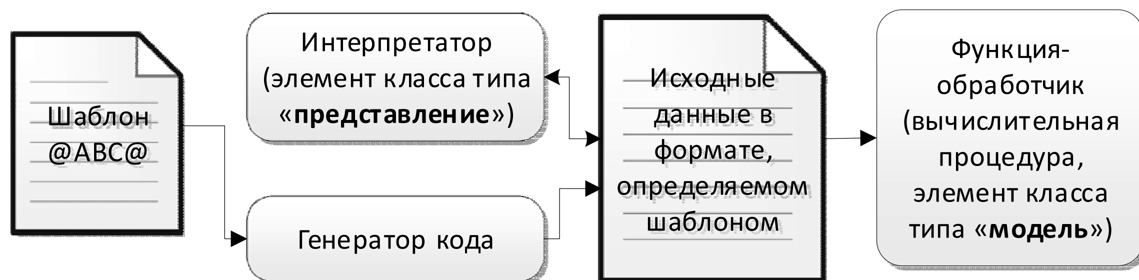


Рис. 9. Применение шаблонов файлов исходных данных

Представленные шаблоны использовались как основа для формирования графического пользовательского интерфейса [25] и далее для формирования файла исходных данных, сохранения введенных значений параметров и их дальнейшей загрузки функциями обработки данных.

3.2. Шаблон документации на программную реализацию вычислительного метода

Одной из прикладных программных разработок, формализующих процесс описания алгоритма вычислительного метода, является графоориентированный подход [27]. Программная реализация вычислительного метода (создание "решателя" задачи) с его помощью сводится к построению графовой модели метода, определению файла исходных данных и реализации множества функций перехода [28].

Применение формализованной процедуры разработки позволило создать шаблон SLR документации на произвольный реализуемый таким образом метод. На рис. 10 (см. вторую сторону обложки) представлен пример использования шаблона SLR (листинг 16) генератором кода при подготовке документации по одному из разрабатываемых решателей. На рис. 10 (см. вторую сторону обложки) выделены некоторые подставленные значения: красным цветом — значения скалярных параметров, синим цветом — значения векторных параметров, зеленым цветом — значения скалярных функциональных параметров.

3.3. Шаблон отчета о проведенных патентных исследованиях согласно ГОСТ Р 15.011—96

В рамках большинства научно-исследовательских работ, проводимых в России в рамках государственных заказов, на отдельных этапах их реализации возникает необходимость проведения патентных исследований в соответствии с требованиями ГОСТ Р 15.011—96. Еще одним примером прикладного использования разработанного программного инструментария стала задача автоматизации построения отчета о проведенных патентных исследованиях.

В зависимости от задачи патентные исследования подразделяют на следующие исследования:

1) тенденций и уровня развития объекта хозяйственной деятельности;
2) патентоспособности;
3) патентной чистоты;
4) с представлением карты патентного ландшафта.

Построение отчета о проведенном патентном исследовании первого вида может быть автоматизировано в связи с незначительным объемом неформализуемого материала, включенного лишь в раздел, связанный с анализом найденных патентных документов, и в заключение. Таким образом, был разработан шаблон, позволяющий генерировать отчет о проведенном патентном исследовании первого из отмеченных выше видов в соответствии с требованиями ГОСТ Р 15.011—96 с использованием языка верстки LaTeX. Источником данных стала разрабо-

```
   Описание инструмента решения задачи (решателя) \textbf{@Solver.slver@}.
   \begin{tabular}{|>{\raggedright}p{0.4\linewidth}|p{0.6\linewidth}|} \hline
   Идентификатор инструмента решения задачи (решателя) & \textbf{@Solver.slver@} \\ \hline
   Описание решателя & @Solver.dscra@ \\ \hline
   Идентификатор связанной сетевой модели (графа) & \textbf{@Solver.ntmdl@}\\
   \hline
   \end{tabular}
   \subsection{Зависимые решатели от @Solver.slver@}
   {\small %#com.sub _ slvrs _ v2[par _ slver = '@Solver.slver@']:TEX#%}
   \begin{landscape}
   \subsection{Описания функций решателя @Solver.slver@}
   На рисунках \ref{fig:@Solver.slver@}, \ref{fig:@Solver.slver@ _ full} представлены соответ-
ственно краткая и полная сетевые модели решателя \textbf{@Solver.slver@}. Описания каждой
функции обработки данных и функций-предикатов представлены в таблице ниже.
```

Листинг 16. Фрагмент шаблона SLR

танная в рамках РВС GCD подсистема проведения патентных исследований.

С использованием разработанного шаблона были подготовлены отчеты о патентных исследованиях в рамках серии государственных и хоздоговорных НИР и ОКР, выполненных в МГТУ им. Н. Э. Баумана, участниками которых были авторы. В результате применения генератора кода и созданного шаблона удалось обеспечить автоматическое построение каркаса отчета и наполнить его данными. При этом были автоматизированы процессы, позволяющие в среднем сформировать более 90 % всего содержания отчетов. Отчеты прошли успешную проверку в организациях-мониторах.

Заключение

Созданный программный инструментарий лег в основу автоматизированной методологии разработки программного обеспечения инженерного анализа, разрабатываемого авторами. Анализ результатов проведенных ими поисковых исследований и выполнения прикладных работ позволяет сделать следующие выводы.

- Представляется целесообразным при разработке программного обеспечения инженерного анализа применение "гибридных" подходов, а именно: для разработки общесистемных стандартных функциональных возможностей следует использовать генераторы кода на основе шаблонов, тогда как при разработке программных реализаций алгоритмически сложных вычислительных процедур следует использовать генерацию кода на основе графовых представлений алгоритмов [28], что является предметом текущих разработок.
- Применение средств поддержки процессов разработки и средств генерации кода позволяет систематизировать процесс разработки многофункциональных программных комплексов. Представляется обоснованным применение таких средств при создании прикладного наукоемкого программного обеспечения.
- Актуальными и востребованными на практике являются программные механизмы предоставления удаленного доступа к разработанному программному инструментарию и библиотеке шаблонов посредством web-приложения. Решение такой задачи позволит сформировать основу средств автоматизации процессов разработки программного обеспечения, документирования, безбумажного документооборота, а также позволит предоставить доступ к созданному инструментарию широкому кругу пользователей. В работах по созданию web-приложения участвует студент И. А. Лаишевский, который подключился к развитию программного инструментария с 2018 г.

Разработка представленного в настоящей статье программного инструментария выполнена авторами в инициативном порядке в МГТУ им. Н. Э. Баумана при разработке ядра распределенной вычислительной системы GCD.

Список литературы

1. **Орлов С.** Технологии разработки программного обеспечения: учебник. СПб.: Питер, 2002. 464 с.
2. **Лаврищева Е. М.** Software Engineering компьютерных систем. Парадигмы, технологии и CASE-средства программирования. Киев: Наукова думка, 2013. 283 с.
3. **Jörges S.** Construction and evolution of code generators 7747. Ch. 2 // The state of the art in code generation. Berlin Heidelberg: Springer; 2013. P. 11—38.
4. **Syriani E., Luhunu L., Sahraoui H.** Systematic mapping study of template-based code generation // Computer Languages, Systems and Structures. 2018. Vol. 52. P. 43—62.
5. **Rosales-Morales V. Y., Alor-Hernández G., García-Alcaráz J. L.** et al. An analysis of tools for automatic software development and automatic code generation // Revista Facultad de Ingenieria. 2015. Issue 77. P. 75—87.
6. **Федотова Д. Э., Семенов Ю. Д., Чижик К. Н.** CASE-технологии. М.: Горячая линия — Телеком, 2005. 160 с.
7. **Lúcio L., Amrani M., Dingel J.** et al. Model transformation intents and their properties // Softw. Syst. Model. 2014. Vol. 15, No. 3. P. 685—705.
8. **Fleischer D., Beine M., Eisemann U.** Applying model-based design and automatic production code generation to safety-critical system development // SAE International Journal of Passenger Cars — Electronic and Electrical Systems. 2009. Volume 2, Issue 1, P. 240—248.
9. **Comparison** of code generation tools. Wikipedia. The Free Encyclopedia. URL: https://en.wikipedia.org/wiki/Comparison_of_code_generation_tools
10. **Luhunu L., Syriani E.** Comparison of the Expressiveness and Performance of Template-Based Code Generation Tools//SLE'17, October 23—24, 2017, Vancouver, Canada, 2017. P. 206—216.
11. **Shinde K., Sun Y.** Template-Based Code Generation Framework for Data-Driven Software Development // Proceedings of 2016 4-th International Conference on Applied Computing and Information Technology (ACIT 2016), December 12—14, 2016, University of Nevada, Las Vegas, USA, 2016. P. 55—60.
12. **Feng J.-D., Zhan D.-C., Nie L.-S., Xu X.-F.** Pattern based code generation method for the business component // Tien Tzu Hsueh Pao/Acta Electronica Sinica. 2008. Vol. 36, Issue SUPPL. P. 19—24.
13. **Andersson P., Höst M.** UML and SystemC — A Comparison and Mapping Rules for Automatic Code Generation // Embedded Systems Specification and Design Languages / Eds. E. Villar. Amsterdam, Netherlands: Springer, 2008. P. 199—209.
14. **Wehrmeister M. A., De Freitas E. P., Binotto A. P. D., Pereira C. E.** Combining aspects and object-orientation in model-driven engineering for distributed industrial mechatronics systems // Mechatronics. 2014. Volume 24, Issue 7. P. 844—865.
15. **Sasi Bhanu S. J., Vinaya Babu A., Trimurthy P.** Code generation for semantic evolution of embedded systems // ARPN Journal of Engineering and Applied Sciences. 2015. Vol. 10, Issue 20. P. 9382—9394.
16. **Деев Д. В., Окуловский Ю. С.** Система порождения документов в форматах HTML, MHT и WordProcessingML // Научно-технический вестник Санкт-Петербургского государственного университета информационных технологий, механики и оптики. 2010. Т. 70, № 6. С. 77—81.
17. **Kołek K., Piątek K.** Rapid algorithm prototyping and implementation for power quality measurement // EURASIP Journal on Advances in Signal Processing. 2015. Article 19. DOI: 10.1186/s13634-015-0192-3
18. **Kai Hu, Zhangbo Duan, Jiye Wang, Lingchao Gao, Lihong Shang.** Template-based AADL automatic code generation // Frontiers of Computer Science. 2018. pp. 2095—2236. DOI: 10.1007/s11704-017-6477-y
19. **Александров А. Е., Шильманов В. П.** Инструментальные средства разработки и сопровождения программного обеспечения на основе генерации кода. Бизнес-информатика. 2012. Т. 22, № 4. С. 10—17.
20. **Самохвалов Э. Н., Ревунков Г. И., Гапанюк Ю. Е.** Генерация исходного кода программного обеспечения на основе многоуровневого набора правил // Вестник Московского государственного технического университета им. Н. Э. Баумана. Серия: Приборостроение. 2014. Т. 98, № 5. С. 77—87.
21. **Софин Н. А., Кавалеров М. В**. Среда разработки для автоматизированной генерации исходного кода и документации // Master's Journal. 2016. № 2. С. 275—262.

22. **Соколов А. П., Шпакова Ю. В., Першин А. Ю.** Проектирование распределенной программной системы GCD численного моделирования композитов. // Математические методы в технике и технологиях — ММТТ-25: сб. трудов XXV Международной научной конференции: в 10 т. Т. 5. Секция 8, 9 / под. общ. ред. А. А. Большакова. Волгоград: ВолГУ, 2012. С. 79—80.
23. **Соколов А. П., Макаренков В. М., Шевцов А. С.** Разработка программы gcdcli_plg_CodeGenerator автоматизации процессов формирования электронных документов // Материалы XIV Всероссийской конференции молодых ученых по математическому моделированию и информационным технологиям. Томск, Россия, Томский филиал Института вычислительных технологий СО РАН, 14—18 октября 2013. С. 46—47.
24. **Свид.** 2014612782 Российская Федерация. Свидетельство об официальной регистрации программы для ЭВМ. Программа gcdcli_plg_CodeGenerator для автоматического формирования документов и программных объектов на основе шаблонов / Соколов А. П., Макаренков В. М.; заявитель и правообладатель: Соколов А. П., Макаренков В. М. (RU). № 2013617477, заявл. 07.08.2013, опубл. 06.03.2014, Реестр программ для ЭВМ. 1 с.
25. **Соколов А. П., Першин А. Ю.** Программный инструментарий для создания подсистем ввода данных при разработке систем инженерного анализа // Программная инженерия. 2017. Т. 8, № 12. С. 543—555.
26. **Димитриенко Ю. И., Соколов А. П.** Система автоматизированного прогнозирования свойств композиционных материалов // Информационные технологии. 2008. № 8. С. 31—38.
27. **Патент** на изобретение RU 2681408. Способ и система графо-ориентированного создания масштабируемых и сопровождаемых программных реализаций сложных вычислительных методов; заявитель и правообладатель: Соколов А. П., Першин А. Ю. (RU). Заявка № RU 2017 122 058 A, приоритет 22.07.2017, опубликовано 22.02.2019, Реестр изобретений Роспатента. 1 с.
28. **Соколов А. П., Першин А. Ю.** Графоориентированный программный каркас для реализации сложных вычислительных методов // Программирование. 2019. Т. 47, № 5. С. 43—55.
29. **Соколов А. П., Щетинин В. Н.** Идентификация упругих свойств адгезионного слоя дисперсно-армированных композитных материалов на основе экспериментальных данных // Механика композиционных материалов и конструкций. 2018. Т. 24, № 4. С. 555—581.
30. **Соколов А. П., Щетинин В. Н., Сапелкин А. С.** Применение технологии GBSE для разработки программного обеспечения автоматизации процесса распределения электрических и тепловых нагрузок на ТЭЦ // Материалы XX Юбилейной международной конференции по вычислительной механике и современным прикладным программным системам (ВМСППС'2017), 24—31 мая 2017 г., Алушта. М.: Изд-во МАИ, 2017. С. 108—110.

Development of Template-Based Code Generation Software for Development of Computer-Aided Engineering System

A. P. Sokolov, alsokolo@bmstu.ru, @**V. M. Makarenkov**, **A. Yu. Pershin**, apershin@bmstu.ru, **I. A. Laishevskiy**, ivanlaish@yandex.ru, Bauman Moscow State Technical University, Moscow, 105005, Russian Federation

Corresponding author:

Sokolov Alexandr P., Associate Professor, Bauman Moscow State Technical University, Moscow, 105005, Russian Federation

E-mail: alsokolo@bmstu.ru

Received on August 21, 2018
Accepted on August 12, 2019

The use of automation tools for software development or so-called computer-aided software engineering (CASE) tools is a sign of a high level of maturity of the development team, and is especially necessary when creating large-scale distributed software. Among others, CASE tools often include code generation software. Template-based code generation involves automatic building of source code or some text result corresponding to a pre-defined format called a template. The article presents the architecture and software tools for computer-aided prototyping of projects of software modules and documents of different types and purposes. The basis of the created technical solutions is the use of the following, developed by the authors: a specialized format of representation of templates, aINI format of source data files and a library of functions that allows to interpret the templates and create objects of different types based on them. The algorithm of generation and the method of automatic integration of the created software tools in the framework of a software complex "Distributed computational system GCD" are presented. Examples of practically significant created templates and their corresponding created objects are given, namely: a) template of initial data file; b) template of the documentation of the problem solver implemented using graph-based software engineering approach; c) template of the report of the conducted patent research according to GOST R 15.011—96.

Keywords: *computer-aided software engineering; rapid software and documentation prototyping; template-based code generation; model-driven engineering; model-to-text transformation; aINI; code generation; automatic programming; interpretation of text data; program processors; technologies for development of computer-aided engineering systems*

Acknowledgements: *This work was carried out on an initiative basis by authors. The development of the template of the result of the simulation of the kinetics of a chemical reaction was funded by RFBR according to the research project No. 18-07-00341.*

For citation:

Sokolov A. P., Makarenkov V. M., Pershin A. Yu., Laishevskiy I. A. Development of Template-Based Code Generation Software for Development of Computer-Aided Engineering System, *Programmnaya Ingeneria,* 2019, vol. 10, no. 9—10, pp. 400—416.

DOI: 10.17587/prin.10.400-416

References

1. **Orlov S.** *Technologies of software development,* 2002, Saint-Petersburg, Piter, 464 p. (in Russian).
2. **Lavrishcheva E. M.** *Software Engineering of computer systems. Paradigms, technologies and CASE-software*, Kiev, Naukova dumka, 2013. 283 p. (in Russian).
3. **Jörges S.** Construction and evolution of code generators 7747. Ch. 2. *The state of the art in code generation,* Berlin Heidelberg, Springer, 2013, pp. 11—38.
4. **Syriani E., Luhunu L., Sahraoui H.** Systematic mapping study of template-based code generation. *Computer Languages, Systems and Structures,* 2018, vol. 52, pp. 43—62.
5. **Rosales-Morales V. Y., Alor-Hernández G., García-Alcaráz J. L., Zatarain-Cabada R., Barrón-Estrada M. L.** An analysis of tools for automatic software development and automatic code generation*, Revista Facultad de Ingenieria*, 2015, issue 77, pp. 75—87.
6. **Fedotova D. E., Semenov Yu. D., Chizhik K. N.** *CASE-technologies,* Moscow, Goryachaya liniya — Telekom, 2005, 160 p. (in Russian).
7. **Lúcio L., Amrani M., Dingel J.** et al. Model transformation intents and their properties, *Softw. Syst. Model.*, 2014, vol. 15, no. 3, pp. 685—705.
8. **Fleischer D., Beine M., Eisemann U.** Applying model-based design and automatic production code generation to safety-critical system development, *SAE International Journal of Passenger Cars — Electronic and Electrical Systems,* 2009, vol. 2, issue 1, pp. 240—248.
9. **Comparison** of code generation tools. Wikipedia. The Free Encyclopedia, available at: https://en.wikipedia.org/wiki/Comparison_of_code_generation_tools
10. **Luhunu L., Syriani E.** Comparison of the Expressiveness and Performance of Template-Based Code Generation Tools, *SLE'17,* October 23—24, 2017, Vancouver, Canada, 2017, pp. 206—216.
11. **Shinde K., Sun Y.** Template-Based Code Generation Framework for Data-Driven Software Development, *Proceedings of 2016 4-th International Conference on Applied Computing and Information Technology (ACIT 2016)*, December 12—14, 2016, University of Nevada, Las Vegas, USA, 2016, pp. 55—60.
12. **Feng J.-D., Zhan D.-C., Nie L.-S., Xu X.-F.** Pattern based code generation method for the business component, *Tien Tzu Hsueh Pao/Acta Electronica Sinica,* 2008, vol.36, issue SUPPL, pp. 19—24.
13. **Andersson P., Höst M.** UML and SystemC — A Comparison and Mapping Rules for Automatic Code Generation, *Embedded Systems Specification and Design Languages* / Eds. E. Villar, Amsterdam, Netherlands, Springer, 2008, pp. 199—209.
14. **Wehrmeister M. A., De Freitas E. P., Binotto A. P. D., Pereira C. E.** Combining aspects and object-orientation in model-driven engineering for distributed industrial mechatronics systems, *Mechatronics*, 2014, vol. 24, issue 7, pp. 844—865.
15. **Sasi Bhanu S. J., Vinaya Babu A., Trimurthy P.** Code generation for semantic evolution of embedded systems, *ARPN Journal of Engineering and Applied Sciences*, 2015, vol. 10, issue 20, pp. 9382—9394.
16. **Deyev D. V., Okulovskiy Yu. S.** Document generation system in HTML, MHT and WordProcessingML formats, *Nauchno-tekhnicheskiy vestnik Sankt-Peterburgskogo gosudarstvennogo universiteta informatsionnykh tekhnologiy, mekhaniki i optiki,* 2010, vol. 70, no. 6, pp. 77—81 (in Russian).
17. **Kołek K., Piątek K.** Rapid algorithm prototyping and implementation for power quality measurement, *EURASIP Journal on Advances in Signal Processing,* 2015, Article 19, DOI: 10.1186/s13634-015-0192-3
18. **Kai Hu, Zhangbo Duan, Jiye Wang, Lingchao Gao, Lihong Shang.** Template-based AADL automatic code generation, *Frontiers of Computer Science,* 2018, pp. 2095—2236, DOI: 10.1007/s11704-017-6477-y
19. **Aleksandrov A. E., Shilmanov V. P.** Software tools for computer-aided support of software development based on code generation, *Biznes-informatika*, 2012, vol. 22, no. 4, pp. 10—17 (in Russian).
20. **Samohvalov E. N., Revunkov G. I., Gapanyuk Yu. E.** Software source code generation based on a multi-level set of rules, *Vestnik Moskovskogo gosudarstvennogo tekhnicheskogo universiteta im. N. E. Baumana. Seriya: Priborostroenie*, 2014, vol. 98, no. 5, pp. 77—87 (in Russian).
21. **Sofin N. A., Kavalerov M. V.** Integrated development environment for code and documentation generation, *Master's Journal,* 2016, no. 2, pp. 275—262 (in Russian).
22. **Sokolov A. P., Shpakova Yu. V., Pershin A. Yu.** Design of distributed software system GCD of numerical modeling of composites, *Matematicheskie metody v tehnike i tehnologijah (MMTT-25). Trudi XXV Mezhdunarodnoj nauchnoj konferencii*, VolGU, Volgograd, Russia, 2012, vol. 5, no. 10, pp. 79—80 (in Russian).
23. **Sokolov A. P., Makarenkov V. M., Shevtsov A. S.** Development of software gcdcli_plg_CodeGenerator for automatized generation of electronic documents, *Materialy XIV Vserossiyskoy konferentsii molodykh uchenykh po matematicheskomu modelirovaniyu i informatsionnym tekhnologiyam*, Tomsk, Russia, Tomsk branch of Computational Technologies Institute SB RAS, October 14—18, 2013, pp. 46—47 (in Russian).
24. **Certificate** 2014612782 Russian Federation. Certificate of official computer program registration. Software gcdcli_plg_CodeGenerator for automatized generation of documents and software objects based on templates. Applicant and rightholder: Sokolov A. P., Makarenkov V. M. (RU). No. 2013617477, application 07.08.2013, published 06.03.2014, Catalog of Computer Programs of Rospatent. 1 p. (in Russian).
25. **Sokolov A. P., Pershin A. Yu.** Software tools for development of input data subsystems of computer-aided engineering complexes, *Programmnaya ingeneria*, 2017, vol. 8, no. 12, pp. 543—555 (in Russian).
26. **Dimitrienko Yu. I., Sokolov A. P.** System for automated prediction of properties of composite materials, *Informacionnye tehnologii*, 2008, no. 8, pp. 31—38 (in Russian).
27. **The patent** application for the invention RU 2681408. Method and system of graph-based development of scalable and maintainable software implementations of complex computational methods. Applicant and rightholder: Sokolov A. P., Pershin A. Yu., Application RU 2017 122 058 A. Priority date: 22.07.2017, published 22.02.2019, Register of inventions of Rospatent. 1 p. (in Russian).
28. **Sokolov A. P. and Pershin A. Yu.** Graph oriented software framework for implementing complex computational methods, *Programmirovanie*, 2019, vol. 47, no. 5, pp. 43—55 (in Russian).
29. **Sokolov A. P., Shchetinin V. N.** Identification of elastic properties of the adhesive layer of dispersion-reinforced composite materials on the basis of experimental data, *Mekhanika kompozicionnyh materialov i konstrukcij*, 2018, vol. 24, no. 4, pp. 555—581 (in Russian).
30. **Sokolov A. P., Schetinin V. N., Sapelkin A. S.** Application of technology GBSE for developing software of automation of process of distribution of electric and thermal loads for CHP plant, *Proceedings of the XX International conference on computational mechanics and to-date applied software systems (VSPPS in 2017)*, 24—31 May 2017, Alushta, Moscow, Mai Publishing house, 2017, pp. 108—110 (in Russian).

ООО "Издательство "Новые технологии". 107076, Москва, Стромынский пер., 4
Технический редактор *Е. М. Патрушева.* Корректор *Н. В. Яшина*

Сдано в набор 15.08.2019 г. Подписано в печать 23.09.2019 г. Формат 60×88 1/8. Заказ PI9-1019
Цена свободная.

Оригинал-макет ООО "Адвансед солюшнз". Отпечатано в ООО "Адвансед солюшнз".
119071, г. Москва, Ленинский пр-т, д. 19, стр. 1. Сайт: **www.aov.ru**