ISSN 2220-3397

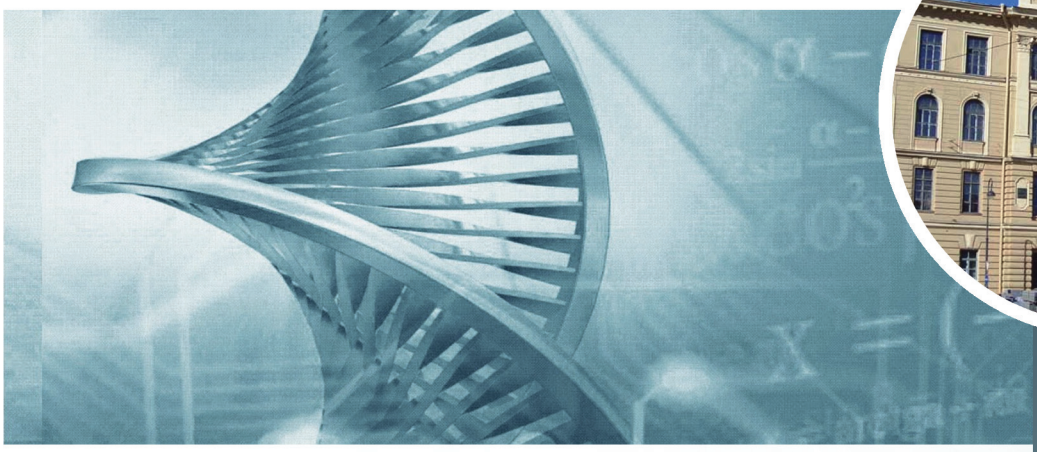
ТЕОРЕТИЧЕСКИЙ И ПРИКЛАДНОЙ НАУЧНО-ТЕХНИЧЕСКИЙ ЖУРНАЛ

Программная инженерия

Пр ИН
5
2018
Том 9

ММТТ-31

Посвящается 190-летию СПбГТИ(ТУ)



Работа конференции будет проходить в одном из старейших и ведущих вузов России – Санкт-Петербургском государственном технологическом институте (техническом университете)

XXXI Международная научная конференция
МАТЕМАТИЧЕСКИЕ МЕТОДЫ В ТЕХНИКЕ И ТЕХНОЛОГИЯХ –

ММТТ-31

Петербургская мультиконференция

10 – 14 сентября 2018 г.

Перечень секций

1. Качественные и численные методы исследования дифференциальных и интегральных уравнений
2. Оптимизация, автоматизация и оптимальное управление технологическими процессами
3. Математическое моделирование технологических и социальных процессов
4. Математическое моделирование и оптимизация в задачах САПР, аддитивных технологий, цифрового производства
5. Математические методы в задачах радиотехники, радиоэлектроники и телекоммуникаций, геоинформатики, авионики и космонавтики
6. Математические методы и интеллектуальные системы в робототехнике и мехатронике
7. Математические методы в медицине, биотехнологии и экологии
8. Математические методы в экономике и гуманитарных науках
9. Информационные и интеллектуальные технологии в технике и образовании
10. Обсуждение квалификационных работ
11. Симпозиум «Разработка интеллектуальных тренажеров для различных отраслей промышленности и образования»

ШМУ

1. Интеллектуализация управляемых технологических процессов
2. Информатизация технических систем и процессов

Одновременно с ММТТ-31 проводится конференция
2018 IEEE Northwest Russia Conference on Mathematical Methods in Engineering and Technology (MMET NW 2018)
2018 IEEE Северо-Западная конференция России по математическим методам в технике и технологиях (СЗКР ММТТ 2018)
по первым восьми секциям ММТТ-31.

Сайт конференции: http://mmtt.sstu.ru/

ТЕОРЕТИЧЕСКИЙ И ПРИКЛАДНОЙ НАУЧНО-ТЕХНИЧЕСКИЙ ЖУРНАЛ

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

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 (индексы: по каталогу агентства "Роспечать" — **22765**, по Объединенному каталогу "Пресса России" — **39795**) 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", 2018

THEORETICAL AND APPLIED SCIENTIFIC AND TECHNICAL JOURNAL

SOFTWARE ENGINEERING

PROGRAMMNAYA INGENERIA

Vol. 9
N 5
2018

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

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

УДК 004 DOI: 10.17587/prin.9.195-208

В. А. Васенин, д-р физ.-мат. наук, проф., e-mail: vasenin@msu.ru,
Механико-математический факультет МГУ имени М. В. Ломоносова,
А. А. Иткес, канд. физ.-мат. наук, науч. сотр., e-mail: itkes@imec.msu.ru,
НИИ механики МГУ имени М. В. Ломоносова

Внедрение реляционной модели логического разграничения доступа в web-приложения информационных систем, разработанных на основе библиотеки Django

На примере наукометрической системы "ИСТИНА" рассмотрены подходы к внедрению программных механизмов реляционной модели логического разграничения доступа к ресурсам информационно-аналитических систем, управляемых web-приложениями, созданными с использованием библиотеки Django. Представлен механизм включения в код такой системы описания модели разграничения доступа к ее объектам, обеспечивающий высокую степень эргономичности процесса сопровождения системы и допускающий при этом возможность статического анализа правил безопасности.

Ключевые слова: *разграничение доступа, информационная безопасность, реляционная модель, web-приложения*

Введение

В настоящей работе рассмотрены подходы к внедрению механизмов реляционной модели логического разграничения доступа (ЛРД), представленной в работах [1, 2], к объектам многопользовательской системы управления сетевым контентом, разработанной на основе библиотеки Django [3]. Решение этой задачи представлено на примере информационно-аналитической системы (ИАС) "ИСТИНА", описанной в работе [4]. Даны некоторые общие сведения об ИАС "ИСТИНА" и о библиотеке Django, позволяющие лучше оценить предлагаемые подходы. Рассмотрены задачи, которые необходимо решить для обеспечения эффективного использования реляционной модели логического разграничения доступа в подобных системах. К их числу относится разработка средств автоматизированного тестирования механизмов управления доступом к объектам целевой системы и средств описания модели разграничения доступа к ее объектам, наиболее удобных для ее сопровождения.

Статья состоит из четырех разделов. В разд. 1 кратко представлены основные сведения о библиотеке Django и описаны особенности структуры ИАС "ИСТИНА", важные в контексте целей и назначения системы [4]. В разд. 2 дано определение и описаны основные свойства реляционной модели ЛРД. Также изложены особенности работы компилятора реляционной модели ЛРД — программы для автоматической генерации модуля проверки прав доступа к объектам системы на основе описания реляционной модели. Сам формат описания такой модели, являющийся входным для ее компилятора, представлен в разд. 3. Здесь проанализированы недостатки этого формата, затрудняющие его использование в больших и сложно организованных системах. В разд. 3 также представлен альтернативный формат описания реляционной модели, который более удобен в сопровождении и предоставляет возможность его преобразования в исходный формат для передачи на вход компилятора реляционной модели ЛРД. В разд. 4 описаны некоторые методы и результаты автоматизированного тестирования механизмов реляционной модели ЛРД при их использовании в целевой системе.

1. Описание используемых программных средств

В этом разделе представлены краткие описания свойств ИАС "ИСТИНА" и лежащей в основе ее разработки библиотеки Django, важных в контексте целей настоящей статьи.

1.1. Основные сведения о структуре данных ИАС "ИСТИНА"

В связи с тем обстоятельством, что процесс внедрения реляционной модели ЛРД рассматривается в настоящей работе на примере ИАС "ИСТИНА", следует более подробно рассмотреть структуру данных этой системы. Система "ИСТИНА" используется для сбора, хранения и анализа информации о результатах

научно-исследовательской и педагогической деятельности работников организаций сферы науки и образования в целях ее использования для оперативного контроля и стимулирования их профессиональной деятельности. С учетом изложенного в ИАС "ИСТИНА" предусмотрены следующие объекты.

- Зарегистрированные в системе пользователи.
- Зарегистрированные в системе организации и их подразделения.
- Работники зарегистрированных в системе организаций, для которых далее используется термин "сотрудники". Не все сотрудники обязаны быть зарегистрированы в системе как пользователи. Например, когда пользователь добавляет в систему информацию о своей статье, то может внести также и информацию о соавторах, не являющихся пользователями системы.
- Результаты научно-исследовательской деятельности зарегистрированных сотрудников, включая их статьи, книги, доклады на конференциях, защищенные диссертации, научно-исследовательские работы (НИР) и другие результаты.
- Объекты, представляющие собой коллекции связанных результатов научно-исследовательской деятельности, которыми могут быть журналы (в каждом из которых опубликованы определенные статьи), конференции (на каждой из которых сделаны определенные доклады), диссертационные советы (на каждом из которых защищены определенные диссертации) и другие объекты подобного рода.
- Вспомогательные объекты, служащие в качестве связок между объектами, естественным образом связанными друг с другом как "многие ко многим", например, объект "место работы", соответствующий записи о месте работы одного пользователя в одном подразделении и включающий такие данные, как название должности и даты начала и окончания работы.
- Ряд других объектов, которые в настоящей работе не рассматриваются.

С учетом изложенного выше важно отметить, что в системе разделены понятия "пользователь" и "сотрудник". Сотрудником называется любой работник сферы науки и образования, результаты научно-исследовательской и педагогической деятельности которого хранятся в системе. Пользователем является зарегистрированный в системе сотрудник, который может самостоятельно осуществлять операции с объектами системы. С каждым из пользователей ассоциирована соответствующая ему учетная запись сотрудника. Однако некоторые из учетных записей сотрудника при этом не принадлежат ни одному из зарегистрированных в системе пользователей.

Среди "связующих" объектов одним из наиболее важных в контексте целей и задач настоящей статьи является объект "ответственный по подразделению за сопровождение информации в системе", в дальнейшем для краткости изложения именуемый "ответственным". Данный объект связывает какого-либо пользователя системы с подразделением, для которого этот пользователь является ответственным. В обязанности ответственного по подразделению входит подтверждение и корректировка данных, вносимых в систему сотрудниками этого подразделения. Такой пользователь имеет право редактировать практически все результаты научно-исследовательской деятельности сотрудников подконтрольного ему подразделения, а также открытые в сети Интернет профессиональные профили самих сотрудников.

Следует также заметить, что для большинства объектов (исключая объекты-связки) в системе присутствует главная страница с информацией об объекте. Кроме прочих данных об объекте, на этой странице представлен список операций, которые просматривающий ее пользователь имеет право совершать над объектом. Каждое из названий операций при этом является ссылкой на страницу системы, используемую для совершения соответствующего действия. Этот факт означает, что процесс формирования страницы с информацией об объекте включает в себя большое число проверок прав на совершение разных видов доступа для одного и того же пользователя и одного и того же целевого объекта. С учетом этого факта желательным свойством механизмов управления доступом к объектам ИАС "ИСТИНА" является возможность вычислить множество всех операций, которые пользователь имеет право осуществлять над конкретным объектом быстрее, чем это можно было бы сделать, проверяя отдельно право на совершение каждой из таких операций. В разд. 2 показано, что существующие программные механизмы реализации реляционной модели ЛРД предоставляют такую возможность.

1.2. Основные сведения о библиотеке Django

Подробная информация об использовании библиотеки средств разработки программного обеспечения Django доступна в работе [3]. В настоящей публикации представлены лишь минимальные сведения, необходимые для понимания механизмов управления доступом к объектам информационных систем, разработанных на основе Django.

1.2.1. Представление записей в базе данных в виде объектов языка Python

При использовании библиотеки средств разработки Django каждый из объектов данных разрабатываемой системы является объектом класса, унаследованного от django.db.models.Model. В терминологии Django такие классы называются моделями. Каждой из моделей системы соответствует таблица в базе данных, а каждому из объектов этого класса — своя строка в этой таблице. При описании библиотеки Django строка в соответствующей таблице называется записью этой модели.

Система "ИСТИНА" является примером системы, разработанной на основе библиотеки Django. Разным классам ее объектов соответствуют разные классы-модели. Например, для пользователей системы используется модель User, которая определена в модуле django.contrib.auth.models библиотеки Django. Сотруднику соответствует модель Worker, определенная в модуле workers.models самой ИАС "ИСТИНА". Заметим, Django ориентирована на использование в сложных системах, состоящих из минимально зависимых друг от друга подсистем, называемых приложениями. Каждая из моделей должна быть определена внутри

одного из приложений. Модель при использовании Django должна однозначно определяться двумя своими свойствами — именем класса (User или Worker в представленных примерах) и именем приложения, внутри которого определена эта модель. Подсистема, содержащая модель User, называется auth, а подсистема, содержащая модель Worker — workers. Другими примерами моделей Django для различных объектов ИАС "ИСТИНА" являются модель Organization, соответствующая зарегистрированной в системе организации, и модель Department, соответствующая подразделению.

Внутри отдельного класса модели описываются поля, каждое из которых соответствует одному из столбцов таблицы. Эти поля соответствуют атрибутам объекта в терминах реляционной модели ЛРД. Каждое из полей может содержать первичный ключ (поле, автоматически получающее уникальное значение при создании нового объекта), числовое, строковое или логическое значение, а также значение даты, времени или значение внешнего ключа, ссылающегося на объект другого класса. Такие поля соответствуют отношениям между объектами системы, например, модель Worker имеет поле departments, с помощью которого можно обратиться к списку всех подразделений, в которых работает или работал ранее данный сотрудник. Модель Department, в свою очередь, содержит поле organization, указывающее на организацию, к которой относится это подразделение.

В качестве примера приведем следующий значительно упрощенный аналог модели данных для зарегистрированных в ИАС "ИСТИНА" сотрудников.

```
class Worker (models.Model):
    # Поле первичного ключа, автоматически получающее
    # уникальное значение
    id=models.AutoField(primary _ key=True, db _ column='id')
    # Два поля текстовых данных, в терминах реляционной
    # модели ЛРД называемых атрибутами
    firstname = models.StringField(db _ column='firstname')
    lastname = models.StringField(db _ column='lastname')
    # Поле внешнего ключа, в терминах реляционной
    # модели ЛРД определяющее отношение
    user = models.ForeignKey(to=User,
                          db _ column='user _ id',
                          related _ name='worker')
    # Это отношение вида "многие ко многим"
    departments = models.ManyToManyField(to=Department,
                                      through=Employment,
                                      related _ name='workers')
    # Имя соответствующей таблицы в базе данных
    # определяется во вложенном классе Meta
    class Meta:
       db _ table='Workers'
```

Этот пример не является частью устроенного значительно сложнее класса сотрудников ИАС "ИСТИНА" (описание некоторых классов ИАС "ИСТИНА", включая и класс "сотрудник", представлено в подразд. 1.1). Однако он позволяет получить представление о том, как может быть устроен класс, применяемый для той же цели. В терминологии реляционной модели данная структура содержит имя таблицы класса Worker (переменная db_table в классе Meta) и имена и типы четырех атрибутов (строковых атрибутов firstname и lastname, а также целочисленных атрибутов id и user). Атрибут id является первичным ключом, а атрибут user — внешним ключом пользователя, связанного с сотрудником. Для этих атрибутов указаны также имена столбцов в базе данных, хранящих их значения. Здесь также указано отношение, связывающее сотрудника с соответствующим ему пользователем. Это отношение соответствует ссылающемуся на пользователя внешнему ключу в таблице сотрудников. Параметр related_name для этого поля означает, что с помощью фиктивного поля User.worker можно обратиться к сотруднику, соответствующему данному пользователю, хотя в таблице для пользователей и отсутствует внешний ключ, указывающий на сотрудника. Таким образом, каждое поле внешнего ключа создает в системе два отношения, обратных друг к другу.

Последнее поле, создаваемое с помощью команды ManyToManyField, используется для отношения вида "многие ко многим". В данном случае таким отношением является отношение места работы, связывающее сотрудника с подразделением, в котором он работает. Каждый сотрудник может иметь несколько мест работы, и в каждом подразделении работает большое число сотрудников. Использование отношения "многие ко многим" требует обязательного объекта-связки, которым в представленном примере является объект места работы. Соответствующая

модель в системе называется Employment. Место работы также является моделью Django, которая должна содержать внутри себя поля внешних ключей, указывающих на сотрудника и на подразделение. Для того чтобы соответствующее поле работало корректно, определение класса Employment должно содержать в себе следующие строки:

```
class Employment (models.Model):
   ...
   worker=models.ForeignKey(to=Worker,...)
   department=models.ForeignKey(to=Department,...)
```

В этом случае значение worker.departments содержит все значения типа Department, для которых существует запись в модели Employment, содержащая соответствующее значение поля department с заданным значением поля worker. Также как и внешний ключ, поле "многие ко многим" может иметь обратное поле, имя которого определяется параметром related_name. В случае, если он задан как в представленном выше примере, объекты класса Department автоматически получают поле с именем workers, ссылающееся на список сотрудников данного подразделения. При этом в списке полей модели Department поле workers в явном виде не указывается. Заметим, что при отсутствии указания параметра related_name поле для обратного отношения будет иметь автоматически сгенерированное имя, которое может быть получено путем поиска в списке полей, связанных с моделью. Подобные способы использования Django лежат за рамками рассмотрения настоящей работы.

1.2.2. Метамодель ContentType

Особое место среди моделей Django занимает модель ContentType, входящая в приложение contenttypes. Эта модель представляет собой список всех моделей, используемых в системе (точнее, каждая из записей этой модели взаимно однозначно соответствует одной из моделей системы). Эта модель может использоваться для создания в моделях полей, которые могут ссылаться на запись любой другой модели. Для примера предположим, что каждый пользователь системы, аналогичной ИАС "ИСТИНА", может выбрать один из своих результатов научно-исследовательской деятельности как наиболее значимый. Этот результат может быть статьей, книгой, докладом на конференции или записью иного типа, но должен быть единственным. Для этого в модель Worker должны быть внесены следующие поля:

```
class Worker (models.Model):
   ...
   content _ type = models.ForeignKey(to=ContentType,...)
   object _ id = models.IntegerField(...)
   content _ object = GenericForeignKey(content _ type, object _ id)
```

Таким образом, поле content_object будет ссылкой на запись, имеющую первичный ключ, равный object_id в модели, определяемой внешним ключом content_type. Более важным с точки зрения задачи разграничения доступа к объектам системы является пример использования команды GenericForeignKey для присвоения особых меток безопасности, дающих определенному пользователю право заданного вида доступа к определенному объекту вне зависимости от прочих свойств пользователя и целевого объекта. При этом целевой объект может иметь любой тип. Одним из требований к механизмам использования реляционной модели ЛРД в информационных системах, разработанных на основе Django, является возможность контроля доступа к объекту на основании объектов, ссылающихся на него подобным образом.

1.2.3. Другие возможности библиотеки Django

Заметим, что рассмотренный выше механизм работы со структурами данных (называемый объектно-реляционной моделью) представляет лишь небольшую часть функциональных возможностей библиотеки Django. Кроме объектно-реляционной модели, Django включает в себя язык шаблонов для описания html-страниц, в определенные места которых автоматически вставляются данные, взятые из определенной модели, а также удобные механизмы задания форм для ввода данных, заготовки классов для формирования страниц со списками объектов данных, удовлетворяющих определенным условиям, и ряд других функциональных возможностей. Эти и многие другие механизмы Django в настоящей публикации не рассматриваются, а более подробно ознакомиться с ними можно, например, в работе [3].

2. Реляционная модель логического разграничения доступа

В настоящем разделе кратко представлена реляционная модель ЛРД к объектам информационных систем. Эта модель была специально разработана для управления доступом к объектам многопользовательских систем управления сетевым контентом. При

разработке реляционной модели ЛРД были приняты во внимание такие характерные особенности подобных систем, как непостоянное множество пользователей системы, большое количество разновидностей объектов и отношений между ними. Принималась во внимание также необходимость учитывать при предоставлении пользователю прав доступа к объектам опосредованные взаимосвязи между пользователем и целевым объектом. Такие взаимосвязи возникают вследствие существования объекта, связанного определенным образом как с пользователем, так и с целевым объектом. Примером такого опосредованного отношения является предоставление ответственному по подразделению прав доступа к учетным записям сотрудников этого подразделения.

Далее представлено краткое описание основных аспектов реляционной модели ЛРД. Подробное описание этой модели приведено в работах [1, 2].

2.1. Определение реляционной модели логического разграничения доступа

В настоящем подразделе кратко рассмотрены основные элементы реляционной модели ЛРД.

Определение 1. Пусть в информационной системе заданы следующие множества.

- *Objects* — множество объектов системы. Для ИАС "ИСТИНА" объектами являются пользователи системы, а также зарегистрированные в системе организации и подразделения, их сотрудники (в том числе и не зарегистрированные как пользователи) и их результаты научно-исследовательской деятельности (например, статьи, монографии, доклады на конференциях), информация о которых хранится в системе.
- $Users \subset Objects$ — множество пользователей системы, входящее в множество ее объектов.
- *Classes* — множество классов объектов системы. В ИАС "ИСТИНА" входят такие классы, как "пользователь", "сотрудник", "статья" и др.
- $Relations = PrimitiveRelations \sqcup InducedRelations$ — множество видов отношений между объектами. Примитивными отношениями (также называемыми базовыми или прямыми) называются взаимосвязи между объектами, указанные в базе данных системы. Примерами таких отношений являются отношение места работы, связывающее сотрудника с подразделением, и отношение авторства, связывающее сотрудника со статьей. Для каждого вида отношения жестко заданы классы объектов, которые могут быть соединены этим отношением. В системе также заданы порожденные отношения, для которых пары связанных объектов определяются с помощью цепочек отношений. Данное понятие описано далее.
- *Actions* — множество возможных действий над объектами. Политика безопасности определяет набор правил, согласно которому для каждой конкретной тройки $(u, a, o) \in Users \times Actions \times Objects$ разрешается либо запрещается доступ a пользователя u к объекту o. Для некоторых классов объектов некоторые виды доступа могут не иметь смысла — например, доступ "просмотр рейтинга" возможен только для сотрудников. Неприменимость какого-либо вида доступа к определенному объекту означает его запрет для всех пользователей.
- *Chains* — множество правил, определяющих два объекта системы как связанные определенным порожденным отношением, если существует последовательность объектов, связывающая эти два объекта определенной последовательностью базовых отношений.
- $AllowedActions \subset Relations \times Actions$ — множество пар (r, a), означающее, что если пользователь u связан с объектом o отношением r, то он имеет право доступа a к объекту o, для всех таких видов доступа a, что $(r, a) \in AllowedActions$.

Тогда будем считать, что в системе задана реляционная модель логического разграничения доступа.

Данное определение можно наглядно описать следующим образом. Система представляется в виде графа, называемого социальным графом, вершинами которого являются ее объекты. Два объекта соединены ребром, если они связаны каким-либо отношением, причем с каждым ребром ассоциировано имя соответствующего отношения. Заметим, что ребра могут быть кратными, т. е. два объекта могут быть связаны несколькими разными отношениями. Доступ a пользователя u к объекту o разрешается в том и только том случае, если в графе существует путь от вершины u к вершине o, соответствующий одной из цепочек отношений, разрешающих доступ a.

Заметим, что в случае ИАС "ИСТИНА", как и других систем, разработанных на основе Django, социальный граф является нагруженным, т. е. с вершинами и ребрам могут быть ассоциированы дополнительные атрибуты. Такими атрибутами являются, например, дата начала и конца периода работы сотрудника в подразделении для отношения места работы, или дата публикации для статьи. Значения атрибутов объектов могут учитываться при принятии решения о предоставлении пользователю права доступа к объекту. Для этого с некоторыми из цепочек отношений ассоциированы предикаты, зависящие от атрибутов объектов и отношений, входящих в цепочку. Если две вершины социального графа соединены путем, соответствующим данной цепочке отношений, то они считаются соединенными порожденным цепочкой отношением только в том случае, если для вершин и ребер, входящих в этот путь, соответствующий предикат истинен.

Определение 2. Пусть $c = (r_1, ..., r_n)$ — цепочка отношений, u — пользователь системы, а o — объект. Последовательность $(u = o_0; o_1, ..., o_n = o)$ называется цепочкой объектов, соответствующей цепочке отношений c, если для любого $i \in \{1, ..., n\}$ объекты o_{i-1} и o_i связаны отношением r_i и при этом для атрибутов объектов $o_1, ..., o_n$ выполнено условие цепочки c. Пользователь u называется связанным с объектом o цепочкой отношений c, если существует такая соответствующая c цепочка объектов $(o_0, ..., o_n)$, что $u = o_0$ и $o = o_n$.

Пример цепочки отношений

Рассмотрим пример. На рисунке представлен пример цепочки отношений, позволяющей ответственному по подразделению редактировать публикации сотрудников этого подразделения и его дочерних подразделений. Отношение "ответственного по статье", разрешающее доступ на редактирование, порождается цепочкой из четырех отношений: отношения ответственности пользователя по подразделению; отношения вложенности подразделений (для простоты здесь считается, что это отношение также связывает подразделения с дочерними подразделениями любого уровня вложенности, т. е. является рефлексивным и транзитивным); отношения места работы сотрудника в подразделении; отношения авторства статьи. Эта цепочка имеет предикат, означающий, что она порождает отношение "ответственного по статье" только в том случае, если дата публикации статьи входит в период работы сотрудника в подразделении. В действительности с этой цепочкой связан еще один предикат, не отмеченный на рисунке, а именно право ответственного по подразделению должно быть действительным в данный момент, т. е. текущая дата должна лежать между датами начала и конца действия полномочий, ассоциированными с отношением ответственного по подразделению. Если для пользователя u и статьи a существуют такие подразделения d_1, d_2 и сотрудник w, что цепочка объектов (u, d_1, d_2, w, a) соответствует представленной на рисунке цепочке отношений, то пользователь u считается ответственным по статье a и имеет право редактировать ее.

Следует заметить, что, как было отмечено в подразд. 1.1 настоящей статьи, среди объектов системы есть объекты, определяющие связи между другими объектами, например, объект связки подразделения с ответственным пользователем, а также объект, означающий связь между статьей и ее автором (в дальнейшем называемый объектом авторства статьи), или между сотрудником и подразделением, в котором он работает (в дальнейшем — объект места работы). При построении модели системы такие объекты могут рассматриваться как объекты, но они также могут рассматриваться и как отношения. Каждый из вариантов имеет свои достоинства и недостатки. Например, использование отношения места работы сотрудника в подразделении в качестве отдельного объекта требует внесения в систему дополнительного класса объекта "место работы" и двух новых отношений, одно из которых связывает место работы с сотрудником, к которому оно относится, а другое — с подразделением, к которому это место работы относится. Отношение места работы, связывающее сотрудника с подразделением, в котором он работает, при этом становится порожденным цепочкой, состоящей из двух указанных базовых отношений. Таким образом, выделение места работы сотрудника в подразделении в отдельный объект требует включения в модель одного дополнительного класса объектов, двух дополнительных отношений и одной дополнительной цепочки отношений. Вместе с тем использование места работы в качестве объекта позволяет регламентировать доступ к нему отдельно от доступа к связанному с ним сотруднику, либо к подразделению.

Для некоторых видов "связующих объектов" выделение в отдельный класс необходимо по причине того, что эти объекты могут использоваться, даже не имея одной из полагающихся им "связей". Например, объект "авторство статьи" содержит ссылки на статью, к которой относится, и на сотрудника, являющегося ее автором. Однако ссылка на сотрудника может быть пустой и не указывать ни на какого зарегистрированного в системе сотрудника, если при добавлении статьи пользователь не смог выбрать ее автора из предложенного системой списка сотрудников, соответствующих указанному в статье имени автора. Для такой статьи пользователь, имя которого соответствует имени, указанному в записи об авторстве статьи, не ассоциированной с конкретным сотрудником, имеет право заявить о своем авторстве. Формулировка этого правила для разрешения доступа "заявить о своем авторстве" требует, чтобы запись об авторстве статьи считалась отдельным объектом.

Таким образом, для достижения наибольшей детализации правил разграничения доступа к объектам системы имеет смысл любое отношение между объектами вида "многие ко многим" считать порожденным. Для записи в таблице этого отношения создается отдельный класс объектов, такой как, например, "место работы" или "авторство статьи", для которого в системе регистрируются отношения, связывающие его с объектами, на которые он ссылается. Например, место работы имеет ссылки на объекты сотрудника и подразделения. Само отношение места работы, связывающее подразделение с сотрудником, при этом

является порожденным цепочкой из двух отношений, первое из которых связывает подразделение со всеми записями о месте работы сотрудников в этом подразделении, а второе — место работы с соответствующим ему сотрудником. В подразд. 3.3 будет показано, что такая структура социального графа системы наилучшим образом соответствует описанию структуры данных, требуемому для использования библиотеки Django. Использование Django требует, чтобы для каждой из таблиц системы, в том числе и для таблиц отношений "многие ко многим", был зарегистрирован соответствующий класс объектов. В подразд. 3.3 будут также описаны способы разрешения возникающего при таком подходе вопроса о росте сложности описания политики безопасности.

2.2. Соответствие терминов, принятых при описании реляционной модели логического разграничения доступа и библиотеки Django

При использовании реляционной модели ЛРД в системах, разработанных на основе Django, термины, принятые при описании реляционной модели, соответствуют терминам, принятым при использовании библиотеки.

- **Класс** реляционной модели ЛРД является **моделью** Django. При этом модель является классом в терминах языка Python. С точки зрения СУБД каждая модель соответствует таблице в базе данных.
- **Объект** реляционной модели является **записью** в соответствующей модели Django. Он также является объектом соответствующего класса языка Python. С точки зрения СУБД каждая запись является строкой в таблице, соответствующей его модели.
- **Атрибутами** объектов некоторого класса являются **поля** данных в соответствующей модели. Исключением являются поля типа ForeignKey и ManyToManyField, которые, с точки зрения реляционной модели ЛРД, являются **примитивными отношениями**. Заметим, что каждое поле ForeignKey и ManyToManyField может задавать два взаимно обратных друг к другу отношения, имя одного из которых совпадает с именем поля, а имя другого задается параметром related_name при определении поля.

2.3. Программные механизмы реляционной модели логического разграничения доступа

В данном подразделе кратко рассмотрены основные программные механизмы, которые используются для внедрения реляционной модели ЛРД в web-приложения на языке Python. Авторами настоящей статьи разработано приложение, предназначенное для автоматического создания подпрограммы проверки прав доступа пользователя к объекту на основе описания реляционной модели ЛРД в XML-подобном формате, который будет описан в подразд. 3.1. Такое программное средство, называемое компилятором реляционной модели ЛРД, считывает из входного файла описание модели ЛРД, реализующей политику безопасности подконтрольной системы. Это описание включает определения используемых классов, отношений и описания цепочек отношений. На основе описания создается несколько функций на языке Python, одна из которых проверяет право конкретного пользователя осуществлять конкретную операцию над определенным целевым объектом, а другая возвращает список всех операций, которые имеет право выполнять этот пользователь над определенным целевым объектом. Алгоритмы работы этих функций выглядят аналогичным друг другу образом. Функция, проверяющая право пользователя на совершение одиночной операции над объектом, последовательно выполняет запросы к базе данных, которые, в свою очередь, проверяют существование цепочек объектов, связывающих пользователя с целевым объектом каждой из допустимых цепочек отношений. В случае, если для одной из цепочек отношений, разрешающих пользователю запрашиваемый им вид доступа к объекту, существует соответствующая цепочка объектов, связывающая пользователя с целевым объектом, функция возвращает значение True, означающее, что доступ разрешается. В случае, если ни одна из таких цепочек не соединяет пользователя с целевым объектом, возвращается значение False, означающее, что доступ запрещается.

Функция, возвращающая список всех операций, разрешенных определенному пользователю для заданного целевого объекта, работает аналогичным образом. Для каждой из цепочек отношений, которые могут соединять пользователя с объектами того же класса, что и целевой объект, выполняется запрос к базе данных, в ответ на который проверяется существование цепочки объектов, связывающей пользователя с целевым объектом этой цепочкой отношений. В случае, если такая цепочка объектов существует, в список разрешенных видов доступа добавляются виды доступа, разрешенные для отношений, порожденных этой цепочкой. В случае, если в системе применяются запрещающие правила, также ведется общий список запрещенных видов доступа, в который добавляются виды доступа, запрещенные отношениями, порожденными данной цепочкой. После проверки всех цепочек отношений функция возвращает разность между итоговыми множествами разрешенных и запрещенных видов доступа.

Важно отметить, что функции, созданные компилятором реляционной модели ЛРД, не содержат циклов и условных переходов, и поэтому выполняются быстро и не могут зацикливаться. Эти функции выполняют фиксированную последовательность запросов к базе данных системы, в каждом из которых изменяются только значения идентификаторов пользователя и целевого объекта. В зависимости от того, пуст или не пуст результат каждого из запросов к базе данных, модифицируется возвращаемое значение. Эти функции таким образом обращаются к базе данных системы в обход моделей Django, что увеличивает производительность их реализации за счет отсутствия затрат времени на вызов методов "промежуточного слоя" Django. Вместе с тем для создания таких функций требуется, чтобы описание

модели ЛРД системы включало в себя имена всех используемых таблиц и полей базы данных системы. Необходимость в таких данных вызвана тем обстоятельством, что функция проверки прав доступа не получает информации о структуре базы данных из моделей Django. Таким образом, информация о структуре базы данных оказывается дублированной. Имена таблиц и полей базы данных системы должны быть указаны как в описании модели данных в исходном коде системы, так и в описании модели ЛРД. В разд. 3 будут рассмотрены способы избежать нежелательного дублирования данных.

Создание описанного выше модуля проверки прав доступа может потребовать значительных преобразований исходного описания модели, включая операцию, именуемую приведением множества цепочек отношений к каноническому виду. Такое преобразование заключается в том, что любая цепочка, содержащая порожденное отношение, заменяется на цепочку, в которой вхождение этого отношения заменяется на подцепочку, порождающую это отношение. Таким образом, результирующее множество цепочек системы состоит из цепочек, включающих в качестве элементов только примитивные отношения. Приведение множества цепочек отношений к указанному каноническому виду необходимо для того, чтобы для каждой из цепочек вычислить SQL-запрос, выполняющий проверку существования соответствующей этой цепочке отношений цепочки объектов. Алгоритм работы компилятора реляционной модели ЛРД описан в работах [1, 2].

3. Описание реляционной модели логического разграничения доступа для приложений на основе Django

В данном разделе представлены различные поддерживаемые к настоящему времени форматы описания политики безопасности, использующей реляционную модель ЛРД. В подразд. 3.1 рассмотрен исторически первый из разработанных форматов описания модели, использующий XML-подобный синтаксис. Показано, что данный формат описания реляционной модели ЛРД является неудобным для сопровождения в силу ряда причин. В подразд. 3.2 рассмотрен альтернативный вариант описания модели, использующий в качестве основы язык Python и не имеющий недостатков, присущих XML-подобному формату. В подразд. 3.3 представлен ряд соображений, позволяющих сделать описание модели разграничения доступа к объектам системы максимально удобным и интегрировать его с исходным текстом целевой системы, сохранив при этом такие преимущества использования исходного способа описания политики, как возможность ее статического анализа и верификации.

3.1. Формат входного файла компилятора реляционной модели логического разграничения доступа

В данном подразделе рассмотрен формат описания реляционной модели ЛРД, являющийся в определенном смысле базовым для всех остальных. Как отмечено в подразд. 2.3, именно этот формат ожидается на входе компилятора реляционной модели ЛРД для создания модуля проверки прав доступа. Другие форматы описания модели, представленные в подразд. 3.2 и 3.3, конвертируются в данный формат перед их передачей на вход компилятора реляционной модели ЛРД.

Базовый формат описания реляционной модели ЛРД использует XML-подобный синтаксис. Описание модели ЛРД целевой системы заключено в тег policy, содержимое которого разделено на следующие секции:

- список классов;
- список отношений;
- список цепочек отношений;
- список типов;
- список ролей.

Здесь ролью называется свойство пользователя, на основании которого ему разрешаются некоторые виды доступа ко всем объектам, вне зависимости от связывающих их отношений. Примерами ролей являются роль активного пользователя (неактивному пользователю запрещен вход в систему и, как следствие, все действия с объектами) и часто используемая в приложениях, основанных на Django, роль суперпользователя, которому разрешено большинство видов доступа ко всем объектам. Типом является свойство объекта определенного класса, часто используемое при проверке прав доступа к этому объекту. Использование типов является способом сокращения записи правил модели и может быть заменено включением соответствующего свойства целевого объекта в правила цепочек отношений.

В описание модели ЛРД может быть также включено описание переменных окружения, т. е. данных, не содержащихся в объектах системы, но используемых при проверке прав доступа к объектам. В настоящее время единственными используемыми переменными окружения являются текущая дата и текущее время.

Следует заметить, что представленная структура описания политики безопасности не соответствует структуре разделения обязанностей разработчиков ИАС "ИСТИНА", среди которых, как правило, каждый разработчик отвечает за работу с определенными классами целевых объектов. По этой причине разумным способом структурирования списка правил разграничения доступа к объектам целевой системы была бы их группировка по классам целевого объекта доступа.

3.2. Преимущества описания правил реляционной модели логического разграничения доступа на языке Python

В данном подразделе представлено описание альтернативного формата представления реляционной модели ЛРД, не имеющего недостатков, присущих формату, представленному в подразд. 3.1. С учетом того обстоятельства, что в значительной части исходный код ИАС "ИСТИНА" и библиотеки Django

написан на языке Python, при разработке альтернативного формата было принято решение взять за основу именно этот язык, как хорошо знакомый специалистам, ответственным за сопровождение и развитие системы. Для этого был реализован отдельный программный модуль специализированной библиотеки на языке Python, включающий функции, необходимые для формирования структуры модели ЛРД, основанной на реляционных принципах, в целях ее сохранения в файл в формате, который был представлен в подразд. 3.1. Таким образом, описанием модели служит сама последовательность команд, формирующая объект такой модели. Использование языка Python при этом позволяет использовать его возможности для создания сокращенных форм описания часто повторяющихся фрагментов правил разграничения доступа. В случае ИАС "ИСТИНА" примером такого часто повторяющегося фрагмента является фрагмент вида "поле начальной даты в некотором объекте меньше текущей даты, а поле конечной даты — больше текущей даты, либо не задано". При проверке прав доступа пользователей к различным объектам системы подобные условия могут использоваться для дат начала и конца срока действия полномочий ответственного по подразделению, для периода работы сотрудника в подразделении, периода членства сотрудника в диссертационном совете, для даты начала и конца сроков действия, которые ассоциированы со многими другими объектами системы.

3.3. Интеграция описания политики безопасности с описанием модели данных системы

Как было отмечено в подразд. 1.2, приложение на основе библиотеки Django включает в себя описание всех структур данных системы в виде классов, производных от django.db.models.Model. С учетом этого факта одним из способов значительно упростить описание реляционной модели ЛРД является отказ от дублирования в этом описании сведений, уже содержащихся в исходном коде целевой системы, например, имен таблиц и столбцов в базе данных. Для этой цели можно интегрировать библиотеку для создания структуры реляционной модели ЛРД в код целевой системы. Библиотека Django позволяет выполнять некоторые подпрограммы web-приложения, запуская их из командной строки без запуска web-сервера. Таким образом, включение в код системы библиотеки для работы с реляционной моделью ЛРД позволит запускать ее в режиме создания файла такой модели без возможности доступа к этой функции через web-интерфейс. Вместе с тем библиотека для создания модели ЛРД получит возможность обращаться к моделям Django, получая из них информацию об используемой каждой моделью таблице в базе данных и ее взаимосвязях с другими моделями. Таким образом, можно избежать необходимости указывать эту информацию в описании политики безопасности системы, существенно сократив ее объем и упростив ее сопровождение.

Например, для предоставления пользователю доступа на редактирование соответствующей ему учетной записи сотрудника в примере модели сотрудника, представленном выше, может использоваться следующая команда:

```
policy.grantModelAccess (Worker,
                         nodes = 'worker',
                         actions = 'edit'])
```

Эта команда означает, что пользователь получает право доступа edit к объекту типа Worker, с которым пользователя связывает фиктивное поле модели worker. Это поле объявлено в представленном выше примере как обратное отношение к полю user в модели Worker, которому соответствует поле user_id в таблице Workers.

Одной из трудностей, возникающих при таком подходе, является то, что имена полей внешних ключей и обратных к ним, используемые в Django, не могут однозначно идентифицировать отношение в пределах всей системы. Поле внешнего ключа с именем worker может содержаться и в других классах, содержащих ссылку на единственного пользователя, например, в классе для места работы или авторства статьи. Аналогичным образом внешний ключ с именем user есть во многих классах. По этой причине в модель было введено понятие псевдонима отношения. Если имя отношения является строкой, однозначно идентифицирует отношение среди всех отношений, зарегистрированных в системе, то псевдоним однозначно определяет отношение среди множества исходящих отношений одного класса, т. е. отношений, для которых "слева" должен стоять объект определенного класса. В этом случае в качестве псевдонима отношения может использоваться имя соответствующего поля в модели Django. В случае, если имя отношения не указано при описании политики безопасности в явном виде, в качестве имени может использоваться комбинация наподобие имя_левого_класса__псевдоним.

Таким образом, описание реляционной модели ЛРД встраивается в целевую систему в виде последовательности команд, формирующих объект политики безопасности. Особой командой этот объект сохраняется в XML-файл, который обрабатывается компилятором реляционной модели ЛРД, что дает на выходе файл на языке Python, содержащий функции для проверки права определенного доступа пользователя к объекту и для вычисления всех видов доступа определенного пользователя к определенному объекту. Эти функции выполняются быстро и гарантированы от зацикливания в силу того обстоятельства, что каждая из них выполняет фиксированную последовательность SQL-запросов к базе данных системы, в которые подставляются только идентификаторы пользователя и целевого объекта. Созданный файл затем включается в исходный код системы, и именно эти функции выполняют проверку прав доступа к объектам в процессе работы web-приложения.

Использование такой технологии имеет приведенные далее преимущества перед непосредственным выполнением команд проверки доступа к объектам, написанным на языке Python.

- Использование промежуточного, более простого по сравнению с описанием на Python, формата хранения правил разграничения доступа к объектам делает возможным статический анализ и верификацию этих правил. Например, компилятор реляционной модели ЛРД проверяет цепочки отношений на наличие зацикленных правил порождения отношений, т. е. ситуации, при которой отношение *a* порождается цепочкой, содержащей отношение *b*, а *b* — цепочкой, содержащей *a*. Выполнение алгоритма проверки прав доступа, описанного на Python, непосредственно при поступлении запроса пользователя на доступ к объекту приведет при этом к зависанию системы. Очевидно, что подобную ситуацию возможно выявить при автоматизированном тестировании системы. Вместе с тем, если зацикливание проверки прав доступа происходит в редких случаях, выявление проблемы путем модульного тестирования может потребовать настолько большого тестового покрытия, что выполнение тестов займет неприемлемо много времени. Вопросы построения оптимального тестового покрытия для автоматизации тестирования механизмов разграничения доступа к объектам системы будут рассмотрены в подразд. 4.2. Статический анализ используемой модели логического разграничения доступа к объектам системы эффективно дополняет автоматическое модульное тестирование, выявляя некоторые ошибки в описании модели, даже если вероятность того, что эта ошибка действительно произойдет, крайне мала.
- Функции проверки прав доступа, созданные компилятором реляционной модели ЛРД, являются более быстрыми по сравнению с непосредственно написанными в коде целевой системы.
- Ограниченность средств, которые можно использовать при описании модели разграничения доступа к объектам системы, снижает вероятность некоторых ошибок при описании модели.

Вместе с тем при описании модели разграничения доступа к объектам целевой системы описанным выше способом можно использовать все механизмы языка Python для создания, например, сокращенных форм записи часто используемых видов правил. В качестве примера представлена функция, возвращающая условие, налагаемое на промежуточный объект цепочки отношений — указанная в объекте дата начала срока его действия должна быть раньше текущей даты, а дата конца срока действия — позже текущей даты:

```
def currentDateInRange (begin, end, current _ date = 'current _ date'):
   u"""
   Возвращает строку условия принадлежности текущей даты
   периоду между датами begin и end. При этом для полей
   begin и end допускается значение Null, которое для
   begin соответствует бесконечно удаленному прошлому,
   а для end — бесконечно удаленному будущему
   :param str begin:
       Имя атрибута объекта или отношения, хранящего
       начальную дату целевого периода
   :param str end:
       Имя атрибута объекта или отношения, хранящего
       конечную дату целевого периода
   :param str current _ date':
       Имя переменной окружения, хранящей текущую дату
   """
   return ("((%s <= env (%s) or isNull (%s))" +
       "and (%s >= env (%s) or isNull (%s)))") \
       % (begin, current _ date, begin, \
       end, current _ date, end)
```

Заметим, что в этом примере обе даты могут быть равны NULL, что означает дату в бесконечно удаленном прошлом — для даты начала срока действия объекта и дату в бесконечно удаленном будущем — для даты конца срока действия объекта.

Данное правило часто используется в различных цепочках отношений для таких объектов, как запись об ответственном по подразделению, о месте работы сотрудника и других объектов, для которых задана дата начала и конца срока их действия.

4. Автоматизация процедуры тестовых испытаний программных механизмов реляционной модели логического разграничения доступа

В настоящем разделе рассмотрено решение задачи автоматизированного тестирования механизмов логического разграничения доступа, реализованных в целевой системе. Существует ряд причин, по которым в систему следует внедрить механизмы

такого тестирования. Первая из этих причин состоит в том, что автоматический тест корректности работы системы можно выполнять после каждого серьезного изменения в ее исходном коде. Ручное тестирование корректности механизмов ЛРД при этом потребовало бы от разработчиков слишком много времени. Вторая причина состоит в том, что при сложной политике безопасности гарантия корректности ее соблюдения требует проверки на большом числе пользователей и объектов доступа. Это означает, что ручное тестирование механизмов безопасности системы потребует слишком много времени. Оценка количества пользователей и целевых объектов, на примере которых необходимо проводить тестирование подсистемы ЛРД, представлена далее в подразд. 4.2. Наконец, для оценки производительности механизмов управления доступом также необходимо большое число прогонов тестов для обеспечения статистической значимости данных о времени выполнения функции, проверяющей права доступа пользователя к объекту. Это обстоятельство также приводит к необходимости автоматизации тестовых испытаний механизмов безопасности системы.

4.1. Методы автоматизации тестирования приложений на основе Django

Библиотека Django содержит ряд инструментальных средств, которые можно использовать для тестирования различных подсистем использующего ее приложения. Такие средства имеет смысл разделить на две группы. Первую группу составляют средства тестирования приложений, входящие в стандартную библиотеку языка Python, которые были незначительно доработаны в целях их эффективного использования для тестирования модулей web-приложений. Эта группа средств автоматизации тестирования приложений будет рассмотрена в подразд. 4.1.1. Вторая группа средств автоматизации тестирования приложений на основе Django включает в себя тестовый web-клиент, позволяющий эмулировать получения тестируемой системой определенных HTTP-запросов и анализировать ее ответы. Использование данного средства будет рассмотрено в подразд. 4.1.2.

4.1.1. Основные средства тестирования

Для тестирования поведения конкретных функций или классов системы используется модуль django.test библиотеки Django. При использовании этого модуля необходимо создать класс, производный от django.test.TestCase и включающий в себя методы, имена которых начинаются с префикса test_, каждый из которых выполняет тестирование поведения определенной функции в определенной ситуации. Любое приложение на основе Django можно запустить в режиме самотестирования с помощью команды

```
python./manage.py test
```

При этом будут выполнены все тестирующие методы во всех классах, производных от TestCase, находящихся в файлах с именами tests.py и других файлах, имена которых начинаются со слова test, во всех модулях системы. При необходимости выполнить тестирование функций, определенных в других файлах, имя файла указывается явно.

Класс django.test.TestCase является унаследованным от класса unittest.TestCase, входящего в стандартную библиотеку языка Python и описанного в работе [5]. В число методов этого класса входят методы, имена которых начинаются с префикса assert, например, метод assertTrue, проверяющий истинность своего аргумента и прерывающий выполнение тестового метода с сообщением об ошибке в случае, если аргумент интерпретируется как ложный. Аналогичный ему метод assertEqual принимает два аргумента и прерывает тестирование с сообщением об ошибке, если аргументы не равны. Похожим образом действуют методы assertFalse, assertNotEqual, assertAlmostEqual, assertGreater и другие методы, описания которых доступны в документации по классам django.test.TestCase, либо unittest.TestCase. Заметим, что большинство этих методов принимают параметр msg, который содержит текст сообщения об ошибке, выводимого на экран при непрохождении теста. Этот параметр позволяет узнавать дополнительную полезную информацию о причинах возникшей ошибки.

Среди других методов класса django.test.TestCase следует упомянуть методы setUp и tearDown, а также метод skipTest, позволяющий отказаться от выполнения теста с указанием причины. Метод skipTest может быть вызван, например, если в базе данных системы отсутствуют объекты с необходимыми для тестирования свойствами. Метод setUp выполняется перед выполнением каждого из тестирующих методов класса и может включать подготовку данных, необходимых большинству из них. Метод tearDown выполняется после каждого тестового метода и уничтожает вспомогательные данные. Таким образом, для каждого из методов тестового класса, имя которого начинается с test_, выполняется следующая последовательность действий: создается экземпляр тестового класса, для него выполняется метод setUp, затем сам тестовый метод, после него — метод tearDown, затем объект тестового класса уничтожается и для выполнения следующего тестового метода создается новый объект. По этой причине тестовый метод не должен зависеть от состояния тестового объекта после выполнения другого тестового метода — каждый из них выполняется в своем тестовом объекте.

Существуют два способа обеспечить наличие в базе данных системы объектов со свойствами, обусловливающими поведение, которое необходимо протестировать. Первый из них заключается в том, чтобы создать эти объекты перед началом тестирования и удалить после его окончания. Второй способ заключается в том, чтобы найти объекты с нужными свойствами в постоянно работающей базе данных, предназначенной специально для тестирования

новых функциональных возможностей системы. Первый из этих вариантов является более универсальным, однако выполняется значительно дольше в случае необходимости выполнить тест для большого числа объектов. Заметим, что второй вариант требует внесения некоторых модификаций в механизмы тестирования Django, так как по умолчанию тестирование всегда проводится для создаваемой специально для него и изначально пустой базы данных. Для создания и удаления тестовой базы данных используется объект класса django.test.runner.DiscoverRunner, и для того чтобы проводить тестирование системы с использованием долговременной базы данных, необходимо создать класс-потомок этого класса с пустыми реализациями методов setup_databases и teardown_databases.

При использовании реляционной модели ЛРД вопрос о формировании списка свойств объектов, для которых необходимо выполнить тестирование, может являться достаточно сложным. Теоретически идеальным вариантом является для каждого из классов целевых объектов и каждой цепочки отношений выполнить тестирование для двух пар, состоящих из пользователя и объекта доступа. В первом случае пользователь должен быть связан с целевым объектом указанной цепочкой отношений, а во втором случае пользователь и целевой объект не должны быть связаны. Выполнение полного набора тестов может быть затруднено двумя обстоятельствами. Первое из этих обстоятельств заключается в том, что при наличии в системе большого числа цепочек отношений тестирование может занимать весьма продолжительное время. Второе обстоятельство обусловлено тем, что для цепочек отношений, имеющих условия, обеспечение полноты тестового покрытия требует также проверки прав доступа пользователя к объекту, связанному с ним цепочкой отношений нужного вида, для которой однако не выполнены соответствующие условия, зависящие от атрибутов входящих в цепочку объектов. Более подробно вопрос расчета оптимального тестового покрытия будет рассмотрен в подразд. 4.2.

4.1.2. Использование встроенного тестового клиента Django

Вторым способом автоматизации тестирования приложений на основе Django является использование эмулятора web-клиента. В этом режиме тестирующая подпрограмма формирует HTTP-запрос, после чего выполняются тестируемые функции, которые выполнялись бы при получении такого запроса от пользователя. Эти функции формируют HTTP-ответ, который анализируется тестирующей подпрограммой, проверяющей соответствие вида передаваемой пользователю информации ожидаемому виду. Данный метод тестирования является более высокоуровневым по сравнению с представленным в подразд. 4.1.1, так как тестируется вид страницы с точки зрения конечного пользователя и при этом игнорируются детали происходящего в процессе формирования страницы. Тестирование приложений данным способом необходимо в связи с тем обстоятельством, что Django позволяет использовать косвенно-вызываемые функции, выполняющие как предобработку входящих HTTP-запросов, так и постобработку ответов системы. Таким образом, разработчику может оказаться затруднительным видеть сразу все фрагменты кода системы, выполняемые в ответ на определенные запросы пользователя. В случае, если формирование HTTP-ответа является сложной процедурой, данный метод тестирования обнаруживает ошибку, на какой бы из стадий формирования ответа она не произошла. В этом случае, однако, тестирование не дает информации о локализации ошибки, а только сообщает об ее наличии. В целях локализации ошибки впоследствии потребуется проверка работы каждой из вызываемых при формировании ответа системы функций.

4.2. Методы разработки наиболее полного тестового покрытия для реляционной модели логического разграничения доступа

Как было отмечено в подразд. 4.1.1, при автоматизации тестирования механизмов реляционной модели ЛРД построение набора тестов, обеспечивающих проверку корректности обработки всех возможных цепочек отношений, представляет собой нетривиальную задачу. Необходимо обеспечить тестирование механизмов разграничения доступа для пользователей и целевых объектов, связанных различными предусмотренными в системе цепочками отношений. Необходимо также обеспечить тестирование цепочек объектов, которые попарно связанны между собой примитивными отношениями, составляющими цепочку, для атрибутов которых не выполнено условие цепочки, например, удостовериться, что ответственный по подразделению не имеет право редактировать объекты, связанные с подразделением, если истек срок действия его полномочий.

Рассмотрим пример. Цепочка отношений, представленная на рисунке, включает четыре примитивных отношения и условие, связывающее атрибут целевого объекта с атрибутами одного из примитивных отношений, включенных в цепочку. При этом одно из отношений — отношение вложенности объектов — является рефлексивным и транзитивным.

С учетом изложенного для представленной на рисунке цепочки необходимо проверить корректность предоставления пользователю права доступа к набору целевых объектов (статей), содержащему не менее одного объекта, удовлетворяющего каждому из перечисленных далее условий.

- Статья, не имеющая автора, работающего в каком-либо подразделении, для которого пользователь является ответственным.

- Статья, имеющая автора, работающего в каком-либо подразделении, для которого пользователь непосредственно назначен ответственным.
- Статья, имеющая автора, работающего в каком-либо подразделении, дочернем для подразделения, для которого пользователь непосредственно назначен ответственным.
- Статья, имеющая автора, работающего хотя бы в одном подразделении, дочернем для одного из дочерних подразделений подразделения, для которого пользователь непосредственно назначен ответственным.
- Статья, имеющая автора, который работает или работал в подразделении, подконтрольном пользователю, но не работал в нем на момент написания статьи.

При такой проверке существует большое число способов увеличить детализацию тестового покрытия. Последнее условие может быть, например, комбинировано с тремя предыдущими, что даст в результате семь пар отношений, включающих пользователя и целевой объект. Кроме того, условие попадания даты публикации статьи в период работы пользователя в подразделении может быть нарушено разными способами. Статья может быть опубликована как до поступления автора на работу в подразделение, подконтрольное пользователю, так и после его увольнения. Кроме того, автор статьи может работать в подразделении, подконтрольном пользователю, в настоящее время, однако также возможна и ситуация, при которой сотрудник не работает в подразделении в настоящее время, но работал в нем на момент выхода статьи. Таким образом, попытка учесть все комбинации атрибутов пользователя и целевого объекта, влияющие на права доступа к этому объекту, приводит к недопустимо большому набору тестовых данных. По результатам анализа такого набора можно предложить перечисленные далее подходы к формированию тестовых наборов данных.

- Выделение групп пользователей, которые в большинстве случаев получают права доступа к объектам за счет цепочек отношений, имеющих некоторое количество одинаковых элементов в начале. К таким группам пользователей относятся, например, следующие группы: "суперпользователи", "ответственные по организациям или подразделениям", "ученые секретари диссертационных советов" или "руководители НИР". При этом в набор данных для тестирования механизмов разграничения доступа обязательно должен быть включен хотя бы один пользователь, входящий в каждую из таких групп, и хотя бы один пользователь, не входящий в каждую из таких групп.
- Регулярное тестирование на максимально широком наборе данных через большие временные промежутки, например, перед выпуском очередной стабильной версии программной системы. При этом после внесения модификаций в политику безопасности системы тестирование проводится только для тех объектах, права доступа к которым с большой вероятностью могут измениться после внесения этих модификаций.
- Наиболее тщательное тестирование функций, которые можно считать с большей вероятностью содержащими ошибки. В работе [6] представлен ряд подходов к задаче оценки вероятности сбоев в том или ином модуле программной системы. В настоящее нельзя сделать вывод о применимости методов оценки количества дефектов программного обеспечения, рассмотренных в работе [6], к исходному коду ИАС "ИСТИНА". Анализ применимости таких методов в ИАС "ИСТИНА" является одним их перспективных направлений дальнейших исследований.

Заключение

Рассмотрены постановки и предложенные авторами подходы к решению ряда задач, возникающих при внедрении в практику реляционной модели логического разграничения доступа к объектам целевой системы, основанной на web-приложении, разработанном с использованием библиотеки Django. Рассмотрен ряд способов задания реляционной модели логического разграничения доступа к объектам целевой системы. Предложен наиболее эффективный с точки зрения удобства сопровождения целевой системы вариант включения в ее код описания используемой модели разграничения доступа к объектам. Предложены подходы к автоматизации процедур тестирования механизмов реляционной модели логического разграничения доступа в рамках целевой системы.

Список литературы

1. **Васенин В. А., Иткес А. А., Шапченко К. А., Бухонов В. Ю.** Реляционная модель логического разграничения доступа на основе цепочек отношений // Программная инженерия. 2015. № 9. С. 11—19.
2. **Васенин В. А., Иткес А. А., Бухонов В. Ю., Галатенко А. В.** Модели логического разграничения доступа в многопользовательских системах управления наукометрическим контентом // Программная инженерия. 2016. Т. 7, № 12. С. 547—558.
3. **Дронов В.** Django: практика создания Web-сайтов на Python. СПб.: БХВ-Петербург, 2016. 528 с.
4. **Садовничий В. А., Афонин С. А., Бахтин А. В.** и др. Интеллектуальная система тематического исследования научно-технической информации ("ИСТИНА"). М.: Изд-во МГУ, 2014. 262 с.
5. **Сузи Р. А.** Язык программирования Python. М.: Интернет-Университет Информационных Технологий, БИНОМ, 2006. 328 с.
6. **Идеальная** разработка ПО. Рецепты лучших программистов / Под ред. Э. Орама и Г. Уилсона. СПб.: Питер, 2012. 592 с.

Using Relation-Based Access Control Model within Django-Based Web Applications

V. A. Vasenin, e-mail: vasenin@msu.ru, Faculty of Mechanics and Mathematics, Lomonosov Moscow State University, Moscow, 119991, Russian Federation, **A. A. Itkes,** e-mail: itkes@imec.msu.ru, Scientific Research Institute of Mechanics, Lomonosov Moscow State University, Moscow, 119192, Russian Federation

Corresponding author:

Vasenin Valery A., Professor, Moscow State University, Moscow, 119191, Russian Federation, E-mail: vasenin@msu.ru

Received on February 26, 2018
Accepted on March 14, 2018

This article examines the approaches to implementation of relation-based access control model within Web applications based on the Django framework. It introduces the mechanisms of describing the access control models within the code of such systems providing ability to analyze the access control rules statically. At the same time the proposed access control implementation mechanism provides an easy way to maintain the access control model without need of writing access control rules on other languages but Python, which the Django library is written on. Also this article describes some ways of building automated tests for the relation-based access control mechanism.

Keywords: *access control, information security, relation-based access control model, web applications*

For citation:

Vasenin V. A., Itkes A. A. Using Relation-Based Access Control Model within Django-Based Web Applications, *Programmnaya Ingeneria,* 2018, vol. 9, no. 5, pp. 195—208.

DOI: 10.17587/prin.9.195-208

References

1. **Vasenin V. A., Itkes A. A., Shapchenko K. A., Bukhonov V. Yu.** Reljacionnaja model' logicheskogo razgranichenija dostupa na osnove cepochek otnoshenij (Relational Access Control Model Based on Chains of Relations), *Programmnaya Ingeneria*, 2015, no. 9, pp. 11—19 (in Russian).

2. **Vasenin V. A., Itkes A. A., Bukhonov V. Yu., Galatenko A. V.** Modeli logicheskogo razgranichenija dostupa v mnogopol'zovatel'skih sistemah upravlenija naukometricheskim kontentom (Access Control Models in Multiuser Scientometric Content Management Systems), *Programmnaya Ingeneria*, 2016, vol. 7, no. 12, pp. 547—558 (in Russian).

3. **Dronov V.** *Django: praktika sozdanija Web-sajtov na Python* (Django: the practice of creating Web sites on Python), Saint-Petersburg, BHV-Peterburg, 2016, 528 p. (in Russian).

4. **Sadovnichij V. A., Afonin S. A., Bahtin A. V., Buhonov V. Ju., Vasenin V. A., Gankin G. M., Gasparjanc A. Je., Golomazov D. D., Itkes A. A., Kozicyn A. S., Tumajkin I. N., Shapchenko K. A.** *Intellektual'naja sistema tematicheskogo issledovanija nauchno-tehnicheskoj informacii* ("ISTINA") (Intelligent system case studies of scientific and technical information ("ISTINA")), Moscow, Izd-vo MGU, 2014, 262 p. (in Russian).

5. **Suzi R. A.** *Jazyk programmirovanija Python* (Python programming language), Moscow, Internet-Universitet Informacionnyh Tehnologij, BINOM, 2006, 328 p. (in Russian).

6. ***Ideal'naja*** *razrabotka PO. Recepty luchshih programmistov* (The ideal software development. Recipes of the best programmers), Eds. by Je. Orama, G. Uilson, Saint Petersburg, Piter, 2012, 592 p. (in Russian).

УДК 004.434 DOI: 10.17587/prin.9.209-220

В. М. Ицыксон, канд. техн. наук, доц., зав. кафедрой, e-mail: vlad@icc.spbstu.ru,
Санкт-Петербургский политехнический университет Петра Великого, г. Санкт-Петербург

LibSL — язык спецификации компонентов программного обеспечения

Бóльшая часть современного программного обеспечения широко использует сторонние библиотеки и компоненты. Такой подход позволяет упросить проектирование и разработку, сократить стоимость создания программного обеспечения и время выхода на рынок. Основная трудность в использовании сторонних библиотек заключается в отсутствии приемлемой документации, описывающей эти библиотеки. Это, в свою очередь, является причиной низкого качества создаваемого программного обеспечения и может приводить к различным программным дефектам. В статье рассмотрены существующие подходы к формальному языковому описанию программных библиотек. Такие языковые описания могут быть основой автоматизации решения определенного класса задач программной инженерии, связанного с многокомпонентными проектами, в которых отсутствует доступ к исходному коду компонентов. На базе сформулированных требований разрабатывается синтаксис и определяется семантика нового языка частичных спецификаций библиотек LibSL. Язык основан на формализме взаимодействующих расширенных конечных автоматов и призван описывать интерфейс и видимое поведения библиотек, необходимые при автоматизации решения ряда задач программной инженерии, таких как поиск ошибок в программных системах, использующих библиотеки, проверка корректности протокола доступа к библиотекам, проверка совместимости библиотек, реинжиниринг программного обеспечения и т. п.

Ключевые слова: *программная библиотека, компонент программного обеспечения, частичная спецификация, язык спецификации поведения*

Введение

Индустрия разработки программного обеспечения в последние годы находится на подъеме. Этот подъем обусловливается многими факторами. В первую очередь он связан с проникновением программного обеспечения во все сферы жизнедеятельности человека. Особенно этому способствует бурное развитие беспроводных сетей, носимой электроники (*wearable devices*) и интернета вещей (*Internet of Things*). Следствием увеличения числа устройств является резкое увеличение потребности в разнообразном программном обеспечении. При этом требования к срокам выпуска новых приложений или обновлению старых также резко ужесточились. Цикл выпуска многих продуктов сократился с нескольких лет до нескольких недель.

Не менее важной тенденцией является проникновение программного обеспечения в критически важные сферы деятельности человека, связанные с его здоровьем, безопасностью, финансами, энергетикой и т. п. Все это предъявляет серьезные требования к таким характеристикам качества программного обеспечения, как надежность и безопасность.

Все представленные выше факторы приводят к изменениям технологий разработки программного обеспечения, которые выражаются:

- в широком использовании при проектировании готовых компонентов, хорошо зарекомендовавших себя ранее при решении определенного класса задач;
- в применении новых методов обеспечения качества программных систем, основанных на формальных моделях.

Исследование, результаты которого представлены в настоящая работе, посвящено развитию методов надежного компонентного программирования, базирующихся на формальных описаниях поведения отдельных компонентов программного обеспечения и протоколов их взаимодействия. В основе подхода лежит формализм, описанный в работе [1]. Поведение отдельных компонентов (библиотек) задается с помощью системы взаимодействующих расширенных конечных автоматов, которые, по сути, являются частичными спецификациями библиотек. Они в явном виде отображают видимое извне поведение компонентов, абстрагируясь при этом от реализации этих библиотек. Формализм, описанный в работе [1], трудно использовать непосредственно. Необходимо

иметь удобный инструментарий, позволяющий авторам компонентов или их активным пользователям формировать спецификации библиотек, не вникая в математические основы формализмов, лежащих в основе предлагаемого подхода. В таких случаях оптимальным решением является разработка специализированного предметно-ориентированного языка (*Domain Specific Language*, DSL), позволяющего выражать все особенности библиотек с помощью понятных языковых средств. Таким образом, целью работы является создание предметно-ориентированного языка частичных спецификаций компонентов программного обеспечения.

Статья организована следующим образом. В разд. 1 приведено сравнение существующих языковых средств для описания компонентов. Разд. 2 посвящен формированию требований к языку. В разд. 3 описано проектирование предметно-ориентированного языка LibSL (*Library Specification Language*). В разд. 4 рассмотрены варианты применения разработанного языка. В заключении сделаны выводы и сформулированы направления дальнейших исследований.

1. Существующие подходы к языковому описанию библиотек

Идея использовать специальные языки для описания библиотек родилась довольно давно. Первые языки, в той или иной мере описывающие библиотеки, появились в начале 90-х гг. XX века, после чего были разработаны десятки языков, которые можно отнести к языкам спецификации библиотек. Языки спецификации библиотек относятся к классу языков поведенческого описания интерфейсов (*Behavioral Interface Specification Languages*). Подробный обзор таких языков можно найти в работе [2]. При этом все существующие языки можно условно поделить на несколько групп.

Языки описания отдельных функций библиотек являются наименее интересными, так как описания отдельных функций входят как составная часть во все остальные группы языков. Языки первой группы не могут решить задачи данной работы, поэтому здесь не рассмотрены.

Языки описания API библиотек предназначены для полного или частичного описания API (*Application Programming Interface*) библиотек. При этом зачастую описание API рассматривается в расширенном толковании, когда специфицируются не только сигнатуры функций библиотеки, но также и типы данных, необходимые при ее использовании. Развитие языков этой группы тесно связано с развитием в 90-х гг. XX века распределенных объектных технологий типа CORBA и DCOM. Языки этой группы обычно называют языками описания интерфейсов (*Interface Definition Language*, IDL). Были разработаны целые семейства языков: CORBA IDL, Microsoft IDL. Основная задача этих языков — детальная спецификация API компонента, выражающаяся в описании сигнатур функций или методов, описании используемых в параметрах функций типов данных, а также иногда в описании типов данных, создаваемых до или в результате вызова функций API. Такая детализация важна при создании распределенных приложений, где специфицированные данные для передачи по сети могут быть подвергнуты сериализации. Ограничением таких языков в контексте данной работы является невозможность описывать динамическую семантику библиотек, т. е. особенность их поведения.

Языки описания условий использования функций библиотек предназначены для спецификаций состояния внешней программы до и после выполнения функции библиотеки. Обычно для этого используются предикаты, заданные в форме логики первого порядка. Такими предикатами могут описываться пред- и постусловия функций, инварианты. Предусловия могут быть использованы статическими и динамическими чекерами для проверки корректности условий вызова функций API из программы, а постусловия и инварианты — для проверки выполнения ими определенных правил и гарантирования некоторых свойств программы. К языкам этой группы может быть отнесен язык ADL (*Assertion Definition Language*) [3], который разработан для языка программирования C и расширяет языки описания интерфейсов контрактами использования функций. В последнее время языки описаний условий использования функций являются частью более мощных языков спецификаций. Характерным примером является *Java Modelling Language* (JML) [4, 5], являющийся внутренним предметно-ориентированным языком аннотаций для программ на языке программирования Java. Синтаксически аннотации выполнены в виде структурированных комментариев и встраиваются непосредственно в текст программы. Схожие подходы используются и для спецификации свойств языка программирования C. В языке ACSL (*Ansi C Specification Language*) [6, 7] аннотации также встраиваются в C-программу в виде специализированных комментариев. В подходе VCC, развиваемом Microsoft Research [8], аннотации являются расширением языка, которое обрабатывается специальным препроцессором. В рамках решаемой в данной работе задачи все такие языки имеют два ограничения. Во-первых, в силу того, что спецификации встраиваются непосредственно в код программ, их затруднительно использовать в случае, когда исходный код библиотеки недоступен. Во-вторых, предусловия, постусловия, инварианты, утверждения (*assertions*), выражаемые формулам логики первого порядка, задают только часть динамической семантики, не позволяя при этом специфицировать все видимое поведение функции или библиотеки.

Языки, специфицирующие допустимое использование библиотек, предназначены для использования в составе систем обнаружения ошибок. Примером языка из этой группы может служить язык metal, разработанный в ходе исследовательского проекта Стэнфордского университета по созданию средства статического анализа исходного кода xgcc/xg++ [9—11].

Архитектура xgcc подразумевает не только использование стандартных проверок, зашитых в анализатор, но и возможность добавления своих проверок (*checkers*) пользователем. Такие проверки в первую очередь необходимы, когда в состав анализируемого программного проекта входят внешние компоненты, исходный код которых недоступен, но имеется необходимость описания поведения этих компонентов для проверки корректности их использования. Для этой цели был разработан язык metal, который, по сути, является языком, специфицирующим допустимое использование библиотек. Язык использует конечно-автоматные модели, основанные на состояниях, и имеет средства сопоставления с образцом (*pattern matching*) для выявления мест применения проверок. В полной мере язык metal не может использоваться для целей, поставленных в настоящей работе, так как не содержит полноценных средств описания библиотек, сигнатур функций и т. п. Однако некоторые идеи по организации работы с автоматами могут быть использованы при разработке языка спецификации библиотек.

Похожий подход используется в технологии SLAM [12—14], разработанной в Microsoft Research и применяемой в компании Microsoft для контроля качества системного программного обеспечения, в том числе драйверов. Для этого на языке SLIC [15] описывается множество правил проверки корректности использования системных API (*checkers*), с помощью которых контролируется качество разработанных сторонними вендорами драйверов. Драйверы, не прошедшие проверки, не могут использоваться поставщиками. В состав языка SLIC включены директивы сигнализирования об ошибках, которые индицируют нарушения каких-либо правил использования системных API. Для целей, преследуемых в данной работе, язык SLIC не подходит ввиду того, что, во-первых, не предлагает средств для формального описания структуры и поведения библиотек, а во-вторых, использует слишком мощную тьюринг-полную вычислительную модель, которая ограничивает возможности ее автоматизированного анализа и преобразования.

Языки, специфицирующие семантику библиотек, предназначены для детального описания структуры и поведения библиотеки. Примером такого языка является разработанный ранее при участии автора язык PanLang [16]. Язык PanLang проектировался для статического анализатора программ на языке C Aegis, а после был модифицирован для решения частных задач реинжиниринга программного обеспечения [17]. Язык содержит средства спецификации API библиотеки и укрупненные алгоритмы работы функций библиотеки, существенные для использования. При этом могут описываться побочные эффекты функций, такие как выполнение каких-либо семантических действий или индикация наличия ошибочной ситуации. Для решения задач данной работы PanLang не может быть использован в полной мере в силу следующих имеющихся ограничений. Во-первых, язык PanLang ориентирован только на библиотеки, написанные на языке C. Во-вторых, поведение каждой функции библиотеки описывается отдельно в отрыве от логики функционирования библиотеки в целом. Как следствие, динамическая семантика всей библиотеки задается только неявно.

2. Требования к разрабатываемому языку

Разрабатываемый авторами и описанный в настоящей работе язык спецификаций проектируется в соответствии с формализмом и требованиями к языку, изложенными в работе [1]. В этой работе введен формализм, позволяющий описывать структуру и поведение библиотеки в виде системы взаимодействующих расширенных конечных автоматов, а также определены основные требования к языку спецификаций. Дополнительно к перечисленным в работе [1] к разрабатываемому языку предъявляются приведенные далее требования.

1. Поддержка процедурных и объектно-ориентированных языков. Большинство языков спецификаций ограничены одним классом целевых языков. В разрабатываемом языке необходимо обеспечить поддержку обоих типов языков и различный синтаксис вызовов функций API.

2. Модульность. Современные библиотеки часто достаточно сложные, как следствие, описание их семантики довольно громоздкое. В связи с этим обстоятельством необходимо предусмотреть возможность размещать части описания в отдельных файлах.

3. Наглядность языка. Структура описания библиотеки и синтаксис отдельных конструкций языка должны быть прозрачными и самодокументируемыми.

4. Простота проведения разбора. Грамматика языка должна обеспечивать быстрый разбор спецификации библиотеки стандартными генераторами парсеров и компиляторами компиляторов.

Выполнение этих требований будет залогом простоты применения языка при разработке и интеграции программного обеспечения.

3. Проектирование языка спецификации библиотек

Разрабатываемый предметно-ориентированный язык получил название LibSL — Library Specification Language. В соответствии с формализмом [1] описание библиотеки на языке LibSL состоит из следующих элементов:

- заголовок библиотеки;
- описание импортируемых описаний;
- описание семантических типов;
- описание классов автоматов, соответствующих создаваемым объектам библиотеки;
- создание и инициализация глобальных переменных и объектов;
- описание функций API библиотеки.

Структура описания библиотеки на языке LibSL приведена на листинге 1.

```
library <Название библиотеки>;
// Импорт описаний из внешних файлов
import <Имя файла>;
 ...
// Секция описания семантических типов
types {
 ...
}
// Описания автоматов
automaton <Класс автомата>: <Тип> {
 ...
}
 ...
// Описание функций API
fun <Имя функции>(<Параметры>): <Тип> {
 ...
}
 ...
// Определение и инициализация глобальных переменных и объектов
var <Имя переменной>: <Тип> = <Значение>;
 ...
// Определение и инициализация экземпляра основного автомата библиотеки
int main = new Main(<Начальное состояние>)
```

Листинг 1. Структура спецификации библиотеки на языке LibSL

3.1. Заголовок библиотеки и импорт внешних описаний

Файл спецификации библиотеки начинается с указания имени библиотеки и списка импортируемых внешних спецификаций. Возможность импортирования внешних файлов описаний позволяет создавать модульные спецификации.

3.2. Описание семантических типов

При описании функций библиотеки обычно необходимо не только специфицировать сигнатуру функции API, но также указать семантическую нагрузку каждого параметра и возвращаемого значения. Это означает, что кроме типа языка программирования в сигнатуре функции необходимо также указывать дополнительные данные для возможности проведения корректной семантической интерпретации аргументов. При этом важно вводить семантическую разметку не только для типов аргументов функций, но и для конкретных значений типа этих аргументов. В языке LibSL для этого используются так называемые семантические типы, полностью унаследованные от семантических типов языка PanLang [17]. Для описания типов в языке имеется секция *types*. Помимо описания семантических типов она содержит также описания псевдонимов типов, которые позволяют задавать семантическую эквивалентность встроенных в язык LibSL типов и типов различных языков программирования.

Пример описания нескольких семантических типов и псевдонимов типов приведен на листинге 2, где введены два семантических типа: `PROTOCOL_TYPE` — тип, задающий идентификатор протокола сокета, и представляемый целым числом; `SOCKET_TYPE` — задающий идентификатор типа сокета, причем отдельно специфицирована семантика четырех значений этого типа, в случае если в разных библиотеках конкретные значения могут отличаться.

```
types {
  PROTOCOL_TYPE (int); // Идентификатор протокола сокета
  SOCKET_TYPE (int) {  // Тип создаваемого сокета
    STREAM: 1;       // Потоковый сокет
    DGRAM: 2;        // Дейтаграммный сокет
    RAW: 3;          // Сырой сокет
    SEQPACKET: 5;    // Потоковый пакетный сокет
  };
  SIZE (int) {
     ERROR:  -1;
  }
  SOCKET (int);
  BUFFER (*void);
  LENGTH (int);
  int = int32; // Тип int эквивалентен типу int32
  unsigned = unsigned32;
  byte = unsigned8;
}
```

Листинг 2. Пример секции семантических типов

Дополнительно вместе с типами могут определяться конструкторы типов — правила построения элементов типа. Конструкторы типов являются самостоятельным сложным элементом языка, их описание выходит за рамки настоящей статьи.

3.3. Описание автоматов

Основными сущностями, описывающими функциональные возможности библиотек, являются автоматы. Они задают жизненный цикл самой библиотеки и объектов, создающихся внутри нее. Автоматы состоят (согласно [1]) из *состояний*, *переходов,* инициируемых *функциями переходов,* и *локальных переменных*.

Состояния описываются ключевым словом *state* и уникальным в пределах автомата именем. Если состояние автомата является завершающим, при достижении которого экземпляр автомата уничтожается, то такие состояния задаются ключевым словом *finishstate*.

Переходы автомата задаются ключевым словом *shift*, за которым следуют исходное состояние, новое состояние, функция API, инициирующая срабатывание перехода и при необходимости охраняемое условие. Для сокращения записи в языке имеется несколько синтаксических упрощений:

- для описания переходов, у которых совпадают исходное и новое состояния автомата (петля), новое состояние обозначается ключевым словом *self*;
- в качестве исходного состояния может указываться множество состояний, если вызов метода API валиден во всех состояниях этого множества; для задания всех состояний автомата используется ключевое слово *any*.

Пример разных вариантов описания переходов автомата приведен на листинге 3.

Если охраняемое условие не задано, то оно считается истинным. Если охраняемое условие не выпол-

```
shift Created -> Connected by connect; // Обычный переход
shift Listening -> self by accept; // Петля
shift (One, Two) -> self by get; // get может быть вызвана в любом из двух состояний
shift any -> Closed by close; // close может быть вызвана в любом состоянии
shift Esatblished -> self by read(Mode==mRead); // Переход с охраняемым условием
```

Листинг 3. Примеры задания переходов автомата

```
automaton BSD_SOCKET: int {
  var blocked: boolean; // Локальная переменная автомата
// Состояния автомата
  state Created;
  state Bound;
  state Established;
  state Listening;
  finishstate Closed;
// Переходы автомата
  shift Create->Bound by bind;
  shift Bound->Create by close;
  shift Bound->Listening by listen;
  shift Listening->Bound by close;
  shift Listening->self by accept;
  shift Established->Created by close;
  shift Established -> self by recv;
  shift Established -> self by send;
  shift any->Closed by shutdown;
}

automaton TCP_SOCKET: int {
  state Established;
  finishstate Closed;
  shift Established -> self by recv;
  shift Established -> self by send;
  shift Established -> Closed by shutdown;
}
```

Листинг 4. Пример описания автоматов библиотеки bsd-socket

няется, то переход не происходит, а автомат остается в прежнем состоянии.

Создаваемые в процессе функционирования библиотеки автоматы описываются с помощью конструкции *automaton*, которая задает название класса автомата (по сути — автоматный тип, определяющий целый класс автоматов), тип автоматной переменной, а также содержит определения локальных переменных автомата, состояний автомата и переходов.

На листинге 4 приведен пример, демонстрирующий описание двух автоматов из библиотеки bsd-socket.

3.4. Описание глобальных переменных и объектов

Глобальные переменные — это созданные при инициализации библиотеки переменные или объекты. Тип и значение глобальных переменных необходимы статическому анализатору для решения задачи поиска ошибок в программе, использующей библиотеки. Кроме того, некоторые библиотеки при своей инициализации выполняют какие-либо операции или создают экземпляры объектов. Так, например, библиотеки работы с файлами создают три предопределенных дескриптора стандартных потоков ввода—вывода*: stdin*, *stdout* и *stderr.*

```
var errno: int = 0; //Создание глобальной переменной
var status: int = 1;
var stdin: int = new File(Created, mRead); // Стандартный поток ввода
var stdout: int;
stdout = new  File(Created, mWrite); //Стандартный поток вывода
```

Листинг 5. Примеры объявления и инициализация глобальных объектов

Объявление и инициализация глобальных объектов в языке LibSL осуществляются внутри тела библиотеки (листинг 5).

3.5. Описание функций API

Язык предполагает задание спецификации *видимого поведения* функций API библиотеки и *полного описания* их интерфейса. Интерфейс функции задается расширенной сигнатурой. В расширенную сигнатуру входят:

- имя функции;
- тип объекта, на котором одна вызывается, если используется объектно-ориентированный синтаксис;
- список аргументов функции с указанием типов;
- тип возвращаемого значения.

Все типы, используемые в описании сигнатуры, могут быть как обычными, так и семантическими. В тех случаях, когда аргумент функции API или возвращаемое значение имеет важную смысловую нагрузку, его семантика отображается семантическим типом, задаваемым пользователем.

Для поддержки всех типов библиотек допускаются и процедурный, и объектно-ориентированный синтаксис описания функций API.

Спецификация видимого поведения функции является частичной спецификацией поведения функции и отражает основное предназначение функции, видимое извне. Видимое поведение функции задается в языке LibSL с помощью описания семантических действий, управления внутренними состояниями автоматов и создания/уничтожения экземпляров автоматов.

По аналогии с функциями и методами языков программирования видимое поведение описывается в виде тела функции. Доступ к аргументам функции осуществляется по именам, доступ к результату функции осуществляется с помощью ключевого слова *result*.

Часто вызов функции API библиотеки непосредственно связан с каким-либо экземпляром объекта библиотеки, который отображается в языке LibSL с помощью экземпляра автомата. Переходы именно в этом автомате инициирует вызов функции, и локальные состояния этого автомата она может изменять. Язык предоставляет два способа указания целевого экземпляра автомата при спецификации функций: с помощью аннотирования аргументов и посредством явного задания объекта, отвечающего за контекст экземпляра автомата.

Аннотирование осуществляется символом "@", предваряющим аргумент функции, являющийся автоматным, или класс, на котором вызывается метод в случае, когда специфицируемая библиотека — объектно-ориентированная. При явном задании используется конструкция *target*, параметром которой является адрес экземпляра автомата. Если для функции не определен экземпляр автомата одним из указанных способов, то такая функция не может быть инициатором перехода автомата.

Примеры описания интерфейсов функций API с двумя способами задания экземпляра автомата для процедурной и объектно-ориентированной нотации приведены на листинге 6.

Семантические действия — это механизм явного указания семантики функции API библиотеки с помощью задания семантического домена библиотеки [1, 17]. Семантические действия задаются конструкцией *action*, за которой следуют имя действия и его параметры. Имена действий (по соглашению записываются заглавными буквами) неявно задают или расширяют семантический домен библиотеки. Если действия имеют параметры, то семантический домен расширяется параметризированным действием.

Действия, наряду с созданием экземпляров автоматов и семантическими типами, формируют динамическую семантику библиотеки.

На листинге 7 приведен пример описания параметризируемого действия, выполняемого при вызове методов *recv* и *recvbyte*.

Экземпляры автоматов создаются либо во время вызова функций, либо при инициализации библиотеки. Экземпляр автомата создается с помощью конструкции *new*, параметрами которой являются класс автомата, начальное состояние и начальные значения всех внутренних атрибутов автомата. Синтаксически создание экземпляра автомата похоже на создание экземпляра класса в объектно-ориентированных языках с помощью конструктора. Результирующий экземпляр автомата связывается с переменной (глобальной переменной, параметром функции или внутренним атрибутом автомата), которая может использоваться для доступа к автомату.

Автомат, задающий состояние самой библиотеки, создается аналогичным образом в теле библиотеки вместе с инициализацией прочих глобальных переменных. Для основного автомата библиотеки предусмотрено предопределенное название класса

```
fun recv(@s: SOCKET; buf: BUFFER; len: LENGTH; flags: int): SIZE {
  // Экземпляр автомата - s
  ...
}
fun recv(s: SOCKET; buf: BUFFER; len: LENGTH; flags: int): SIZE {
  target s;  // Экземпляр автомата - s
  ...
}
fun @SOCKET.recv(buf: BUFFER; len: LENGTH; flags: int): SIZE {
  // Экземпляр автомата – класс, на котором вызван метод (this)
  ...
}
fun SOCKET.recv(buf: BUFFER; len: LENGTH; flags: int): SIZE {
  target this.socketID;  // Экземпляр автомата – this.socketID
  ...
}
```

Листинг 6. Примеры аннотирования автоматной переменной

```
fun recv(@s: SOCKET; buf: BUFFER; len: LENGTH; flags: int): SIZE {
  action RECEIVE(s, buf, len);
// При вызове функции recv выполняется семантическое действие RECEIVE,
// заключающееся в получении из сокета s массива длиной len и размещение его в buf
}
fun recvbyte(@s: SOCKET, b: byte, len: MSG_SIZE, FLAGS) : byte {
  action RECEIVE(s, b, 1);
}
// При вызове функции recvbyte выполняется семантическое действие RECEIVE,
// заключающееся в получении из сокета s одного байта и размещение его в b
}
```

Листинг 7. Пример описания действия с параметрами

автомата — *Main*. Примеры создания экземпляров автомата приведены на листинге 8.

Экземпляр автомата может быть уничтожен двумя способами: если сам экземпляр в своем жизненном цикле достигнет завершающего состояния (*fihishstate*), либо если извне будет вызвана конструкция *free*.

Если для функции API задан экземпляр автомата, то эта функция может иметь доступ к внутренним атрибутам этого автомата. Доступ осуществляется по имени, как к полям класса внутри метода в объектно-ориентированных языках. Функция может как читать, так и изменять значения внутренних атрибутов автомата. Для доступа к внутренним атрибутам другого автомата (например, автомата библиотеки) необходимо явно указывать его имя. Примеры обращений к локальным переменным автоматов приведены на листинге 9.

```
fun socket(domain: DOMAIN, type: SOCKET_TYPE, proto: PROTOCOL_TYPE): SOCKET {
  result = new BSD_SOCKET(Created); // Создание нового BSD_сокета
}
fun accept(@s: SOCKET, addr: SOCK_ADDR, addrlen: SOCK_LEN): SOCKET {
  // Создание нового TCP-сокета в состоянии Established
  result = new TCP_SOCKET(Established);
}

var stderr: int = new File(Created, mWrite); // Стандартный поток ошибок
var main: int= new Main(Initialised); //Создание автомата библиотеки
```

Листинг 8. Примеры создания экземпляров автомата

```
mode = rwMode;    // Изменение внутреннего атрибута автомата
main.status = 0;  // Изменение внутреннего атрибута автомата main
```

Листинг 9. Доступ к локальным переменным автоматов

Для описания различных поведений функции API, зависящих от каких-либо условий, в языке предусмотрено два типа ветвлений: *if* и *when*. Примеры использования этих конструкций приведены на листинге 10.

Здесь необходимо явно отметить, что язык является сугубо декларативным, императивная форма записи тела функции с ветвлениями используется только для удобства разработчика и наглядности. Фактически после трансляции условия ветвлений

```
fun send(s: SOCKET, msg: MSG, len: MSG_SIZE, FLAGS) : SIZE {
  if (len>0)
    action SEND(s, msg, len);
  else
    action ERROR("Ошибка параметра");
}
fun fopen(pathname: FILENAME, mode: MODE): FILE {
  when (mode) {
    "r" -> result = new FILE(Created, mRead);
    "r+" -> result = new FILE(Created, mReadWrite);
    "w" -> result = new FILE(Created, mWrite);
    "w+" -> result = new FILE(Created, mReadWriteAppend);
    "a" -> result = new FILE(Created, mAppend);
    "a+" -> result = new FILE(Created, mReadAppend);
    else -> action ERROR();
  }
}
```

Листинг 10. Применение ветвлений в теле функций API

превращаются в логические предикаты, как такового исполнения тела функции не происходит.

Одна и та же функция API может вызываться в разных состояниях автомата. Различие в поведении при этом будет заключаться только в охраняемых условиях, установленных на разных переходах автомата. Если охраняемое условие не выполняется, то функция API не запускается, а автомат остается в исходном состоянии.

4. Применение спроектированного языка

Представленный язык может использоваться для решения нескольких классов задач программной инженерии:

1) аппроксимация поведения библиотечных функций при статическом анализе программного обеспечения;

2) аппроксимация поведения отдельных функций программы для снижения размерности статического анализа сложных программ;

3) обнаружение ошибок протокола использования библиотек;

4) определение совместимости двух библиотек;

5) автоматизированный реинжиниринг программного обеспечения при портировании на новые библиотеки.

В рамках решения первого класса задач функции всех используемых в программном обеспечении библиотек заменяются на их описание на языке LibSL, которое транслируется во внутреннее представление статического анализатора. При этом решается вопрос отсутствия исходных текстов библиотек, а также обеспечивается возможность диагностирования ошибочных ситуаций, происходящих в самих библиотеках. Подобный подход используется, например, в статическом анализаторе Borealis [18].

При статическом анализе сложных программных проектов одним из наиболее ресурсоемких процессов является межпроцедурный анализ. Основной метод борьбы с высокой ресурсоемкостью — использование аппроксимаций отдельных функций. Аппроксимации поведения могут строиться как автоматически с определенными ограничениями (например, с помощью интерполяции Крейга [19]), так и вручную. Во втором случае разработчику необходимы средства описания поведения отдельных функций. Язык LibSL может использоваться как такое средство для решения второго класса задач.

Третий класс задач связан с решением задачи проверки корректности использования библиотеки. Для этого анализатор (статический или динамический) использует семантическую модель библиотеки, построенную на основе LibSL-описания. Семантическая поведенческая модель библиотеки представляет собой конечно-автоматную абстракцию, описывающую всевозможные валидные протоколы доступа к библиотеке [1]. Анализатор использует эту абстракцию для выявления возможных нарушений протокола доступа к библиотеке.

Четвертый класс задач связан с определением совместимости пары библиотек, которая необходима при решении задачи преобразования программы, использующей одну библиотеку в новую программу, использующую альтернативную библиотеку вместо первой. Для корректного решения задачи совместимости необходимо иметь формальные спецификации искомых библиотек, содержащие описание поведения в одном семантическом домене. Язык LibSL может использоваться для описания семантики библиотек и привязки семантики к одному семантическому домену.

Пятый класс задач — портирование программ — одно из ключевых направлений использования представленного языка. Созданные с его помощью формальные спецификации библиотек проходят проверку на совместимость, после чего (в случае успеха) сформированные на их основе модели библиотек используются как исходные данные для алгоритма портирования программного проекта [20].

Заключение

В результате проведения исследования был разработан предметно-ориентированный язык LibSL, предназначенный для спецификации интерфейса и видимого поведения программных библиотек. Язык полностью соответствует созданному ранее формализму и может применяться для спецификации компонентов программного обеспечения при решении целого ряда задач программной инженерии, включая поиск ошибок, портирование, проверку корректности интеграции и ряд других.

Формат и объем статьи не позволяют привести исчерпывающее описание синтаксиса и семантики языка LibSL. За рамками статьи остались детали системы типов, работа с внутренними объектами и т. п. Кроме того, отдельного описания требуют языковые конструкции для правил трансляции семантических доменов различных библиотек, которые являются предметом отдельной публикации.

Дальнейшие исследования в этой области связаны в первую очередь с практической реализацией разработанных концепций и интеграцией разработанного языка в инструментальные средства портирования программного обеспечения на новые библиотеки [20], в средства проверки соблюдения протокола доступа к библиотекам, а также в статические анализаторы, например, такие как Borealis [18].

Список литературы

1. **Ицыксон В. М.** Формализм и языковые инструменты для описания семантики программных библиотек // Моделирование и анализ информационных систем. 2016. Том 23, № 6. С. 754—766. DOI: 10.18255/1818-1015-2016-6-754-766.

2. **Hatcliff J., Leavens G. T., Leino K. R. M., Müller P., Parkinson M.** Behavioral interface specification languages // ACM Comput. Surv. June 2012. Vol. 44, Issue 3. Article 16. 58 p. DOI: 10.1145/2187671.2187678.

3. **Sankar S., Hayes R.** ADL — an interface definition language for specifying and testing software // In Proceedings of

the Workshop on Interface Definition Languages (IDL '94) / Eds. by J. M. Wing, R. L. Wexelblat. ACM, New York, NY, USA, 1994. P. 13—21. DOI: 10.1145/185084.185096.

4. **Leavens G. T., Baker A. L., Ruby Cl.** JML: A Notation for Detailed Design, Behavioral Specifications of Businesses and Systems, Kluwer, 1999, chapter 12, p. 175—188.

5. **Huisman M., Ahrendt W., Bruns D., Hentschel M.** Formal specification with JML. Technical Report. 2014. 51 p.

6. **Baudin P., Filliâtre J. C., Hubert T., Marché C., Monate B., Moy Y., Prevosto V.** ACSL: ANSI/ISO C Specification Language, v1.6, Sept. 2012. URL: http://frama-c.com/acsl.html.

7. **Delahaye M., Kosmatov N., Signoles J.** Common Specification Language for Static and Dynamic Analysis of C Programs // Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC 2013), 2013, Coimbra, Portugal. ACM. 2013. P. 1230—1235.

8. **Cohen E., Dahlweid M., Hillebrand M., et al.** VCC: A Practical System for Verifying Concurrent C // In Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs '09) / Eds by S. Berghofer, T. Nipkow, C. Urban, M. Wenzel. Springer-Verlag, Berlin, Heidelberg, 2009. P. 23—42. DOI: 10.1007/978-3-642-03359-9_2.

9. **Engler D., Chelf B., Chou A., Hallem S.** Checking system rules using system-specific, programmer-written compiler extensions// In Proceedings of the 4th conference on Symposium on Operating System Design & Implementation — (OSDI'00). USENIX Association, Berkeley, CA, USA. 2000. Vol. 4. P. 1—16.

10. **Chelf B., Engler D., Hallem S.** How to write system-specific, static checkers in meta // In Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE '02). ACM, New York, USA, 2002. P. 51—60.

11. **Hallem S., Chelf B., Xie Y., Engler D.** A system and language for building system-specific, static analyses // SIGPLAN Not. 2002. Vol. 37, No. 5. P. 69—82. DOI: 10.1145/543552.512539.

12. **Ball T., Rajamani S. K.** Automatically validating temporal safety properties of interfaces // In Proceedings of the 8th international SPIN workshop on Model checking of software (SPIN '01)/ Eds by M. Dwyer. Springer-Verlag New York, Inc., New York, NY, USA, 2001. P. 103—122.

13. **Ball T., Levin V., Rajamani S. K.** A decade of software model checking with SLAM // Commun. ACM. 2011. Vol. 54, No. 7. P. 68—76. DOI: 10.1145/1965724.1965743.

14. **Ball T., Bounimova E., Levin V., Kumar R., Lichtenberg J.** The Static Driver Verifier Research Platform // Computer Aided Verification — CAV 2010 / Eds by T. Touili, B. Cook., P. Jackson. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg. 2010. Vol. 6174. P. 119—122.

15. **Ball T., Rajamani S. K.** Slic: a Specification Language for Interface Checking (of C). Microsoft Research, January 10, 2002, Technical Report, MSR-TR-2001-21.

16. **Ицыксон В. М., Глухих М. И.** Язык спецификаций поведения программных компонентов // Научно-технические ведомости СПбГПУ. Информатика. Телекоммуникации. Управление. 2010. № 3 (101). С. 63—70.

17. **Ицыксон В. М., Зозуля А. В.** Автоматизированная трансформация программ при миграции на новые библиотеки // Программная инженерия. 2012. № 6. С. 8—14.

18. **Akhin M., Belyaev M., Itsykson V.** Borealis. Bounded Model Checker: The Coming of Age Story // Present and Ulterior Software Engineering / Eds by M. Mazzara, B. Meyer. Springer, Cham, 2017. P. 119—137. DOI: 10.1007/978-3-319-67425-4_8.

19. **Akhin M., Kolton S., Itsykson V.** Random model sampling: Making craig interpolation work when it should not // Automatic Control and Computer Sciences. 2015. Vol. 49, Issue 7. P. 413—419.

20. **Алексюк А. О., Ицыксон В. М.** Семантически-ориентированная миграция Java-программ: опыт практического применения // Моделирование и анализ информационных систем. 2017. Том 24, № 6. С. 677—690. DOI: 10.18255/1818-1015-2017-6-677-690.

LibSL: Language for Specification of Software Libraries

V. M. Itsykson, vlad@icc.spbstu.ru, Peter the Great St. Petersburg Polytechnic University, Saint-Petersburg, 195251, Russian Federation

Corresponding author:

Itsykson Vladimir M., Chair of Computer Systems & Software Engineering department, Peter the Great St. Petersburg Polytechnic University, Saint-Petersburg, 195251, Russian Federation, E-mail: vlad@icc.spbstu.ru

Received on February 04, 2018
Accepted on February 20, 2018

Most of modern software applications widely use third-party components and libraries. It can simplify design and development and can reduce cost and time to market. The main problem of using third-party components is a lack of acceptable documentation. This lack is a cause of low quality of such multicomponent projects and leads to different program defects. The article reviews modern approaches to formal language-based specification of software libraries. Such language-based specifications can be used for automation of some class of software engineering tasks related to multicomponent projects when source code of components is unavailable. Based on formulated requirements syntax and semantics of new language LibSL for partial specification of libraries were developed. LibSL is based on formalism of interacting extended finite state machines which was created by author before. The created language can specify library interface and its visible behavior. Each public function from library API is described by its signature and high-level behavior. Interacted extended finite state machines are defined explicitly by means of states and shifts descriptions.

Such specifications can help automate several actual tasks of software engineering such as defect detection of software which uses external libraries, library access protocol correctness verification, libraries compatibility check, software reengineering etc.

Keywords: *software library, component of software, partial specification, behavior specification language*

For citation:

Itsykson V. M. LibSL: Language for Specification of Software Libraries, *Programmnaya Ingeneria,* 2018, vol. 9, no. 5, pp. 209—220.

DOI: 10.17587/prin.9.209-220

References

1. **Itsykson V. M.** Formalizm i jazykovye instrumenty dlja opisanija semantiki programmnyh bibliotek (The Formalism and Language Tools for Semantics Specication of Software Libraries), *Modelirovanie i analiz informacionnyh system,* 2016, vol. 23, no. 6, pp. 754—766. DOI: 10.18255/1818-1015-2016-6-754-766 (in Russian).
2. **Hatcliff J., Leavens G. T., Leino K. R. M., Muller P., Parkinson M.** Behavioral interface specification languages, *ACM Comput. Surv.,* 2012, vol. 44, issue 3, article 16, 58 p. DOI: 10.1145/2187671.2187678.
3. **Sankar S., Hayes R.** ADL — an interface definition language for specifying and testing software, *Proceedings of the Workshop on Interface Definition Languages* (IDL '94) / Eds by J. M. Wing, R. L. Wexelblat. ACM, New York, NY, USA, 1994, pp. 13—21. DOI: 10.1145/185084.185096.
4. **Leavens G. T., Baker F. L., Ruby C.** *JML: A Notation for Detailed Design, Behavioral Specifications of Businesses and Systems*, Kluwer, 1999, chapter 12, pp. 175—188.
5. **Huisman M., Ahrendt W., Bruns D., Hentschel M.** Formal specification with JML. Technical Report. 2014. 51 p.
6. **Baudin P., Filliatre J. C., Hubert T., Marche C., Monate B., Moy Y., Prevosto V.** ACSL: ANSI/ISO C Specification Language, v1.6, Sept. 2012, available at: http://frama-c.com/acsl.html.
7. **Delahaye M., Kosmatov N., Signoles J.** Common Specification Language for Static and Dynamic Analysis of C Programs, *Proceedings of the 28th Annual ACM Symposium on Applied Computing SAC 2013 — 28th ACM Symposium on Applied Computing*, 2013, Coimbra, Portugal. ACM, 2013, vol. 2, pp. 1230—1235.
8. **Cohen E., Dahlweid M., Hillebrand M., Leinenbach D., Moskal M., Santen T., Schulte W., Tobies S.** VCC: A Practical System for Verifying Concurrent C., *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics* (TPHOLs '09) / Eds by S.Berghofer, T. Nipkow, C. Urban, M. Wenzel, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 23—42. DOI: 10.1007/978-3-642-03359-9_2
9. **Engler D., Chelf B., Chou A., Hallem S.** Checking system rules using system-specific, programmer-written compiler extensions, *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation — Volume 4* (OSDI'00), USENIX Association, Berkeley, CA, USA. 2000, vol. 4, pp. 1—16.
10. **Chelf B., Engler D., Hallem S.** How to write system-specific, static checkers in metal, *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (PASTE '02). ACM, New York, NY, USA, 2002, pp. 51—60.
11. **Hallem S., Chelf B., Xie Y., Engler D.** A system and language for building system-specific, static analyses, *SIGPLAN Not.,* 2002, vol. 37, no. 5, pp. 69—82. DOI: 10.1145/543552.512539.
12. **Ball T., Rajamani S. K.** Automatically validating temporal safety properties of interfaces, *Proceedings of the 8th international SPIN workshop on Model checking of software* (SPIN '01) / Eds by M. Dwyer, Springer-Verlag New York, Inc., NY, USA, 2001, pp. 103—122.
13. **Ball T., Levin V., Rajamani S. K.** A decade of software model checking with SLAM, *Commun. ACM,* 2011, vol. 54, no. 7, pp. 68—76. DOI: 10.1145/1965724.1965743.
14. **Ball T., Bounimova E., Levin V., Kumar R., Lichtenberg J.** The Static Driver Verifier Research Platform, *Computer Aided Verification — CAV 2010 /* Eds by T. Touili, B. Cook., P. Jackson. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg. 2010, vol. 6174, pp. 119—122.
15. **Ball T., Rajamani S. K.** Slic: a Specification Language for Interface Checking (of C). Microsoft Research, January 10, 2002, Technical Report, MSR-TR-2001-21.
16. **Itsykson V., Glukhikh M.** Jazyk specifikacij povedenija programmnyh komponentov (A program component behavior specification language), *Nauchno-tehnicheskie vedomosti SPbGPU. Informatika. Telekommunikacii. Upravlenie*, 2010, vol. 3 (101), pp. 73—70 (in Russian).
17. **Itsykson V. M., Zozulia A. V.** Avtomatizirovannaja transformacija programm pri migracii na novye biblioteki (Automated software transformation when migrating to new libraries**)**, *Programmnaya Ingeneria,* 2012, no. 6, pp. 8—14 (in Russian).
18. **Akhin M., Belyaev M., Itsykson V.** Borealis. Bounded Model Checker: The Coming of Age Story, *Present and Ulterior Software Engineering* / Eds by M. Mazzara, B. Meyer. Springer, Cham, 2017, pp. 119—137. DOI: 10.1007/978-3-319-67425-4_8.
19. **Akhin M., Kolton S., Itsykson V.** Random model sampling: Making craig interpolation work when it should not, *Automatic Control and Computer Sciences*, 2015, vol. 49, issue 7, pp. 413—419.
20. **Aleksyuk A. O., Itsykson V. M**. Semanticheski-orientirovannaja migracija Java-programm: opyt prakticheskogo primenenija (Semantics-Driven Migration of Java Programs: a Practical Experience), *Modelirovanie i analiz informacionnyh sistem*, 2017, vol. 24, no. 6, pp. 677—690. DOI: 10.18255/1818-1015-2017-6-677-690 (in Russian).

Продолжается подписка на журнал "Программная инженерия" на второе полугодие 2018 г.

Обращаем внимание, что во втором полугодии журнал выйдет 3 раза — в августе, октябре и декабре

Оформить подписку можно через подписные агентства или непосредственно в редакции журнала.

Подписные индексы по каталогам:

Роспечать — 22765; Пресса России — 39795

Адрес редакции: 107076, Москва, Стромынский пер., д. 4,
Издательство "Новые технологии",
редакция журнала "Программная инженерия"

Тел.: (499) 269-53-97. Факс: (499) 269-55-10. E-mail: prin@novtex.ru

УДК 004.056.55 DOI: 10.17587/prin.9.221-227

С. О. Супрунюк, аспирант, email: sofia-92@mail.ru, **Е. А. Курганов,** аспирант, e-mail: kuev@yandex.ru, МГУ имени М. В. Ломоносова

О глубине аппаратной реализации потокового шифра ZUC

Исследуется глубина аппаратной (схемной) реализации потокового шифра ZUC. Рассматриваются также способы ее минимизации, после чего приводится сравнение полученной реализации шифра с результатами, представленными в публикациях других исследователей.

Ключевые слова: *аппаратная реализация, оптимизация глубины схем, потоковые шифры, шифр ZUC, регистр сдвига с обратной линейной связью, поле*

Введение

В настоящее время для создания цифровых устройств широко используют интегральные схемы. Сферами их применения обычно являются цифровая обработка сигналов, цифровая видео-аудиоаппаратура, высокоскоростная передача данных, криптография. Это, в частности, связано с тем обстоятельством, что программные реализации алгоритмов не всегда могут обеспечить необходимую скорость обработки.

В настоящей работе рассмотрен потоковый шифр ZUC. Он был разработан в центре DACAS (*Data Assurance and Communication Security Research Center*) Китайской академии наук. Этот шифр лежит в основе мобильных 3GPP-стандартов 128-EEA3 (для шифрования) и 128-EIA3 (для контроля целостности) [1, 2]. Шифр ZUC входит в состав стандарта мобильной связи четвертого поколения LTE (*Long Term Evolution*).

В настоящей работе речь пойдет об аппаратной реализации шифра ZUC. Определяющим параметром производительности таких реализаций является глубина схемы. Под глубиной понимается длина максимального простого пути схемы. Рассмотрен базис из элементов конъюнкции, дизъюнкции, отрицания и задержки. При этом отрицание игнорируется при вычислении глубины (вычисление проводится по тем же правилам, что и в работе [3]).

В начале статьи дано подробное описание шифра ZUC. Далее приведены оценки глубины преобразований, используемых в шифре, после чего описана самая простая реализация шифра, приведена ее глубина. Затем дан обзор существующих на данный момент исследований. В заключительной части статьи рассмотрены некоторые оптимизирующие по глубине преобразования и приведена итоговая реализация шифра. Полученные результаты кратко изложены в работе [4].

Алгоритм ZUC

Шифр ZUC принимает на вход ключ шириной 128 бит и вектор инициализации (IV) шириной 128 бит. Алгоритм состоит из трех уровней: регистра сдвига с линейной обратной связью (LFSR, *Linear Feedback Shift Register*), реорганизации бит (BR, *Bit-Reorganization*) и нелинейной функции F (рис. 1). Алгоритм работает в двух режимах — рабочем и режиме инициализации. Каждый такт в рабочем режиме шифр генерирует 32 бита выходного потока.

Далее опишем более подробно, как устроен каждый из отмеченных уровней.

LSFR: в алгоритме ZUC используется один регистр сдвига с линейной обратной связью, который состоит из 16 31-битных ячеек $s_0, ..., s_{15}$, содержимое которых интерпретируется как число в диапазоне $\{1, ..., 2^{31} - 1\}$ (важно, что содержимое ни одной из ячеек не может быть равно нулю). Работу данного регистра в режиме инициализации и в рабочем режиме описывают представленные далее Алгоритм 1 и Алгоритм 2 соответственно.

Алгоритм 1. Работа LFSR в режиме инициализации

Шаг 1. $v := 2^{15}s_{15} + 2^{17}s_{13} + 2^{21}s_{10} + 2^{20}s_4 + 2^8 s_0 + s_0 \bmod(2^{31} - 1)$.

Шаг 2. $s_{16} := (v + u) \bmod(2^{31} - 1)$; /*$u$ берется с выхода F*/

Шаг 3. if $s_{16} = 0$ **then**

Шаг 4. $s_{16} \leftarrow 2^{31} - 1$.

Шаг 5. $(s_1, s_2, ..., s_{15}, s_{16}) \rightarrow (s_0, s_1, ..., s_{14}, s_{15})$.

Алгоритм 2. Работа LFSR в рабочем режиме

Шаг 1. $v := 2^{15}s_{15} + 2^{17}s_{13} + 2^{21}s_{10} + 2^{20}s_4 + 2^8 s_0 + s_0 \bmod(2^{31} - 1)$.

Шаг 2. if $s_{16} = 0$ **then**

Шаг 3. $s_{16} \leftarrow 2^{31} - 1$.

Шаг 4. $(s_1, s_2, ..., s_{15}, s_{16}) \leftarrow (s_0, s_1, ..., s_{14}, s_{15})$.

Число $2^{31} - 1$ является простым. Следовательно, кольцо вычетов по модулю данного числа является полем. Следует отметить, что в данном поле умножение на 2^n — это циклический сдвиг влево на n позиций [1, 2].

BR: на данном уровне из LFSR извлекаются клетки $s_0, s_2, s_5, s_7, s_9, s_{11}, s_{14}, s_{15}$, на основе которых формируются четыре слова X_0, X_1, X_2, X_3 шириной 32 бита по следующему правилу: $X_0 = s_{15H} \parallel s_{14L}$,

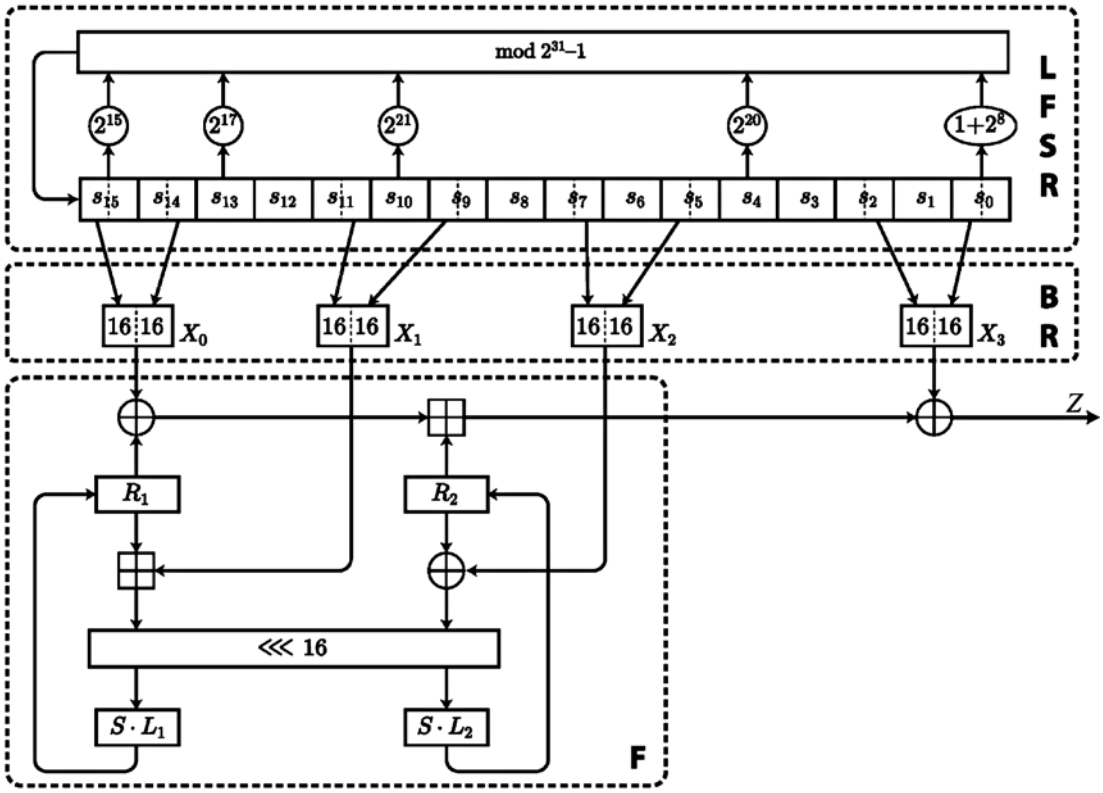


Рис. 1. Устройство шифра ZUC

$X_1 = s_{11L} \| s_{9H}$, $X_2 = s_{7L} \| s_{5H}$, $X_3 = s_{2L} \| s_{0H}$, где s_{iH} означает биты 30, ..., 15, s_{iL} — биты 15, ..., 0 клетки s_i, а $\|$ означает конкатенацию.

Три первых слова поступают на вход нелинейной функции F, а последнее используется для формирования выходного потока.

Нелинейная функция F: в процессе вычисления данной функции используются две 32-битных клетки памяти (R_1, R_2); один S-блок 32×32 бита (S), который состоит из четырех S-блоков 8×8 бит, два линейных преобразования (L_1, L_2) и три входных слова шириной 32 бита (X_0, X_1, X_2), полученных на уровне BR. Выход данной функции — 32-битное слово W. Поток 32-битных выходных слов Z получается побитовым сложением со словом X_3: $Z = W \oplus X_3$. Функция F определяется следующим образом:

$F(X_0, X_1, X_2)\{$

1) $W = (X_0 \oplus R_1) \boxplus R_2$;
2) $W_1 = R_1 \boxplus X_1$;
3) $W_2 = R_2 \oplus X_2$;
4) $R_1 = S(L_1(W_{1L} \| W_{2H}))$;
5) $R_2 = S(L_2(W_{2L} \| W_{1H}))$; },

где $\boxplus$ означает сложение по модулю 2^{32}.

В начале работы алгоритма на основе ключа и вектора инициализации IV заполняются ячейки LFSR s_0, ..., s_{15}. Для этого ключ представляется в виде $k = k_0 \| ... \| k_{15}$, где каждый k_i имеет ширину 8 бит. Аналогично представляется вектор инициализации $iv = iv_0 \| ... \| iv_{15}$. На данном этапе используется также константа D шириной 240 бит, которая тоже представляется как конкатенация шестнадцати строк шириной 15 бит: $D = d_0 \| ... \| d_{15}$. Значения констант d_0, ..., d_{15} указаны ниже.

d_0 100010011010111
d_1 010011010111100
d_2 110001001101011
d_3 001001101011110
d_4 101011110001001
d_5 011010111100010
d_6 111000100110101
d_7 000100110101111
d_8 100110101111000
d_9 010111100010011
d_{10} 110101111000100
d_{11} 001101011110001
d_{12} 101111000100110
d_{13} 011110001001101
d_{14} 111100010011010
d_{15} 100011110101100

Значения ячеек LFSR получаются следующим образом: $s_i = k_i \| d_i \| iv_i$, $i = 0, ..., 15$.

Работа алгоритма после инициализации ячеек LFSR может быть описана последовательным выполнением представленных ниже алгоритмов 3, 4 и 5.

Алгоритм 3. Стадия инициализации алгоритма ZUC

Шаг 1. $ctr = 0$;
Шаг 2. **repeat**
Шаг 3. Вычислить результат BR.
Шаг 4. Вычислить значение функции F, используя слова X_0, X_1, X_2, полученные на предыдущем шаге.
Шаг 5. Запустить Алгоритм 1.
Шаг 6. $ctr \leftarrow ctr + 1$.
Шаг 7. **until** $ctr = 32$.

Алгоритм 4. Первая итерация алгоритма ZUC в рабочем режиме

Шаг 1. Вычислить результат BR.
Шаг 2. Вычислить значение функции F, используя слова X_0, X_1, X_2, полученные на предыдущем шаге.
Шаг 3. Проигнорировать выход W функции F.
Шаг 4. Запустить Алгоритм 2.

Алгоритм 5. Работа алгоритма ZUC в рабочем режиме

Шаг 1. **repeat**
Шаг 2. Вычислить результат BR.
Шаг 3. Вычислить значение функции F, используя слова X_0, X_1, X_2, полученные на предыдущем шаге.
Шаг 4. Вычислить выходной поток $Z = W \oplus X_3$.
Шаг 5. Запустить Алгоритм 2.
Шаг 6. $ctr \leftarrow ctr + 1$.
Шаг 7. **until** будет выработано столько слов ключевого потока, сколько требуется.

Простая реализация и оценка ее глубины

Прежде чем привести самую простую реализацию алгоритма ZUC, оценим глубину каждого нетривиального преобразования, используемого в нем.

Сложение по модулю $2^{31} - 1$. Существует несколько реализаций данного сумматора. В плане глубины лучшей является реализация, представленная в работе [5].

Убедимся, что указанный сумматор вычисляет сумму по модулю $2^{31} - 1$ с глубиной 13. Приведем данную конструкцию. Пусть $A = \{a_{30}, a_{29}, ..., a_0\}$, $B = \{b_{30}, b_{29}, ..., b_0\}$ и $Z = \{z_{30}, z_{29}, ..., z_0\}$ — сумма A и B по модулю $2^{31} - 1$. Пусть также $g_k = a_k b_k$, $p_k = a_k \vee b_k$ и $h_k = a_k \oplus b_k$ для $k = 0, 1, ..., 30$. Определим оператор $\circ$ следующим образом: $(g, p) \circ (g', p') = (g \vee pg', pp')$. Пусть

$$(G_k, P_k) = (g_k, p_k) \circ (g_{k-1}, p_{k-1}) \circ ... \\ ... \circ (g_0, p_0) \circ (g_{30}, p_{30}) \circ ... \circ (g_{k+1}, p_{k+1}). \quad (1)$$

Тогда $z_k = h_k \oplus c_{k-1}$, $k = 0, 1, ..., 30$, где $c_k = G_k$, причем $c_{-1} = c_{30}$.

Заметим, что

$$\{(g_2, p_2) \circ (g_1, p_1)\} \circ \{(g_1, p_1) \circ (g_0, p_0)\} \Leftrightarrow \\ \Leftrightarrow \{(g_2, p_2) \circ (g_1, p_1) \circ (g_0, p_0)\}.$$

Так на первом уровне можно вычислить все

$$\{(g_{30}, p_{30}) \circ (g_{29}, p_{29})\}, ..., \{(g_1, p_1) \circ (g_0, p_0)\}, \\ \{(g_0, p_0) \circ (g_{30}, p_{30})\}.$$

На втором уровне вычисляются композиции из четырех пар и так далее. На пятом уровне вычисляются все (G_k, P_k). Каждый уровень имеет глубину 2, также все g_k и p_k вычисляются с глубиной 1. И при вычислении z_k глубина увеличивается еще на 2.

Таким образом, глубина сложения $D(\mathrm{mod}(2^{31} - 1)) \leqslant$ $\leqslant 13$. Утверждение доказано.

Сложение по модулю 2^{32}. Для того чтобы вычислить сумму двух 32-битных векторов по модулю 2^{32}, можно воспользоваться методом золотого сечения, описанным в работе [6]. Тогда данную операцию можно реализовать с глубиной 11.

***S*-блок.** S-блок алгоритма ZUC размером 32×32 бита состоит из четырех S-блоков размером 8×8 бит, т. е. $S = (S_0, S_1, S_2, S_3)$, причем $S_0 = S_2$, $S_1 = S_3$. Описание данных S-блоков можно найти в работах [1, 2].

Любой S-блок размером 8×8 бит можно реализовать с глубиной 10, воспользовавшись совершенной дизъюнктивной нормальной формой (СДНФ) [7]. Подробнее об этом можно прочитать в работе [8].

Функция *F*. Линейные преобразования L_1 и L_2 имеют вид $L_1(X) = X \oplus (X \lll_{32} 2) \oplus (X \lll_{32} 10) \oplus (X \lll_{32} 18) \oplus$ $\oplus (X \lll_{32} 24)$, $L_2(X) = X \oplus (X \lll_{32} 8) \oplus (X \lll_{32} 14) \oplus$ $\oplus (X \lll_{32} 22) \oplus (X \lll_{32} 30)$. Их глубина $D(L_i) \leqslant 6$.

Нетрудно заметить, что максимальную глубину в функции F имеет путь

$$R_1 \rightarrow \boxplus \rightarrow (\lll 16) \rightarrow SL_1 \rightarrow \mathrm{mux} \rightarrow R_1,$$

где mux обозначает мультиплексор, который осуществляет выбор начальных или старых данных (рис. 2). Следовательно, глубина функции F: $D(F) \leqslant D(L_i(\boxplus)) + D(S) + 1$. Таким образом, $D(F) \leqslant$ $\leqslant 11 + 6 + 10 + 1 = 28$.

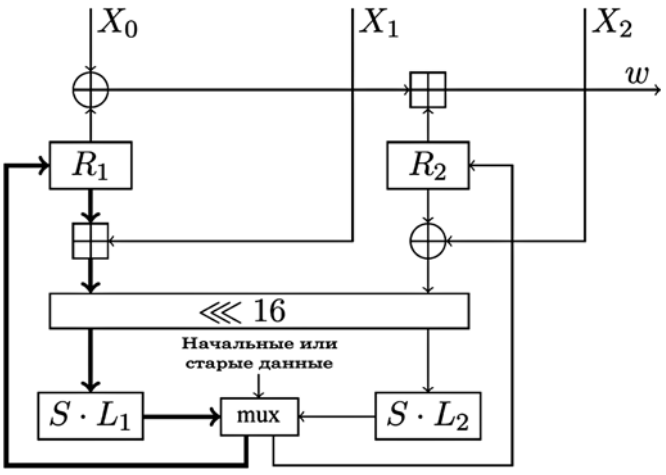


Рис. 2. Нелинейная функция *F*

LFSR. Заметим, что в LFSR присутствует шесть или семь слагаемых, в зависимости от режима, в котором работает алгоритм.

Непосредственной проверкой можно убедиться, что глубину сложения можно уменьшить, сделав из трех слагаемых два следующим образом:

$A_1 + A_2 + A_3 \bmod (2^{31} - 1) = A + B \bmod(2^{31} - 1)$, где $A = A_1 \oplus A_2 \oplus A_3$, $B = \{b_{30}, b_{29}, ..., b_0\}$, $b_i = \mathrm{Maj}(a_{1,\,i-1}, a_{2,\,i-1}, a_{3,\,i-1}) := a_{1,\,i-1}a_{2,\,i-1} \vee a_{2,\,i-1}a_{3,\,i-1} \vee a_{1,\,i-1}a_{3,\,i-1}$, причем $a_{j,\,-1} = a_{j,\,30}$, $j = 1, 2, 3$.

При этом A вычисляется с глубиной 4, B — с глубиной 3, следовательно, глубина данной функции меньше, чем глубина суммы двух слагаемых по модулю $2^{31} - 1$.

Обозначим через $D(A_i)$, i=1, 2, 3, глубину вычисления A_i. Заметим, что если аргументы рассматриваемой функции таковы, что $D(A_2) < D(A_1)$ и $D(A_3) < D(A_1)$, то A можно вычислить с глубиной $D(A) \leqslant D(A_1)+3$, B — с глубиной $D(B) \leqslant D(A_1)+2$. Если же $D(A_2) < D(A_1)-1$ и $D(A_3) < D(A_1)-1$, то и A, и B можно вычислить с глубиной $D(A_1)+2$.

Следует также отметить, что если в данном преобразовании одно из слагаемых является константой, то A вычисляется на глубине 2, а B — на глубине 1. Действительно, пусть $A_3 = \mathrm{const}$. Тогда $a_i = a_{1,\,i} \oplus a_{2,\,i}$, если $a_{3,\,i} = 0$ и $a_i = \overline{a_{1,i} \oplus a_{2,i}}$ иначе, $i = 0, 1, ..., 30$. Аналогично, $b_i = b_{1,\,i}b_{2,\,i}$ или $b_i = b_{1,\,i} \vee b_{2,\,i}$, $i = 0, 1, ..., 30$.

Для того чтобы при $s_{16} = 0$ сделать $s_{16} = 2^{31} - 1$, можно воспользоваться функцией

$$g = \overline{s_{16,\,30}} \,\&\, \overline{s_{16,\,29}} \,\&\, ... \overline{s_{16,\,0}},$$

которая равна 1 тогда и только тогда, когда $s_{16} = 0$. Глубина данной функции равна 5.

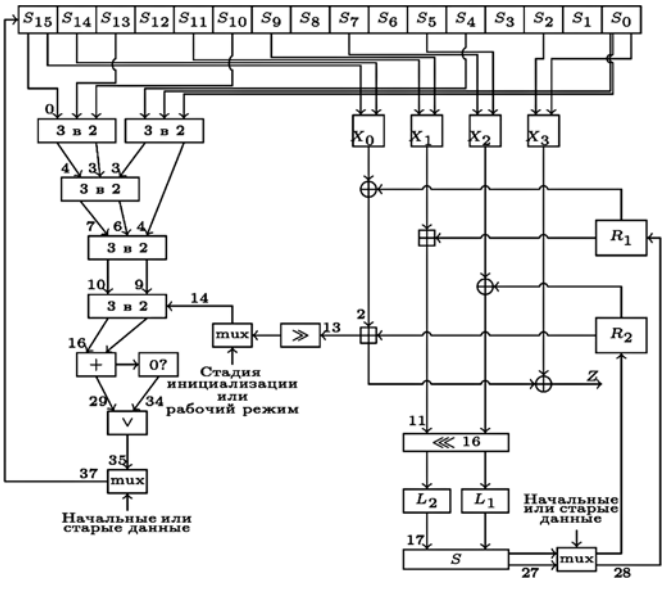


Рис. 3. Простая реализация шифра ZUC

Теперь чтобы в случае, когда $s_{16} = 0$, определить $s_{16} = 2^{31} - 1$, нужно каждый получившийся после сложения в LFSR бит $s_{16,\,i}$ заменить на $s_{16,\,i} \vee g$, $i = 0, 1, ..., 30$.

Общая глубина. Используя полученные выше оценки, можно получить общую реализацию с глубиной 37 (рис. 3). В данной реализации используются три мультиплексора. Два из них выбирают между входными данными и нулем, поэтому реализуются на глубине 1. Последний мультиплексор осуществляет выбор между двумя ненулевыми числами, поэтому реализуется на глубине 2.

Существующие исследования

В настоящее время ведутся исследования по оптимизации аппаратной реализации алгоритма ZUC. Среди множества работ по данной теме можно выделить работу [9]. В данной работе представлена реализация ZUC, для которой глубина одного раунда составляет 31.

Оптимизация по глубине

Покажем несколько преобразований, которые помогут уменьшить итоговую глубину реализации.

Функция F. Заметим, что глубину вычисления выражения $L_1(A \boxplus B)$ можно сократить на 2. Запишем его в следующем виде:

$$L_1(A \boxplus B) = L_1(A \oplus B \oplus C) = L_1(A \oplus B) \oplus L_1(C),$$

где $C = \{c_0, c_1, ..., c_{31}\}$ — биты переноса из сложения по модулю 2^{32}. Глубина нахождения c_{31} равна 9 [6], $L_1(A \oplus B)$ считается с глубиной 8. Перепишем более детально выражение для $L_1(A \boxplus B)$:

$$L_1(A \boxplus B) = L_1(A \oplus B) \oplus C \oplus (C \lll_{32} 2) \oplus$$
$$\oplus (C \lll_{32} 10) \oplus (C \lll_{32} 18) \oplus (C \lll_{32} 24).$$

Нетрудно заметить, что данное выражение считается на глубине 15.

***S*-блок.** Каждый S-блок 8×8 бит можно рассматривать как восемь булевых функций от восьми переменных (i-й выходной бит S-блока является значением i-й функции, i = 1, 2, ..., 8). Для каждой из этих восьми функций можно вычислить минимальную ДНФ (см., например, [10]). Посчитав для каждой булевой функции S-блоков S_0, S_1 минимальную ДНФ, нетрудно убедиться, что преобразование S можно реализовать с глубиной 9. Действительно, для всех минимальных ДНФ, полученных для S-блоков S_0, S_1, максимальное число дизъюнктов составляет 49 (дизъюнкция 49 элементов считается на глубине 6). В свою очередь, каждый дизъюнкт является конъюнкцией не более чем 8 элементов (считается на глубине 3). После применения данных преобразований глубина максимального пути в функции F составляет 25.

Сложение по модулю $2^{31} - 1$. Так как ноль в поле $GF(2^{31} - 1)$ может быть представлен с помощью 31 бита как 0 и как $2^{31} - 1$, сумматор может представить ноль, полученный после сложения двух чисел по модулю $2^{31} - 1$, любым из этих способов. Докажем два утверждения, которые потребуются при оптимизации ZUC по глубине.

Утверждение 1. *Выход схемы сумматора по модулю* $2^{31} - 1$, *представленного в работе* [5], *равен нулю тогда и только тогда, когда оба слагаемых равны нулю.*

Доказательство. Пусть как прежде, A, B и Z — 31-битные числа, A и B являются входами, а Z — выходом сумматора. Очевидно, что если $A = B = 0$, то $Z = 0$. Это вытекает из того, что в этом случае $g_k = p_k = 0$ для всех $k = 0, 1, ..., 30$, поэтому из выражения (1) следует, что все $c_k = G_k$ также равны нулю. Значит, $z_k = a_k \oplus b_k \oplus c_{k-1}$ для всех $k = 0, 1, ..., 30$, что равносильно тому, что $Z = 0$.

Докажем утверждение в обратную сторону. Пусть $Z = 0$, т. е. $z_k = h_k \oplus c_{k-1} = 0$ для всех $k = 0, 1, ..., 30$. Тогда $A + B = 0 \bmod(2^{31} - 1)$, что верно в одном из трех случаев: $A = B = 2^{31} - 1$, $A + B = 2^{31} - 1$ или $A = B = 0$.

Пусть $A = B = 2^{31} - 1$. Это условие равносильно тому, что $a_k = b_k = 1$ для всех $k = 0, 1, ..., 30$, следовательно, $g_k = p_k = 1$, $k = 0, 1, ..., 30$. Поэтому из выражения (1) все $c_k = G_k$ также равны 1. Значит, $z_k = a_k \oplus b_k \oplus c_{k-1} = 1$ для всех $k = 0, 1, ..., 30$, что противоречит предположению.

Рассмотрим второй случай, когда $A + B = 2^{31} - 1$. Заметим, что в этом случае $A \neq B$, так как $2^{31} - 1$ — нечетное число. Следовательно, существует целое n, $0 \leqslant n \leqslant 30$, такое, что $a_n \neq b_n$, что равносильно тому, что $h_n = 1$, из чего вытекает, что $g_n = 0$ и $p_n = 1$. Можно также заметить, что $c_{n-1} = 1$, так как $z_n = h_n \oplus c_{n-1} = 0$. Из того, что $g_n = 0$ $p_n = 1$ следует, что

$$(G_{n-1},\ P_{n-1}) = (g_{n-1},\ p_{n-1}) \circ (g_{n-2},\ p_{n-2}) \circ ...$$
$$... \circ (g_0,\ p_0) \circ (g_{30},\ p_{30}) \circ ... \circ (g_{n+1},\ p_{n+1}) \circ (g_n,\ p_n) =$$
$$= (g_{n-1},\ p_{n-1}) \circ (g_{n-2},\ p_{n-2}) \circ ... \circ (g_0,\ p_0) \circ$$
$$\circ (g_{30}, p_{30}) \circ ... \circ (g_{n+1},\ p_{n+1}),$$

$$(G_n,\ P_n) = (g_n,\ p_n) \circ (G_{n-1},\ P_{n-1}).$$

Так как $c_{n-1} = G_{n-1} = 1$, получаем, что $c_n = G_n = g_n \vee p_n G_{n-1} = g_n \vee p_n = 1$. Из этого следует, что $h_{n+1} = 1$, так как $z_{n+1} = h_{n+1} \oplus c_n = 0$, что означает, что $g_{n+1} = 0$ и $p_{n+1} = 1$. Таким образом, мы доказали, что если $h_n = 1$, $g_n = 0$ и $c_{n-1} = 1$, то $h_{n+1} = 1$, $g_{n+1} = 0$ и $c_n = 1$. Действуя аналогично по циклу, приходим к выводу, что $c_k = 1$, $g_k = 0$ для всех $k = 0, 1, ... 30$. Это, очевидно, противоречит выражению (1), так как если все g_k равны нулю, то все $G_k = c_k$ также равны нулю.

Остается только случай, когда $A = B = 0$. Утверждение доказано.

Утверждение 2. *Во время работы алгоритма ZUC в его реализации всегда хотя бы одно из слагаемых сумматора по модулю* $2^{31} - 1$ *не равно нулю.*

Доказательство. Для начала рассмотрим преобразование 3 в 2. Легко увидеть, что оба выхода одновременно равны нулю тогда и только тогда, когда все три входа равны нулю.

Действительно, $b_i = 0 \Leftrightarrow$ только одно $a_{k_i} = 1$ или $a_{1_i} = a_{2_i} = a_{3_i} = 0$, $k = 1, 2, 3$, $i = 0, 1, ..., 30$. Если только одно $a_{k_i} = 1$, то $a_i = 1$. Значит, все $a_{k_i} = 0$.

Значит, оба слагаемых в сумматоре по модулю $2^{31} - 1$ равны нулю тогда и только тогда, когда все три слагаемых в преобразовании 3 в 2 равны нулю.

Ясно, что на первом такте работы алгоритма сумма $2^{15}s_{15} + 2^{17}s_{13} + 2^{21}s_{10} + 2^{20}s_4 + 2^8 s_0 + s_0 \bmod(2^{31} - 1)$ не будет равна нулю, так как изначально в ячейках $s_0, ..., s_{15}$ находятся ненулевые значения (так как константа D не равна нулю).

Поэтому новое значение ячейки s_{15} на первом такте будет отлично от нуля. Следовательно, на втором и всех последующих тактах ситуация будет аналогичной — в ячейках $s_0, ..., s_{15}$ всегда будут находиться ненулевые значения. Утверждение доказано.

Эти утверждения позволяют исключить проверку равенства нулю после сложения по модулю $2^{31} - 1$.

Дополнительные преобразования. Сумму двух чисел A и B по модулю 2^{32} можно представить следующим образом:

$$(A \boxplus B) \gg 1 \equiv (A \gg 1) + (B \gg 1) + \epsilon \bmod(2^{31} - 1),$$

где A, B — 32-битовые числа, $\epsilon \in \{-1, 0, 1\}$.

В этом нетрудно убедиться. Рассмотрим все возможные случаи. Пусть c_{32} и c_1 — биты переноса из сумматора по модулю 2^{32}. Тогда:

- если $c_{32} = 0$, $c_1 = 0$, то

$$(A \boxplus B) \gg 1 = (A \gg 1) + (B \gg 1) \bmod 2^{31} =$$
$$= (A \gg 1) + (B \gg 1) \bmod(2^{31} - 1);$$

- если $c_{32} = 0$, $c_1 = 1$, то

$$(A \boxplus B) \gg 1 = (A \gg 1) + (B \gg 1) + 1 \bmod 2^{31} =$$
$$= (A \gg 1) + (B \gg 1) + 1 \bmod(2^{31} - 1);$$

- если $c_{32} = 1$, $c_1 = 0$, то

$$(A \boxplus B) \gg 1 = (A \gg 1) + (B \gg 1) - 1 \bmod(2^{31} - 1),$$

так если $c_{32} = 1$, то $A + B \bmod(2^{31} - 1) = A + B \bmod(2^{31} - 1)$.

- если $c_{32} = 1$, $c_1 = 1$, то

$$(A \boxplus B) \gg 1 = (A \gg 1) + (B \gg 1) + 1 - 1 \bmod(2^{31} - 1) =$$
$$= (A \gg 1) + (B \gg 1) \bmod(2^{31} - 1).$$

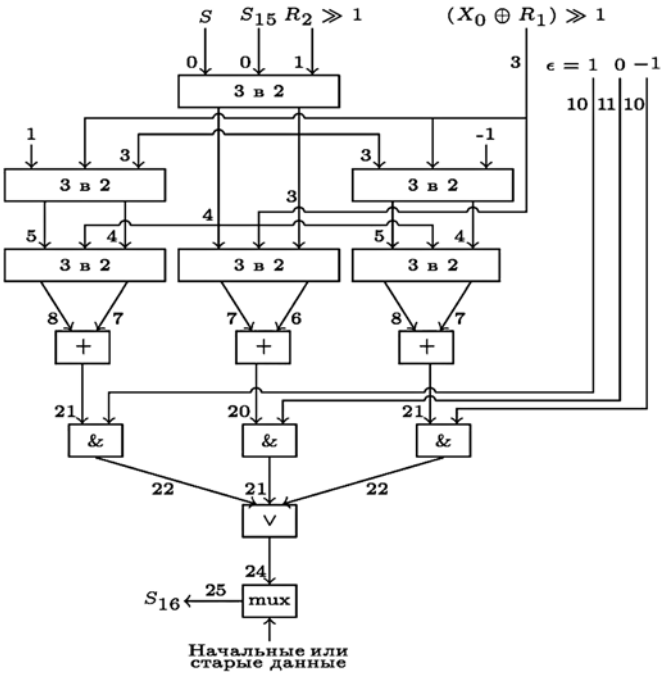


Рис. 4. Конечная реализация LFSR в шифре ZUC

Определим функцию

$$h_k = \begin{cases} 1, & \text{если } \epsilon = k \\ 0, & \text{иначе.} \end{cases}$$

Более подробно:

- $\epsilon = -1$, если $c_{32}=1$, $c_1=0 \Rightarrow h_{-1}=c_{32} \,\&\, \overline{c_1}$;
- $\epsilon = 0$, если $c_{32} = 0$, $c_1 = 0$ или $c_{32} = 1$, $c_1=1 \Rightarrow h_0=c_{32} \,\&\, c_1 \vee \overline{c_{32}} \,\&\, \overline{c_1}$;
- $\epsilon = 1$, если $c_{32} = 0$, $c_1=1 \Rightarrow h_1=\overline{c_{32}} \,\&\, c_1$.

Глубина вычислений c_{32} и c_1 равна 9 и 1 соответственно. Следовательно, глубина вычисления h_{-1}, h_1 равна 10, глубина вычисления h_0 — 11.

Обозначим $S = 2^{15}s_{15} + 2^{17}s_{13} + 2^{21}s_{10} + 2^{20}s_4 + + 2^8 s_0 + s_0 \bmod(2^{31} - 1)$. Данную сумму можно считать каждый такт работы алгоритма, чтобы на следующий такт пользоваться результатом S. Глубина вычисления данной суммы составляет 22.

Нетрудно также заметить, что последняя дизъюнкция и мультиплексор представляются следующим образом:

$$(A_1 \vee A_2 \vee A_3)C \vee B\overline{C},$$

что можно переписать как

$$A_1C \vee A_2C \vee A_3C \vee B\overline{C}.$$

Очевидно, что данное выражение вычисляется на глубине 3. Оптимизированная реализация уровня LFSR изображена на рис. 4.

Используя все преобразования и утверждения, показанные выше, можно получить реализацию алгоритма ZUC, для которой глубина одного раунда равна 25.

Заключение

Приведены результаты исследования аппаратной реализации потокового шифра ZUC и способы ее оптимизации. В качестве параметра оптимизации выбрана глубина реализующей шифр схемы.

Приведены преобразования и доказаны утверждения, позволяющие уменьшить глубину схемы. Для итогового варианта реализации глубина одного раунда составляет 25. Данная оценка на шесть позиций лучше, чем в реализациях, представленных в работах других исследователей.

Авторы выражают благодарность своему научному руководителю, канд. физ.-мат. наук, доц. А. В. Галатенко за внимание к работе и высказанные в ходе подготовки статьи замечания.

Список литературы

1. **Specification** of the 3GPP Condentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 2: ZUC Specication. Version: 1.6. 2011.

2. **Specification** of the 3GPP Condentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 4: Design and Evaluation Report. Version: 2.0. 2011.

3. **Болотов А. А., Галатенко А. В., Гринчук М. И.** и др. Методы оптимизации глубины реализации хэш-функций // Интеллектуальные системы. 2013. Т. 17, Вып. 1—4. С. 224—228.

4. **Супрунюк С. О., Курганов Е. А.** Оптимизация схемной реализации потокового шифра ZUC // Интеллектуальные системы. 2016. Т. 20, Вып. 3. С. 236—241.

5. **Dimitrakopoulos G., Vergos H. T., Nikolos D., Efstathiou C.** A systematic methodology for designing area-time efficient parallel-prex modulo $2^n - 1$ adders // Proc. IEEE Int. Symp. on Circuits and Systems. 2003. P. 225—228.

6. **Гашков С. Б., Гринчук М. И., Сергеев И. С.** О построении схем сумматоров малой глубины // Дискрет. анализ и исслед. опер., сер. 1. 2007. № 14:1. С. 19—25.

7. **Яблонский С. В., Гаврилов Г. П., Кудрявцев В. Б.** Функции алгебры логики и классы Поста. М.: Наука, 1966. 90 с.

8. **Курганов Е. А.** О глубине аппаратной реализации блочного шифра Кузнечик // Интеллектуальные системы. 2016. Т. 20, Вып. 1. С. 61—76.

9. **Lingchen Z., Luning X., Zongbin L., Jiwu J., Yuan M.** Evaluating the optimized implementations of SNOW 3G and ZUC on FPGA // Proceedings of IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom). 25—27 June 2012. P. 436—442.

10. **Яблонский С. В.** Введение в дискретную математику. М.: Наука, 1986. 384 с.

On Depth of the ZUC Stream Cipher Hardware Implementation

S. O. Suprunyuk, sofia-92@mail.ru, **E. A. Kurganov,** kuev@yandex.ru, Moscow State University, Moscow, 119991, Russian Federation

Corresponding author:

Kurganov Evgeny A. Postgraduate Student, Moscow State University, Moscow, 119991, Russian Federation, E-mail: kuev@yandex.ru

Received on January 04, 2018
Accepted on January 23, 2018

A programmable logic device (PLD) is an electronic component that is used to build digital circuits that are reprogrammable. PLD is widely used in different implementations of information protection facilities. It offers high operating speed and possibility to quickly change compromised or obsolete encryption algorithms due to flexibility of programmatic methods and power of hardware implementations, such as ASIC.

This paper deals with a hardware implementation of ZUC cipher. This implementation can be implemented by using logical conjunction, disjunction, negation and delay blocks. The main indicator of productivity of such implementations is a circuit depth, namely the maximum length of a simple way of the circuit (negation elements are not taken into account).

ZUC is a stream cipher designed by the Data Assurance and Communication Security Research Center (DACAS) of the Chinese Academy of Sciences. The cipher forms the core of the 3GPP mobile standards 128-EEA3 (for encryption) and 128-EIA3 (for message integrity). It was proposed for inclusion in the Long Term Evolution (LTE) or the 4th generation of cellular wireless standards (4G).

A detailed description of ZUC cipher is provided at the beginning of the article. The depth estimation of basic components (Adder, S-box, etc.) and the simplest cipher implementation are presented. The article concludes with several optimizing circuit transformations and statements which contribute to reduction of implementation depth. The article also includes the review of existing studies.

Keywords: *hardware implementation, circuit depth optimization, stream ciphers, ZUC cipher, linear feedback shift registers, field*

For citation:

Suprunyuk S. O., Kurganov E. A. On Depth of the ZUC Stream Cipher Hardware Implementation, *Programmnaya Ingeneria,* 2018, vol. 9, no. 5, pp. 221—227.

DOI: 10.17587/prin.9.221-227

References

1. **Specification** of the 3GPP Condentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 2: ZUC Specication. Version: 1.6, 2011.

2. **Specification** of the 3GPP Condentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 4: Design and Evaluation Report. Version: 2.0, 2011.

3. **Bolotov A. A., Galatenko A. V., Grinchuk M. I., Zolotykh A. A., Ivanovich L.** Metody optimizacii glubiny realizacii hesh-funkcij (Techniques for depth optimization of hash-functions implementations), *Intellektualnye Sistemy*, 2013, vol. 17, no. 1—4, pp. 224—228 (in Russian).

4. **Suprunyuk S. O., Kurganov E. A.** Optimizaciya shemnoj realizacii potokovogo shifra ZUC (Optimization of depth of ZUC stream cipher hardware implementation), *Intellektualnye Sistemy*, 2016, vol. 20, no. 3, pp. 236—241 (in Russian).

5. **Dimitrakopoulos G., Vergos H. T., Nikolos D., Efstathiou C.** A systematic methodology for designing area-time efficient parallel-prex modulo $2^n - 1$ adders, *Proc. IEEE Int. Symp. on Circuits and Systems*, 2003, pp. 225—228.

6. **Gashkov S. B., Grinchuk M. I., Sergeev I. S.** O postroenii shem summatorov maloj glubiny (Circuit design of an adder of small depth), *Diskretnyi Analiz i Issledovanie Operatsii*, 2007, vol. 14, no. 1, pp. 27—44 (in Russian).

7. **Jablonskii S. V., Gavrilov G. P., Kudryavcev V. B.** *Funkcii algebry logiki i klassy posta* (Functions of the logic algebra and Post classes), Moscow, Nauka, 1966, 90 p. (in Russian).

8. **Kurganov E. A.** O glubine apparatnoj realizacii blochnogo shifra Kuznechik (On depth of the hardware implementation of block cipher Kuznechik), *Intellektualnye Sistemy*, 2016, vol. 20, no. 1, pp. 61—76 (in Russian).

9. **Lingchen Z., Luning X., Zongbin L., Jiwu J., Yuan M.** Evaluating the optimized implementations of SNOW 3G and ZUC on FPGA, *Proceedings of IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom).* June 25—27, 2012, pp. 436—442.

10. **Jablonskii S. V.** *Vvedenie v diskretnuyu matematiku* (Introduction to discrete mathematics), Moscow, Nauka, 1986, 384 p. (in Russian).

УДК 338.364.2 DOI: 10.17587/prin.9.228-234

Д. С. Пащенко, канд. техн. наук, MBA, независимый консультант
в области разработки программного обеспечения, Москва, e-mail: denpas@rambler.ru

Основные ошибки в управлении проектами заказной разработки программного обеспечения

Наблюдаемый последние годы рост спроса на услуги разработки программного обеспечения (ПО) по требованиям заказчика обусловливает появление на российском рынке новых компаний, предлагающих такие услуги. Часть из них приходит из смежных IT-сфер — дистрибьюции аппаратного обеспечения, консалтинга, рекламных услуг, разработки web-сайтов. Такие компании, несмотря на почти полное отсутствие соответствующих компетенций, участвуют в тендерах и в случае победы спешно набирают на рынке труда новую команду для реализации проекта. В статье описаны типичные ошибки таких команд в управлении проектами заказной разработки ПО: от планирования проекта до управления продуктом и человеческим капиталом команды. Приведены способы выявления таких ошибок с помощью анализа проектных артефактов: проектного плана, реестра рисков, документов внутреннего учета трудозатрат в командах и архитектурных диаграмм. Для каждой группы выявляемых ошибок даны рекомендации по корректирующим воздействиям, направленным на уменьшение их негативного влияния. Рекомендации по улучшению процессов и артефактов в проектах разработки ПО будут полезны и для руководителей в компаниях-заказчиках, и для руководителей проектов, возглавляющих недавно сформированные команды разработки в IT-компаниях из смежных сфер.

Ключевые слова: *управление проектами, программные продукты, проекты разработки ПО, организация производства, заказная разработка ПО*

Введение и постановка проблемы

Во всем мире проекты заказной разработки ПО являются значительной частью рынка информационных услуг [1]. При этом российские аутсорсинговые компании предоставляют такие услуги уже долгие годы, как для внутреннего рынка, так и для иностранных заказчиков. Так в прошлом российская (ныне швейцарская) компания Люксофт стала настоящим лидером отечественного рынка: в ее портфеле линейки проектов для международных лидеров, включая корпорации Боинг и Дойтче Банк. Другая российская компания 1С создала популярный на пространстве СНГ инструментарий (язык, среду разработки) для создания на его основе заказного ПО. Этот инструментарий поддерживается десятками тысяч региональных разработчиков. Несмотря на это, общая доля российских компаний на мировом рынке услуг заказной разработки ПО очень мала и продолжает снижаться: за последние 10 лет российские компании еще сильнее отдалились от лидеров экспорта — Индии и Китая, а теперь уступают уже региональным лидирующим экспортерам ПО Израилю и Филиппинам [2].

При этом Россия обладает существенным внутренним рынком информационных технологий, где протекционные меры государства направлены на получение отечественными исполнителями нерыночных конкурентных преимуществ [3]. Этот локальный рынок продолжает свой рост, несмотря на экономический кризис, и все больше компаний из разных секторов экономики заказывают разработку ПО по собственным уникальным требованиям. Постоянный рост спроса привлекает на рынок услуг заказной разработки ПО новых исполнителей, в том числе "случайные" компании, в основном web-студии и рекламные агентства, успехи которых связаны с созданием простых web-сайтов и интернет-магазинов. Вместе с тем все чаще крупные системные интеграторы и поставщики аппаратного обеспечения, не обладающие существенным опытом собственной разработки информационных систем, участвуют в конкурсах на проекты заказной разработки. Такие компании обладают подходящими формальными показателями для тендеров, например, "годовой оборот" или "количество лет активной деятельности компании". В процессе тендера они активно демпингуют, надеясь, что прибыли от продажи сопутствующего аппаратного обеспечения в целом покроют убытки от проекта разработки ПО. Такие компании без стеснения вводят в заблуждение потенциального заказчика заявляемыми производственными возможностями (поддерживаемые технологии, опыт самостоятельной разработки похожего ПО).

Такие типы новых игроков рынка разработки ПО часто получают проекты разработки систем, значительно превосходящие их производственные мощ-

ности. Для краткости далее в статье "случайные" команды таких проектов будем называть "команды IT-компаний из смежных сфер".

Основная цель настоящей статьи состоит в том, чтобы провести обзор основных ошибок в управлении проектами разработки заказного ПО командами из IT-компаний, не занимающихся профессиональной заказной разработкой, а пришедших на волне повышенного спроса на данный рынок из смежных сфер деятельности. К таким компаниям относятся web-студии (специализация — сайты, интернет-магазины), поставщики аппаратного обеспечения (специализация — продажа серверов, персональных компьютеров, хостинг web-сайтов) и в меньшей степени — системные интеграторы (специализация — интеграция и настройка готовых систем). Обзор таких ошибок дополнен методикой их выявления и предложениями по корректирующим воздействиям, направленным на уменьшение их влияния. Анализ ошибок и предложенные меры реагирования на них позволят, по мнению автора, как представителям заказчика, так и менеджерам команды исполнителя рассчитывать на более успешное завершение проекта и сохранение команды исполнителя для последующего развития продукта проекта.

Актуальность данной темы заключается во весьма значительном проникновении таких, как кажется на первый взгляд, "случайных" компаний на рынок заказной разработки ПО. Этот факт подтверждается как информацией на сайтах подобных работах [4, 5], так и личным опытом автора, изложенном в более ранних статьях [6]. Размывание специализации IT-компаний и победы в тендерах по "непрофильным проектам" — это вовсе не развитие бизнеса, как иногда кажется их менеджменту, а абсолютно негативное явление, которое в силу необходимости выполнения полученных в тендере обязательств приводит к следующим последствиям:

- фактическое несовпадение ожидаемого заказчиком и предоставляемого исполнителем набора проектных параметров: качества продукта проекта, сроков его реализации, бюджета — в каждой итерации проекта;
- возрастающая нагрузка на проектную команду внутри исполнителя, приводящая в долгосрочной перспективе к низким результатам ее работы, демотивации разработчиков и периодически повторяющемуся процессу полураспада проектной команды с увольнением уставших сотрудников;
- финансовые потери заказчика, вынужденного за свой счет компенсировать отсутствие профессионализма исполнителя в производственном аспекте — оплата перерасходов бюджета исполнителем, имиджевые потери от срыва сроков и низкого качества ПО для конечных пользователей.

Также следует отметить, что проекты заказной разработки ПО очень часто становятся серией проектов. Причина в том, что развитие продукта только начинается после его выхода в промышленную эксплуатацию, а это приводит к тому, что ошибки и оплошности непрофессиональной команды разработчиков будут повторяться из итерации в итерацию.

В единичных случаях такие компании и команды успешно проходят все трудности и создают новое направление бизнеса на базе текущей проектной команды. Однако в большинстве случаев после нескольких очень сложных лет заказчик меняет исполнителя, иногда просто нанимая команду профессионалов в свой штат. Такая замена также ведет к серьезным расходам. Перестройка архитектуры уже созданной системы и внедрение современных технологий разработки ПО, замещающих наработки "любительского уровня" IT-компаний из смежных сфер деятельности, могут оказаться дороже, чем повторное создание аналогичной системы "с нуля".

Основные ошибки в управлении проектами заказной разработки ПО командами IT-компаний из смежных сфер связаны с самими истоками поспешного создания таких команд (групп). При планировании работ такой группы необходимо учитывать следующие обстоятельства:

- специалисты собраны в группу в спешке на высоко конкурентном трудовом рынке — создание полноценной команды происходит уже в проекте;
- специалисты обладают разрозненным опытом, они ничего не знают о практиках производственного и управленческого учета в текущей компании;
- руководство компании исполнителя не обладает собственным релевантным опытом разработки промышленного ПО по требованиям заказчика, имеет неверное или поверхностное представление о специфических рисках, производственных технологиях и учете трудозатрат в таких проектах.

Далее в статье приведем перечень типичных ошибок таких команд, методику их выявления и предложим целесообразные управляющие воздействия, направленные на снижение влияние таких ошибок на основные проектные параметры, включая качество программного продукта, сроки завершения проекта и его бюджет.

Типичные ошибки и методика их выявления

В организации работы поспешно собранной на внешнем рынке команды и ее взаимодействии с руководством исполнителя и заказчика проявляются следующие группы ошибок:

1) ошибки планирования проекта;
2) ошибки производственного учета;
3) ошибки управления продуктом;
4) ошибки управления человеческим капиталом проекта.

Ошибки планирования проекта легко выявляются при анализе типичных артефактов проектного управления вне зависимости от применяемых подходов к разработке ПО. Ошибки планирования, прежде всего, связаны со слабым представлением команды проекта о потенциальных рисках управления сроками релизов и скоростью разработки командой продукта проекта. Предыдущий опыт компании-исполнителя не связан с разработкой сложных информационных систем. Это означает, что весь на-

копленный корпоративный опыт не только не приносит пользы, а даже наоборот вреден. Он создает иллюзию, что текущая разработка идет на должном качественном уровне до самых последних фаз, когда стоимость исправлений становится самой высокой.

Выявление ошибок этой группы начинается с анализа плана выполнения проекта в формате спринта (для "гибких" подходов) или классической диаграммы Ганта (для итерационных RUP-образных подходов). Типичные ошибки команды из IT-компании из смежной сферы заключаются в отсутствии: четкого плана реализации проекта; учета рисков в плане; формализованного представления команды о важных контрольных точках выполнения проекта. Методика нахождения таких ошибок заключается в анализе параметров плана выполнения проекта. Такой анализ предполагает выполнение следующих действий.

1. Оценка степени детальности плана.
   a. При явном преобладании в плане текущего этапа работ длительностью более трех рабочих дней (выраженные в днях, часах или относительных символических единицах), план нуждается в доработке и детализации.
   b. При наличии оценок работ с продолжительностью менее четырех часов, такие работы должны быть переоценены: при разработке ПО на заказ не существует атомарных задач, которые занимают менее половины рабочего дня. Чаще всего такие оценки можно увидеть в сметах web-студий, которые переносят свой опыт оценок примитивных функций web-сайтов в заказную разработку промышленного ПО.
   c. При наличии в плане текущего этапа проекта любых работ с продолжительностью более 8—10 рабочих дней, такие работы должны быть детализированы и переоценены. Оценки в 60...80 и более часов показывают непонимание командой объема функций, которые должна реализовать система в рамках этой задачи.

2. Определение в плане пулов задач (технических историй и т. п.), относящихся к доработке архитектуры и обработке "технического долга" программного продукта и проекта в целом. На любом этапе развития информационной системы необходимо продолжать совершенствование не только функциональной, но и технологической составляющей [7]. Отсутствие таких работ в этапе или спринте свидетельствует о слабом понимании командой жизненного цикла развития программного продукта и грядущих неизбежных сложностях — производственных и инфраструктурных.

3. Оценка выделенного времени и трудозатрат для первоначальной стабилизации продукта проекта в промышленной эксплуатации. Вне зависимости от подхода к разработке ПО, уже кажущееся "готовое решение" потребует еще не менее 20...25 % бюджета и временных затрат от общей оценки проекта на стабилизацию в продуктивной среде и успешный запуск для конечных потребителей. Занижение оценки усилий (времени, бюджета) для стабилизации создаваемой информационной системы — это типичная ошибка команд, не обладающих реальным опытом создания систем по требованиям заказчика. Такие команды неверно оценивают как изменение требований и функциональных возможностей в ходе реализации самого проекта, так и всю полноту затрат на организацию сред развертывания программного продукта.

4. Оценка степени влияния рисков на рабочий план. Такое влияние должно заключаться в наличии резервов времени на каждом этапе, наличии активностей по анализу текущей производительности команды и уровня качества продукта, степени параллелизма выполняемых задач, правильной логической последовательности реализуемых функциональных возможностей продукта.

5. Оценка наличия зависимостей между различными работами: в любых используемых подходах к разработке ПО различные работы имеют взаимные зависимости, которые должны быть визуализированы в артефактах планирования.

6. Оценка наличия контрольных точек, сопровождающих проекты по разработке заказного ПО. Отсутствие конечных и промежуточных контрольных точек свидетельствует о поверхностном понимании командой сценария создания продукта проекта, о пренебрежении формальными согласованиями промежуточных результатов с заинтересованными участниками (демопоказы, прототипы решений и интерфейсов, согласование документов и т. п.).

После выявления описанных выше ошибок необходимо привлечь руководителя проекта со стороны исполнителя (или Scrum-мастера) к диалогу для того, чтобы убедиться в одинаковом понимании запланированного хода проекта, обязать исполнителя к внесению исправлений в артефакты проекта. Изменение артефактов планирования должно осуществляться итерационно, пока все выявленные ошибки не будут устранены.

Вторая группа типичных ошибок — ошибки производственного учета в командах из IT-компаний, работающих в смежных областях. Следует отметить, что это вовсе не внутренние затруднения этих IT-компаний, как считают многие заказчики. Все понесенные убытки будут обязательно включены в следующие оценки и следующие этапы проекта. Поэтому и исполнитель, и заказчик заинтересованы в организации правильного учета затрат в таких проектах. Заказчик вправе на регулярной основе узнавать о том, как происходит производственный учет у исполнителя, опираясь как на соответствующие артефакты, так и на косвенные признаки вроде "текучки кадров" команды проекта.

Одной из ключевых методик оценки бюджета проекта является стоимостная оценка, основанная на рейтах (ставках) специалистов, занятых в проектной команде. Порядок такой оценки можно описать следующим образом: этапы работ или релизы выпуска продукта разбиваются на отдельные функции, составляется производственный график их реализации исполнителями, оцениваются соответствующие трудозатраты специалистов. Далее благодаря поправочным и рисковым коэффициентам из оцененных трудозатрат получаются скорректированные временные оценки, и с помощью часовых/дневных рейтов

(ставок) получается общая оценка стоимости услуг по разработке. Другой методикой бюджетирования является принцип Time and Material, когда заказчик оплачивает определенное время команды, а команда выполняет "укладывание" реализуемых функциональных возможностей в оплаченное время.

При реализации обоих отмеченных выше методик оценки проектов допускаются следующие типичные ошибки:

1) проводят календарный (помесячный, ежеквартальный) учет рабочего времени специалистов в отрыве от производственных результатов команды;

2) проводят учет реализованных функций в отрыве от достигнутых бизнес-целей;

3) при простоях специалистов стремятся направить их усилия на дополнительные работы, оплачиваемые заказчиком дополнительно, вместо усиления команды в нужных направлениях.

Учет ежедневного рабочего времени инженеров в командах, разрабатывающих ПО, уходит своими корнями в индустриальную эпоху заводов и фабрик. Несмотря на убедительные примеры отказа от этой практики в ведущих мировых компаниях (Google, Apple, Яндекс, Finastra), она сохраняется во многих российских системных интеграторах и web-студиях. В таких компаниях низкая маржинальность сервисного бизнеса заставляет менеджмент считать каждый оплачиваемый час сотрудников, поэтому они переносят эти практики в заказную разработку промышленного ПО. В результате такого учета можно получить отчеты сотрудников с тем, сколько часов было потрачено на определенную работу, сколько часов ушло на обучение и простои, сколько часов сотрудник провел в офисе. Учитывая уникальные требования заказчика и уникальность создаваемых решений в каждом проекте, такая информация обладает очень низкой бизнес-значимостью. Ее анализ сильно затруднен и повторное использование вряд ли адекватно. С точки зрения пользы для бизнеса IT-компании в целом, по опыту автора, такие системы учета делятся на следующие две категории:

- отчеты содержат значительный объем правдивой информации, создание отчетов отняло у сотрудников проекта значительное время и уменьшило усилия инженеров и менеджеров, предназначенные для реализации продукта;
- отчеты сделаны быстро, но "для галочки", тогда почти все выводы на их основе неверны и не позволяют улучшать бизнес-процессы.

Оба варианта всегда бесполезны для проектной команды, чьи цели лежат вне области анализа времени присутствия сотрудников в офисе. Часто они малополезны и для IT-компании в целом, особенно если они не используются для оценки реализации типовых функций разрабатываемого ПО. Примером таких типовых функций могут быть, например, интеграция с социальными сетями или реализация базовой ролевой модели в новых аналогичных проектах.

Следующая типичная ошибка описываемых команд разработчиков — это анализ созданных ими компонентов и функций информационной системы без анализа достигнутых заказчиком бизнес-целей. На стороне исполнителя распределенный по месяцам в году учет работ, направленных на реализацию функциональных возможностей, означает "соблюдение бюджета проекта по итерациям", а на стороне заказчика — "плавное расходование средств в календарном году". На практике отношение к разрабатываемому продукту как к набору ранее оцененных и реализованных функций в среднесрочной и долгосрочной перспективах приводит к следующим негативным последствиям:

- неминуемые перерасходы бюджета и стремительные изменения требований внутри итераций, так как реальная эксплуатация системы всегда выявляет неучтенные факторы и требует немедленной реакции;
- ухудшение общего качества программного решения, так как невостребованные и "неиспользуемые" функции часто содержат дефекты, которые не выявлены на этапах тестирования и опытно-промышленной эксплуатации.

Производственный учет реализованных функций должен быть всегда связан с достижением заказчиком поставленных бизнес-целей с помощью созданного продукта. Модели, в которых исполнитель соблюдает бюджеты релизов, но заказчик не достигает бизнес-целей, не могут считаться разумными. В перспективе они приводят к провалу проекта и досрочному завершению сотрудничества с серьезными репутационными потерями для команды исполнителя.

Небольшие web-студии, не обладающие солидной финансовой устойчивостью, ревностно относятся с каждому дню простоя специалистов. Итерационные (RUP, MSF и т. п.) и тем более водопадные модели производства ПО, на первый неискушенный взгляд, предполагают разную степень загруженности инженеров разной специализации в течение одной итерации проекта. Разработчики в процессе реализации таких моделей "не востребованы" до создания требований, аналитики "выключаются из проекта" после согласования технического задания, инженеры качества "не нужны" до появления первого релиза и т. п. Данное восприятие загруженности специалистов команды вызвано непониманием процессов в жизненном цикле разработки ПО и степени влияния реальных рисков на успех в создании конечного продукта. Поэтому при таких возникающих простоях менеджмент IT-компаний из смежных областей предпочитает временно перевести сотрудника в другой проект вместо того чтобы усилить традиционно слабые места в разработке ПО. По опыту автора примерно половина простоев инженеров — это недостаточное внимание к задачам, на решение которых должно быть потрачено соответствующее время. К их числу относятся:

1) проектирование и последующее совершенствование архитектуры ПО;

2) анализ и улучшение программного кода;

3) управление требованиями, включая их актуализацию;

4) анализ эргономики уже реализованных пользовательских интерфейсов;

5) создание и детализация тестовой модели, тестовых сценариев, автоматизация тестирования;

6) верификация и валидация уже реализованных функций, в том числе на предмет соответствия бизнес-требованиям.

Однако в типичных командах IT-компаний из смежных областей предпочитают направлять усилия в сторону оплаченных заказчиком работ, придумывая такие работы для частично освободившихся сотрудников и не фокусируя внимание на общем качестве продукта и документации. Это естественное продолжение производственного учета "по релизам" и без привязки к достигаемому заказчиком бизнес-результату.

По убеждению автора, заказчик должен представлять себе производственный учет исполнителя в проекте если не по проектным артефактам, то по косвенным признакам. Следует учитывать изложенные далее рекомендации:

- управляющие воздействия, направленные на изменение трудозатрат в проекте, не должны быть основаны на календарном производственном учете, а должны быть привязаны к значимым событиям в проекте (релизам, спринтам, итерациям) и быть связаны с реализацией бизнес-целей проекта;
- нежелательно направление простаивающих сотрудников исполнителя на выполнение дополнительных работ, в то время как остаются нереализованными мероприятия, повышающие качество и стабильность работы продукта проекта.

Третья группа ошибок — это ошибки в управлении продуктом проекта, как в части его функциональных возможностей, так и с точки зрения технологий. Частым заблуждением является неверное распределение ответственности за развитие продукта между заказчиком и исполнителем. При таком подходе полное перекладывание ответственности за функциональное или технологическое развитие ПО на чью-то одну сторону не имеет никаких оправданий. Однако даже при совместной работе над развитием продукта у команд IT-компаний из смежных сфер можно наблюдать характерные ошибки.

Для выявления технологических ошибок в управлении процессом создания продукта следует рассматривать совокупность следующих артефактов:

- архитектурную компонентную диаграмму;
- конфигурационный план и детализацию сред развертывания;
- актуальную модель нефункциональных требований.

Анализ данных артефактов позволяет выявить следующие типичные ошибки:

1) выбор технологий и сред разработки из стека, к которым привык исполнитель, вместо использования таких, которые соответствуют нефункциональным требованиям к системе;

2) использование проприетарных жестко ограниченных технологий, не позволяющих расширять и усложнять решение по мере его развития;

3) излишняя монолитность системы, когда набор функций соединен вместе в компоненты не ввиду их взаимной логической или технологической связи, а потому что так удобнее разрабатывать систему;

4) отсутствие требований к средам тестирования и опытно-промышленной эксплуатации;

5) отсутствие требований к автоматизированным сценариям развертывания системы в различных средах;

6) отсутствие актуальности и неполнота нефункциональных требований.

Разрешение пула вопросов, обусловленных перечисленными ошибками, силами бизнес-заказчика вряд ли возможно, поэтому единственной рекомендацией может служить дополнительная внешняя экспертиза архитектурного решения незаинтересованным специалистом в области разработки промышленного ПО. При этом ошибки в управлении функциональными возможностями продукта в командах IT-компаний из смежных сфер совместными усилиями с заказчиком могут быть исправлены довольно легко. Далее приведены самые типичные проблемные вопросы, которые можно легко выявить из анализа бэк-лога продукта и актуальных бизнес-требований:

- планирование процессов реализации новых функций в отрыве от бизнес-целей заказчика и технологического совершенствования подлежащей разработке информационной системы;
- слабое влияние обратной связи от конечных потребителей продукта на его развитие;
- отсутствие регулярной работы по усовершенствованию пользовательских интерфейсов.

Рекомендации по разрешению таких вопросов лежат в области формально строгого описания процессов планирования, направленных на развитие продукта. Вне зависимости от применяемого подхода в разработке ПО необходимо выработать следующий алгоритм планирования разработки ПО на заказ.

1. Выделение и приоритизация групп задач для разработки на стратегическом уровне, согласование с бизнес-целями и учет необходимости постоянного технологического совершенствования.

2. Учет основных бизнес-рисков и текущих ограничений в развитии продукта.

3. Декомпозиция задач таким образом, чтобы в каждый новый релиз входили задачи:

- направленные на достижение тактических целей в развитии функциональных возможностей;
- улучшающие эргономические свойства и дизайн уже реализованных функций;
- сокращающие технический долг решения [8];
- учитывающие уже полученную обратную связь от конечных потребителей.

4. Выделение ресурсов на организацию обратной связи с конечными потребителями в рамках анализа выпущенного релиза.

Такой формализованный подход, вовлекающий во взаимодействие при развитии программного продукта и исполнителя, и бизнес-заказчика уменьшает такие типичные риски, как:

- невостребованность у конечных потребителей создаваемого продукта;
- перерасход бюджета на разработку;
- создание нестабильного продукта в проектах, когда скорость разработки ошибочно становится важнее качества.

Как было отмечено выше, еще одна типичная ошибка при разработке заказного ПО в IT-компаниях из смежных сфер — это недостаточное внимание к человеческому капиталу команды. Разработка сложного промышленного ПО не только требует погружения в предметную область, но и подразумевает некоторую историю проб и ошибок проектной

команды. Однако web-студии, опирающиеся на взаимозаменяемых удаленных специалистов и их разовые работы, или системные интеграторы с более высокой атомарностью задач, в своей основной деятельности в меньше степени нуждаются в полноценных проектных командах. Этот опыт они переносят и в доставшиеся им проекты заказной разработки, не укрепляя при этом проектные команды и не обращая должного внимания на сплочение и мотивацию сотрудников. Напомним, что такой подход сочетается с другими просчетами, к числу которых относятся:

- отсутствие релевантного опыта у компании исполнителя;
- поспешный сбор группы сотрудников для реализации проекта;
- фактическое несовпадение ожиданий заказчика и возможностей исполнителя практически по всем проектным параметрам.

Вместо последовательного укрепления проектной команды и копирования практик управления персоналом у ведущих компаний отрасли заказной разработки (Люксофт, ЭПАМ и др.) IT-компании из смежных сфер применяют следующие привычные им методы:

- управляющие воздействия на основе аналитических выводов из результатов календарного почасового производственного учета;
- ротации специалистов команды, включая замены руководителей групп специалистов и всего проекта;
- "спихивание" ответственности за успех на руководителя проекта без выделения дополнительных ресурсов.

На "перегретом" трудовом рынке IT-специалистов такая стратегия управление командами, занятыми реализацией проектов, приводит к регулярным полураспадам проектных групп, сопровождающимся уходом демотивированных и уставших сотрудников. Они с легкостью покидают проект, унося с собой экспертизу в предметной области и знания о накопленном техническом долге информационной системы.

Самым очевидным свидетельством сложностей в проектной группе, связанных с недостаточным вниманием к человеческому капиталу, являются частые ротации ведущих специалистов и длительные больничные инженеров.

Для заказчика программного продукта это вовсе не набор отстраненных бизнес-проблем чужого бизнеса, а наиболее болезненный удар по скорости разработки и качеству получаемого решения. Кроме смены парадигмы мышления менеджмента самих IT-компаний, полезными могут быть следующие рекомендации:

- укрепление личных горизонтальных связей между специалистами заказчика и исполнителя;
- участие в регулировании режимов "сверхнапряженной" и "расслабленной" работы проектной команды, в том числе путем изменения оперативных требований и ожиданий заказчика;
- повышение формализации процессов и документирование ключевых параметров — требований, прототипов, описания технического долга;
- создание артефактов с перспективным описанием информационной системы — планируемых изменений в архитектуре, мероприятий по технологическому совершенствованию, планированию следующих релизов.

Заключение

В начале каждого большого проекта разработки программного обеспечения по требованиям заказчика сотрудники исполнителя и заказчика, как правило, плохо знакомы. Они по-разному представляют себе цели и задачи проекта, процесс разработки продукта. Если в результате тендера победила IT-компания, не специализирующаяся на заказной разработке ПО, то следует ожидать повторения ее командой всех приведенных выше типичных ошибок: от поверхностного планирования до недостаточно внимательного отношения менеджмента этой компании к человеческому капиталу создаваемой "с нуля" проектной команды. Снижение влияния таких ошибок лежит в плоскости соблюдения заинтересованными руководителями рекомендаций, представленных в настоящей работе:

- проводить анализ артефактов планирования и управления, подготовленных исполнителем, и устранять найденные ошибки — повышать детализированность плана, вносить влияние специфических рисков, создавать промежуточные ключевые точки, отводить достаточное время на стабилизацию продукта проекта в продуктивной среде;
- анализировать и модернизировать применяемый производственный учет в IT-компании исполнителя, принимая во внимание ключевые точки в проекте, взаимосвязь развития продукта с бизнес-целями и влияние данного учета на мотивацию проектной команды;
- анализировать параметры использования труда и талантов сотрудников команды для достижения главных бизнес-целей, отсекая любые попытки отвлечения усилий на дополнительные, долгосрочные и второстепенные работы, уделяя при этом максимум внимания немедленному улучшению продукта проекта и в функциональном, и в технологическом аспектах;
- прилагать регулярные усилия по сохранению и усилению проектной команды и человеческого капитала, создавая основу для постоянного сокращения сроков разработки новых релизов и совершенствования качества программного продукта.

К сожалению, российский (и мировой) рынок разработки ПО растет быстрее, чем объективные возможности лидирующих игроков, обладающих зрелыми процессами и опытом заказной разработки. В сочетании с постоянным ожиданием снижения цен на услуги по разработке ПО это приводит к невероятным исходам даже в самых серьезно подготовленных тендерах: победам демпингующих IT-компаний, не обладающих реальным опытом и ресурсами для выполнения сложных проектов. Описанные в статье ошибки являются типичными для таких IT-компаний и наносят, по мнению автора, наиболее серьезный

урон продукту проекта и бизнес-планам заказчиков. Исправление ошибок на ранних этапах — это обязательная совместная работа заказчика и исполнителя. При этом стоит отметить, что исполнитель практически в каждом даже неуспешном проекте зарабатывает деньги, а значит, его мотивация к совершенствованию процессов планирования проектов и разработке ПО должна обеспечиваться постоянными усилиями заказчика.

Список литературы

1. **Boggs R.** Worldwide Small and Medium-Sized Business Forecast, 2015—2019: IT Spending by Company Size and Region for Key Hardware, Software, and Services Categories. July 2015. URL: https://www.idc.com/getdoc.jsp?containerId=US41381617 (дата обращения: 15.12.2017).
2. **ICT** service exports (BoP, current US$). Data of The Word bank — 2011—2016. URL: https://data.worldbank.org/indicator/BX.GSR.CCIS.CD?end=2016&locations=RU-IN-IL-CN-PH&start=2011&view=chart (дата обращения 12.02.2018).
3. **Приказ** Минкомсвязи России 01.04.2015 № 96 "Об утверждении плана импортозамещения программного обеспечения" // Портал Минкомсвязи. URL: http://minsvyaz.ru/ru/documents/4548/ (дата обращения 12.02.2018).
4. **ADV/web-инжиниринг** — разработка и создание сайтов, поисковое продвижение и привлечение трафика. URL: https://adv.ru/#Technologies (дата обращения 12.02.2018).
5. **Инфосистемы** Джет — системный интегратор широкого профиля. URL: https://jet.su/services/software-development/ (дата обращения 12.02. 2018).
6. **Пащенко Д. С.** О факторах, тормозящих развитие российских IT-компаний в фазе зрелости // Менеджмент и бизнес-администрирование. 2013. № 3. С. 123—128.
7. **Suryanarayana G.** Refactoring for Software Design Smells (1st ed.). Morgan Kaufmann. 2014. 258 p.
8. **Allman E.** Managing Technical Debt // Communications of the ACM. 2012. Vol. 55, No. 5. P. 50—55.

Basic Mistakes in Project Management in Custom Software Development

D. S. Pashchenko, denpas@rambler.ru, Moscow, 123368, Russian Federation,

Corresponding author:

Pashchenko Denis S., Ph. D., MBA, Independent consultant in software development, Moscow, 123368, Russian Federation
E-mail: denpas@rambler.ru

Received on February 22, 2018
Accepted on March 13, 2018

The growth of demand for custom software development services predetermines the appearance on the Russian market of new companies offering this service. Some of them come from related IT-spheres — distribution of hardware, consulting, advertising services, development of web-sites. Such companies participate in tenders and, in case of victory, hastily recruit a new team in the labor market to implement the project. This article describes typical errors of such teams in the management of projects of custom software development: from project planning to product and HR-management in the team. The article provides ways to identify such errors by analyzing project artifacts: project plan, risk plan, internal labor accounting documents in the team and architecture diagrams. For each group of detected errors the recommendations to correct the effects are presented, proven to reduce their negative impact. Recommendations for improving processes and artifacts in custom software development projects will be useful for executive managers in customer companies and project managers who lead newly formed software development teams in IT-companies from related fields.

Keywords: *project management, software products, software development projects, production organization, custom software development*

For citation:

Pashchenko D. S. Basic Mistakes in Project Management in Custom Software Development, *Programmnaya Ingeneria,* 2018, vol. 9, no. 5, pp. 228—234.

DOI: 10.17587/prin.9.228-234

References

1. **Boggs R.** Worldwide Small and Medium-Sized Business Forecast, 2015—2019: IT Spending by Company Size and Region for Key Hardware, Software, and Services Categories. July 2015, available at: https://www.idc.com/getdoc.jsp?containerId=US41381617
2. **ICT** service exports (BoP, current US$). Data of The Word bank — 2011—2016, available at: https://data.worldbank.org/indicator/BX.GSR.CCIS.CD?end=2016&locations=RU-IN-IL-CN-PH&start=2011&view=chart
3. **Prikaz** Minkomsvjazi Rossii 01.04.2015 № 96 "Ob utverzhdenii plana importozameshhenija programmnogo obespechenija" (Order of the Ministry of Communications of Russia 01.04.2015 No. 96 "On approval of the plan for import substitution of software"), *Portal of Ministry of Communications of Russia*, available at: http://minsvyaz.ru/ru/documents/4548/ (in Russian).
4. **ADV/web-inzhiniring** — razrabotka i sozdanie sajtov, poiskovoe prodvizhenie i privlechenie trafika (ADV/ web-engineering — development and creation of sites, search promotion and attraction of traffic), available at: https://adv.ru/#Technologies (in Russian).
5. **Infosistemy** Jet — sistemnyj integrator shirokogo profilja (Jet Infosystems — a system integrator of a wide profile), available at: https://jet.su/services/software-development/ (in Russian).
6. **Pashchenko D. S.** O faktorah, tormozjashhih razvitie rossijskih IT-kompanij v faze zrelosti (About the factors hindering the development of IT-companies in the maturity phase), *Menedzhment i biznes-administrirovanie*, 2013, no. 3, pp. 123—128 (in Russian).
7. **Suryanarayana G.** *Refactoring for Software Design Smells* (1st ed.), Morgan Kaufmann, 2014, 258 p.
8. **Allman E.** Managing Technical Debt, *Communications of the ACM*, 2012, vol. 55, no. 5, pp. 50—55.

УДК 004.421.6 DOI: 10.17587/prin.9.235-240

С. Д. Махортов, д-р физ.-мат. наук, зав., e-mail: msd_exp@outlook.com,
Воронежский государственный университет

О выразительных возможностях схем описания структуры строк

Алгоритмы вычислений на строках находят все возрастающее применение в различных прикладных областях. В статье рассмотрены две наиболее известные и эффективные схемы анализа и описания структуры строк — массивы граней и Z-блоков. Проведен сравнительный анализ их выразительных возможностей. Предложен и обоснован новый линейный алгоритм преобразования массива граней в массив Z-блоков, обсуждены некоторые его модификации, способные повысить эффективность работы.

Ключевые слова: *строка, поиск вхождений, препроцессинг, префикс-функция, Z-функция, взаимные преобразования, анализ сложности*

Введение

Алгоритмы вычислений на строках находят все возрастающее применение в различных разделах прикладных наук, таких как обработка текстов, сжатие данных, криптография, распознавание речи, вычислительная геометрия, молекулярная биология. Их актуальность существенно повысилась в последние десятилетия, когда появились новые предметные области, требующие *эффективной* обработки *гигантских* символьных последовательностей. К таким областям, в частности, относятся компьютерное зрение, информационный поиск в глобальных сетях, вычислительная геномика и некоторые другие.

Рассмотрим задачу вычисления точных вхождений образца P (т. е. некоторой заданной подстроки) длины m в текст T (т. е. большую строку) длины n. Последовательно, начиная с первой позиции, "прикладывая" образец к тексту и сравнивая с образцом встречающиеся в тексте фрагменты, получим решение. Этот "примитивный" алгоритм имеет вычислительную сложность $O(mn)$. Как известно, алгоритмы с полиномиальной сложностью, тем более квадратичной, в теории имеют "высокий статус" [1]. Однако, например, в задачах исследования цепочек ДНК [2] возможны такие значения, как $m = 300$, $n = 3\,000\,000$, поэтому даже алгоритм с квадратичной сложностью оказывается слишком "дорогостоящим".

Задолго до возникновения подобных проблем были известны алгоритмы, решающие указанную задачу за время $O(n)$ (например, Кнута—Морриса—Пратта, Бойера—Мура и т. д. [1]). Более того, использование специальной структуры "суффиксное дерево" (П. Вайнер) [2], которая может быть построена за линейное по n время, позволяет выявить все вхождения подстроки выполнением $O(m$ + число вхождений) операций. Если когда-то такие алгоритмы имели в основном теоретическое значение, то в наше время их изучение обрело ярко выраженный практический смысл. То же самое можно сказать и о ряде других задач на строках (общая подстрока, оптимальное выравнивание и т. д. [2]), требующих точного или приближенного решения.

В основе многих эффективных алгоритмов сравнения и анализа текстов лежит *препроцессинг* — предварительное исследование структуры строки. В результате такой обработки (в некотором роде, подобно индексу в реляционной базе данных), строится некоторая вспомогательная конструкция, позволяющая существенно сократить число избыточных сравнений символов. На этой стадии обычно исследуется лишь одна из обрабатываемых строк (например, образец P или текст T), а другая сравниваемая строка может оставаться неизвестной. При этом важно, чтобы препроцессинг занимал приемлемое время. Далее следует фаза непосредственного поиска или анализа подстрок, на которой полученная ранее информация о строке используется для сокращения работы.

В настоящей работе рассмотрены две наиболее известные и эффективные схемы предварительного анализа и описания структуры строк — массивы граней и Z-блоков [2, 3]. Проведен сравнительный анализ их выразительных возможностей. Предложен и обоснован новый линейный алгоритм преобразования массива граней в массив Z-блоков, обсуждены некоторые его модификации, способные повысить эффективность работы.

1. Строки — основные понятия

Вначале напомним некоторые понятия и обозначения, связанные со строками [1—3].

Пусть задан *алфавит* — некоторое фиксированное множество различимых элементов (*символов*). *Строка* на алфавите A — это конечная последовательность символов, каждый из которых принадлежит A. Через $n = |S|$ обозначается *длина* (число символов) строки S. Для теоретической "завершенности" вводится понятие *пустой* строки ε (нулевой длины), которая не содержит ни одного символа.

Как обычно, будем интерпретировать строку S в виде массива символов $S[0...n-1]$. Заметим, что

в известной литературе, посвященной алгоритмам на строках [1—3], в соответствующих массивах традиционно принята нумерация символов с 1. Наш же выбор (нумерация с 0) обусловлен ориентацией на синтаксис наиболее распространенных С-подобных языков программирования.

Под *равенством*, или *совпадением*, двух строк $S_1 = S_2$ (определенных на общем алфавите) подразумевается посимвольное равенство двух массивов одинаковой длины.

Подстрока некоторой строки — это строка, полученная как непрерывная подпоследовательность исходной строки. Для любой строки S через $S[i...j]$ обозначается подстрока в массиве S, которая начинается в позиции i и заканчивается в позиции j. В частности, $S[0...i]$ называется *префиксом*, а $S[j...n-1]$ — *суффиксом* строки S.

При $i > j$ строка $S[i...j]$ пуста. Очевидно, длина непустой строки $S[i...j]$ равна $j - i + 1$. Подстрока называется *собственной*, если она не равна S. По определению любая непустая строка в качестве собственной подстроки в любой позиции содержит пустую строку.

Наиболее известны две эффективные схемы предварительного анализа строк — нахождение граней и Z-блоков. О них пойдет речь в следующих разделах.

2. Грани строк

Определение. *Гранью* строки называется любой собственный префикс этой строки, равный ее же суффиксу.

Ограничение "собственный" исключает из рассмотрения грань, совпадающую со всей строкой. Любая непустая строка имеет пустую грань (длины 0).

Например, строка ***ABAABABAABAAB*** содержит две непустые грани: *AB* и *ABAAB* (подчеркиванием выделена наибольшая из двух упомянутых граней).

Выделяют *наибольшую грань* строки, т. е. имеющую среди всех граней наибольшую длину. В предыдущем примере это ***ABAAB***. Строка *ABAABAAB* имеет точно такие же грани, но в этом случае наибольшая грань *abaab* частично перекрывает саму себя (точнее, перекрываются части — префикс и суффикс, образующие эту грань).

В приложениях к исследованию строк важную роль играет *массив граней* (*border of prefixes array*) $bp[0..n-1]$, содержащий длины наибольших граней для всех подстрок $S[0...i]$, $i = 0, 1, ..., n-1$, т. е. для префиксов в S. Например, строке *ABAABABAABAAB* соответствует массив 0, 0, 1, 1, 2, 3, 2, 3, 4, 5, 6, 4, 5. Заметим, что в ряде источников (например, в работе [1]) для такого массива используется название "префикс-функция".

Существует эффективный алгоритм вычисления массива граней, работающий за линейное время [3]. Он основан на достаточно очевидных свойствах граней и их массива. Ниже понадобятся некоторые из них. Для иллюстрации рассмотрим следующую строку: ***AB****AAAB*ACBCA***AB****AAAB* (подчеркиванием выделена наибольшая грань строки, полужирным шрифтом — грань этой грани).

Свойство 1. Если подстрока $S[0...i]$ $(0 < i < n)$ имеет грань длины $k > 0$, то подстрока $S[0..i-1]$ имеет, по крайней мере, грань длины $k-1$. Отсюда следует, что при переходе от $i-1$ к i значение $bp[i]$ увеличивается не более чем на 1, т. е. $bp[i] \leqslant bp[i-1] + 1$.

Свойство 2. Равенство $bp[i] = bp[i-1] + 1$ выполнено тогда и только тогда, когда $S[i] = S[bp[i-1]]$ (поскольку $bp[i-1]$ — позиция символа справа от префикса $S[0..bp[i-1]-1]$, который является наибольшей гранью подстроки $S[0...i-1]$).

Замечание 1. Согласно свойствам 1—2, массив bp состоит из нескольких *серий* возрастающих на 1 целочисленных подпоследовательностей, возможно, разделенных нулями.

Построение массива граней позволяет, в частности, более эффективно найти вхождения образца P в текст T. Возьмем некоторый символ #, не принадлежащий алфавиту A, и построим строку $S = P\#T$. Далее для строки S вычислим массив bp. В этом массиве достаточно выбрать все значения, равные m (длине P). Индекс i каждого такого значения (уменьшенный на $m + 1$ — длину "добавки" в S) укажет правую координату вхождения P в T.

Заметим, что при построенной таким образом строке S в рекуррентном алгоритме вычисления массива граней [3] все обращения к вычисленным граням будут приводить внутрь P. Поэтому для решения поставленной задачи достаточно сохранять лишь элементы массива bp с индексами 1, ..., $m-1$.

В некоторых приложениях (например, в алгоритме Кнута—Морриса—Пратта) целесообразно использование модифицированного массива граней bpm строки S. Его содержимое отличается от рассмотренного выше массива bp дополнительным требованием о несовпадении символов, следующих за частями грани. Точнее, $bpm[i]$ — это длина наибольшей грани подстроки $S[0...i]$, удовлетворяющей условию $S[bpm[i]] \neq S[i+1]$.

Рассматривая структуру массивов граней, можно получить линейные алгоритмы преобразования bp в bpm и обратно, не обращающиеся к символам исходной строки S [3]. Отсюда следует вывод: массивы bp и bpm имеют равные выразительные возможности.

3. Z-блоки

С Z-блоками связан альтернативный подход к описанию структуры строк [2].

Определение. Для строки S и ее позиции $i > 0$ определяется значение $Z_i(S)$ (короче — Z_i) как длина наибольшей подстроки, которая начинается в i и совпадает с префиксом строки S. Сама такая подстрока называется *Z-блоком*. При $i = 0$ полагают $Z_0(S) = 0$.

Например, если $S = AABCAABXAAZ$, то

$Z_4(S) = 3$ (*AAB**C**...AAB**X**...*),

$Z_5(S) = 1$ (*A**A**...A**B**...*),

$Z_6(S) = Z_7(S) = 0$,

$Z_8(S) = 2$ (*AA**B**...AA**Z***).

В работе [2] достаточно подробно описан и обоснован линейный алгоритм вычисления массива

Z-значений (по-другому — "*Z*-функция") заданной строки *S*. Нам потребуется его запись в следующей специальной форме. Будем использовать вспомогательную функцию *SubstrComp* (*S*, *n*, *i*1, *i*2), которая вычисляет наибольшую длину двух совпадающих подстрок строки *S* (длины *n*), начинающихся с позиций *i*1, *i*2.

Здесь и в дальнейшем алгоритм записан с помощью псевдокода — подмножества языка С без указания типов данных.

```
   Prefix-Z-Values (S, zp)
{
   n = strlen (S); l = r = 0; zp[0] = 0;
   for (i = 1; i < n; ++i)
   {
      zp[i] = 0;
      if (i >= r)
      { // Позиция i не покрыта ранним
      // Z-блоком – Z[i] вычисляется от начала
      zp[i] = SubstrComp (S, n, 0, i); l = i;
r = l + zp[i];
      }
      else
      { // Позиция i покрыта ранним Z-блоком –
      // он используется
        j = i – l;
      if (zp[j] < r – i) zp[i] = zp[j]; // Новый
      // Z-блок закончился левее раннего
      else { zp[i] = r – i + SubstrComp (S, n,
      r – i, r); l = i; r = l + zp[i]; }
    }
   }
}

   SubstrComp (S, n, i1, i0)
{ // Предполагается, что i0 >= i1
   eqLen = 0;
   while (i0 < n && S[i1++]==S[i0++]) ++eqLen;
   return eqLen;

}
```

Замечание 2. Контекст используемых в алгоритме двух вызовов функции *SubstrComp* описан комментариями. Функция "честно", т. е. последовательно сравнивая символы, вычисляет длину всего *Z*-блока (первый вызов) или размер продолжения *Z*-блока, имеющего известную начальную длину *i*1 (второй вызов), с заданной позиции текста *i*0.

Вычисление массива *Z*-блоков, подобно массиву граней, дает эффективный способ решения задачи поиска вхождений образца *P* в текст *T*. Выберем символ #, не принадлежащий исходному алфавиту, и сформируем строку *S* = *P*#*T*. Для *S* выполним алгоритм построения массива *zp*. В нем каждое значение *zp*[*i*], равное *m* (т. е. длине *P*), дает индекс *i* начальной координаты вхождения *P* в *T* (ее нужно скорректировать на *m* + 1).

Следует отметить, что при построенной таким образом строке *S* длины всех ее *Z*-блоков не превосходят *m*. Следовательно, в алгоритме *Prefix-Z-Values* обращения к обработанным ранее блокам (индекс *j*) всегда будут приводить внутрь *P*. Поэтому для поиска вхождений образца достаточно хранить лишь элементы массива *zp* с индексами 1, ..., *m* – 1.

Сложность данного алгоритма — $\Theta(m + n)$.

4. Сопоставление двух схем описания структуры строк

В предыдущих разделах были представлены две схемы описания структуры строк — массивы граней и *Z*-блоков. Показаны также примеры их непосредственного применения к оптимизации поиска вхождений образца в текст. Нетрудно заметить, что эти подходы взаимосвязаны: оба массива содержат длины подстрок в *S*, совпадающих с префиксами *S*. Основное отличие в том, что в *zp* отмечаются начальные индексы таких подстрок, а в *bp* и *bpm* — конечные.

В фундаментальной монографии [2] формально обоснованы возможности преобразования массива *Z*-блоков в массивы граней (как обычный, так и модифицированный) с помощью линейных алгоритмов. В этих алгоритмах не используются непосредственные сравнения символов исходной строки *S*. Тем самым автор концепции *Z*-блоков Д. Гасфилд показал, что введенный им массив *zp* обладает не меньшими выразительными возможностями, чем массивы граней *bp* и *bpm*. Указанные результаты автор возвел в статус теорем. Обратные же преобразования массивов в работе [2] рассмотрены не были.

Подобных рассмотрений, по-видимому, пока нет и в других общедоступных научно-технических печатных изданиях. Следует признать, что алгоритмы получения массива *Z*-значений из массива граней можно обнаружить в Интернете. Однако некоторые из них не являются непосредственными (например, предварительно осуществляют стадию восстановления варианта исходной строки *S* [4]), другие — не имеют строгого формального обоснования [5, 6]. В частности, последний факт признает и сам автор публикации [5]. Алгоритм [6] представляет собой модификацию алгоритма [5].

Ниже будет представлен новый алгоритм преобразования массива граней в массив *Z*-значений. Он работает за линейное время и имеет достаточно простое обоснование.

Вначале вернемся к алгоритму *Prefix-Z-Values* из разд. 3, который "напрямую" вычисляет массив *Z*-значений строки *S*. Он записан в такой форме, что непосредственное использование символов исходной строки целиком сосредоточено в выделенной функции *SubstrComp*. Предлагаемое здесь решение задачи состоит в том, чтобы заменить *SubstrComp* другой функцией, которая бы находила аналогичный результат, но вместо строки *S* работала с массивом граней *bp*.

С указанной целью рассмотрим семантику некоторых параметров, используемых в алгоритме *Prefix-Z-Values*. На каждом шаге *i* он вычисляет очередной блок *Z*[*i*], при этом используя "ранние" *Z*-блоки, т. е. полученные на предыдущих шагах. Точнее, *r* — это максимальная правая граница для всех предыдущих *Z*-блоков, *l* — начало соответствующего ей блока. Обозначим этот блок [*l*, *r*), чтобы показать включение/исключение обеих границ. Если несколько ранних *Z*-блоков заканчиваются в одной и той же позиции *r* – 1, то *l* вычисляется как максимум среди таковых.

Как уже подчеркивалось (замечание 2 из разд. 3), функция *SubstrComp*(*S*, *n*, *i*1, *i*0) вычисляет размер продолжения (т. е. увеличения) с заданной позиции *i*0 *Z*-блока, имеющего заранее известную часть длины *i*1 ⩾ 0. Другими словами, когда в позиции *i*0 − *i*1 строки *S* известна часть *Z*-блока, простирающаяся по позицию *i*0 − 1 включительно, требуется "достроить" этот блок, проверив последовательные совпадения символов с позиции *i*0. Следующее утверждение обосновывает замену этой функции на другую, которая не обращается к самим символам исходной строки, но взамен использует массив граней *bp*.

Лемма. В алгоритме *Prefix-Z-Values* результат выполнения каждого из двух вызовов функции *SubstrComp*(*S*, *n*, *i*1, *i*0) равен максимальной длине монотонно возрастающей на 1 подпоследовательности в массиве *bp*, начинающейся в его позиции *i*0 со значения *i*1 + 1.

Доказательство. Вначале напомним, что возрастание последовательно расположенных значений в массиве *bp* возможно не более чем на 1 (см. разд. 2, замечание 1).

Согласно свойствам 1—2 из разд. 2, наличие положительной возрастающей последовательности в массиве *bp*, начинающейся в некоторой позиции *k* и имеющей длину *m*, означает, что символы исходной строки *S* с индексами *k*, *k* + 1, ..., *k* + *m* − 1 попарно совпадают с соответствующими символами в позициях *bp*[*k*] − 1, *bp*[*k*], *bp*[*k*] + 1, ..., *bp*[*k*] + *m* − 2.

Если в позиции *i*0 − *i*1 исходной строки известна часть *Z*-блока, имеющая длину *i*1 ⩾ 0, то его увеличение на 1 с позиции *i*0 гарантировано при наличии грани *bp*[*i*0] = *i*1 + 1. Если же в *i*0 не окажется грани, имеющей размер *i*1 + 1, то *Z*-блок фиксируется как есть: его продолжения не существует.

Возможность дальнейшего продолжения рассматриваемого блока обосновывается индуктивно и в итоге определяется общей длиной возрастающей последовательности граней, начатой в *i*0 со значения *i*1 + 1.

Однако в общем случае допустима ситуация, когда в позиции *i*0 окажется грань большего размера, чем требуемое выше значение *i*1 + 1. Тогда в массиве *bp*, по определению хранящем наибольшие грани, элемент *bp*[*i*0] не даст полезной информации, ведь в позиции *i*0 может одновременно существовать и меньшая грань *i*1 + 1, дающая продолжение нашему *Z*-блоку. Покажем, что в контексте имеющихся в алгоритме *Prefix-Z-Values* двух вызовов функции *SubstrComp* это невозможно, т. е. наибольшая грань в позиции *i*0 всегда не превосходит *i*1 + 1.

Очевидно, что наличие большей грани в позиции *i*0 означает существование "раннего" (вычисленного на предыдущих шагах алгоритма) *Z*-блока, покрывающего эту позицию. Как отмечено выше, правую границу всех таких блоков [*l*, *r*) содержит переменная *r*.

Первый вызов функции *SubstrComp* (*S*, *n*, 0, *i*) (здесь *i*1 = 0, *i*0 = *i*) осуществляется в ситуации *r* ⩽ *i*, т. е. когда текущая позиция *i* не покрыта ни одним "ранним" *Z*-блоком. Поэтому большая грань, чем начинающаяся в той же позиции *i* и равная 1, невозможна. Второй вызов функции *SubstrComp* (*S*, *n*, *r* − *i*, *r*) (при *i*1 = *r* − *i*, *i*0 = *r*) также исключает присутствие упомянутой большей грани. Дело в том, что здесь в качестве позиции *i*0 фигурирует величина *r*, которая, будучи внешней правой границей всех "ранних" *Z*-блоков, не может быть покрыта ни одним из них.

Таким образом, утверждение леммы доказано.

Ниже приведен алгоритм непосредственного преобразования массива граней в массив *Z*-блоков. Его корректность обосновывается следующим фактом. Он получен из алгоритма *Prefix-Z-Values* заменой обоих вызовов *SubstrComp* на функцию *ValGrow* (*nArr*, *n*, *nVal*, *i*0). Эта функция вычисляет длину максимальной возрастающей серии в массиве *nArr* размера *n*, которая должна стартовать со значения *nVal* в позиции *i*0. Следовательно, согласно доказанной выше лемме, результат работы нового алгоритма совпадает с результатом исходного алгоритма *Prefix-Z-Values*.

BP-to-ZP (*bp*, *zp*, *n*)

```
{
   l = r = 0; zp[0] = 0;
   for (i = 1; i < n; ++i)
    {
      zp[i] = 0;
      if (i > = r)
      { // Позиция i не покрыта ранним
      // Z-блоком — Z[i] вычисляется от начала
      zp[i] = ValGrow (bp, n, 1, i); l = i;
      r = l + zp[i];
      }
      else
      { // Позиция i покрыта ранним Z-блоком —
      // он используется
       j = i - l;
       if (zp[j] < r - i) zp[i] = zp[j]; // Новый
       // Z-блок закончился левее раннего
       else { zp[i] = r - i + ValGrow (bp, n,
       r - i + 1, r); l = i; r = l + zp[i]; }
     }
   }
}
```

ValGrow (*nArr*, *n*, *nVal*, *i*0)

```
{
   nSeqLen = 0;
   while  (i0  <  n&&nVal++==nArr[i0++])
   ++nSeqLen;
   return nSeqLen;
}
```

Замечание 3. Время работы функций *ValGrow* и *SubstrComp* не только одинаково оценивается (*O*(*n*)), но и совпадает при соответствующих наборах фактических параметров. Действительно, их (по лемме равные) результаты совпадают с числом повторений единственного цикла *while*. Отсюда следует, что вычислительная сложность последнего алгоритма эк-

вивалентна сложности исходного алгоритма *Prefix-Z-Values* и соответственно равна $\Theta(n)$.

Далее рассмотрим возможности оптимизации алгоритма *BP-to-ZP*.

Как замечено выше, новая функция *ValGrow* работает с той же скоростью, что и функция *SubstrComp*, которая выполняла "наивные" последовательные сравнения символов подстрок. Такая ситуация не может быть признана удовлетворительной, ведь предварительное построение массива граней *bp* фактически содержит некоторое исследование структуры исходной строки. Почему бы не использовать его результаты?

Эти результаты выражены в следующем знании: исследуемая функцией *ValGrow* последовательность символов представляет собой отрезок арифметической прогрессии. Ее можно тестировать с произвольным шагом h, а не только равным 1 (как в функции *ValGrow*). В этих целях может быть использован тот факт, что в растущей на единицу арифметической прогрессии $A[0...n]$ при корректных индексах справедливо соотношение $A[k + h] = A[k] + h$.

Представленная далее функция решает такую задачу, получая в качестве параметра шаг перемещения по исследуемой последовательности.

ValGrowH(*nArr*, *n*, *nVal*, *i*0, *h*)

```
{
   nSeqLen = 0;
   if (nVal! = nArr[i0]) return nSeqLen;
   // "Крупные" перемещения с заданным шагом
   nPosHigh = n - h;
   while (i0 < nPosHigh)
   {
     nVal += h; i0 + = h; nSeqLen += h;
     if (nVal!= nArr[i0]) break;
   }
   // Уточнение позиции внутри последнего шага
   if (nVal == nArr[i0])
   {
     ++nSeqLen;
     while (++nVal == nArr[++i0]) ++nSeqLen;
   }
   else while (--nVal!=nArr[--i0])--nSeqLen;
   return nSeqLen;
}
```

Функция *ValGrowH* не дает качественного улучшения времени работы, сохраняя его линейную оценку. Однако она способна существенно повысить скорость работы алгоритма для случаев, когда в S существуют многочисленные грани большой длины.

Возникает вопрос, можно ли определить длину монотонной последовательности быстрее, чем за линейное время? Решение дает классика, а именно — алгоритм бинарного поиска [1]. На его основной идее построена следующая модификация функции *ValGrow*, время работы которой оценивается *логарифмически*.

ValGrow2 (*nArr*, *n*, *nVal*, *i*0)

```
{
   nLeft = i0 - 1; nRight = n; nL = nVal - i0;
   while (nLeft < nRight - 1)
   {
    nMid = (nLeft + nRight) >> 1;
    // Деление на 2 с помощью сдвига
    if (nL + nMid == nArr[nMid]) nLeft = nMid;
    else nRight = nMid;
   }
   return nLeft - i0 + 1;
}
```

Заключение

Рассмотрены две наиболее известные и эффективные схемы предварительного анализа и описания структуры строк — массивы граней и Z-блоков. Проведен сравнительный анализ их выразительных возможностей. Установлено, что указанные массивы имеют равные выразительные возможности в том смысле, что их непосредственное взаимное преобразование может быть реализовано за линейное время без непосредственного использования символов исходной строки.

В рамках исследования предложен и обоснован новый линейный алгоритм преобразования массива граней в массив Z-блоков. Рассмотрены некоторые модификации этого алгоритма, способные повысить скорость работы одного из компонентов до логарифмической оценки.

Полученные результаты могут быть (и были) применены при практическом исследовании больших символьных последовательностей, в частности, в области биоинформатики [7].

Список литературы

1. **Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К.** Алгоритмы: построение и анализ: пер. с англ. М.: Вильямс, 2016. 1328 с.

2. **Гасфилд Д.** Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология: пер. с англ. СПб: Невский диалект, 2003. 654 с.

3. **Смит Б.** Методы и алгоритмы вычислений на строках: пер. с англ. М: Вильямс, 2006. 496 с.

4. **Code** Review. Prefix- and *z*-functions in C++ (string algorithms). URL: https://codereview.stackexchange.com/questions/87349/prefix-and-z-functions-in-c-string-algorithms

5. **Codeforces.** Переход между *Z*- и префикс-функциями. URL: http://codeforces.ru/blog/entry/9612/

6. **Университет** ИТМО. *Z*-функция. URL: https://neerc.ifmo.ru/wiki/index.php?title=Z-функция

7. **Quispe-Tintaya W., Gorbacheva T., Lee M., Makhortov S. D., Popov V. N., Vijg J., Maslov A. Y.** Quantitative detection of low-abundance somatic structural variants in normal cells by high-throughput sequencing // Nature Methods. 2016. No. 13. P. 584—586.

On Expressive Possibilities of Schemes for Strings Structure Description

S. D. Makhortov, msd_exp@outlook.com, Voronezh State University, Voronezh, 394018, Russian Federation

Corresponding author:

Makhortov Sergey D., Head of Department of Applied and System Software, Voronezh State University, Voronezh, 394018, Russian Federation
E-mail: msd_exp@outlook.com

Received on March 22, 2018
Accepted on April 05, 2018

The algorithms for computation on strings find increasing use in various applied fields, such as word processing, data compression, cryptography, speech recognition, computational geometry, molecular biology. Their relevance has significantly increased in recent decades with the appearance of new subject areas that require efficient processing of giant character sequences. These areas, in particular, include computer vision, information search in global networks, computational genomics and some others.

Many effective algorithms for texts comparing and analyzing are based on preprocessing — a preliminary study of the structure of a string. As a result of this processing, some auxiliary construction is built, which allows one to significantly reduce the number of redundant character comparisons. At this stage, only one of the processed strings (for example, a sample or text) is usually examined, and the other string compared can remain unknown. It is important that preprocessing takes an acceptable amount of time. This is followed by the phase of direct search or analysis of substrings, on which the earlier obtained information about the string is used to reduce the work.

This article considers two of the most well-known and effective schemes for preliminary analysis and description of the strings structure — borders arrays (prefix-function) and Z-blocks (Z-function). The author carries out a comparative analysis of their expressive possibilities. He also proposes and justifies a new linear algorithm for transforming a borders array into an array of Z-blocks, discusses some of its modifications that can improve the efficiency of work.

The results obtained can be applied to practical study of large character sequences.

Keywords: *string, string matching, preprocessing, prefix-function, Z-function, complexity analysis*

For citation:

Makhortov S. D. On Expressive Possibilities of Schemes for Strings Structure Description, *Programmnaya Ingeneria,* 2018, vol. 9, no. 5, pp. 235—240.

DOI: 10.17587/prin.9.235-240

References

1. **Cormen T. H., Leiserson C. E., Rivest R. L., Stein C**. *Introduction to Algorithms*, Third Edition (3rd ed.), The MIT Press, 2009. 1312 p.

2. **Gusfield D.** *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology,* Cambridge University Press, New York, NY, USA, 1997. 550 p.

3. **Smyth B.** *Computing Patterns in Strings*, Addison-Wesley, UK, 2003, 440 p.

4. **Code** Review: Prefix- and z-functions in C++ (string algorithms), available at: https://codereview.stackexchange.com/questions/87349/prefix-and-z-functions-in-c-string-algorithms

5. **Codeforces:** Conversion between *Z*-function and Prefix-function, available at: http://codeforces.ru/blog/entry/9612/ (in Russian).

6. **ITMO** University: Z-function, available at: https://neerc.ifmo.ru/wiki/index.php?title=Z-функция (in Russian).

7. **Quispe-Tintaya W., Gorbacheva T., Lee M., Makhortov S. D., Popov V. N., Vijg J., Maslov A. Y.** Quantitative detection of low-abundance somatic structural variants in normal cells by high-throughput sequencing, *Nature Methods*, 2016, no. 13, pp. 584—586.

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

Сдано в набор 15.03.2018 г. Подписано в печать 25.04.2018 г. Формат 60×88 1/8. Заказ PI518
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**

ПУМСС-2018

3–6 сентября 2018 года

Самара, Россия

ПРОБЛЕМЫ УПРАВЛЕНИЯ И МОДЕЛИРОВАНИЯ В СЛОЖНЫХ СИСТЕМАХ

XX Международная конференция

УВАЖАЕМЫЕ КОЛЛЕГИ!

Федеральное государственное бюджетное учреждение науки Институт проблем управления сложными системами Российской академии наук в период с 3 по 6 сентября 2018 года проводит в городе Самаре очередную XX Международную научную конференцию «Проблемы управления и моделирования в сложных системах».

Целью конференции является повышение эффективности научных исследований в области решения проблем управления и моделирования в сложных системах путем обсуждения последних научных достижений и лучших практик их применения в различных областях.

Направления конференции:

- Методы управления и оптимизации в сложных технических системах
- Интеллектуальные технологии в сложных системах
- Измерения, контроль и диагностика в экстремальных условиях
- Процессы управления в обществе
- Цифровое сельское хозяйство и агрокибернетика
- Биокибернетика и биоинформатика

Формы участия:

- Обзорные и обобщающие доклады (30 минут)
- Доклады, посвященные частным вопросам (10 минут)

Языки конференции:

- Русский
- Английский

Дополнительная информация

ИПУСС РАН
ул. Садовая, 61, г. Самара, 443020
тел. +7 (846) 333 27 70
http://www.iccs.ru/cscmp/cscmp.html e-mail: cscmp@iccs.ru

По итогам конференции издается сборник трудов, индексируемый РИНЦ.

Издательство «НОВЫЕ ТЕХНОЛОГИИ»

выпускает *научно-технические журналы*

Теоретический и прикладной научно-технический журнал

ПРОГРАММНАЯ ИНЖЕНЕРИЯ

В журнале освещаются состояние и тенденции развития основных направлений индустрии программного обеспечения, связанных с проектированием, конструированием, архитектурой, обеспечением качества и сопровождением жизненного цикла программного обеспечения, а также рассматриваются достижения в области создания и эксплуатации прикладных программно-информационных систем во всех областях челове ческой деятельности.

Подписные индексы по каталогам:
«Роспечать» – 22765; «Пресса России» – 39795

Подписные индексы по каталогам:
«Роспечать» – 72656;
«Пресса России» – 94033

Ежемесячный теоретический и прикладной научно-технический журнал

ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

В журнале освещаются современное состояние, тенденции и перспективы развития основных направлений в области разработки, производства и применения информационных технологий.

Ежемесячный междисциплинарный теоретический и прикладной научно-технический журнал

НАНО- и МИКРОСИСТЕМНАЯ ТЕХНИКА

В журнале освещаются современное состояние, тенденции и перспективы развития нано- и микросистемной техники, рассматриваются вопрос разботки и внедрения нанои микросистем в различные области науки, технологии и производства.

Подписные индексы по каталогам:
«Роспечать» – 79493;
«Пресса России» – 27849

Подписные индексы по каталогам:
«Роспечать» – 79492;
«Пресса России» – 27848

Ежемесячный теоретический и прикладной научно-технический журнал

МЕХАТРОНИКА, АВТОМАТИЗАЦИЯ, УПРАВЛЕНИЕ

В журнале освещаются достижения в области мехатроники, интегрирующей механику, электронику, автоматику и информатику в целях совершенствования технологий производства и создания техники новых поколений. Рассматриваются актуальные проблемы теории и практики автомати ческого и автоматизированного управления техническими объектами и технологическими процессами в промышленности, энергетике и на транспорте.

Научно-практический и учебно-методический журнал

БЕЗОПАСНОСТЬ ЖИЗНЕДЕЯТЕЛЬНОСТИ

В журнале освещаются достижения и перспективы в области исследований, обеспечения и совершенствования защиты человека от всех видов опасностей производственной и природной среды, их контроля, мониторинга, предотвращения, ликвидации последствий аварий и катастроф, образования в сфере безопасности жизнедеятельности.

Подписные индексы по каталогам:
«Роспечать» – 79963;
«Пресса России» – 94032

Адрес редакции журналов для авторов и подписчиков:
107076, Москва, Стромынский пер., 4. Издательство "НОВЫЕ ТЕХНОЛОГИИ".
Тел.: (499) 269-55-10, 269-53-97. Факс: (499) 269-55-10. E-mail: antonov@novtex.ru

ISSN 2220-3397. Программная инженерия. 2018. Том 9. № 5. 193–240. Индексы: «Роспечать» – 22765, «Пресса России» – 39795