ИСП Институт Системного Программирования
Российской Академии наук

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

Труды Института Системного Программирования РАН

Proceedings of the Institute for System Programming of the RAS

Том 29, выпуск 3

Volume 29, issue 3

Москва 2017

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

Труды Института системного программирования РАН

Proceedings of the Institute for System Programming of the RAS

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

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

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

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

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

Труды ИСП РАН реферируются и/или индексируются в:

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

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

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

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

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

УДК004.45

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

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

Содержание

Table of Contents

О проблеме представления формальной модели политики безопасности операционных систем

П.Н. Девянин <peter_devyanin@hotmail.com>
Федеральное учебно-методическое объединение высших учебных заведений России по образованию в области информационной безопасности, г. Москва

Аннотация. В связи с начавшимся процессом внедрения ФСТЭК России «Требований безопасности информации к операционным системам» в работе анализируются пути выполнения требований функциональной компоненты ADV_SPM.1 «Формальная модель политики безопасности», в том числе по определению языка, глубины и детализации представления модели политики безопасности управления доступом и информационными потоками. При этом приводятся предложения по составу основных элементов модели, использованию для ее верификации инструментальных средств. Практическая возможность применения предлагаемых подходов рассматривается на примере представления описания и верификации МРОСЛ ДП-модели, как основы механизма управления доступом в ОССН Astra Linux Special Edition.

Ключевые слова: информационная безопасность; политики безопасности; формальные модели

DOI: 10.15514/ISPRAS-2017-29(3)-1

Для цитирования: Девянин П.Н. О проблеме представления формальной модели политики безопасности операционных систем. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр 7-16. DOI: 10.15514/ISPRAS-2017-29(3)-1

1. Введение

Хотя моделирование безопасности управления доступом и информационными потоками можно считать первым научным направлением, заложившим фундамент современной теории компьютерной безопасности [1, 2], в рамках которого уже разработаны десятки, если не сотни формальных моделей, а модель Белла-ЛаПадулы [3] более 40 лет назад была применена в качестве основы механизма управления доступом операционной системы (ОС) *Multics*, до сих пор научным сообществом, представителями регуляторов и разработчиков в области информационной безопасности в полной мере не определён ни сам термин «формальная модель политики безопасности», ни

тем более не сформировано чётких критериев наличия представления такой модели при сертификации средств защиты информации.

Эта проблема становится особенно актуальной с учётом начавшегося процесса внедрения ФСТЭК России «Требований безопасности информации к операционным системам» [4], в которых, а также в разработанных на их основе в соответствии с ГОСТ Р ИСО/МЭК 15408 [5] профилях защиты и заданиях по безопасности для некоторых, возможно, высоких классов защиты ОС, как предполагается, будут явно указаны требования функциональной компоненты ADV_SPM.1 «Формальная модель политики безопасности».

В описании функционального компонента ADV_SPM.1 указывается, что формальная модель должна быть изложена в формальном стиле (с использованием, например, математического языка), должно быть определено понятие «безопасность» для объекта оценки (ОО) и должно быть представлено формальное доказательство того, что ОО не может перейти в небезопасное состояние, а также должно быть продемонстрировано соответствие между какой-либо функциональной спецификацией, используемой ОО, и моделью. Кроме того, указывается, что испытательная лаборатория при выполнении соответствующей проверки должна руководствоваться п. 10.7.1 ГОСТ Р ИСО/МЭК 18045 [6]. Однако в нем не даётся содержательных пояснений, как выполнить данную проверку.

Таким образом, можно говорить о наличии проблемы определения языка, применяемых научных подходов для обоснования безопасности, критериев наличия представления формальной модели политики безопасности, обоснования корректной реализации модели непосредственно в программном коде механизма управления доступом при реализации новых требований безопасности информации к ОС, решение которой может потребоваться в самое ближайшее время.

Очевидно, что язык представления модели может быть либо математическим [2], либо формализованным [7]. Тем более, что уже существуют примеры использования таких языков при разработке формальной модели для отечественной защищённой операционной системы специального назначения (ОССН) *Astra Linux Special Edition* [8]. На математическом языке изложена мандатная сущностно-ролевая ДП-модель (МРОСЛ ДП-модели) [9], на формализованный язык (нотацию) *Event-B* (*Rodin Platform*) эта же модель не только переведена, но и верифицирована [10, 11].

Однако самым существенным вопросом, требующим ответа, по-видимому, здесь является определение достаточной «глубины» проработки формальной модели. Можно ли считать удовлетворительным, например, следующее «математическое» представление механизма управления доступом ОС в рамках модели в виде кортежа (*V, T*), где *V* – множество состояний системы, как-то задающее доступы (текущие доступы или права доступа) субъектов из множества S к объектам из множества *O*, а *T* – какая-то функция переходов системы из состояния в состояние, без какой-либо детализации?

Или довольно популярной среди разработчиков отечественных защищённых ОС до сих пор является модель Белла-ЛаПадулы. Можно ли признать её адекватной современным ОС и достаточной для представления в профиле защиты, например, механизма мандатного управления доступом, реализуемого в защищённых ОС, принадлежащих семейству *Linux*? При том, что в этой модели не содержится средств описания информационных потоков по времени, иерархии сущностей (адекватной файловым системам ОС), функционально ассоциированных с субъектами сущностей, мандатного контроля целостности, различий в условиях функционирования доверенных и недоверенных субъектов и др. Ответ, очевидно, отрицательный.

2. Требования к представлению формальной модели политики безопасности управления доступом

В связи с изложенным для удовлетворение требованиям компоненты ADV_SPM.1, обеспечения должной «глубины» и детализации целесообразно предложить следующие требования к представлению формальной модели политики безопасности управления доступом, которое должно включать описание на математическом и формализованном языке:

- Множеств учётных записей пользователей, субъектов, объектов (сущностей), устанавливающих классификацию элементов этих множеств, связи между этими множествами или внутри них функций (отношений), заданных на этих множествах отношений иерархии;
- Множеств реализуемых прав доступа и доступов субъектов к сущностям, используемых для задания прав доступа и доступов (непосредственно, с использованием групп, ролей, типов, атрибутов) множеств, функций (отношений);
- Решётки уровней целостности (для большинства современных ОС без мандатного контроля целостности трудно достичь необходимого уровня защищенности), используемых для задания уровней целостности учётных записей пользователей, субъектов, сущностей функций (отношений);
- Решётки уровней конфиденциальности (при необходимости реализации в ОС мандатного управления доступом), используемых для задания уровней доступа учётных записей пользователей и субъектов, уровней конфиденциальности сущностей функций (отношений);
- Множеств, функций (отношений), используемых для задания сущностей, функционально ассоциированных с доверенными субъектами или параметрически ассоциированных с учётными записями пользователей;
- Множеств, функций (отношений), используемых для задания

сущностей-контейнеров, доступ к содержащимся в которых сущностях субъектами может быть разрешён без учёта уровней целостности или без учёта уровней конфиденциальности (при необходимости) таких сущностей-контейнеров;

- Видов информационных потоков (как минимум по памяти), используемых для задания информационных потоков между сущностями и субъектами множеств, функций (отношений);
- Элементов состояний, моделирующей ОС абстрактной системы, используемых для этого множеств, функций (отношений);
- Условий предоставления субъектам прав доступа и доступов к сущностям или субъектам и условий выполнения иных правил преобразования состояний (команд, операций, функций перехода) над учётными записями пользователей, субъектами и сущностями (создание, удаление, переименование, получение параметров), заданных для этого специальных элементов ОС (привилегией, ролей, административных ролей);
- Условий возникновения информационных потоков, за счёт реализации субъектами доступов к сущностям или субъектами, или получения субъектами контроля над другими субъектами;
- Условий получения субъектами контроля над другими субъектами за счёт использования сущностей, функционально ассоциированными с субъектами или параметрически ассоциированными с учётными записями пользователей, и информационных потоков между ними;
- Правил преобразования состояний (команд, операций, функций перехода), моделирующей ОС абстрактной системы, включая параметры каждого правила, условия и результаты его применения. Как минимум должны быть описаны: правила администрирования (создания, удаления, переименования, изменения прав доступа, уровней целостности, доступа или конфиденциальности (при необходимости), получения параметров) учётных записей пользователей, субъектов и сущностей; правила предоставления доступов субъектов к сущностям и субъектам; правила создания информационных потоков и получения субъектами контроля над другими субъектами;
- Доказательства выполнения при применении (корректности задания) правил преобразования состояний (команд, операций, функций перехода), моделирующей ОС абстрактной системы: условий предоставления субъектам прав доступа и доступов к сущностям или субъектам; условий выполнения иных правил преобразования состояний (команд, операций, функций перехода) над учётными записями пользователей, субъектов и сущностей (создание, удаление,

переименование, получение параметров);

- Доказательства в рамках моделирующей ОС абстрактной системы того, что реализованный мандатный контроль целостности позволяет обеспечить защиту от несанкционированного изменения субъектом-нарушителем параметров или данных в сущностях, параметров или функциональности субъектов (захватить контроль над субъектом) с более высоким, чем у него уровнем целостности, и в результате нарушить целостность программно-аппаратной среды ОС;
- При необходимости реализации мандатного управления доступом доказательство в рамках моделирующей ОС абстрактной системы того, что реализованные мандатные контроль целостности и управление доступом позволяют обеспечить защиту от запрещённых информационных потоков (как минимум по памяти) от сущностей с более высоким уровнем конфиденциальности к сущностям с более низким уровнем конфиденциальности (защиту от информационных потоков «сверху-вниз»).

Кроме того, с учётом сложности такого представления формальной модели политики безопасности управления доступом целесообразно в соответствии с требованиями компоненты ADV_SPM.1 для формального доказательства того, что ОС не может перейти в небезопасное состояние, а также для демонстрации соответствия между какой-либо функциональной спецификацией, используемой ОС, и моделью требовать её верификации с применением инструментальных средств. Для этого представление модели должно включать описание:

- Основных функциональных возможностей, формализованного языка и порядка применения использованных для верификации модели политики безопасности управления доступом инструментальных средств;
- Представления модели политики безопасности управления доступом с использованием формализованного языка инструментальных средств верификации. При этом на формализованном языке должны быть выражены:
    - элементы состояний, моделирующей ОС абстрактной системы, используемые для этого множества, функции (отношения);
    - правила преобразования состояний (команды, операции, функции перехода), включая параметры каждого правила, условия и результаты его применения;
    - условия выполнения мандатного контроля целостности и мандатного управления доступом (при необходимости);
- Если формализованный язык инструментальных средств не может

точно выразить некоторые элементы модели политики безопасности управления доступом, то описание всех таких элементов и полуформальное обоснование того, что это не влияет на итоговый результат верификации;

- Результатов верификации модели политики безопасности управления доступом с использованием инструментальных средств верификации с указанием того, какие элементы модели были верифицированы в автоматическом режиме, а какие в полуавтоматическом (ручном) режиме;
- Результатов верификации с применением инструментальных средств выполнения следующих условий при применении (корректности задания) правил преобразования состояний (команд, операций, функций перехода), моделирующей ОС абстрактной системы:
    - условий предоставления субъектам прав доступа и доступов к сущностям или субъектам;
    - условий выполнения иных правил преобразования состояний (команд, операций, функций перехода) над учётными записями пользователей, субъектами и сущностями (создание, удаление, переименование, получение параметров);
- Результатов верификации с применением инструментальных средств того, что в рамках моделирующей ОС абстрактной системы реализованный мандатный контроль целостности позволяет обеспечить защиту от несанкционированного изменения субъектом-нарушителем параметров или данных в сущностях, параметров или функциональности субъектов (захватить контроль над субъектом) с более высоким, чем у него уровнем целостности, и в результате нарушить целостность программно-аппаратной среды ОС;
- При необходимости результатов верификации с применением инструментальных средств того, что в рамках моделирующей ОС абстрактной системы реализованные мандатные контроль целостности и управление доступом позволяют обеспечить защиту от запрещённых информационных потоков (как минимум по памяти) от сущностей с более высоким уровнем конфиденциальности к сущностям с более низким уровнем конфиденциальности (защиту от утечки конфиденциальных данных, от информационных потоков «сверху-вниз»).

3. Заключение

Возможность практического выполнения этих условий подтверждается опытом разработки и верификации МРОСЛ ДП-модели [9-11]. В

совокупности эти условия позволяют сформулировать чёткие, научно обоснованные критерии наличия представления формальной модели политики безопасности ОС в соответствии с требованиями функциональной компоненты ADV_SPM.1 при её включении в профиль защиты или задание по безопасности при сертификации средств защиты информации в рамках реализации новых требований ФСТЭК России [4].

Список литературы

[1]. Bishop M. Computer Security: art and science. ISBN 0-201-44099-7, 2002. 1084 p.

[2]. Девянин П.Н. Модели безопасности компьютерных систем. Управление доступом и информационными потоками. Учебное пособие для вузов. 2-е изд., испр. и доп. М.: Горячая линия — Телеком, 2013. 338 с.: ил.

[3]. Bell D.E., LaPadula L.J. Secure Computer Systems: Unified Exposition and Multics Interpretation. Bedford, Mass.: MITRE Corp., 1976. MTR-2997 Rev. 1.

[4]. Информационное сообщение об утверждении Требований безопасности информации к операционным системам от 18 октября 2016 г. No 240/24/4893/ ФСТЭК России. URL: http://fstec.ru/component/attachments/download/1051.

[5]. ГОСТ Р ИСО/МЭК 15408-2013. Информационная технология. Методы и средства обеспечения безопасности. Критерии оценки безопасности информационных технологий.

[6]. ГОСТ Р ИСО/МЭК 18045-2013. Информационная технология. Методы и средства обеспечения безопасности. Методология оценки безопасности информационных технологий.

[7]. Abrial J.-R. Modeling in Event-B: System and Software Engineering. Cambridge University Press, 2010.

[8]. Операционная система Astra Linux. URL: http://www.astra-linux.ru/.

[9]. П.В. Буренин, П.Н. Девянин, Е.В. Лебеденко и др.; Под редакцией доктора техн. наук П.Н. Девянина. Безопасность операционной системы специального назначения Astra Linux Special Edition. Учебное пособие для вузов. 2-е издание, стереотипное. М.: Горячая линия – Телеком, 2016. 312 с.

[10]. P.N. Devyanin, V.V. Kuliamin, A.K. Petrenko, A.V. Khoroshilov, I.V. Shchepetkov. Using Refinement in Formal Development of OS Security Model. In Lecture Notes in Computer Sciences #9609 "Perspectives of System Informatics: 10th International Andrei Ershov Informatics Conference", Springer International Publishing, 2016 pp. 107-115. DOI: 10.1007/978-3-319-41579-6_9.

[11]. Petr N. Devyanin, Alexey V. Khoroshilov, Victor V. Kuliamin, Alexander K. Petrenko, Ilya V. Shchepetkov. Comparison of Specification Decomposition Methods in Event-B. Programming and Computer Software, 2016, Vol. 42, No. 4, pp. 198–205 DOI: 10.1134/S0361768816040022

On the problem of representation of the formal model of security policy for operating systems

P.N. Devyanin <peter_devyanin@hotmail.com>
Federal Educational and Methodological Association of Higher Educational Institutions of Russia for Education in Information Security
Russia, Moscow

Abstract. In connection with the process of implementation by the Federal Service for Technical and Export Control of Russia "Information Security Requirements for Operating Systems", the work analyzes the ways of fulfilling the requirements of the functional component ADV_SPM.1 "Formal Security Policy Model", including defining the language, depth and detail of the presentation of the access control policy and information flows. Among other things, proposals are given on the composition of the main elements of the model, the use of tools for its verification. The practical possibility of applying the proposed approaches is considered by the example of the presentation of the description and verification of the mandatory entity-role security model for logical access control and information flows as the basis of the access control mechanism in the special-purpose operating system Astra Linux Special Edition.

Keywords: information security, security policies, formal models

DOI: 10.15514/ISPRAS-2017-29(3)-1

For citation: Devyanin P.N. On the problem of representation of the formal model of security policy for operating systems. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 3, 2017. pp. 7-16 (in Russian). DOI: 10.15514/ISPRAS-2017-29(3)-1

References

[1]. Bishop M. Computer Security: art and science. ISBN 0-201-44099-7, 2002. 1084 p.
[2]. Devyanin P.N. Security models for computer systems. Control of access and information flows. Textbook for higher schools. 2nd ed. M.: Goryatchaya liniya – Telecom, 2013. 338 p (in Russian)
[3]. Bell D.E., LaPadula L.J. Secure Computer Systems: Unified Exposition and Multics Interpretation. Bedford, Mass.: MITRE Corp., 1976. MTR-2997 Rev. 1.
[4]. Information message on the approval of information security Requirements for operating systems, October 18, 2016. No 240/24/4893/ FSTEK Russian. URL: http://fstec.ru/component/attachments/download/1051.
[5]. GOST R ISO / IEC 15408-2013. Security techniques. Evaluation criteria for IT security. (in Russian).
[6]. GOST R ISO / IEC 18045-2013. Information technology - Security techniques - Methodology for IT security evaluation (in Russian)
[7]. Abrial J.-R. Modeling in Event-B: System and Software Engineering. Cambridge University Press, 2010.
[8]. Operating system Astra Linux. URL: http://www.astra-linux.ru/ (in Russian).

[9]. P.V. Burenin, P.N. Devyanin, E.V. Lebedenko and others; Under the editorship of P.N. Devyanin. Security of the special-purpose operating system Astra Linux Special Edition. Textbook for high schools. 2nd edition, stereotyped. M .: Goryatchaya liniya – Telecom, 2016, 312 p. (in Russian)

[10]. P.N. Devyanin, V.V. Kuliamin, A.K. Petrenko, A.V. Khoroshilov, I.V. Shchepetkov. Using Refinement in Formal Development of OS Security Model. In Lecture Notes in Computer Sciences #9609 "Perspectives of System Informatics: 10th International Andrei Ershov Informatics Conference", Springer International Publishing, 2016, pp. 107-115. DOI: 10.1007/978-3-319-41579-6_9.

[11]. Petr N. Devyanin, Alexey V. Khoroshilov, Victor V. Kuliamin, Alexander K. Petrenko, Ilya V. Shchepetkov. Comparison of Specification Decomposition Methods in Event-B. Programming and Computer Software, 2016, Vol. 42, No. 4, pp. 198–205. DOI: 10.1134/S0361768816040022

Комплекс алгоритмов функционирования системы безопасного исполнения программного кода

А.В. Козачок <a.kozachok@academ.msk.rsnet.ru>
Е.В. Кочетков <e.kochetkov@academ.msk.rsnet.ru>
Академия Федеральной службы охраны Российской федерации,
302034, Россия, г. Орёл, ул. Приборостроительная, д. 35

Аннотация. В настоящей статье представлен комплекс алгоритмов, составляющих основу функционирования системы безопасного исполнения программного кода. Функциональным предназначением данной системы является проведение исследования произвольных исполняемых файлов операционной системы в условиях отсутствия исходных кодов с целью обеспечения возможности контроля исполнения программного кода в рамках заданных функциональных требований. Представленный в работе комплекс алгоритмов включает в себя: алгоритм функционирования системы безопасного исполнения программного кода и алгоритм построения модели программы, пригодной для проведения верификации и сохраняющей свойства исходной программы.

Ключевые слова: алгоритм; вредоносное программное обеспечение; model checking; security automata

DOI: 10.15514/ISPRAS-2017-29(3)-2

Для цитирования: Козачок А.В., Кочетков Е.В. Комплекс алгоритмов функционирования системы безопасного исполнения программного кода. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 17-30. DOI: 10.15514/ISPRAS-2017-29(3)-2

1. Введение

В настоящее время становятся все более актуальными вопросы формального доказательства свойств безопасности систем. Математическое доказательство свойств безопасности позволяет доверять исследуемым объектам, но его применение возможно только к ограниченному классу объектов, а именно – позволяющих построить формализованную модель с сохранением важных для проведения исследования свойств. Актуальной задачей обеспечения информационной безопасности сетей и объектов критической информационной инфраструктуры (КИИ) является определение степени

доверия к объектам, приходящим из внешних источников [1]. Полное функциональное содержание таких объектов априорно неизвестно.

На текущий момент допуск к использованию во внутреннем контуре сети объектов недоверенного происхождения сводится к принятию решения на основе результатов антивирусного сканирования. Однако результаты тестирования средств антивирусной защиты (САВЗ) показывают, что вероятность пропуска цели достаточно высока [2]. Это связано с постоянным совершенствованием возможностей вредоносного программного обеспечения (ВПО) злоумышленниками и принципиальными ограничениями заложенных в САВЗ методов обнаружения [3]. Авторами предлагается подход к оценке степени безопасности функционирования программного кода в операционной системе (ОС) за счет построения модели программы и оценки ее соответствия заданным функциональным требованиям.

2. Предшествующие работы

В ранней работе авторов [4] предлагалось построение системы безопасного исполнения программного кода (СБИПК), являющейся расширенной композицией таких подходов к защите от ВПО, как применение метода формальной верификации "Model checking" и использование автомата безопасности для контроля над выполнением функциональных ограничений программы как на статическом, так и динамическом этапе исследования. Ключевой особенностью такой системы является построение модели безопасного исполнения программного кода (МБИПК). Под безопасностью исполнения в данном случае подразумевается выполнение ряда ограничений, выраженных на языке формальной логики и представляющих собой спецификацию, описывающую безопасное поведение программы [5].

Формальная верификация – это процесс математического доказательства соответствия определенной модели некоторым заданным свойствам (спецификации), основанный на строгих математических принципах [6]. При этом для описания свойств модели и задания требований к ней применяются специализированные языки, а для проведения верификации – специализированные программные средства – верификаторы. Одним из широко распространенных методов формальной верификации является метод "Model checking". Его суть состоит в построении некоторой модели реактивной системы (системы, способной менять свое состояние под воздействием внешних факторов); задании спецификации работы данной системы, отражающей свойства, выполнение которых требуется доказать; проведении процесса верификации. Результатом верификации является решение о согласованности модели заданной спецификации.

Вопросами обнаружения ВПО с использованием методов формальной верификации, в частности методом "Model checking", в разное время занимались такие ученые как J. Kinder, Song Fu, P. Beaucamps, A. Holzer [7–12].

Общим направлением перечисленных исследований является применение метода "Model checking" для построения модели ВПО, которая позволит в будущем обнаруживать семейства вредоносных программ данного вида. Недостатком данного подхода является принципиальная невозможность построения универсальной модели, которая бы позволила обнаруживать штаммы различных семейств ВПО.

Важным этапом при проведении такого исследования является построение модели, пригодной для проведения формальной верификации. Такой моделью является модель Крипке [6], которая описывается четверкой элементов:

$$(S, s_0, R, L), \tag{1}$$

где S – конечное множество состояний; $s_0 \subseteq S$ – начальное состояние; $R \subseteq S \times S$ – отношение переходов, которое должно быть тотальным, т.е. для каждого состояния $s \in S$ должно существовать такое состояние $s' \in S$, что имеет место $R(s, s')$; $L : S \to 2^{AP}$ – функция оценки состояния $s \in S$.

Структура Крипке представляет собой модель, которая может быть верифицирована на соответствие спецификации, выраженной с использованием операторов темпоральной логики. Для применения метода формальной верификации "Model checking" в СБИПК требуется построить модель, позволяющую произвести такую верификацию.

Состояние в модели Крипке – это мгновенное значение всех рассматриваемых в рамках модели переменных. Попытка применения данного подхода к описанию поведения бинарной программы приводит к «комбинаторному взрыву», который вызван чрезмерно большим количеством различных значений рассматриваемых переменных. Сокращение числа переменных позволит построить модель и применить к ней подход "Model checking". В работе [5] была введена модель функционирования процесса в ОС, которая основывается на принципе разделения всех взаимодействующих элементов ОС на субъекты и объекты. К субъектам относятся процессы (множество P), совершающие действия по отношению к объектам (множество O). Объектами являются ресурсы ОС и процессы, в отношении которых совершаются действия субъектами. Все объекты были разделены на следующие категории: "Процесс", "Оперативная память", "Внешняя память", "Периферийные устройства", "Сетевая подсистема". Действия субъектов по отношению к объектам были выделены следующие: $create$, $open$, $delete$, $read$, $write$. Согласно [5] поведение процесса в ОС можно описать последовательностью действий $\{s_i\}_{i \in \mathbb{N}}$, выполняемых субъектом ${p_i}^{k_p}$, каждое из которых в операторном виде можно представить как:

$${p_i}^{k_p} \xrightarrow{a_o} {o_y}^{k_o}, \tag{2}$$

где ${p_i}^{k_p}$ – субъект категории k_p с идентификатором i; a_o – некоторое действие субъекта по отношению к объекту; ${o_y}^{k_o}$ – некоторый объект категории k_o с идентификатором y.

3. Комплекс алгоритмов

Комплекс алгоритмов безопасного исполнения программного кода включает в себя следующие алгоритмы:

- алгоритм функционирования системы безопасного исполнения программного кода, отличающийся применением расширенной композиции таких подходов к обнаружению ВПО, как формальная верификация методом "Model checking" и использование автомата безопасности;
- алгоритм преобразования бинарной исполняемой программы в структуру Крипке, позволяющую производить формальную верификацию на соответствие функциональным требованиям, отличающийся применением интеллектуального фаззера для увеличения степени покрытия бинарного кода исследуемой программы.

3.1 Алгоритм функционирования системы безопасного исполнения программного кода

Исходными данными для работы алгоритма функционирования СБИПК являются следующие параметры:

- B – исследуемый бинарный файл;
- FR – функциональные требования, заданные на формально логическом языке [13];
- L – функция оценки;
- $SCTT$ ("System Call Translate Table") – таблица соответствия системных вызовов ОС и действий процесса.

Функциональные требования по выполнению программы строятся на основе аксиом безопасного исполнения программного кода. Под аксиомой безопасного исполнения программного кода AX понимается описание параметров разрешенных действий $\{s_k\}_{k\in\mathbb{N}}$ для процесса ${p_i}^{k_P}$ во время его функционирования, позволяющее исключить потенциальную возможность выполнения программой вредоносных действий:

$$AX_j = \cup_k s_k, k \in \mathbb{Q}_j, \quad (3)$$

где $\mathbb{Q}_j$ – множество индексов безопасных действий для процесса ${p_i}^{k_P}$ при заданных функциональных требованиях FR_j.

Функция оценки $L(x)$ определяет, какое подмножество из множества атомарных предикатов AP является истинным в заданном состоянии.

Таблица соответствия системных вызовов ОС и действий процесса ($SCTT$) формируется на основе экспертных данных путем задания соответствия между системными вызовами и функциями декодирования их параметров. Данная таблица формируется отдельно для каждого семейства ОС, а также ее

версии, если при этом были внесены изменения в параметры вызовов системных функций. Структура $SCTT$ показана в выражении 4. Она представляет собой множество кортежей, первым элементом которого является наименование системного вызова, вторым – соответствующее действие в ОС, а третьим – объект, над которым совершается действие.

$$\left\{ \begin{array}{c} (name_1; action_1; object_1), \\ (name_2; action_2; object_2), \\ \dots \\ (name_N; action_N; object_N) \end{array} \right\} \quad (4)$$

Алгоритм преобразования системного вызова в действие процесса в ОС представлен на рис. 1. Первым этапом работы данной функции является получение категории отслеживаемого процесса путем вызова функции $getProcessCategory$ (шаг 2), которая согласно [5] может принимать следующие значения: "Системный процесс", "Привилегированный процесс", "Пользовательский процесс". Далее в цикле (шаги 3–10) происходит перебор элементов таблицы $SCTT$. В случае совпадения имени системной функции с именем записи в таблице $SCTT$ происходит присвоение переменной a значения соответствующего действия (шаг 5) и переменной o значения соответствующего объекта (шаг 6), а также декодирование параметров системной функции $getObjInfo$ (шаг 7), возвращающей категорию объекта k_o. Результатом работы функции в целом являются сведения, достаточные для формирования действия процесса в рамках модели функционирования процесса в ОС.

```
1  Function TranslateSysCall(sysCall, SCTT)
       Result: (k_p, a, o, k_o)
2      k_p = getProcessCategory()
3      foreach record ∈ SCTT do
4          if record.name = sysCall.name then
5              a ← record.action
6              o ← record.object
7              k_o ← getObjInfo(sysCall)
8              return (k_p, a, o, k_o)
9          end
10     end
11 return ∅
```

Рис. 1. Функция преобразования системного вызова в действие в ОС
Fig. 1. Function that transforms a system call into an action

Обобщенный алгоритм функционирования СБИПК представлен на рис. 2. Первым этапом работы алгоритма функционирования СБИПК является проверка соответствия исполняемого файла ряду требований (функция $checkMinCriterias$):

- программа не должна быть упакована, зашифрована или каким-либо иным образом защищена от анализа;
- в программе должны отсутствовать антиотладочные механизмы;
- бинарный файл программы должен быть предназначен для исполнения в ОС из семейства, для которого задана $SCTT$.

```
1  Function SecureCodeExec(B, FR, L, SCTT)
       Result: Decision
2      // Проверка ограничений
3      if checkMinCriterias(B) ≠ true then
4          return false
5      end
6      // Построение модели
7      M ← buildModel(B, SCTT, L)
8      // Статический этап
9      φ ← buildSpec(FR)
10     if Verify(M, φ) ≠ true then
11         return false
12     end
13     // Динамический этап
14     safeConfig ← buildSecurityAutomata(φ)
15     if SecurityAutomataMonitor(B, safeConfig, M) ≠ true then
16         return false
17     end
18 return true
```

Рис. 2. Алгоритм функционирования СБИПК
Fig. 2. Secure code execution system operation algorithm

Функция $checkMinCriterias$ (шаг 3) принимает на вход исполняемый бинарный файл B и возвращает результат проверки. Если бинарный файл B не удовлетворяет минимальным требованиям для проведения исследования, то он признается потенциально опасным и дальнейший анализ не производится. В этом случае результатом работы алгоритма является решение о недопуске исследуемого исполняемого файла к работе на объектах КИИ.

В случае успешного прохождения проверки на соответствие минимальным требованиям производится построение модели исполняемого файла в функции $buildModel$ (рис. 3). Результатом ее работы является структура Крипке, позволяющая производить формальную верификацию.

Следующим этапом работы алгоритма (шаги 9–12) является статическая верификация модели программы на соответствие требованиям безопасного исполнения в рамках заданных функциональных требований FR. Для реализации этого этапа необходимо преобразовать множество FR в

множество формул φ, задающих спецификацию безопасного исполнения. Данное преобразование производится функцией $buildSpec(\varphi)$ [13].

В функции $Verify$ производится формальная верификация на основе метода "Model checking" с использованием полученных ранее модели программы M и спецификации φ. Результатом работы данной функции является решение о соответствии модели и спецификации. В случае несоответствия – исполняемый файл B признается небезопасным, дальнейший анализ не производится.

Для контроля за исполнением функциональных требований и аксиом безопасности в процессе работы исполняемого файла строится автомат безопасности [14]. Для работы автомата безопасности необходимо задать его конфигурацию. Построение конфигурации автомата безопасности осуществляется функцией $buildSecurityAutomata$, принимающей в качестве аргумента спецификацию φ.

Функция $SecurityAutomataMonitor$ осуществляет подготовку и запуск исследуемого исполняемого файла B с отслеживанием его выполнения в рамках конфигурации безопасного исполнения $safeConfig$. Каждый вызов системной функции, производимый исполняемым файлом, изменяет состояние автомата безопасности. В случае перехода автомата в состояние, отсутствующее в модели, но не противоречащее спецификации, исполнение программы продолжается, а данная трасса добавляется во множество трасс исполнения T, полученных на этапе построения модели M. В случае перехода автомата в запрещенное состояние в рамках спецификации выполнение исполняемого файла приостанавливается.

Текущая трасса исполнения сохраняется для дальнейшего анализа. Таким образом, предлагаемый подход предполагает возможность исключения выполнения потенциально небезопасных действий. Если в течение выполнения функции $SecurityAutomataMonitor$ не возникло исключительных ситуаций, то есть во время работы программы все совершаемые ею последовательности системных вызовов не противоречили заданной конфигурации автомата безопасного исполнения, то функция возвращает значение "$true$", означающее, что анализируемый исполняемый файл соответствует модели безопасного исполнения программного кода.

3.2 Алгоритм преобразования бинарной исполняемой программы в структуру Крипке

Алгоритм преобразования бинарной исполняемой программы в структуру Крипке реализует функцию $buildModel$, функциональным предназначением которой является построение модели исполняемого файла на основе полученных методом фаззинга трасс исполнения. Результатом работы данной функции является четверка элементов, задающая модель Крипке и

позволяющая производить формальную верификацию методом "Model checking" (рис. 3).

```
1  Function buildModel(B, SCTT, L)
       Result: {S, R, L}
2      // Начальная инициализация
3      T ← ∅
4      R ← ∅
5      S ← {s_0}
6      // Выделение потока управления
7      D ← Dissasembler(B)
8      // Оценка степени покрытия кода
9      while Cover(D, T) ≤ C_min do
10         repeat
11             data ← Mutation(data, T)
12             trace ← Fuzzer(B, data)
13         until trace ∉ T
14         T ← T ∪ trace
15         C ← split(trace)
16         seq ← {s_0}
17         foreach c ∈ C do
18             sysCall ← ExtractSysCall(c)
19             if sysCall = ∅ then
20                 continue
21             end
22             s ← TranslateSysCall(sysCall, SCTT)
23             seq ← seq‖s
24             if s ∉ S then
25                 S ← S ∪ s
26             end
27         end
28         for i ← 0 to | seq | −2 do
29             s_1 ← seq[i]
30             s_2 ← seq[i + 1]
31             r ← (s_1, s_2)
32             if r ∉ R then
33                 R ← R ∪ r
34             end
35         end
36     end
37 return {S, R, L}
```

Рис. 3. Преобразование множества трасс исполнения программы в модель
Fig. 3. Converting a set of program execution traces into a model

На первом этапе (шаги 3–5) производится инициализация переменных начальными значениями: T – множество трасс исполнения программы; R – множество связей между действиями программы; S – множество состояний модели (по определению модели Крипке включает в себя начальное состояние s_0).

Следующим этапом (шаг 7) является статическое преобразование бинарного исполняемого файла в модель потока управления D, отражающую связи между элементарными блоками операций. D представляет собой направленный граф, вершинами которого являются условные операторы, а с каждым ребром ассоциирован блок элементарных операций.

Цикл построения модели программы повторяется до тех пор, пока истинно условие в выражении 5.

$$Cover(D,T) \leq C_{min}, \tag{5}$$

функция $Cover$ – определяет степень покрытия графа потока управления D трассами из множества T; C_{min} – задает минимальный порог уровня точности построения модели.

Каждая итерация данного алгоритма начинается с получения от фаззера [15] трассы исполнения программы $trace$ (шаг 12), представляющей собой последовательность ассемблерных инструкций. Следует отметить, что получаемая на очередном шаге цикла трасса исполнения программы должна отличаться от предыдущих, то есть не входить во множество T (шаг 13). Это обеспечивается за счет работы функции $Mutation$ (шаг 11), в основе работы которой лежит анализ множества уже пройденных трасс T и исходных данных $data$, подаваемых на вход программе B на предыдущей итерации. Затем трасса $trace$ добавляется ко множеству T (шаг 14).

Результатом работы функции $split$ (шаг 15) является множество последовательностей ассемблерных команд C, полученных из $trace$ путем разделения в местах вызова команды $call$. Правило преобразования данной функции представлено в выражении 6.

$$\begin{aligned} &split(\{a_1, a_2, \ldots, call_1, a_{N+1}, a_{N+2}, \ldots, call_2\}) = \\ &= \{\{a_1, a_2, \ldots, call_1\}, \{a_{N+1}, a_{N+2}, \ldots, call_2\}\}, \end{aligned} \tag{6}$$

где a_i – ассемблерная команда (отличная от $call$); $call_j$ – команда вызова подпрограммы.

По определению из [6], структура Крипке имеет начальное состояние s_0, из которого выходят все остальные ветви, поэтому оно добавляется ко множеству seq (шаг 16).

Важным этапом работы данного алгоритма является преобразование множества последовательностей ассемблерных команд C в системные вызовы

функцией $ExtractSysCall$ (шаг 18). Результатом ее работы в случае обнаружения системного вызова является структура, представленная в выражении 7, иначе – функция возвращает пустое множество.

$$\begin{cases} name, \\ params = \{(p_1, v_1), (p_2, v_2), \ldots, (p_N, v_N)\}; \end{cases} \quad (7)$$

где $name$ – имя системной функции; $params$ – параметры вызова системной функции.

После преобразования системного вызова в действие процесса в ОС посредством вызова функции $TranslateSysCall$ (шаг 22) переменная s содержит формальное описание действия в ОС.

Следующим этапом является конкатенация полученного действия процесса s и сформированной последовательности seq (шаг 23). Затем производится проверка наличия действия s во множестве S, и его добавление в случае отсутствия (шаг 24).

После прохождения по всем элементам списка C переменная seq содержит упорядоченную последовательность действий процесса, которую можно представить следующим выражением:

$$seq = \{s_0, s_1, s_2, \ldots, s_{N-1}\}.$$

Согласно определению модели Крипке, множество R содержит информацию о переходах между состояниями в виде двухместных кортежей. В каждом кортеже на первом месте находится текущее состояние, а на втором – состояние, в которое осуществляется переход:

$$R = \{(s_i; s_j), (s_f; s_z), \ldots, (s_x; s_y)\}. \quad (8)$$

Формирование множества R производится в цикле для i равного от 0 до $|\, seq \,| - 2$ (шаги 28–35). Из последовательности действий процесса выделяются два следующих друг за другом элемента (i , $i + 1$). На их основе формируется переход r и проверяется, входит ли он во множество R (шаг 31). Если такого перехода во множестве R нет, то он добавляется. Данная последовательность действий повторяется до тех пор, пока не будут пройдены все пары элементов. На этом обработка текущей трассы считается завершенной, и осуществляется проверка условия окончания цикла – оценка степени покрытия кода (шаг 9).

Результатом работы алгоритма является структура Крипке, позволяющая производить верификацию на предмет соответствия заданной спецификации, выраженной в виде темпоральной логики.

4. Заключение

В настоящей статье рассмотрен комплекс алгоритмов, лежащих в основе функционирования СБИПК. Подробно рассмотрен алгоритм функционирования СБИПК в целом, алгоритм преобразования бинарной

исполняемой программы в структуру Крипке. Применение предложенного комплекса алгоритмов для реализации СБИПК позволяет построить формальную модель объектов, в частности исполняемых файлов, приходящих из внешних источников и произвести их формальную верификацию с использованием метода "Model checking" и автомата безопасности. Данное исследование позволит допускать к использованию на объектах КИИ только программы, уровень доверия к которым достаточен для работы с ними.

Список литературы

[1] Козачок А. В., Бочков М. В., Фаткиева Р. Р., Туан Л. М. Аналитическая модель защиты файлов документальных форматов от несанкционированного доступа. Труды СПИИРАН, т. 43, № 6, 2015, стр. 228–252

[2] Anti-Virus Comparative Summary Report. 2017. URL: https://www.avcomparatives.org/wp-content/uploads/2017/02/avc_sum_201612_en.pdf

[3] Козачок А.В. Распознавание вредоносного программного обеспечения на основе скрытых марковских моделей: дис. канд. техн. наук: 05.13.19. Орел, 2012

[4] Козачок А.В., Кочетков Е.В. Обоснование возможности применения верификации программ для обнаружения вредоносного кода. Вопросы кибербезопасности, т. 16, № 3, 2016, стр. 25–32

[5] Козачок А.В., Кочетков Е.В. Формальная модель функционирования процесса в операционной системе. Труды СПИИРАН, т. 51, № 2, 2017, стр. 78–96

[6] Clarke Edmund M, Grumberg Orna, Peled Doron. Model checking. MIT press, 1999

[7] Kinder Johannes, Katzenbeisser Stefan, Schallhart Christian, Veith Helmut. Detecting malicious code by model checking. International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, 2005, pp. 174–187.

[8] Song Fu, Touili Tayssir. Pushdown model checking for malware detection.International Journal on Software Tools for Technology Transfer, vol. 16, no. 2, 2014, pp. 147–173.

[9] Song Fu, Touili Tayssir. Efficient malware detection using model-checking. International Symposium on Formal Methods. Springer, 2012, pp. 418–433

[10] Song Fu, Touili Tayssir. PoMMaDe: pushdown model-checking for malware detection. Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, 2013, pp. 607–610

[11] Beaucamps Philippe, Gnaedig Isabelle, Marion Jean-Yves. Abstraction-based malware analysis using rewriting and model checking. European Symposium on Research in Computer Security. Springer, 2012. pp. 806–823

[12] Holzer Andreas, Kinder Johannes, Veith Helmut. Using verification technology to specify and detect malware. Computer Aided Systems Theory–EUROCAST. 2007, pp. 497–504

[13] Kozachok A.V., Bochkov M.V., Lai M.T., Kochetkov E.V. First Order Logic For Program Code Functional Requirements Description. Вопросы кибербезопасности, т. 21, № 3, 2017, стр. 2–7

[14] Schneider Fred B Enforceable security policies. ACM Transactions on Information and System Security (TISSEC), vol. 3, no. 1, 2000, pp. 30–50.

[15] Jesse Hertz Project Triforce: Run AFL on Everything! 2016. URL: https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/.

Secure code execution system operation algorithm

A.V. Kozachok <a.kozachok@academ.msk.rsnet.ru>
E.V. Kochetkov <e.kochetkov@academ.msk.rsnet.ru>
Academy of Federal Guard Service,
35, Priborostroitelnaya st., Oryol, 302034, Russia

Abstract. The article presented a set of algorithms that form the basis of the safe code execution system. The functional purpose of it is to investigate arbitrary executable files of the operating system in the absence of source codes in order to provide the ability to control the execution of the program code within the specified functional requirements. The set of algorithms presented in this work includes: the algorithm for the functioning of a system for the safe execution of the program code; the algorithm for constructing a program model suitable for verification, which accurately preserves the properties of the source program.

Keywords: algorithm; malware; model checking; security automata

DOI: 10.15514/ISPRAS-2017-29(3)-2

For citation: Kozachok A.V., Kochetkov E. V. Secure code execution system operation algorithm . Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 3, 2017, pp. 17-30 (in Russian). DOI: 10.15514/ISPRAS-2017-29(3)-2

References

[1] Kozachok A.V., Bochkov M.V., Fatkieva R.R, Tuan L.M. Analytical Model for Protecting Documentary File Formats from Unauthorized Access. SPIIRAS Proceedings, vol. 43, no. 6, 2015, pp. 228–252 (in Russian)
[2] Anti-Virus Comparative Summary Report. 2017. URL: https://www.avcomparatives.org/wp-content/uploads/2017/02/avc_sum_201612_en.pdf
[3] Kozachok A.V. Detection of malicious software based on hidden Markov models: PhD thesis. Oryol, 2012, 209 p. (in Russian)
[4] Kozachok A.V., Kochetkov E.V. Using Program Verification for Detecting Malware. Cybersecurity issues, vol. 16, no. 3, 2016, pp. 25–32 (in Russian)
[5] Kozachok A.V., Kochetkov E.V. Formal model of functioning process in the operating system. SPIIRAS Proceedings, vol. 51, no. 2, 2017, pp. 78–96 (in Russian)
[6] Clarke Edmund M, Grumberg Orna, Peled Doron. Model checking. MIT press, 1999.
[7] Kinder Johannes, Katzenbeisser Stefan, Schallhart Christian, Veith Helmut. Detecting malicious code by model checking. International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer. 2005, pp. 174–187.
[8] Song Fu, Touili Tayssir. Pushdown model checking for malware detection. International Journal on Software Tools for Technology Transfer, vol. 16, no. 2, 2014, pp. 147–173.
[9] Song Fu, Touili Tayssir. Efficient malware detection using model-checking. International Symposium on Formal Methods. Springer, 2012, pp. 418–433
[10] Song Fu, Touili Tayssir. PoMMaDe: pushdown model-checking for malware detection. Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, 2013, pp. 607–610.

[11] Beaucamps Philippe, Gnaedig Isabelle, Marion Jean-Yves. Abstraction-based malware analysis using rewriting and model checking. European Symposium on Research in Computer Security. Springer. 2012, pp. 806–823.

[12] Holzer Andreas, Kinder Johannes, Veith Helmut. Using verification technology to specify and detect malware. Computer Aided Systems Theory–EUROCAST. 2007, pp. 497–504.

[13] Kozachok A.V., Bochkov M.V., Tuan L.M., Kochetkov E.V. First Order Logic For Program Code Functional Requirements Description. Cybersecurity issues, vol. 21, no. 3, 2017, pp. 2–7.

[14] Schneider Fred B. Enforceable security policies. ACM Transactions on Information and System Security (TISSEC), vol. 3, no. 1, 2000, pp. 30–50.

[15] Jesse Hertz. Project Triforce: Run AFL on Everything! 2016. URL: https: //www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/

О представлении результатов обратной инженерии бинарного кода ⋆

В.А. Падарян <vartan@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25
Московский государственный университет имени М.В. Ломоносова,
119991 ГСП-1, Москва, Ленинские горы

Аннотация. В статье рассматривается вопрос представления кода алгоритмов, извлекаемых из бинарного кода в рамках задачи обратной инженерии: как промежуточные представления для автоматического анализа, так и конечные представления, передаваемые пользователю. Разбираются две ключевых подзадачи в области обратной инженерии: автоматический поиск эксплуатируемых дефектов и выявление НДВ. Описана общая схема системы, реализующей автоматический поиск эксплуатируемых дефектов, указаны ключевые свойства промежуточного представления, позволяющего решать такую задачу с точки зрения эффективной генерации системы уравнений для SMT-решателя. Представлен тракт работы системы по выявлению НДВ, состоящий из трех этапов: локализация алгоритма, его представление в удобной для анализа форме и исследование его свойств. Для автоматизации первого этапа применяется построение статико-динамического представления, выделяются события уровня ОС и вызовы библиотечных функций, являющиеся «якорями», от которых отталкивается аналитик при локализации алгоритма. Дальнейшая поддержка локализации осуществляется за счет построения срезов и средств навигации. После того, как локализация алгоритма выполнена, дальнейшая работа разделяется на два направления: диалоговое построение компактного аннотированного представления алгоритма в виде блок-схемы и автоматизированное изучение свойств алгоритма в части определения декларированных и недекларированных потоков данных. Представление алгоритма в виде блок-схемы базируется на построении упрощенных моделей функций, учитывающих входные и выходные буферы, и автоматически выявляемыми связями по данным между буферами различных вызовов функций. Описан общий сценарий работы аналитика с подобной блок-схемой в контексте задачи выявления НДВ, основанный на аннотировании декларированных потоков данных и автоматическом выявлении недекларированных потоков. В завершение статьи приводится пример получаемого представления и перечисляются направления дальнейшей работы.

⋆ Работа поддерживается грантом РФФИ № 16-29-09632

Ключевые слова: бинарный код; комбинированный анализ; промежуточное представление.

DOI: 10.15514/ISPRAS-2016-29(3)-3

Для цитирования: Падарян В.А. О представлении результатов обратной инженерии бинарного кода. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 31-42. DOI: 10.15514/ISPRAS-2016-29(3)-3

1. Введение

Необходимость в анализе безопасности бинарного (исполняемого) кода возникает, когда требуется оценить фактические свойства программы: соответствие заявленным возможностям, отсутствие программных закладок, наличие дефектов и возможности их эксплуатации. Проведение анализа на уровне бинарного кода обусловлено рядом причин. Исходный код может быть потерян или недоступен. Характерные примеры такой ситуации – наследственные системы с частично или полностью утраченной документацией, сторонние библиотеки, вредоносное ПО. Оптимизирующие преобразования кода, проводимые современными компиляторами, способны внести серьезные и потенциально эксплуатируемые дефекты, когда сталкиваются с определенными аспектами поведения программы, выходящими за рамки спецификации языка программирования [1].

Волна исследовательских работ последних 10 лет [2, 3, 4] предлагала новый подход к исследованию безопасности исполняемого кода. Основой разработанных методов стали компиляторные технологии, примененные в обратном направлении, когда входными данными оказывается исполняемый файл, низкоуровневые команды транслируются в промежуточное представление, которое затем анализируется с привлечением классических компиляторных алгоритмов, таких как распространение констант, достижимые определения и др.

Проводимое в 2016 году соревнование Darpa Grand Cyber Challenge [6] выявило победителей, работы которых [5, 4] реализуют приведенный выше подход.

В ИСП РАН на протяжении ряда лет разрабатывается среда анализа бинарного кода [7], базирующаяся на комбинированном подходе, сочетающем динамический и статический анализ. Методы анализа и программные инструменты, реализующие этот подход, широко задействуют компиляторные технологии. В статье рассматриваются некоторые разработанные представления, как промежуточные, рассчитанные на автоматический анализ, так и передаваемые пользователю для интерактивной работы.

2. Выявление эксплуатируемых дефектов

Технологии автоматизированного поиска дефектов в последние годы качественно развились, претендуя на применение в промышленности.

Основанием для развития стали два подхода: символьное выполнение на уровне бинарного кода и фаззинг. Типовой механизм символьного выполнения совмещает конкретное и символьное состояние программы. Символьные значения заводятся только для тех переменных, на которые способны повлиять входные данные, что способствует сохранению относительно небольшого числа переменных и компактности выражений над ними. Отслеживание влияния со стороны входных данных сводится к классической задаче анализа помеченных данных, для которой по-прежнему актуальны проблемы избыточной и недостаточной помеченности.

Поиск дефекта средствами символьной интерпретации заключается проверке совместимости системы условий (уравнений и неравенств) над символьными переменными и нахождением ее решения. Система состоит из предиката пути, задающего условия выполнения определенной последовательности команд, и предиката безопасности, описывающего срабатывания некоторого программного дефекта.

Можно утверждать, что уже устоялась архитектура такого средства поиска дефектов (рис. 1). Выполнение программы, сопровождающееся поддержкой символьного состояния, обеспечивается средствами бинарной инструментации (Valgrind [8], Pin [9], QEMU [10]). Та часть исполняемого кода, которая обрабатывает символьные данные, транслируется в архитектурно независимое представление низкого уровня. Можно описать его как код RISC-машины с минимизированными или полностью исключенными побочными эффектами. Следует отметить, что возможна ситуация и с обратным порядком, когда весь код транслируется в промежуточное представление и уже на нем ведется отслеживание символьных значений.

Операции над символьными переменными приводят к обновлению символьного состояния. Условные переходы формируют предикат пути – систему ограничений над символьными переменными. В целях упрощения последующего решения предикаты формируют в рамках теории QFBV. Предикат пути, дополненный предикатом безопасности, передают SMT-решателю. Возможности современных SMT-решателей, таких как Z3 [11], позволяют на настольном компьютере за приемлемое время оценивать совместность системы из нескольких десятков тысяч формул.

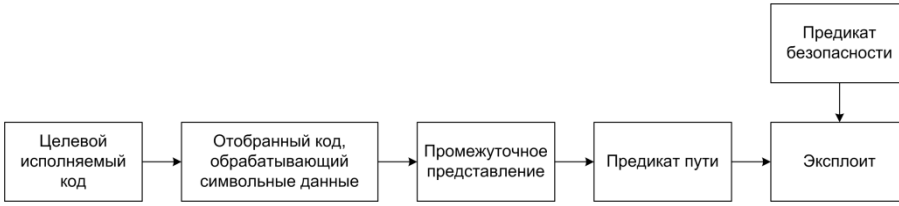


Рис. 1. Последовательность преобразований исполняемого кода при поиске уязвимостей средствами символьного выполнения

Fig. 1. Sequence of transformations of executable code when searching for vulnerabilities by means of symbolic execution

Ключевая особенность описанной архитектуры – трансляция в промежуточное представление (BAP [3], VINE [2], Pivot [12], REIL [13] и др.), которое, затем транслируется в SMT-формулы. Характерная особенность всех упомянутых представлений – крайне малое числом команд, что позволяет задать краткие правила построения SMT-формул. Представление Pivot, например, содержит всего 8 различных операторов. Таким образом, поддержка новой процессорной архитектуры ограничивается разработкой транслятора в промежуточное представление.

Открытой проблемой остается формулировка предикатов безопасности для актуальных видов дефектов [14]. В подавляющем числе открытых публикаций приводятся предикаты, описывающие конкретный механизм эксплуатации дефекта, неприменимый к современным промышленным программам. В научных статьях популярны такие дефекты, как переполнение буфера на стеке или контролируемая форматная строка. При этом игнорируются повсеместно применяемые защитные механизмы, такие как ASLR, DEP, противодействие переполнению буфера со стороны компилятора. Без учета этих факторов точное ранжирование опасности найденных дефектов невозможно.

После подготовки анализируемых материалов поиск программных дефектов не требует участия человека. Тем не менее, для окончательной оценки опасности найденного дефекта необходим ручной экспертный анализ.

3. Выявление НДВ

Возможности автоматизации для поиска программных закладок гораздо скромнее. Исследователь вынужден заниматься ручным анализом бинарного кода, который условно разделяется на три этапа. Необходимо локализовать алгоритм в общей массе кода, представить его в удобной форме и провести исследование свойств. Полная автоматизация первого и последнего этапа невозможна, требуется поддержка действий аналитика.

В ИСП РАН были получены результаты [7], продемонстрировавшие возможность предварительного повышения уровня представления бинарного кода, в рамках комбинированного анализа. По трассам выполнения восстанавливается статическое представление программ с сохранением связи между инструкциями представления и шагами трассы. Предложенный подход не накладывает требований по априорным знаниям об окружении (устройство ОС, язык и система программирования), в котором работает анализируемый алгоритм. Для поддержки анализа кода новой платформы будет достаточно знания о семантике инструкций процессорной архитектуры и карты адресного пространства. Разработанные программные средства позволяют восстановить события уровня ОС и системы программирования, облегчают навигацию. В полученном статико-динамическом представлении выявляются зависимости по управлению, что способствует более точному восстановлению потоков данных и управления.

После проведения предварительного повышения уровня представления трасса размечается, становится доступен поиск вызовов функций, в месте вызова у библиотечных функций восстанавливаются значения параметров. Автоматизированная навигация по вызовам библиотечных функций позволяет найти в трассе выполнения «якорь»: как правило, это некоторая функция ввода/вывода, через нее передаются результаты работы искомого алгоритма.

Посредством отслеживания потока данных в обратном направлении (обратного слайса трассы) выделяется подпоследовательность шагов, участвующих в выработке результата, эти шаги будут отнесены к искомому алгоритму.

Поиск НДВ затрагивает определенные алгоритмы, реализованные в программе; их выбор обусловлен проводимыми тематическими исследованиями. Как правило, выбираются алгоритмы, работающие с чувствительными данными. Для каждого выбранного алгоритма требуется определить, как вырабатывались результаты, какие входные данные были для этого использованы, где именно были размещены результаты, не сопровождалось ли это утечкой чувствительных данных. Помимо того, преобразованиям, проводимым над входными данными необходимо дать некоторую интерпретацию, выраженную в виде аннотации на естественном языке. Поскольку вопрос оценки семантики кода сводится к определению эквивалентных функций, в общем случае он неразрешим. Выработка аннотации остается за человеком. Однако выявление входных и выходных данных, определение зависимостей между ними успешно автоматизируются.

Наиболее близкой работой по высокоуровневому представлению алгоритма является гибридный граф потока данных и управления (Hybrid Information and Control Flow Graph, HI-CFG) [15]. HI-CFG содержит вершины двух типов: вершины, соответствующие фрагментам кода программы, и вершины, соответствующие некоторым структурам данных. Ребра, связывающие вершины первого типа, отражают последовательность передачи управления, а ребра, связывающие вершины второго типа, отражают зависимости по данным. Кроме того, в графе есть специальные виды ребер *производитель*, связывающие участок кода с порождаемой им структурой данных, и *потребитель*, связывающие структуры данных с использующими их участками кода. HI-CFG поддерживает различные уровни детализации: вершины первого типа могут представлять как базовые блоки кода, так и функции, вершины второго типа – как отдельные ячейки памяти, так и крупные ее области.

В отличие от HI-CFG, разработанное в ИСП РАН представление изначально рассчитано на то, что работа с ним будет вестись по двум направлениям. Первое – диалоговое построение компактного аннотированного представления в виде иерархической блок-схемы. Второе – автоматизированное изучение свойств алгоритма в части определения декларированных и недекларированных потоков данных. Помимо того, в

представление включается информация о восстановленном интерфейсе структурных компонент алгоритма – так называемых моделей функций.

Построение представления начинается с восстановления графа зависимостей по данным для подмножества команд трассы, относящихся к описываемому алгоритму. Граф – двудольный, вершины в нем – выполнявшиеся команды и использующиеся ими на запись и чтение ячейки памяти и регистры. Группировка узлов графа управляется посредством проекции на него восстановленного дерева вызовов.

Для описания семантики блоков кода, выполняющихся в исследуемой программе, используется понятие моделей. Модель функции – это её формальное упрощенное описание, позволяющее сгруппировать ячейки данных, с которыми взаимодействует функция, в несколько блоков данных (буферов), соответствующих входным и выходным параметрам функции.

Использование моделей функций позволяет аналитику описывать семантику отдельных функций и их параметров. При этом потоки данных между параметрами экземпляров моделей отслеживаются автоматически.

Начальным источником аннотаций выступает набор ранее описанных моделей библиотечных функций. Более формально под моделью понимается полностью восстановленная функция, описываемая следующими атрибутами: имя функции, модуль, в котором она располагается, смещение команды точки входа в функцию от начала модуля, множество команд выхода из функции, задаваемое смещениями от начала модуля, множество именованных параметров, зависимости между параметрами. Каждый параметр, в свою очередь, описывается атрибутами: имя, тип (входной, выходной, входной/выходной), вырабатываемое значение. Последний атрибут представляет собой арифметическое выражение над адресуемыми ячейками памяти и регистрами. Значения регистров и ячеек памяти берутся в точке входа, точке выхода или в явно заданной команде, соответственно для всех типов параметров.

Модель строится либо вручную по результатам изучения функции, либо автоматизировано при наличии заголовочных файлов и документации к ним. На рис. 2 приведен снимок окна, используемого для управления моделями функций. На снимке выделена модель функции CreateFileW, размещенной в динамической библиотеке kernel32. Функция оперирует четырьмя параметрами: тремя входными (помечены зеленым) и одним выходным (помечен красным). Следует отметить, что один из параметров (fname), является указателем; имя fname используется для описания содержимого второго параметра – буфера памяти непосредственно содержащего имя файла.

Схема алгоритма включает в себя вершины кода, данных и вспомогательные вершины.

Вершины кода бывают трех типов:

- вызовы функций с известной семантикой, то есть вызовы функций, для которых в системе были заведены модели;

- вызовы функций, не имеющих моделей;
- более мелкие фрагменты трассы (блоки, разделенные инструкциями вызова функций и возврата).

Вершины данных соответствуют параметрам моделей.

Вспомогательные вершины (соответствуют началу трассы, концу трассы, либо некоторой позиции, на которой был завершен анализ потока данных).

При этом вершины данных (параметры вызова) связаны ребром с соответствующей этому вызову вершиной-моделью, а потоки данных между параметрами представлены либо ребром графа (в случае, если выходные данные одной функции с моделью непосредственно используются другой функцией с моделью), либо же подграфом-гамаком, содержащим вызовы функций без моделей.

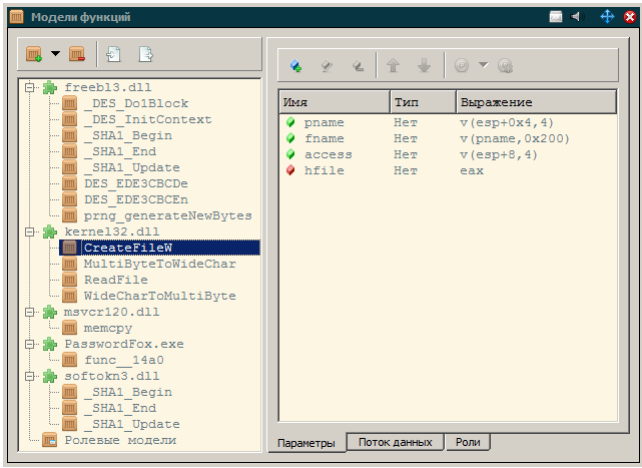


Рис. 2. Окно графического интерфейса, управляющее моделями функций
Fig. 2. The graphical interface window that controls the function models

В случае если какой-либо параметр формировался (использовался) исключительно функциями, не имеющими моделей, в качестве входной (выходной) вершины гамака будет использована вспомогательная вершина.

Особенность используемого представления заключается в том, что одному и тому же фрагменту трассы (вызову функции) может соответствовать несколько вершин, расположенных в разных гамаках.

Уже имеющиеся и дополнительно создаваемые модели применяются к графу зависимостей, сворачивая подграфы, соответствующие вызовам данной функции. В момент свертки выполняется сверка фактических и модельных потоков данных. Для каждого параметра модели выполняется вычисление его значения и происходит их сопоставление с фактическим множеством входных выходных данных. Модельные зависимости между входными и выходными параметрами сопоставляются с фактическими. Свернутый подграф

аннотируется именем модели, а его входные и выходные данные – именами параметров. Процесс изучения кода алгоритма ведется снизу-вверх и заканчивается тогда, когда построена и проверена объемлющая модель для всего отобранного кода.

На рис. 3 приведен фрагмент визуализированного графа, полученного при извлечении из исполняемого кода браузера Firefox алгоритма доступа к локальному хранилищу паролей. Образ исполняемого кода составлял 12МБ, из них 10МБ – системные библиотеки. В результате проведенного обратного слайсинга было отобрано 236 тыс. шагов трассы, при этом учитывались адресные зависимости. Для того чтобы отобранный код стал «обозримым», к представлению было применено 17 моделей функций. В слайсе было обнаружено и свернуто порядка 1500 вызовов функций, описываемых этими моделями. Результирующее представление на верхнем уровне было сведено к графу из 102 узлов.

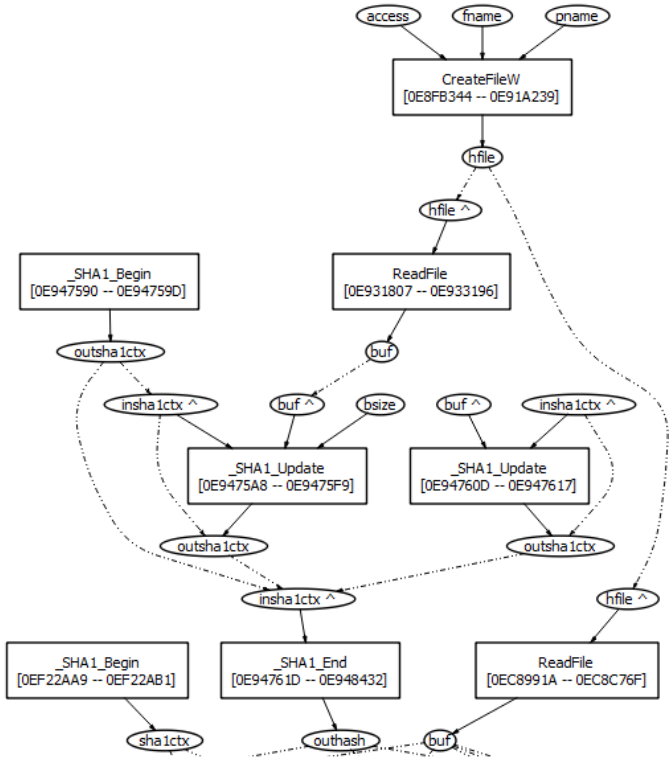


Рис. 3. Фрагмент визуализированного представления алгоритма
Fig. 3. Fragment of the visualized representation of the algorithm

4. Заключение

В работе описаны представления программ, используемые для решения двух основных задач компьютерной безопасности: поиска НДВ и уязвимостей. Представления были разработаны в рамках работ по созданию и развитию среды анализа бинарного кода.

Представления, используемые в поиске уязвимостей, полностью покрывают текущие потребности экспериментальной работы. Это позволило сдвинуть исследовательскую деятельность [14] в область разработки более точных методов оценки критичности найденных программных дефектов.

В части поиска НДВ в настоящее время ведется работа по преодолению ряда ограничений реализации инструмента, поддерживающего описанный способ анализа. В визуализируемом представлении будет добавлена поддержка зависимостей по управлению, а выражения, описывающие значение параметра, планируется развить, добавив возможность описания рекурсивных типов данных.

Список литературы

[1]. Wang X., Zeldovich N., Kaashoek M. F., Solar-Lezama A. A Differntial Approach to Undefined Behavior Detection. ACM Transactions on Computer Systems, 33(1), article 1, 2015, 29 p. DOI: 10.1145/2699678.

[2]. Song D., Brumley D., Yin H. et al. BitBlaze: A new approach to computer security via binary analysis. Information systems security, 2008, pp. 1-25.

[3]. Brumley D., Jager I., Avgerinos T. et al. BAP: A binary analysis platform. International Conference on Computer Aided Verification, 2011, pp. 463-469.

[4]. Shoshitaishvili Y., Wang R., Salls C. et al. Sok: (state of) the art of war: Offensive techniques in binary analysis. Security and Privacy (SP), 2016 IEEE Symposium on, 2016, pp. 138-157.

[5]. Cha S. K., Avgerinos T., Rebert A. et al. Unleashing mayhem on binary code. Security and Privacy (SP), 2012 IEEE Symposium on, 2012, pp. 380-394.

[6]. Defense Advanced Research Projects Agency Program Information: Cyber Grand Challenge (CGC). Доступно по ссылке: http://www.darpa.mil/program/cyber-grand-challenge, 01.06.2017.

[7]. В.А. Падарян, А.И. Гетьман, М.А. Соловьев, М.Г. Бакулин, А.И. Борзилов, В.В. Каушан, И.Н. Ледовских, Ю.В. Маркин, С.С. Панасенко. Методы и программные средства, поддерживающие комбинированный анализ бинарного кода. Труды ИСП РАН, том 26, вып. 1, 2014, стр. 251-276. DOI: 10.15514/ISPRAS-2014-26(1)-8.

[8]. Nethercote N., Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM SIGPLAN notices, 42(6), 2007, pp. 89-100.

[9]. Luk C. K., Cohn R., Muth R. et al. Pin: building customized program analysis tools with dynamic instrumentation. ACM SIGPLAN notices, 40(6), 2005, pp. 190-200.

[10]. Bellard F. QEMU, a fast and portable dynamic translator. USENIX Annual Technical Conference, FREENIX Track, 2005, pp. 41-46.

[11]. De Moura L., Bjørner N. Z3: An efficient SMT solver. Tools and Algorithms for the Construction and Analysis of Systems, 2008, pp. 337-340.

[12]. В.А. Падарян, М.А. Соловьев, А.И. Кононов. Моделирование операционной семантики машинных инструкций. Программирование, № 3, 2011, стр. 50-64.

[13]. Dullien T., Porst S. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. CanSecWest, 2009, 7 pp.

[14]. А.Н. Федотов, В.А. Падарян, В.В. Каушан, Ш.Ф. Курмангалеев, А.В. Вишняков, А.Р. Нурмухаметов. Оценка критичности программных дефектов в условиях работы современных защитных механизмо. Труды ИСП РАН, том 28, вып. 5, 2016, стр. 73-92. DOI: 10.15514/ISPRAS-2016-28(5)-4.

[15]. Caselden D., Bazhanyuk A., Payer M., McCamant S., Song D. HI-CFG: Construction by Binary Analysis and Application to Attack Polymorphism. In Computer Security – ESORICS 2013. Lecture Notes in Computer Science, vol 8134. Springer pp. 164-181

On representation used in the binary code reverse engineering

V.A. Padaryan <vartan@ispras.ru>
Institute for System Programming of Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia
Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia

Abstract. The paper discusses the problem of representation of algorithms extracted from binary code in course of reverse engineering: both representations for automatic analysis and final representations for the user. Two key subproblems of reverse engineering are focused on: automatic search for exploitable defects and discovery of undeclared capabilities. A principal scheme of system that allows automatically finding exploitable defects is described, along with key features of an internal representation employed by such system from the viewpoint of efficient generation of equations for an SMT solver. A sequence of steps for a system that reveals undeclared capabilities is enumerated: algorithm localization, its representation in a form suitable for analysis, and recovery of its properties. In order to automate the first step a static-dynamic representation is built which includes OS-level events and calls to library functions that serve as “anchor points” for the analyst in course of algorithm localization. Further support for localization is provided by means of code slicing and navigation algorithms. Once the algorithm is localized, further work goes in two directions: dialogue-based building of an annotated representation of the algorithm as a flowchart and automated research of characteristics of the algorithm in terms of declared and undeclared data flows. Flowchart representation of an algorithm is based on building simplified function models which describe input and output buffers, and automatic analysis of data flows between buffers of calls of different functions. The general scenario of interaction between an analyst and such a flowchart in context of the undeclared capability revealing problem is described, based on annotating declared data flows and automatically revealing undeclared ones. The paper concludes with an example of such a representation and an enumeration of further work directions.

Keywords: binary code; combined analysis; intermediate representation

DOI: 10.15514/ISPRAS-2017-29(3)-3

For citation: Padaryan V.A. Automated vulnerabilities exploitation in presence of modern defense mechanisms. *Trudy ISP RAN/Proc. ISP RAS*, vol. 29, issue 3, 2017. pp. 31-42 (in Russian). DOI: 10.15514/ISPRAS-2017-29(3)-3

References

[1]. Wang X., Zeldovich N., Kaashoek M. F., Solar-Lezama A. A Differntial Approach to Undefined Behavior Detection. ACM Transactions on Computer Systems, 33(1), article 1, 2015, 29 p. DOI: 10.1145/2699678.

[2]. Song D., Brumley D., Yin H. et al. BitBlaze: A new approach to computer security via binary analysis. Information systems security, 2008, pp. 1-25.

[3]. Brumley D., Jager I., Avgerinos T. et al. BAP: A binary analysis platform. International Conference on Computer Aided Verification, 2011, pp. 463-469.

[4]. Shoshitaishvili Y., Wang R., Salls C. et al. Sok: (state of) the art of war: Offensive techniques in binary analysis. Security and Privacy (SP), 2016 IEEE Symposium on, 2016, pp. 138-157.

[5]. Cha S. K., Avgerinos T., Rebert A. et al. Unleashing mayhem on binary code. Security and Privacy (SP), 2012 IEEE Symposium on, 2012, pp. 380-394.

[6]. Defense Advanced Research Projects Agency Program Information: Cyber Grand Challenge (CGC). Available at: http://www.darpa.mil/program/cyber-grand-challenge, accessed 01.06.2017.

[7]. V.A. Padaryan, A.I. Getman, M.A. Solovyev, M.G. Bakulin, A.I. Borzilov, V.V. Kaushan, I.N. Ledovskich, U.V. Markin, S.S. Panasenko. Methods and software tools for combined binary code analysis. Trudy ISP RAN/Proc. ISP RAS, vol. 26, issue 1, 2014, pp. 251-276 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-8.

[8]. Nethercote N., Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM SIGPLAN notices, 42(6), 2007, pp. 89-100.

[9]. Luk C. K., Cohn R., Muth R. et al. Pin: building customized program analysis tools with dynamic instrumentation. ACM SIGPLAN notices, 40(6), 2005, pp. 190-200.

[10]. Bellard F. QEMU, a fast and portable dynamic translator. USENIX Annual Technical Conference, FREENIX Track, 2005, pp. 41-46.

[11]. De Moura L., Bjørner N. Z3: An efficient SMT solver. Tools and Algorithms for the Construction and Analysis of Systems, 2008, pp. 337-340.

[12]. V.A. Padaryan, M.A. Solov'ev, A.I. Kononov. Simulation of Operational Semantics of Machine Instructions. Programming and Computer Software, 37(3), 2011, pp. 161-170. DOI: 10.1134/S0361768811030030.

[13]. Dullien T., Porst S. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. CanSecWest, 2009, 7 pp.

[14]. A.N. Fedotov, V.A. Padaryan, V.V. Kaushan, Sh.F. Kurmangaleev, A.V. Vishnyakov, A.R. Nurmukhametov. Software defect severity estimation in presence of modern defense mechanisms. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 5, 2016, pp. 73-92 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-4.

[15]. Caselden D., Bazhanyuk A., Payer M., McCamant S., Song D. HI-CFG: Construction by Binary Analysis and Application to Attack Polymorphism. In Computer Security – ESORICS 2013. Lecture Notes in Computer Science, vol. 8134. Springer pp. 164-181

ADV_SPM — Формальные модели политики безопасности на практике

[1,2,3,4]А.В. Хорошилов <khoroshilov@ispras.ru>,
[1]И.В. Щепетков <shchepetkov@ispras.ru>,
[1]ИСП РАН, 109004, Россия, г. Москва, ул. А. Солженицына, дом 25
[2]ВМК МГУ, 119991 ГСП-1 Москва, Ленинские горы,
[3]Московский физико-технический институт,
141700, Московская область, г. Долгопрудный, Институтский пер., 9
[4]НИУ ВШЭ, Россия, Москва, 101000, ул. Мясницкая, д. 20

Аннотация. В статье рассматривается семейство требований доверия к безопасности ADV_SPM «Моделирование политики безопасности», которое определяется стандартом ГОСТ Р ИСО/МЭК 15408-3-2013 «Критерии оценки безопасности информационных технологий. Часть 3. Компоненты доверия к безопасности». Обсуждаются задачи, решаемые этим семейством, и вопросы, которые возникают при попытке интерпретировать его требования. На простом примере представляется подход к формализации политик безопасности при помощи языка формальных спецификаций Event-B и инструментов платформы Rodin.

Ключевые слова: информационная безопасность; политики безопасности; формальные модели.

DOI: 10.15514/ISPRAS-2017-29(3)-4

Для цитирования: Хорошилов А.В., Щепетков И.В. ADV_SPM — Формальные модели политики безопасности на практике. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 43-56. DOI: 10.15514/ISPRAS-2017-29(3)-4

1. Введение

Стандарт ГОСТ Р ИСО/МЭК 15408 [1-3] применяется в качестве основы при проведении сертификации программного обеспечения, ответственного с точки зрения информационной безопасности. Программное обеспечение, подлежащее оценке, в стандарте обозначается как объект оценки (ОО).

Стандарт состоит из трёх частей:

- часть 1 описывает основные принципы обеспечения безопасности, предлагаемые стандартом;
- часть 2 содержит библиотеку функциональных требований безопасности;

- часть 3 содержит каталог требований доверия, которые определяют набор мероприятий, которые должны быть проведены в ходе разработки и оценки ОО, а также набор документов, которые должны быть сформированы в результате этих мероприятий.

Всего в третьей части стандарта ГОСТ Р ИСО/МЭК 15408-3-2013 [3] представлено 39 семейств требований доверия к безопасности, объединённых в 8 классов доверия.

2. Моделирование политики безопасности

В рамках данной статьи основным объектом рассмотрения является семейство требований доверия к безопасности ADV_SPM "Моделирование политики безопасности", являющаяся одним из элементов класса доверия ADV "Разработка" (рис. 1). Также для рассмотрения ADV_SPM нам потребуются элементы класса доверия ASE "Задание по безопасности", поэтому остановимся на этом классе поподробнее.

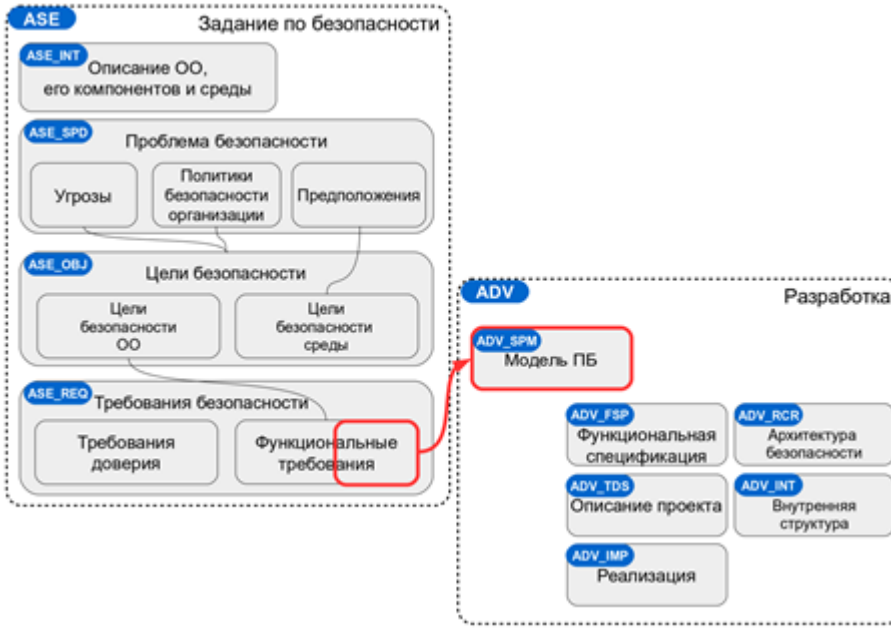


Рис. 1. Место семейства требований ADV_SPM
Fig. 1. Location of the ADV_SPM requirements family

Семейство требований ASE_INT "Введение в задание по безопасности" требует подготовки документа, содержащего описание ОО, его основных компонентов и среды функционирования.

Семейство ASE_SPD "Определение проблемы безопасности" требует явной формулировки проблемы безопасности, включающей в себя:

- описание угроз, с выделением:
  - источника угрозы,
  - способа реализации угрозы,
  - активов, подвергаемых опасности,
  - нарушаемых свойств безопасности (целостность, доступность, конфиденциальность);
- описание политик безопасности организации (ПБОр) — дополнительных требований, специфичных для организации или области применения ОО;
- предположения безопасности, т. е. предположения о среде функционирования ОО, существенные с точки зрения выполнения функций безопасности.

Семейство ASE_OBJ "Цели безопасности" требует определение целей безопасности с разделением их на два класса: отнесённых к ОО и отнесённых к среде. Для каждой цели требуется обоснование, которое сопоставляет её с идентифицированными угрозами, которым будет противостоять ОО, или с политикой безопасности организации, которая будет выполняться ОО. Цели безопасности среды также могут быть сопоставлены с предположениями безопасности. Также семейство ASE_OBJ требует проведение обобщающего логического анализа совокупности сформулированных целей безопасности, демонстрирующего способность противостоять всем идентифицированным угрозам безопасности и выполнять все установленные положения политики безопасности организации.

Семейство ASE_REQ "Требования безопасности" говорит о необходимости описания требований безопасности к ОО, разделяя их на два вида:

- функциональные требования безопасности ОО, которые, в частности, могут использовать компоненты функциональных требований, представленные во второй части стандарта;
- требования доверия к ОО, которые, как правило, составляются из компонентов требований доверия, описанных в третьей части стандарта.

При этом набор сформулированных функциональных требований должен обеспечить достижение всех целей безопасности, отнесённых к ОО. Для семейства требований доверия к безопасности ADV_SPM "Моделирование политики безопасности", функциональные требования безопасности ASE_REQ играют ключевую роль.

Хотя исторические корни термина "политика безопасности" находятся в политиках управления доступом, и формальные модели политик безопасности

появились именно как модели политик управления доступом, в определении семейства ADV_SPM стандарт использует этот термин в более широкой трактовке, которая предлагает рассматривать "политику безопасности" как произвольное множество логически связанных функциональных требований безопасности.

Таким образом, для одного объекта оценки может быть определено множество политик безопасности и, кроме того, одна формальная модель политики безопасности может описывать несколько политик безопасности. При этом формальная модель политики безопасности строится не только посредством формализации функциональных требований безопасности, но и при помощи представления в модели некоторых деталей реализации, необходимых для придания полноты описываемой картине на соответствующем уровне абстракции.

Целью семейства ADV_SPM стандарт заявляет приобретение дополнительного доверия посредством разработки формальной модели политики безопасности и установления соответствия между функциональной спецификацией ОО и этой моделью политики безопасности.

Для достижения этой цели стандарт требует от разработчика выполнения следующих мероприятий с использованием формальной модели политики безопасности (модели ПБ):

- формальное доказательство отсутствия внутренних противоречий в модели ПБ;
- определение понятия небезопасного состояния для каждой политики безопасности и формальное доказательство недостижимости небезопасных состояний;
- формальное доказательство соответствия между формальной функциональной спецификацией и моделью ПБ;
- демонстрация непротиворечивости и полноты функциональной спецификации относительно модели ПБ, а также относительно моделируемых политик безопасности.

Доказательство отсутствия внутренних противоречий помогает выявить неоднозначные, внутренне противоречивые и противоречащие друг другу элементы функциональных требований безопасности, а также позволяет повысить уровень доверия к корректности представления модели, поскольку ошибки и опечатки в описании сложных моделей не так легко выявить при помощи ручного анализа [4].

Доказательство недостижимости небезопасных состояний предназначено для демонстрации того, что моделируемый подход к реализации функциональных требований безопасности удовлетворяет этим требованиям.

Функциональная спецификация определяется в стандарте как описание интерфейса функций безопасности ОО, которое включает:

- описание всех способов, которыми пользователи могут вызвать ту или иную функцию безопасности;
- описание реакций на запросы пользователя;

но не включает описание реализации этих функций.

Демонстрация соответствия между формальной функциональной спецификацией и моделью ПБ должна показать, что функциональная спецификация является непротиворечивой и полной относительно модели ПБ.

3. Формальная модель политики безопасности

Наш практический опыт [5,6] в первую очередь приходится на построение формальной модели политики безопасности операционной системы специального назначения Astra Linux Special Edition, основанной на мандатной сущностно-ролевой ДП-модели управления доступом и информационными потоками (МРОСЛ ДП-модели [7]), и решении с помощью формальной модели первых двух задач: доказательства отсутствия внутренних противоречий и доказательства недостижимости небезопасных состояний.

Для этой цели использовался язык формальной спецификации Event-B [8], так как он позволяет описывать систему при помощи определения возможных наблюдаемых событий вне зависимости от того, где эти события происходят: внутри специфицируемой системы, вне её, или на границе между целевой системой и её окружением, в отличии от многих других языков формальной спецификации.

В соответствии с предложенных подходом формальная модель ПБ описывается в виде спецификации на Event-B, состоящей из следующих элементов:

- Состояние модели — набор переменных, отражающих текущее состояние как объекта оценки, так и его окружения. Состояние описывается как множество значений переменных модели.
- События модели — формальное представление значимых событий, происходящих как в объекте оценки, так и в его окружении. Семантика событий определяется как трансформация состояния модели. События формально описываются при помощи параметров, предусловий и постусловий.
- Инварианты консистентности — предикат над состоянием модели, описывающий требования к внутренней согласованности значений переменных состояния.
- Инварианты безопасности — предикат над состоянием модели, описывающий недостижимость небезопасного состояния для данной политики безопасности.

Верификация модели ПБО заключается в формальном доказательстве её консистентности и недостижимости небезопасных состояний, которое

выполняется при помощи доказательства сохранения всех инвариантов при любом событии модели.

Рассмотрим, как выглядит на примере модель ПБ на языке Event-B, формализующая функциональное требование безопасности FRU_PRS "Приоритет обслуживания", описанное во второй части стандарта (рис. 2).

15.2 Приоритет обслуживания (FRU_PRS)

15.2.1 Характеристика семейства

Требования семейства FRU_PRS позволяют ФБО управлять использованием находящихся под контролем ФБО ресурсов пользователями и субъектами так, что высокоприоритетные операции под контролем ФБО всегда будут выполняться без препятствий или задержек со стороны операций с более низким приоритетом.

15.2.2 Ранжирование компонентов

FRU_PRS.1 «Ограниченный приоритет обслуживания» предоставляет приоритеты для использования субъектами подмножества ресурсов под контролем ФБО.

FRU_PRS.2 «Полный приоритет обслуживания» предоставляет приоритеты для использования субъектами всех ресурсов под контролем ФБО.

15.2.3 Управление: FRU_PRS.1, FRU_PRS.2

Для функций управления из класса FMT могут рассматриваться следующие действия:

a) назначение приоритетов каждому субъекту в ФБО.

15.2.4 Аудит: FRU_PRS.1, FRU_PRS.2

Если в ПЗ/ЗБ включено семейство FAU_GEN «Генерация данных аудита безопасности», то следует предусмотреть возможность аудита следующих действий:

a) Минимальный: отклонение операции на основании использования приоритета при распределении ресурса.

b) Базовый: все попытки использования функции распределения ресурсов с учетом приоритетности обслуживания.

15.2.5 FRU_PRS.1 Ограниченный приоритет обслуживания

Иерархический для: Нет подчиненных компонентов.

Зависимости: отсутствуют

15.2.5.1 FRU_PRS.1.1

ФБО должны установить приоритет каждому субъекту в ФБО.

15.2.5.2 FRU_PRS.1.2

ФБО должны обеспечить доступ к [назначение: *управляемые ресурсы*] на основе приоритетов, назначенных субъектам.

15.2.6 FRU_PRS.2 Полный приоритет обслуживания

Иерархический для: FRU_PRS.1 Ограниченный приоритет обслуживания

Зависимости: отсутствуют.

15.2.6.1 FRU_PRS.2.1

ФБО должны установить приоритет каждому субъекту в ФБО.

15.2.6.2 FRU_PRS.2.2

ФБО должны обеспечить доступ ко **всем совместно используемым ресурсам** на основе приоритетов, назначенных субъектам.

Рис. 2. Требование безопасности FRU_PRS "Приоритет обслуживания"
Fig. 2. The requirements of the FRU_PRS "Priority of service" family

Модели (или спецификации) на языке Event-B состоят из компонентов двух типов: контекстов и машин. Контексты содержат статическую, неизменяемую часть модели: определения множеств и констант, а также аксиом, которые используются для описания свойств множеств и констант.

Контекст модели FRU_PRS (рис. 3) содержит определение множества приоритетов *P*, которое с помощью аксиом задаётся как подмножество множества натуральных чисел, состоящее из двух элементов — *High*, или

высокий приоритет, и *Low*, низкий приоритет. В свою очередь, *High* и *Low* определяются как константы, числа 1 и 0 соответственно.

```
CONTEXT
    C0  ›
CONSTANTS
○   P     ›Множество приоритетов
○   High      ›Высокий приоритет
○   Low   ›Низкий приоритет
AXIOMS
○   axm1:   P⊆ℕ not theorem ›
○   axm2:   High=1 not theorem ›
○   axm3:   Low=0 not theorem ›
○   axm4:   P={High, Low} not theorem ›
END
```

Рис. 3. Контекст модели
Fig. 3. The model context

В отличие от контекстов, машины содержат динамическую часть модели. При этом машины могут использовать определённые в контекстах константы и множества. В машинах определяются переменные, инварианты, события. Значения переменных формируют текущее состояние модели, а инварианты ограничивают его.

```
MACHINE
    M0  ›
SEES
○    C0
VARIABLES
○   S     ›Субъекты
○   SP    ›Функция приоритетов субъектов
○   O     ›Объекты
○   R     ›Текущие активные доступы
○   Q     ›Очередь
INVARIANTS
○   inv1:   S⊆ℕ not theorem ›
○   inv2:   SP∈S→P not theorem ›
○   inv3:   O⊆ℕ not theorem ›
○   inv4:   R∈S↔O not theorem ›
○   inv5:   Q∈S↔O not theorem ›

○   inv6:   ∀s,o·s∈S ∧ o∈O ∧ s↦o∈R ⇒ s↦o∉Q not theorem ›
○   inv7:   ∀s,o·s∈S ∧ o∈O ∧ s↦o∈Q ⇒ s↦o∉R not theorem ›
○   inv8:   ∀s1,s2,o·s1∈S ∧ s2∈S ∧ s1≠s2 ∧ o∈O ∧ s1↦o∈R ⇒ s2↦o∉R not theorem ›
○   inv9:   ∀sr,sq,o·sr∈S ∧ sq∈S ∧ o∈O ∧ sr↦o∈R ∧ sq↦o∈Q ⇒ SP(sr)≥SP(sq) not theorem ›
```

Рис. 4. Переменные и инварианты модели
Fig. 4. Variables and invariants of the model

В машине модели FRU_PRS (рис. 4) определены 5 переменных: множество субъектов *S*; функция приоритетов *SP*, которая ставит в соответствие каждому субъекту его приоритет; множество объектов *O*; множество пар *R* вида "субъект-объект", описывающее текущие активные доступы субъектов к объектам; множество пар *Q*, описывающее очередь на получение доступа. Типы данных переменных задаются в первых пяти инвариантах.

Остальные инварианты описывают следующие требования:

- inv6: каждая пара вида "субъект-объект", которая принадлежит множеству текущих активных доступов *R*, не может одновременно находиться в очереди на получение доступа *Q*;
- inv7: каждая пара вида "субъект-объект", которая находится в очереди *Q*, не может одновременно принадлежать множеству текущих активных доступов *R*;
- inv8: никакие два субъекта не могут одновременно иметь доступ к одному и тому же объекту;
- inv9: приоритет любого субъекта *sr*, который обладает текущим активным доступом к некоторому объекту *o*, должен быть выше приоритета любого субъекта *sq*, который находится в очереди на получение доступа к тому же объекту.

Текущее состояние модели может быть изменено событием. Каждое событие атомарно и обычно состоит из имени, параметров, охранных условий, и действий. Охранные условия являются набором предикатов, которые ограничивают набор возможных состояний модели, при которых данное событие может произойти. Типы параметров события также задаются в блоке охранных условий. Действия изменяют текущее состояние за счёт модификации значений переменных модели.

В модели FRU_PRS определяются четыре события. Рассмотрим подробнее каждое из них.

Событие *change_priority* (рис. 5) моделирует изменение приоритета субъекта. У события два параметра: субъект *s* и новый приоритет *p*. Охранные условия *grd1* и *grd2* задают типы параметров, а условие *grd3* требует, чтобы новый приоритет *p* отличался от старого. Приоритет субъекта изменяется в действии *act1*, причём данное изменение может нарушить некоторые из определённых инвариантов. Например, если приоритет субъекта повышается, и он находится в очереди на получение доступа к некоторому объекту, текущий активный доступ к которому имеется у другого субъекта, приоритет которого меньше *p*, то будет нарушен инвариант *inv9*. Он будет нарушен и в том случае, когда приоритет субъекта уменьшается, и он обладает текущим активным доступом к некоторому объекту, в очереди на получение доступа к которому находится другой субъект, приоритет которого больше *p*. Чтобы избежать этого, в действии *act2* забирается доступ у тех пар "субъект-объект", которые нарушат

инварианты после изменения приоритета. В действии *act3* те же самые пары добавляются в очередь.

```
change_priority:     not extended ordinary ›
ANY
○  s    ›
○  p    ›
WHERE
○  grd1:   s∈S not theorem ›
○  grd2:   p∈P not theorem ›
○  grd3:   p≠SP(s) not theorem ›
THEN
○  act1:   SP(s) = p ›
○  act2:   R = {x↦y|x↦y∈R ∧ (s↦y∈Q ⇒ SP(x)≥p) ∧ (x≠s ∨ (x=s ∧ (∀z·z∈S ∧ z↦y∈Q ⇒ p≥SP(z))))} ›
○  act3:   Q = {x↦y|x↦y∈Q ∨ (x↦y∈R ∧ ((x≠s ∧ s↦y∈Q ∧ SP(x)<p) ∨ (x=s ∧ (∃z·z∈S ∧ z↦y∈Q ∧ p<SP(z)))))
END
```

Рис. 5. Событие «change_priority»
Fig. 5. «Change_priority» event

Событие *unsuccessful_access* (рис. 6) описывает ситуацию, когда субъект хочет получить доступ к объекту, но его приоритет меньше приоритета субъекта, который уже осуществляет доступ к этому объекту (*grd3*). В этом случае пара "субъект-объект" добавляется в очередь (*act1*).

```
unsuccessful_access:     not extended ordinary ›
ANY
○  s    ›
○  o    ›
WHERE
○  grd1:   s∈S not theorem ›
○  grd2:   o∈O not theorem ›
○  grd3:   ∃x·x∈S ∧ x↦o∈R ∧ SP(s)≤SP(x) not theorem ›
THEN
○  act1:   Q = Q ∪ {s↦o} ›
END
```

Рис. 6. Событие unsuccessful_access
Fig. 6. «Unsuccessful_access» event

Событие *access* (рис. 7) описывает успешное получение доступа субъекта *s* к объекту *o*, которые, как и в предыдущих событиях, заданы в виде параметров. Если существует активный доступ к объекту *o* от некоторого другого субъекта, то его приоритет должен быть строго ниже (*grd3*) приоритета *s*. Если существует субъект, который находится в очереди на получение доступа к объекту *o*, то его приоритет должен также должен быть ниже (*grd4*) приоритета *s*. В действии *act1* ко множеству текущих активных доступов *R* добавляется новый доступ от субъекта *s* к объекту *o*, но при этом если какой-то субъект уже обладает доступом к объекту *o*, то его доступ удаляется из *R* и переносится в очередь *Q* (*act2*). Если субъект *s* находился в очереди на получение доступа к объекту *o*, то после успешного получения доступа он будет удалён из очереди (*act2*).

```
access:  not extended ordinary ›
ANY
○  s  ›
○  o  ›
WHERE
○  grd1:  s∈S not theorem ›
○  grd2:  o∈O not theorem ›
○  grd3:  ∀x·x∈S ∧ x↦o∈R ⇒ SP(s)>SP(x) not theorem ›
○  grd4:  ∀x·x∈S ∧ x↦o∈Q ⇒ SP(s)≥SP(x) not theorem ›
THEN
○  act1:  R = {x↦y|(x↦y∈R ∧ y≠o) ∨ (x=s ∧ y=o)} ›
○  act2:  Q = {x↦y|(x↦y∈Q ∧ (x≠s ∨ y≠o)) ∨ (x↦y∈R ∧ y=o)} ›
END
```

Рис. 7. Событие «access»
Fig. 7. «Access» event

Последнее событие называется *free* (рис. 8) и моделирует удаление доступа из множества текущих активных доступов.

```
free:  not extended ordinary ›
ANY
○  s  ›
○  o  ›
WHERE
○  grd1:  s∈S not theorem ›
○  grd2:  o∈O not theorem ›
○  grd3:  s↦o∈R not theorem ›
THEN
○  act1:  R = R\{s↦o} ›
END
```

Рис. 8. Событие «free»
Fig. 8. «Free» event

Корректность изменения состояния модели, которое происходит после каждой модификации значений переменных в результате выполнения события, необходимо доказывать, так как при этом могут нарушиться определённые инварианты. Для этого платформа Rodin [9], которая используется для разработки и верификации моделей на Event-B, генерирует специальные утверждения для доказательства, которые могут быть доказаны формальным образом либо автоматическими инструментами, либо в интерактивном редакторе доказательств (рис. 9). При условии адекватности сформулированных в виде инвариантов требований к модели доказательство всех сгенерированных утверждений подтверждает её консистентность и корректность.

Рис. 9. Формализация и верификация модели FRU_PRS
Fig. 9. Formalizaion and verification of the FRU_PRS model

В рассмотренном примере инвариантом безопасности является *inv9*. Доказательство его сохранности при любом событии означает, что все состояния модели безопасны, и, следовательно, небезопасные состояния недостижимы.

При этом остаётся открытым вопрос о том, насколько корректно ОО будет реализовать формализованную модель ПБ. Для ответа на этот вопрос стандартом ГОСТ Р ИСО/МЭК 15408 предусмотрен анализ соответствия по цепочке модель ПБ — функциональная спецификация — описание проекта — реализация (рис. 1). Но детальное рассмотрение этой цепочки находится вне рамок данной статьи.

4. Заключение

В настоящей работе рассмотрен подход к формализации модели ПБ на языке Event-B, позволяющий решать задачи доказательства отсутствия внутренних противоречий и доказательства недостижимости небезопасных состояний при инструментальном контроле со стороны системы верификации Rodin. Наиболее значимым примером применения этого подхода является формализация и верификация политики безопасности МРОСЛ-ДП, реализованной операционной системе специального назначения Astra Linux Special Edition. Разработанная в рамках данного проекта формальная модель ПБ на языке Event-B обладает следующими количественными характеристиками:

- кол-во констант: 34;
- кол-во аксиом: 30;
- кол-во переменных состояния: 60;
- кол-во инвариантов: 248;
- кол-во событий: 75;
- число уровней уточнения: 4;
- размер спецификации на Event-B: 4393 строк;
- кол-во утверждений для доказательства: 2962.

Таким образом, можно сделать вывод, что рассмотренный подход применим для решения практически значимых задач и может применяться для выполнения требований семейства доверия ADV_SPM "Моделирование политики безопасности", определяемого стандартом ГОСТ Р ИСО/МЭК 15408-3-2013.

Список литературы

[1]. ГОСТ Р ИСО/МЭК 15408-1-2012 Информационная технология. Методы и средства обеспечения безопасности. Критерии оценки безопасности информационных технологий. Часть 1. Введение и общая модель.

[2]. ГОСТ Р ИСО/МЭК 15408-2-2013 Информационная технология. Методы и средства обеспечения безопасности. Критерии оценки безопасности информационных технологий. Часть 2. Функциональные компоненты безопасности.

[3]. ГОСТ Р ИСО/МЭК 15408-3-2013 Информационная технология. Методы и средства обеспечения безопасности. Критерии оценки безопасности информационных технологий. Часть 3. Компоненты доверия к безопасности

[4]. Huynh, N., Frappier, M., Mammar, A., Laleau, R., Desharnais, J.: Validating the RBAC ANSI 2012 standard using B. In: Abstract State Machines, Alloy, B, TLA, VDM, and Z. (2014) 255–270

[5]. Devyanin P.N., Khoroshilov A.V., Kuliamin V.V., Petrenko A.K., Shchepetkov I.V. Formal Verification of OS Security Model with Alloy and Event-B. In A. Yamine and K.-D. Schewe, eds. Abstract State Machines, Alloy, B, TLA, VDM, and Z, LNCS 8477:309-313, Proceedings of ABZ-2014, Toulouse, France, June 2-6, 2014, pp. 309-313. DOI: 10.1007/978-3-662-43652-3_30.

[6]. Devyanin P.N., Khoroshilov A.V., Kuliamin V.V., Petrenko A.K., Shchepetkov I.V. Comparison of Specification Decomposition Methods in Event-B. Programming and Computer Software, 2016, Vol. 42, No. 4, pp. 198–205. DOI: 10.1134/S0361768816040022.

[7]. Буренин П.В., Девянин П.Н., Лебеденко Е.В., Проскурин В.Г., Цибуля А.Н. Безопасность операционной системы специального назначения Astra Linux Special Edition. Учебное пособие для вузов. 2-е изд. М.: Горячая линия — Телеком, 2016, 312 с.

[8]. Abrial J.-R. Modeling in Event-B: System and Software Engineering. Cambridge University Press, 2010.

[9]. Abrial J.-R., Butler M., Hallerstede S., Hoang T. S., Mehta F., Voisin L. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. International Journal on Software Tools for Technology Transfer, 12(6), pp. 447-466, 2010.

ADV_SPM — Formal security policy models in practice

[1,2,3,4]*A.V. Khoroshilov <khoroshilov@ispras.ru>,*
[1]*I.V. Shchepetkov <shchepetkov@ispras.ru>,*
[1]*ISP RAS, 25 Alexander Solzhenitsyn Str., Moscow, 109004, Russia*
[2]*CMC MSU, CMC faculty, 2 educational building, MSU, Leninskie gory str., Moscow 119991, Russia*
[3]*Moscow Institute of Physics and Technology, 9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia*
[4]*FCS NRU HSE, 20 Myasnitskaya str., Moscow 101000, Russia*

Abstract. The paper examines the ADV_SPM "Security policy modelling" assurance family, which is part of the ADV "Development" assurance class and defined by the ISO/IEC 15408-3-2013 "Information technology – Security techniques – Evaluation criteria for IT security – Part 3: Security assurance components" standard. We discuss the objective set by this family, which is to provide additional assurance from the development of a formal security policy model of the target of evaluation security functionality and establishing a correspondence between the functional specification and this security policy model by means of a mathematical proof. We propose an approach to the formalization and verification of security policies using the Event-B modelling notation and the Rodin platform, whose rigour is used to obtain the desired security properties by means of formal machine-checkable proofs. The approach helps to identify and eliminate ambiguous, inconsistent, contradictory, or unenforceable security policy elements. We illustrate this approach with a simplified example of a FRU_PRS "Priority of service" model (defined in the second part of the standard) in which we provide a formal proof that the model contains no inconsistencies, and that an insecure state cannot be reached. We conclude that the approach is applicable for solving practical problems and it can be used to fulfil the requirements of the ADV_SPM assurance family.

Keywords: information security; security policy; formal models

DOI: 10.15514/ISPRAS-2017-29(3)-4

For citation: Khoroshilov A.V., Shchepetkov I.V. ADV_SPM — Formal security policy models in practice. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 3, 2017. pp. 43-56 (in Russian). DOI: 10.15514/ISPRAS-2017-29(3)-4

References

[1]. ISO/IEC 15408-1:2012 Information technology - Security techniques - Evaluation criteria for IT security - Part 1: Introduction and general model (in Russian).
[2]. ISO/IEC 15408-2:2013 Information technology - Security techniques - Evaluation criteria for IT security - Part 2: Security functional requirements (in Russian).
[3]. ISO/IEC 15408-3:2013 Information technology - Security techniques - Evaluation criteria for IT security - Part 3: Security assurance requirements (in Russian).

[4]. Huynh, N., Frappier, M., Mammar, A., Laleau, R., Desharnais, J.: Validating the RBAC ANSI 2012 standard using B. In: Abstract State Machines, Alloy, B, TLA, VDM, and Z. (2014) 255–270

[5]. Devyanin P.N., Khoroshilov A.V., Kuliamin V.V., Petrenko A.K., Shchepetkov I.V. Formal Verification of OS Security Model with Alloy and Event-B. In A. Yamine and K.-D. Schewe, eds. Abstract State Machines, Alloy, B, TLA, VDM, and Z, LNCS 8477:309-313, Proceedings of ABZ-2014, Toulouse, France, June 2-6, 2014, pp. 309-313. DOI: 10.1007/978-3-662-43652-3_30.

[6]. Devyanin P.N., Khoroshilov A.V., Kuliamin V.V., Petrenko A.K., Shchepetkov I.V. Comparison of Specification Decomposition Methods in Event-B. Programming and Computer Software, 2016, Vol. 42, No. 4, pp. 198–205. DOI: 10.1134/S0361768816040022.

[7]. Burenin P.V., Devyanin P.N., Lebedenko E.V., Proskurin V.G., Cibulya A.N. Security of the special purpose Astra Linux Special Edition operating system. Textbook for high schools. 2nd ed. Hot line - Telecom, Moscow [Uchebnoe posobie dlya vuzov. 2-e izd. M.: Goryachaya liniya — Telekom], 2016, 312 p. (in Russian)

[8]. Abrial J.-R. Modeling in Event-B: System and Software Engineering. Cambridge University Press, 2010.

[9]. Abrial J.-R., Butler M., Hallerstede S., Hoang T. S., Mehta F., Voisin L. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. International Journal on Software Tools for Technology Transfer, 12(6), pp. 447-466, 2010.

Анализ программ на языке Java в инструменте Svace

[1] *А.П. Меркулов <steelart@ispras.ru>*
[1] *С.А. Поляков <inly@ispras.ru>*
[1,2] *А.А. Белеванцев <abel@ispras.ru>*
[1] *Институт системного программирования РАН,*
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.
[2] *Московский государственный университет имени М.В. Ломоносова,*
119991, Россия, Москва, Ленинские горы, д. 1.

Аннотация. В статье описываются работы, выполненные для поддержки анализа программ на языке Java в статическом анализаторе Svace, разрабатываемом в ИСП РАН. Приводятся методы построения внутреннего представления для анализа Java, включая изменения в компоненте контролируемой сборки, доработки компилятора OpenJDK, трансляцию байткода Java в окончательное представление для анализа. Описываются особенности анализа Java-программ – алгоритм девиртуализации, спецификации методов стандартной библиотеки Java, некоторые специфичные детекторы. Представлены результаты выполнения анализа для исходного кода операционной системы Android 5.

Ключевые слова: статический анализ; Java; девиртуализация; байткод.

DOI: 10.15514/ISPRAS-2017-29(3)-5

Для цитирования: А.П. Меркулов, С.А. Поляков, А.А. Белеванцев. Анализ программ на языке Java в инструменте Svace. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 57-74. DOI: 10.15514/ISPRAS-2017-29(3)-5

1. Введение

Высокая сложность программ делает практически невозможным создание программного продукта без дефектов. Причём с увеличением размера программного обеспечения возрастает не только количество дефектов, но и их плотность. Поэтому растёт необходимость в инструментах и методах поиска дефектов. Одним из таких методов является статический анализ программ, который осуществляется без их реального выполнения. При этом происходит исследование всего кода программы, в том числе редко достигаемых участков кода, что позволяет найти ошибки, которые сложно воспроизвести, и которые обычно остаются незамеченными в ходе тестирования.

В рамках проекта Svace в Институте системного программирования РАН ведутся работы по реализации статического анализа кода с целью поиска дефектов и ошибок в программах. Изначально инструмент был реализован для анализа кода на языках C и C++. В данной статье описана реализация поддержки анализа программ на языке Java и связанные с этим особенности и проблемы. Описание устройства инструмента Svace можно найти в статьях [1-2]. Кратко поясним основные этапы работы Svace для ясности дальнейшего изложения.

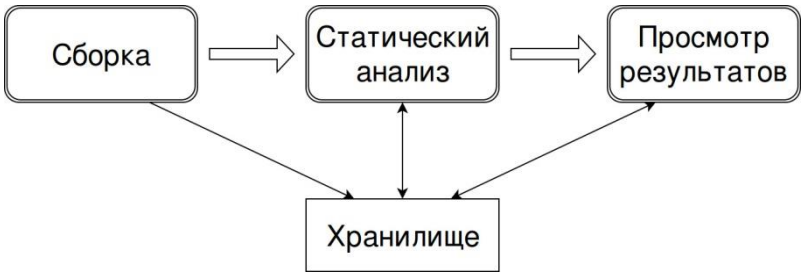


Рис. 1. Ход работы анализатора Svace
Fig. 1. Svace analyzer workflow

Процесс анализа программы Svace можно разбить на 3 этапа, работающие с одним хранилищем (см. рис. 1): контролируемая сборка проекта, статический анализ проекта, исследование результатов анализа пользователем в web-интерфейсе. Сначала Svace выполняет мониторинг оригинальной командой сборки интересуемого проекта. В качестве результата выполнения этой сборки генерируется контейнер с данными для анализа, которые включают в себя скомпилированное промежуточное представление, исходные коды, данные компоновки и так далее. На втором этапе запускается статический анализ для собранного контейнера с данными, его результатом становится контейнер с набором выданных предупреждений. Этот контейнер отправляется в хранилище результатов, которое отлеживает историю предупреждений. На третьем этапе пользователь открывает web-интерфейс и просматривает найденные предупреждения. Благодаря наличию истории становится возможным показывать, к примеру, только новые предупреждения или только пропавшие предупреждения.

Для добавления поддержки языка программирования Java было необходимо модифицировать этап сборки проекта и этап статического анализа проекта. Решение задачи добавления языка Java можно разбить на следующие подзадачи: модификации системы контролируемой сборки, трансляции байт-кода Java [3] во внутреннее представление Svace (раздел 2), модификации компилятора javac [4] (раздел 3), составления спецификаций стандартной библиотеки Java, девиртуализации вызовов (раздел 4). Девиртуализация была реализована перед этапом построения графа вызовов и является новой фазой

по сравнению с анализом кода на языках C и C++. В разделе 5 представлены результаты анализа исходного кода ОС Android 5, а в разделе 6 – заключение.

2. Перехват компиляции Java-программ

Для анализа проекта с помощью Svace необходимо выполнить контролируемую анализатором сборку проекта для компиляции исходных файлов проекта с помощью специально модифицированного компилятора Svace и с включёнными отладочными опциями. Общая схема этапа сборки приведена на рисунке 2. Процесс перехвата запускает оригинальную команду сборки, инструментируя её таким образом, чтобы перехватывать все запускаемые процессы и выделять интересующие нас команды сборки. Важно не влиять при этом на исходную сборку, чтобы ее результаты совпали с выполнением сборки без мониторинга.

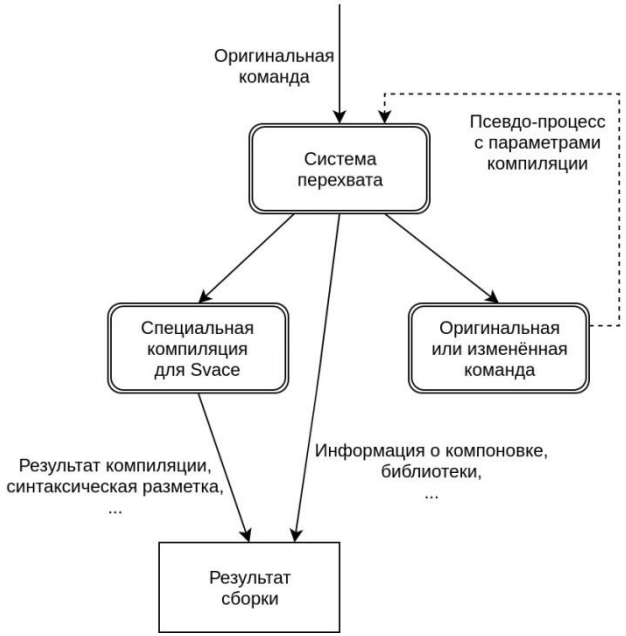


Рис. 2. Контролируемая сборка программы
Fig. 2. Monitored program build

В тот момент, когда в процессе сборки обнаруживается, что была запущена команда компиляции, дополнительно запускается модифицированный компилятор, который собирает код так, чтобы его было удобно впоследствии анализировать. Эта компиляция производится с выключенными оптимизациями и с включённой отладочной информацией. Сам компилятор

при этом модифицируется таким образом, чтобы максимально упростить генерируемый код, увеличить объём отладочной информации, а так же сделать специальные пометки на автогенерированный код. Кроме этого, модифицированный компилятор сохраняет дополнительную информацию о программе, в частности, генерирует файлы с синтаксической разметкой исходного кода, которые понадобятся при показе предупреждений пользователю.

Для компиляции кода на языке Java было решено проводить отладочную сборку с помощью модифицированного компилятора javac из пакета OpenJDK [4], так как OpenJDK является де-факто стандартным компилятором Java. Другим возможным выбором является компилятор ECJ среды Eclipse [9], однако его использование в контексте анализа в Svace не дает особых преимуществ – возможность разбора исходного кода с ошибками, важная для интегрированной среды разработки, на данный момент несущественна для Svace. В результате отладочной компиляции получаются файлы с байткодом Java (class-файлы), карта соответствия файлов промежуточного представления исходным файлам, синтаксическая разметка исходных файлов.

Дополнительной сложностью перехвата компиляций Java является возможность программно вызывать компилятор через Java Compiler API [5] без запуска нового процесса. Этой возможностью активно пользуются среды сборки Java-приложений, в частности, Ant, Maven, Gradle [6,7]. Задача перехвата в данном случае решается с использованием Java-агента [10]. При запуске любого Java-приложения можно указать стандартизированным образом библиотеку, через которую будет фильтроваться весь байткод запускаемого приложения. При этом Java-агент имеет возможность модифицировать этот байткод и, таким образом, запускаемое приложение.

Нами был создан Java-агент, который ищет в байткоде запускаемой программы вызовы компилятора (класса com.sun.tools.javac.main.Main) и инструментирует их так, чтобы дополнительно к компиляции вызывался специальный метод в java-агенте, в который передаются параметры компиляции. Метод формирует запуск специального псевдопроцесса, через параметры которого передаются параметры компиляции. Запуск этого псевдопроцесса перехватывается, как и все остальные процессы, после чего извлекаются параметры компиляции, и по ним формируется команда отладочной компиляции. При этом псевдопроцесс даже не запускается. На рисунке 2 пунктирной стрелкой показан вызов такого процесса. Таким образом, в реализованном решении перехваченные из оригинальной сборки запуски Java-программ модифицируются перед реальным запуском путём указания библиотеки собственного Java-агента.

3. Модификация компилятора javac

В качестве базового компилятора Java был выбран компилятор javac из пакета OpenJDK для Java 8. Оригинальный компилятор пришлось модифицировать,

так как необходимо было решить ряд проблем. Часть информации об исходной программе терялась или искажалась в процессе компиляции. В частности, возникали случаи дублирования байткода, основанного на одном и том же исходном коде, например, в процессе трансляции finally-блоков, а так же трансляции разделов инициализации полей класса в случае, если они инициализируются прямо в строке объявления, такие инициализации дублируются на каждый конструктор.

В случае трансляции finally-блоков необходимость в дублировании байткода возникает из-за того, что необходимо выполнить finally-блок сразу на нескольких путях: на успешном пути выполнения try-блока, на пути пойманного исключения в catch-блоке и на пути непойманного исключения, которое продолжит свой путь выше по стеку вызовов. Во всех этих случаях требуется выполнить один и тот же байткод finally-блока. Логичным решением данной задачи является генерация функции, которая выполняла бы finally-блок. Проблема тут заключается в том, что код finally-блока, как правило, работает с локальными переменными функции, а поскольку в Java байткоде невозможно взять адрес локальных переменных, генерация отдельной функции для обработки finally-блока не представляется возможным.

В ранних версиях Java в байткоде существовали инструкции `jsr/ret`, которые фактически выполняли локальный вызов. Инструкция jsr при этом запоминает адрес возврата в стек и переходит на указанную метку, а инструкция ret, переходит по адресу, указанному в стеке. Таким образом, в компилятор уже получается встроенным механизм генерации finally-блоков без дублирования кода. К сожалению, данный механизм не используется в поздних версиях Java.

Для анализа дублирование байткода неудобно тем, что байткод перестаёт взаимно-однозначно отображаться на исходный код, и необходимо учитывать отладочную информацию для установления, какой же код был продублирован. В противном случае возможны ситуации, при которых будут выданы ложные сообщения об ошибках. Дело в том, что одним из критериев выдачи сообщений в Svace является установление факта наличия ошибки на всех путях, проходящих через определённую точку. То есть, анализатор пытается найти точки в программе, обладающие следующим свойством: если программа попадает в эту точку, то неминуемо произойдёт или уже произошла ошибка – в этом случае данное место в программе либо является мертвым кодом, либо в программе содержится ошибка.

В случае дублирования кода данный критерий теряет свою актуальность, поскольку одна и та же точка в исходной программе будет соответствовать нескольким точкам в байткоде. Тем самым анализатор не может рассчитывать на то, что все точки программы должны хоть когда-то исполняться. В табл. 1 приводится код метода и получающийся стандартным компилятором соответствующий байткод. Как видно из данного примера, код,

соответствующий finally-блоку, дублируется в трёх экземплярах (дублируемые инструкции байт-кода выделены серым, остальной код – мелким шрифтом). И, если не исправить и не учитывать это дублирование, то на выполнении кода на ветке без исключений возникает недостижимый код: переменная `err` будет равна нулю, и далее следует бессмысленное сравнение с единицей.

Табл. 1. Дублирование кода компилятором javac
Table 1. Code duplication performed by javac compiler

```
int err = 0;
try {
    return something();
} catch(RuntimeException e) {
    err = 1;
} finally {
    // Возможное ложное
    // срабатывание
    if (err == 1) {
        report();
    }
    finish();
}
return err;
```

```
Exception table:
from    to  target  type
  2     7    22 RuntimeException
  2     7    41      any
 22    25    41      any
 41    43    41      any

 0: iconst_0
 1: istore_2
 2: aload_0
 3: invokevirtual something:()I
 6: istore_3
 7: iload_2

 8: iconst_1
 9: if_icmpne     16
12: aload_0
13: invokevirtual report:()V
16: aload_0
17: invokevirtual finish:()V

20: iload_3
21: ireturn
22: astore_3
23: iconst_1
24: istore_2
25: iload_2

26: iconst_1
27: if_icmpne     34
30: aload_0
31: invokevirtual report:()V
34: aload_0
35: invokevirtual finish:()V

38: goto   59
41: astore 4
43: iload_2

44: iconst_1
45: if_icmpne     52
48: aload_0
49: invokevirtual report:()V
52: aload_0
53: invokevirtual finish:()V
```

```
56: aload  4
58: athrow
59: iload_2
60: ireturn
```

Другой проблемой является автогенерированный компилятором код. К примеру, оператор отрицания нередко раскрывается в if-ветвление. В результате выдаются ложные срабатывания про недостижимый код. В примере из табл. 2 инструкция №9 является недостижимым кодом, однако более правильно в данном случае сообщать о константном результате вычисления выражения `!x`. Чтобы решить проблему автогенерации, был изменен компилятор javac таким образом, чтобы он генерировал вспомогательный блок информации к функции с разметкой сгенерированных ветвлений (серым цветом в примере помечены инструкции такого ветвления).

Табл. 2. Генерация условного оператора
Table 2. If operator generation

```
if (x)
 {
    // Сообщение о недостижимом
    // коде будет некорректно
    something(!x);
 }
```

```
 0: iload_1
 1: ifeq 17
 4: aload_0
 5: iload_1
 6: ifne 13
 9: iconst_1
10: goto 14
13: iconst_0
14:           invokevirtual
something:(Z)V
17: return
```

4. Статический анализ Java-программ

Этап статического анализа в инструменте Svace можно разбить на несколько последовательных основных фаз: построение графа вызовов, разрыв циклов в графе вызовов и составление топологического порядка обхода графа вызовов в порядке от листовых вершин к корневым, обход графа вызовов с разорванными циклами в топологическом порядке и анализ каждой функции. При этом по результатам анализа функции формируется резюме, которое используется вызывающими функциями, чем достигается межпроцедурность анализа.

Для анализа функции сначала читается промежуточное представление, сгенерированного специальным компилятором, и транслируется в

промежуточное представление Svace. Для каждой анализируемой функции строится граф потока управления и топологически обходится в глубину, начиная с входной точки в функцию. Ядро статического анализа Svace осуществляет символьное исполнение инструкций промежуточного представления. Поиском дефектов занимаются детекторы, которые подписываются на интересующие их события (например, на события разыменования и сравнения для детектора поиска разыменования переменной после её сравнения с нулём). Детекторы декларируют набор атрибутов, которые распространяются по графу потока управления в процессе символьного исполнения, а также корректируются в обработчиках детектора при наступлении интересующих его событий. Проверка на возможный дефект также происходит в обработчиках детектора. Найденные дефекты записываются в предварительный буфер, который впоследствии фильтруется с целью устранить дублирующиеся и похожие сообщения.

Анализ Java-программ происходит по аналогичной схеме. На первом этапе происходит быстрый просмотр всех файлов промежуточного представления. При этом ведётся сбор информации об иерархии наследования, содержании классов, строится граф вызовов. На втором этапе происходит девиртуализация, которая существенно упрощает граф вызовов. По готовому графу вызовов строится порядок обхода функций. На последнем этапе происходит обход этого графа и основной описанный выше анализ.

Таким образом, для добавления поддержки языка Java было необходимо реализовать трансляцию Java байт-кода в промежуточное представление Svace, девиртуализацию, реализацию спецификаций стандартной библиотеки Java, а также ряд специфичных для Java детекторов.

4.1 Трансляция Java байткода в промежуточное представление Svace

Внутреннее представление Svace пригодно для анализа различных языков, но по своему уровню близко к биткоду LLVM, т.к. изначально использовалось для программ на языках C и C++, а собственным компилятором Svace для этих языков является модифицированный Clang. Представление Svace является низкоуровневым трехадресным типизированным ассемблером в SSA-форме. При добавлении поддержки языка Java в Svace было решено использовать байткод Java как входное для статического анализатора представление, а перед анализом транслировать его в имеющееся внутреннее представление.

Это решение было обосновано несколькими факторами. Во-первых, байткод Java является стандартным выходным форматом для компилятора javac, и этот формат поддерживают многие системные утилиты Java, среди которых популярная библиотека ASM. Во-вторых, кроме непосредственно компилируемой программы, в проекте обычно присутствуют сторонние библиотеки в виде JAR-файлов, также содержащие байткод. Таким образом,

становится возможным проанализировать ещё и использующиеся при компиляции библиотеки.

Для чтения байткода используется библиотека ASM. Несмотря на то, что Java байткод представляет из себя стек-машину, эта стек-машина получается из трансляции абстрактного синтаксического дерева метода и потому имеет ряд ограничений. В частности, для каждого базового блока фиксирована входная глубина и максимально возможная глубина стека. Таким образом, возможна трансляция этой стек-машины в обычное трёхадресное представление.

Табл. 3. Трансляция выражения z=x+y
Table 3. Translating z=x+y expression

Байт-код	Тривиальная трансляция	Трансляция с оптимизацией
`iload_1`	`tmp1 = deref addr_x` `pmove tmp1 to stack0`	`tmp1 = deref addr_x`
`iload_2`	`tmp2 = deref addr_y` `pmove tmp2 to stack1`	`tmp2 = deref addr_y`

`iadd`	`tmp3 = deref stack1`
	`tmp4 = deref stack0`
	`tmp5 = tmp3 + tmp4`
	`pmove tmp5 to stack0`
		`tmp3 = tmp1 + tmp1`

`istore_3`	`tmp6 = deref stack0` `pmove tmp6 to addr_z`	`pmove tmp3 to addr_z`

Для трансляции был разработан следующий алгоритм. Достаточно поддерживать для каждой инструкции текущую глубину стека. Для каждой глубины стека заводится соответствующая ячейка памяти. Загрузка значения из стека транслируется, как загрузка значения из ячейки памяти, соответствующей текущей глубине стека. Пример исходного кода из табл. 3 показывает трансляцию выражения `z=x+y` для целочисленных `x`, `y` и `z`.

Здесь была произведена следующая оптимизация. Вместо того, чтобы транслировать каждое обращение в стек и из стека в инструкции обращения к ячейкам памяти, алгоритм трансляции может запоминать промежуточные значения и класть их в ячейки памяти, соответствующие глубине стека, при условии, что транслируемый базовый блок заканчивает выполняться с непустым стеком. Соответственно, брать значения из ячеек памяти можно только тогда, когда неизвестны SSA-значения, которые там должны были лежать. Такая ситуация возможна, только если транслируется базовый блок с изначально непустым стеком, что бывает, например, при трансляции тернарных операторов. В результате в байткоде получается разветвление, каждая ветка которого по-своему заполняет единственный элемент в стеке.

Возможно окончательно избавиться от ячеек памяти, эмулирующих стек. Для этого после построения исходного представления необходимо переместить на псевдорегистры все переменные из памяти, про которые известны их адреса и которые не остаются живыми после окончания работы функции. Подобного рода устранение ячеек памяти реализовано в трансформации mem2reg в рамках инфраструктуры LLVM. Однако при этом необходимо аккуратно сохранять соответствие с исходным кодом программы, чтобы не потерять информацию о том, какие псевдорегистры находились в памяти (например, для корректного поиска утечек памяти). В настоящий момент данное преобразование в инструменте Svace не реализовано.

Следует также отметить ещё несколько тонкостей, связанных с трансляцией. Первой из них является тот факт, что для значений типа `long` и `double` выделяется 2 ячейки памяти. При этом под ссылки на объекты выделяется одна ячейка памяти. Такое деление является устаревшим, так как в 64х-битных системах возможны ситуации, при котором объектов в программе будет больше, чем 2^{32}. Нужно отслеживать тип операндов у операций со стеком для корректной генерации кода.

Второй тонкостью является выделение двух ячеек памяти под локальные переменные типа `long` и `double`. Из-за этого нумерация локальных переменных является не сплошной. Кроме того, локальные переменные для Java-машины по сути реализуют адресуемую память. Соответствие реальным локальным переменным можно построить только по отладочной информации. При этом одной и той же ячейке этой адресуемой памяти могут соответствовать несколько реальных локальных переменных в анализируемой программе, если области видимости этих переменных не перекрываются.

4.2 Девиртуализация

Девиртуализация была реализована перед этапом построения графа вызовов и является новой фазой по сравнению с анализом кода на языках C и C++. На данный момент реализованы простые эвристики девиртуализации, основанные на иерархии классов. Считается, что при анализе доступна вся информации о наследовании. Конечно, это не всегда так, но на реальных проектах это предположение себя оправдывает.

Исходя из иерархии наследования, очень часто можно сказать, из какого именно класса будет вызвана та или иная функция. На рис. 3 изображён пример с иерархией наследования из четырёх классов с базовым классом A и рассмотрена ситуация, как будет девиртуализирован вызов виртуального метода f(). Для объектов, тип которых указан как B, C и D, можно однозначно определить, какая именно функция будет вызвана. В классе C метод f() объявлен как абстрактный, однако поскольку единственной реализацией в примере является реализация из класса D, то можно точно сказать, что эта реализация и будет вызвана. В то же время, базовый класс A имеет реализацию метода f(), однако если переменная имеет тип A, то её реальный

тип может быть A, B или D, что не позволяет девиртуализовать вызовы метода f() у переменных с типом A.

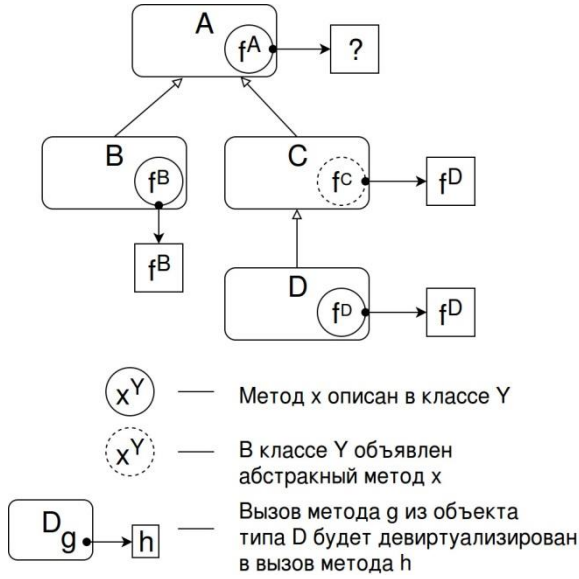


Рис. 3. Пример девиртуализации
Fig. 3. Devirtualization example

Такой метод наиболее эффективен для девиртуализации публичных (`public`) методов, не имеющих переопределений. Поскольку практика указания `final` в описании методов не является распространённой среди Java-программистов, то подобного рода девиртуализация на основе иерархии наследования позволяет устранить избыточную формальную виртуальность в вызовах.

4.3 Спецификации

Статическому анализатору важно знать, какой эффект на программу производят функции стандартной библиотеки, особенно функции выделения ресурсов. Можно пытаться понять смысл функций стандартной библиотеки автоматически по их исходному коду, но таковой чаще всего недоступен для анализа, нет гарантий, что заранее проанализированный код совпадает с используемой библиотекой, и информация, полученная от такого анализа, избыточна. Поэтому в Svace используются спецификации для описания эффектов функций стандартной библиотеки. Спецификация функции стандартной библиотеки представляет из себя краткое резюме эффекта

исполнения этой функции с точки зрения видимых извне действий функции, интересных анализатору.

Стандартная библиотека Java обладает существенно большим размером, нежели стандартная библиотека языка Си. Кроме нее, нами была поддержана библиотека платформы Android [8], так как эта платформа является важной для Java. Основной проблемой при реализации спецификаций стала иерархия классов в стандартной библиотеке. К примеру, необходимо учитывать тот факт, что при вызове метода `close` у объекта через интерфейс базового класса будет очищаться выделенный ресурс. Таким образом, описанной выше девиртуализации становится недостаточно, чтобы отслеживать утечки ресурсов, так как вызов остался бы виртуальным. В текущей реализации задача решается через спецификации интерфейсов и абстрактных функций. Таким образом, виртуальный вызов функции из стандартной библиотеки превращается в подстановку спецификации, что позволяет справиться с виртуальностью в стандартной библиотеке Java.

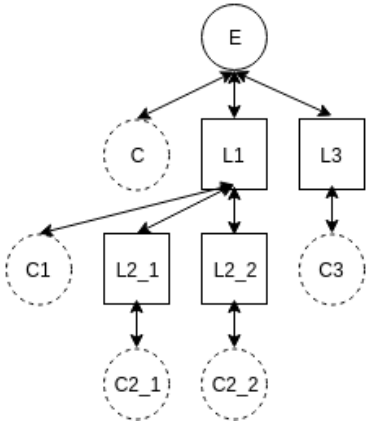
Отличительной особенностью стандартной библиотеки Java является то, что она содержит достаточно много классов-обёрток, таких как `BufferedInputStream`. В результате, надо учитывать тот факт, что вызов вызов `close` из базового интерфейса в случае стандартной библиотеки будет, как правило, закрывать все дочерние ресурсы. Данное предположение реализовано в спецификациях стандартной библиотеки Java, то есть считается, что метод `close` рекурсивно вызывает `close` у всех дочерних объектов. Использование такой эвристики приводит к незначительной потере истинных срабатываний, однако при этом позволяет избежать значительного количество ложных срабатываний или усложнения алгоритмов анализа и девиртуализации.

4.4 Специфичные для Java детекторы

Большая часть детекторов для Java входит в набор детекторов для Си/Си++ в том или ином виде, и их реализация является общей. Однако имеется ряд специфичных для Java детекторов. Прежде всего, можно выделить детекторы на синхронизацию. В Java существует оператор синхронизации, который синтаксически гарантированно всегда будет сбалансирован. Это позволяет писать для Java более простые и эффективные детекторы, чем это пришлось бы делать для Си и Си++. Кратко опишем устройство детекторов ошибок синхронизации в объеме, возможном для данной статьи.

В статическом анализаторе Svace реализованы следующие детекторы для обнаружения ошибок синхронизации: детектор для обнаружения взаимных блокировок DEADLOCK и статистический детектор NO_LOCK.STAT для обнаружения состояний гонки. Работа данных детекторов основана на анализе *дополненного графа вызовов* — модели параллельной программы, описывающей выполнение программы в несколько потоков.

Локальным дополненным графом вызовов будем называть двунаправленный граф, имеющий вершины трех типов: начало метода, захват ресурса, вызов метода. Вершина любого типа содержит в себе ссылку на место в исходном коде программы, где произошло то или иное событие. Вершины начала или вызова метода, кроме того, содержат уникальный идентификатор метода. Вершина захвата ресурса содержит абстрактный ресурс, идентифицирующий ресурс, который необходимо захватить потоку для входа в критическую секцию. Локальный граф строится, соединяя вершины захвата ресурсов согласно путям в графе потока управления, проходящим через них и начало метода, и аналогично соединяя вершины с вызовами методов. Целью является представить информацию о том, какие ресурсы необходимо захватить, чтобы добраться до вызова метода (см. рис. 4 – квадратами показаны вершины второго типа, прерывистыми кругами – третьего).




```
public void E() {
    C();
    synchronized(L1) {
        C1();
        synchronized(L2_1) {
            C2_1();
        }
        synchronized(L2_2) {
            C2_2();
        }
    }
    synchronized(L3) {
        C3();
    }
}
```

Рис. 4. Локальный дополненный граф вызовов
Fig. 4. Local enriched graph

Дополненным графом вызовов будем называть граф, полученный из локальных, путем соединения соответствующих вершин первого и третьего типов. Так как граф вызовов обходится анализом от вызываемых функций к вызывающим, то дополненный граф можно строить инкрементально по ходу анализа – локальные графы для вызываемых функций в точке вызова уже будут известны.

Ошибкой взаимной блокировки потоков в параллельной программе называется ситуация, когда несколько потоков находятся в состоянии бесконечного ожидания ресурсов, занятых самими этими потоками. Детектор DEADLOCK нацелен на обнаружение взаимной блокировки, определенной следующим образом. Пусть t и t′ – два абстрактных потока, а l_1, l_2, l_1', l_2' – инструкции блокировки в исходном коде программы. *Взаимная блокировка* потоков t и t′

будет иметь место, если t и t′ будут ждать освобождения ресурсов r_1 и r_2 при выполнении l_2 и l_2', при этом владея ресурсами r_2 и r_1 после выполнения l_1 и l_1' соответственно. Определенный таким образом дефект легко обнаруживается во время обхода в глубину расширенного графа вызовов.

Перед тем как выдать предупреждение о дефекте, анализатор проводит дополнительные проверки для подавления ложных предупреждений. Основной причиной ложных предупреждений является gate lock — общий для потоков t и t' ресурс, завладеть которым необходимо до выполнения инструкций l_1 и l_1'. Для вершин дополненного графа вызовов, соответствующим l_1 и l_1', формируется множество доминирующих вершин, имеющих тип захват ресурса. Если пересечение полученных множеств не пусто, необходимо отменить выдачу предупреждения.

Использование предложенной модели параллельных программ не ограничивается поиском состояний взаимной блокировки. Ее также можно использовать для обнаружения состояний гонки. Данная ошибка может возникнуть в случае, когда несколько потоков имеют доступ к одному и тому же ресурсу. Детектор NO_LOCK.STAT накапливает статистику обращения к полям классов во время обхода графа потока управления. Если обращение к полю произошло внутри критической секции, записывается позитивный результат в статистику. Также сохраняется информация о том, какой ресурс необходимо захватить потоку, чтобы попасть в данную критическую секцию. Если обращение к полю произошло вне критической секции, записывается негативный результат в статистику. При значительном превышении позитивной статистики над негативной для случаев негативной статистики выдается предупреждение о потенциальном состоянии гонки.

Поскольку алгоритм не учитывает возможные контексты вызова метода, внутри которого произошло небезопасное обращение к полю, возможна выдача ложных предупреждений. Такая ситуация встречается, например, когда контракт метода предусматривает вызов данного метода только при захвате потоком необходимого ресурса. Для уменьшения числа ложных срабатываний используется дополненный граф вызовов. Исследуются все пути в расширенном графе вызовов, ведущие в вершину начала метода, внутри которого произошло небезопасное обращение к полю. Если на каждом пути встречается вершина захвата ресурса, необходимого для обращения к данному полю, то предупреждение о состоянии гонки не будет выдано.

Рассмотрим еще один характерный для Java детектор – NO_BASE_CALL. Он нацелен на поиск методов, которые не вызывают свою реализацию из базового класса, хотя должны это делать. Этот детектор делится на два поддетектора. Детектор NO_BASE_CALL.LIB обладает базой знаний о библиотечных методах, при реализации которых надо вызывать базовый метод. К примеру, при реализации метода `clone` необходимо использовать `clone` родительского класса и так до `Object.clone()`, который создаёт объект нужного типа, возвращает его, а производные классы его заполняют.

Типичной ошибкой при реализации является создание объекта текущего класса. В этом случае производный класс не сможет нормально себя склонировать.

Вторым поддетектором является NO_BASE_CALL.STAT. Этот детектор работает с произвольными методами и статистически пытается понять, когда программист забыл вызвать метод из базового класса, то есть детектор ищет ситуации, когда производный класс вызывает почти везде соответствующий метод базового класса, но есть случаи, когда такого вызова не происходит. В этом случае есть подозрение на забытый вызов, и генерируется предупреждение об ошибке.

5. Экспериментальные результаты

Описанные алгоритмы трансляции программ на языке Java и доработки анализатора были реализованы в инструменте Svace. Основные структуры данных и алгоритмы анализа являлись общими между Java, Си и Си++. По результатам тестирования было установлено, что качество анализа кода на языке Java находится на уровне успешных коммерческих аналогов. Анализ исходного кода ОС Android 5 занимает примерно полтора часа на сервере с 32 логическими ядрами. Результаты истинных срабатываний для некоторых детекторов приведены в табл. 4.

Табл. 4. Результаты тестирования для ОС Android 5
Table 4. Experimental results for OS Android 5

Название детектора	Срабатываний всего	Исследовано срабатываний	TP%
DEREF_AFTER_NULL	143	108	96.3%
DEREF_AFTER_NULL.EX	250	100	98.0%
DEREF_OF_NULL	10	7	100.0%
DEREF_OF_NULL.ASSIGN	245	166	100.0%
DEREF_OF_NULL.CONST	1001	775	100.0%
DEREF_OF_NULL.EX	160	57	96.5%
DEREF_OF_NULL.RET.LIB	574	188	91.2%
DEREF_OF_NULL.RET.LIB.PROC	124	12	100.0%
DEREF_OF_NULL.RET.USER	1407	397	98.0%
DEREF_OF_NULL.RET.USER.PROC	144	43	100.0%
HANDLE_LEAK	1165	268	96.8%
HANDLE_LEAK.EXCEPTION	984	240	96.9%

HANDLE_LEAK.FRUGAL	381	132	98.4%
HANDLE_LEAK.FRUGAL.EXCEPTION	158	46	95.7%
NO_BASE_CALL.LIB	179	73	93.2%
NO_BASE_CALL.STAT	235	82	100.0%
NO_CHECK_IN_LOCK	66	30	86.2%
NO_LOCK.GUARD	47	32	93.8%
NO_LOCK.STAT.EX	1332	101	94.1%
NULL_AFTER_DEREF	219	95	78.6%
UNREACHABLE_CODE	211	80	89.9%
WRONG_LOCK_OBJECT	52	37	100.0%

6. Заключение

В статье была описана реализация поддержки языка программирования Java в рамках инфраструктуры инструмента статического анализа Svace, изначально разработанного для анализа кода на языках C и C++. В целом можно отметить, что базовые алгоритмы анализа подходят и для языка Java, трудности заключаются в организации перехвата компиляций через Java VM, девиртуализации, работе со стандартной библиотекой. Результаты работы на промышленном коде являются приемлемыми, и анализатор используется в компании Samsung для анализа собственных мобильных приложений для ОС Android и собственных ее расширений.

Дальнейшим направлением развития является разработка и внедрение более сложных алгоритмов девиртуализации. В частности, можно попробовать объединять резюме от потенциальных наследников и объединять их в одно резюме. В этом случае граф вызовов становится намного более связным, и потребуется существенно больше удалений рёбер из граф вызовов, чтобы сделать его ациклическим. Чтобы компенсировать удалённые рёбра, может потребоваться делать несколько обходов по ациклическому графу вызовов. Более точную девиртуализацию также можно проводить в случаях, когда на этапе потоково-чувствительного анализа удаётся установить реальный тип объекта или хотя бы сузить диапазон возможных типов.

Список литературы

[1]. В.П. Иванников, А.А. Белеванцев, А.Е. Бородин, В.Н. Игнатьев, Д.М. Журихин, А.И. Аветисян, М.И. Леонов. Статический анализатор Svace для поиска дефектов в исходном коде программ. Труды ИСП РАН, том 26, 2014 г., стр 231–250. DOI: 10.15514/ISPRAS-2014-26(1)-7.

[2]. А.Е. Бородин, А.А. Белеванцев. Статический анализатор Svace как коллекция анализаторов разных уровней сложности, том 27, вып. 6, 2015 г., стр. 111-134. DOI: 10.15514/ISPRAS-2015-27(6)-8.

[3]. Спецификация виртуальной машины Java. http://docs.oracle.com/javase/specs/jvms/se7/html/, дата обращения 20.06.2017
[4]. Компилятор Javac. http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javac.html, дата обращения 20.06.2017
[5]. Программный интерфейс компиляции в Java. http://openjdk.java.net/groups/compiler/guide/compilerAPI.html, дата обращения 20.06.2017
[6]. Система сборки Ant. http://ant.apache.org/, дата обращения 20.06.2017
[7]. Система сборки Maven. https://maven.apache.org/, дата обращения 20.06.2017
[8]. ОС Android. https://source.android.com/, дата обращения 20.06.2017
[9]. Компилятор Eclipse ECJ. https://mvnrepository.com/artifact/org.eclipse.jdt.core.compiler/ecj, дата обращения 20.06.2017
[10]. Инструментация байткода Java через java-агенты. https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html, дата обращения 20.06.2017

Supporting Java programming in the Svace static analyzer

[1] *A.P. Merkulov <steelart@ispras.ru>*
[1] *S.A. Polyakov <inly@ispras.ru>*
[1,2] *A.A. Belevantsev <abel@ispras.ru>*
[1] *Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia*
[2] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia*

Abstract. The paper is devoted to the works performed within the Svace static analysis tool to support Java language. First, the approach to intercept compilation process for transparently building the analyzer internal representation should be extended to cover usage of the Java compiler API that is popular in Ant and Maven tools. We achieve this goal with implementing our custom Java agent that instruments all calls to the compiler API and notifies the analyzer with the actual compilation parameters. Second, the modified Javac compiler builds the analyzer IR. The changes we made to the compiler include avoiding unnecessary bytecode duplication for easier mapping of bytecode instructions to source code and properly marking the code added by the compiler itself. Third, we discuss the process of bytecode translation to the Svace IR proper (which is a low-level 3-address IR akin to the LLVM IR). It is a straightforward code generation algorithm with further code cleanups that treats stack locations as local variables made possible by the fact that we know the maximum stack size consumed by the method. Finally, we discuss the devirtualization heuristics that assume we know the full class hierarchy and specific Java checkers including synchronization issue checkers. Experimental results obtained on Android 5 source code show that the checkers have high quality (more than 80% true positives). It can be seen that the general infrastructure for analysis and checkers implemented in Svace works well for the Java programming language with the adaptations described in the paper.

Keywords: static analysis; Java; bytecode; devirtualization

DOI: 10.15514/ISPRAS-2017-29(3)-5

For citation: Merkulov A.P., Polyakov S.A., Belevantsev A.A. Supporting Java programming in the Svace static analyzer. *Trudy ISP RAN/Proc. ISP RAS*, vol. 29, issue 3, 2017. pp. 57-74 (in Russian). DOI: 10.15514/ISPRAS-2017-29(3)-5

References

[1]. V.P. Ivannikov, A.A. Belevantsev, A.E. Borodin, V.N. Ignatiev, D.M. Zhurikhin, A.I. Avetisyan, M.I. Leonov. Static analyzer Svace for finding of defects in program source code. Trudy ISP RAN/ Procf ISP RAS, vol. 26, issue 1, 2014, pp. 231-250 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-7

[2]. A. Borodin, A. Belevancev. A Static Analysis Tool Svace as a Collection of Analyzers with Various Complexity Levels. Trudy ISP RAS/ Proc. ISP RAS, vol. 27, issue 6, pp. 111-134 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-8

[3]. Java virtual machine specification. http://docs.oracle.com/javase/specs/jvms/se7/html/, accessed 20.06.2017

[4]. The Javac compiler. http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javac.html, accessed 20.06.2017

[5]. Java compiler API. http://openjdk.java.net/groups/compiler/guide/compilerAPI.html, accessed 20.06.2017

[6]. Ant build system. http://ant.apache.org/, accessed 20.06.2017

[7]. Maven build system. https://maven.apache.org/, accessed 20.06.2017

[8]. Android operating system. https://source.android.com/, accessed 20.06.2017

[9]. The Eclipse ECJ compiler. https://mvnrepository.com/artifact/org.eclipse.jdt.core.compiler/ecj, accessed 20.06.2017

[10]. Instrumenting Java bytecode with Java agents. https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html, accessed 20.06.2017

Обзор подходов к улучшению качества результатов статического анализа программ[1]

А.Ю. Герасимов <agerasimov@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. В настоящий момент индустрия создания программ для всевозможного рода вычислительных устройств находится в состоянии бурного развития. Постоянно увеличивающаяся мощность вычислительных систем предоставляет всё новые возможности для создания высокопроизводительных, в том числе – параллельных, программ и программных комплексов. В связи с этим постоянно возрастает сложность программного обеспечения, управляющего вычислительными системами. Из-за высокой сложности программных систем процесс обеспечения качества разрабатываемого программного обеспечения требует новых подходов к процессу проверки корректности программ как на соответствие требованиям пользователей, так и на наличие критических дефектов и уязвимостей безопасности. Одним из методов контроля качества программного обеспечения является применение инструментальных средств программиста, предназначенных для анализа программ. Отрасль создания инструментальных средств статического и динамического анализа программ активно развивается с начала 2000-х годов. Разрабатывается большое количество академических и промышленных сред и инструментов анализа программ. В связи с фундаментальными ограничениями и инженерными компромиссами в угоду производительности и масштабируемости инструменты статического анализа не всегда могут обеспечить отсутствие ошибок первого рода в результатах своей работы. При этом анализ предупреждений инструмента может отнимать значительное время высококвалифицированного эксперта в области разработки и обеспечения качества программного обеспечения. В связи с этим возникает задача улучшения качества результатов работы статических анализаторов программ. Данная статья посвящена обзору методов анализа программ и подходов к улучшению качества работы статических анализаторов. Особое внимание в статье уделяется методам совмещения подходов статического и динамического анализа программ.

Ключевые слова: статический анализ программ; динамический анализ программ; комбинированный анализ программ

[1] Работа проводится при финансовой поддержке Российского фонда фундаментальных исследований. Проект 07-17-00702.

DOI: 10.15514/ISPRAS-2017-29(3)-6

Для цитирования: Герасимов А.Ю. Обзор подходов к улучшению качества результатов статического анализа программ. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 75-98. DOI: 10.15514/ISPRAS-2017-29(3)-6

1. Введение

Область исследований, посвященная методам автоматического анализа программ с целью обнаружения дефектов, активно развивается в последнее время. Появляется большое количество инструментов статического и динамического анализа, а также инструментов, совмещающих оба этих подхода в том или ином виде. В данной работе рассматривается понятие программной ошибки, даётся обзор проблем методов статического анализа программ и приводится обзор подходов к улучшению результатов работы инструментов статического и динамического анализа программ.

2. Программные ошибки

В 1842 году Чарльз Бэббидж (*Charles Babbadge*), английский математик и создатель Аналитической машины (Большой разностной машины), был приглашен в Туринский университет провести семинар о созданной им машине. Луиджи Менабреа (*Luigi Federico Menabrea*), итальянский инженер, записал этот семинар на французском языке, и этот текст в последствии был опубликован в Общественной библиотеке Женевы в октябре 1942 года. Друг Бэббиджа, изобретатель Чарльз Уитстон (*Sir Charles Wheatstone*), попросил графиню Аду Августу Лавлейс (*Ada Augusta, countess of Lovelace*) перевести эти записи Менабреа на английский и сопроводить текст комментариями. С учётом 52 страниц комментариев графини Ады Августы Лавлейс труд под названием «Очерк об аналитической машине, представленной Чарльзом Бэббиджем» [1], опубликованный под акронимом ААЛ оказался более обширным, чем записи Менабреа. В этом очерке Ада Августа графиня Лавлейс в частности заметила: «…анализирующий процесс должен быть одинаково произведен в соответствии с предоставленными Аналитической Машине необходимыми управляющими данными, и это при сём также может быть источником возможной ошибки. Правда в том, что механизм безошибочен в своих процессах, но карты (*прим.* с управляющими данными) могут давать ошибочные команды». Это означает, что уже на заре эры программируемых вычислительных устройств, которые используют внешние данные как управляющие команды, осознавалась возможность наличия ошибок в этих управляющих данных, которые мы в настоящее время называем программами.

В статье «Что мы знаем о методах обнаружения ошибок?» [2] предлагается трактовка программной ошибки (дефекта), как некоторой сущности, которая приводит к одной или нескольким ошибкам в таком артефакте как программный код. В то же время проводится анализ двенадцати исследований

на тему сравнения инспекций кода и тестирования, в которых выделяются дефекты на уровне требований к программному обеспечению, на уровне спецификаций программного обеспечения и в коде программного обеспечения.

Общепринятого определения программного дефекта на данный момент обнаружить не удалось. В различных работах, посвященных обнаружению дефектов в программах, даётся своё определение с целью использования его в исключительно рамках работы. В 2010 году подразделением Компьютерного сообщества (IEEE Computer Society) Института инженеров электротехники и электроники (IEEE – Institute of Electrical and Electronics Engineers) выпущен стандарт IEEE 1044-1999 «Стандартная классификация для программных аномалий» (IEEE Standard Classification for Software Anomalies) [3], в котором даётся несколько определений для терминов, используемых в рамках данной классификации:

- *дефект (defect)* – несовершенство или недостаток в работающем продукте, когда этот работающий продукт не соответствует требованиям или спецификациям и требуется либо его исправление, либо замена;
- *оплошность (error)* – действие человека, которое приводит к некорректному результату;
- *повреждение (failure)* – прекращение возможности продукта выполнять требуемую функцию или неспособность выполнять функцию в ранее указанных ограничениях;
- *ошибка (fault)* – сообщение об ошибке в программе;
- *проблема (problem)* – (A) трудность или неясность, испытанная одним или несколькими пользователями, которая случилась в процессе использования системы. (B) Негативная ситуация, которую необходимо преодолеть.

Стоит отметить, что признать данный набор определений полным не представляется возможным, в связи с тем, что существуют правила, нарушение которых может трактоваться как программный дефект, но его описание не подходит ни под одно из вышеперечисленных определений.

Например, в работе [4] есть «Правило 4», которое гласит, что длина кода функции не должна превышать один стандартный лист, где каждая декларация и каждое выражение размещено на одной строке. Обычно это означает, что текст функции должен быть не более 60 строк. Считается, что трудно оперировать фрагментами кода более этого размера. Однако, нарушение этого правила вряд ли однозначно приведет к наличию ошибки в программе. А стандарт Ассоциации надёжности программного обеспечения в автомобильной индустрии (Motor Industry Software Reliability Association) содержит правило MISRA 2004 [5] 5.1: «Идентификаторы (внутренние и внешние) не должны полагаться на значимость более чем 31 символа».

Нарушение этого правила вряд ли приведет к нарушению работоспособности программного обеспечения, если используемый разработчиками компилятор языка Си поддерживает идентификаторы большего размера. Но программа, предназначенная для работы на борту автомобиля и содержащая идентификаторы с длиной более 31 символа, не пройдёт сертификацию по стандарту MISRA, что может оказаться достаточным условием для отказа прохождения программой и автомобилем в целом внутреннего или внешнего аудита безопасности. В связи с этим, для производителя автомобиля, которому важно соответствие программы правилам MISRA, нарушение правила будет считаться критической ошибкой, пропуск которой может привести к серьезным финансовым потерям. Но, если разработчиками используется компилятор, который учитывает только первые 31 символ идентификатора, и не выводит предупреждение о том, что используется идентификатор большей длины, то это может привести к серьезному нарушению логики работы программы.

С другой стороны, наибольший интерес представляют дефекты в программном обеспечении, которые приводят к некорректной работе или сбою в работе программы.

3. История развития методов статического анализа программ

На заре компьютерной эры достаточно большое внимание уделялось тестированию, как методу обеспечения качества программных систем. В 60-х годах XX века тесты пытались проводить таким образом, чтобы покрыть все возможные участки программы на всех возможных входных данных и показать корректность работы программы (так называемое исчерпывающее тестирование). Но количество возможных путей исполнения сколь-нибудь сложной программы и количество значений входных данных оказывалось настолько велико, что подобный подход к обеспечению качества программного обеспечения был признан непрактичным [6]. В лекции «О надёжности программ» Эдскер Вибе Дейкстра утверждает, что тестирование может использоваться очень эффективно для того, чтобы показать наличие ошибки, но не может показать их отсутствие [7].

Как отмечается в статье «Что мы знаем о методах обнаружения ошибок?» [2], в процессе инспекции кода программ обнаруживается в среднем от 25% до 50% дефектов в программе, а в процессе тестирования от 30% до 60%. То есть в среднем в проверенной опытными разработчиками программного обеспечения и хорошо протестированной программе остаётся около половины дефектов, не обнаруженных на этапах, предназначенных для обеспечения качества программы. При этом экспертом, проводящим инспекцию кода, в среднем обнаруживается от 1 до 2,5 дефектов в час.

В статье, посвященной истории создания языка Си [8], Деннис Ритчи (Dennis M. Ritchie) отмечает, что несмотря на то, что в первой редакции книги «Язык

программирования Си» было указано большинство правил, которые привели систему типов языка Си к его современной форме, многие программы были написаны в старом, более свободном стиле, и компиляторы это позволяли. Для того, чтобы обратить больше внимания разработчиков программ на официальные правила языка, обнаруживать разрешенные, но подозрительные конструкции, и помочь найти несоответствие интерфейсов, не обнаруживаемые простыми механизмами в процессе раздельной компиляции, Стив Джонсон (Steve Johnson) адаптировал свой компилятор *pcc* [9] для создания инструмента *lint* [10], который сканировал файлы исходного кода программы и отмечал сомнительные конструкции. Особенностью программы *lint* являлась возможность сравнивать соответствие и находить отсутствие противоречий во всей программе, путём сканирования множества исходных файлов и сравнения типов аргументов функций в месте вызова с типами аргументов функции, указанных в определении.

Согласно утверждениям технического отчёта «Следующее поколение статического анализа» [11] программу *lint* можно назвать первым инструментом статического анализа кода, но на самом деле *lint* не разрабатывался с целью обнаружения дефектов, которые приводят к проблемам времени исполнения. Скорее, целью создания инструмента было обнаружение подозрительных и непортируемых конструкций в коде с целью помощи разработчикам создавать более согласованный код. Под «подозрительными конструкциями» подразумевается технически корректный, с точки зрения спецификации языка, исходный код (например, Си, Си++), который может привести к такому выполнению программы, которое не подразумевалось разработчиком. Проблема заключалась в том, что такой подозрительный код мог исполняться и часто исполнялся корректно. Поэтому из-за ограниченных возможностей инструмента *lint* уровень шума (ложноположительных предупреждений об ошибках) был экстремально высок, часто превышая соотношение шума и реальных дефектов 10:1.

В начале 2000-х стали появляться инструменты статического анализа программ второго поколения [11], такие как: Coverity Prevent, Klocwork и другие [12]. Основной особенностью данных инструментов было смещение фокуса с «подозрительных конструкций» на «дефекты времени исполнения», что потребовало комбинации изощрённого анализа путей исполнения с межпроцедурным анализом, чтобы понять, что происходит, когда управление передаётся от одной функции к другой внутри программной системы [13]. В связи с постоянной борьбой за точность и масштабируемость анализа инструменты второго поколения могли находить ограниченный набор дефектов достаточно точно, но либо анализ не масштабировался на миллионы строк исходного кода, либо мог работать достаточно быстро, но обладал малой точностью, как инструмент *lint*, привнося уже известные проблемы шума предупреждений [13, 14]. Борьба за иллюзорный баланс между точностью, масштабируемостью и производительностью анализа привела к проблеме ложноположительных предупреждений о дефектах в инструментах

второго поколения. Если для инструментов первого поколения проблема шума в результатах работы препятствовала их внедрению в индустрии, то инструменты второго поколения сдвинули развитие инструментов статического анализа программ с технологической мёртвой точки, позволяя обнаруживать осмысленные дефекты в программах, но разработчики программного обеспечения всё чаще ожидают лучшей точности результатов анализа.

В связи с развитием алгоритмов и появлением инструментов для решения формул в булёвых ограничениях (решателей) [15], в последнее время стали появляться инструменты статического анализа третьего поколения, которые объединяют классические методы анализа путей исполнения программы в комбинации с применением решателей. Согласно исследованию Чельфа и Чу [11] применение подобной комбинации позволило получить дальнейшие технологические преимущества для статического анализа и снизить высокие уровни предупреждений ошибок первого рода (ложноположительных предупреждений) на 15%.

4. Методы улучшения результатов работы инструментов статического анализа программ

Несмотря на постоянное улучшение качества результатов работы инструментов статического анализа программ, например, коммерческого инструмента Coverity Prevent [16], с постоянным уменьшением соотношения ложноположительных предупреждений к общему количеству предупреждений до 9,7% в среднем по всем типам дефектов, для отдельных проектов этот показатель оказывается на достаточно высоком уровне (менее 20% [17]). Это означает, что в среднем каждое пятое предупреждение о дефекте является ложным.

В связи с ограничениями инструментов статического анализа, связанными с компромиссом между обеспечением точности, производительности и масштабируемости анализа, а также принимая во внимание фундаментальную теорему об алгоритмической неразрешимости задачи об определении останова работы алгоритма, сформулированную Аланом Тьюрингом ещё в 30-х годах XX века, возникла задача повышения точности результатов статического анализа путем постобработки результатов работы инструментов анализа и комбинирования статических и динамических методов анализа.

В октябре 2016 года на конференции, посвященной анализу и манипуляциям над исходным кодом программ, была представлена обширная классификация методов постобработки результатов статического анализа [18]. Авторы классификации, применяя методы автоматизированной обработки научных статей и докладов на научных конференциях, собрали и классифицировали 79 статей по предлагаемым методам постобработки результатов работы инструментов статического анализа. Авторы обзора выделяют семь основных категорий методов, из которых один метод – упрощение инспекций

результатов работы инструментов статического анализа на основе обратной связи от пользователя, – является автоматизированным. Ниже кратко будут рассмотрены автоматические методы улучшения результатов работы инструментов статического анализа с целью повышения их качества, приведенные в этой статье. Особое внимание будет уделено методам совмещения статического и динамического анализа. Этот раздел будет расширен дополнительными примерами подходов и инструментов.

4.1 Кластеризация, основанная на схожести или связях

Задача методов кластеризации – объединить в одно множество (кластер) группу дефектов, с дальнейшей целью оценки одного предупреждения о дефекте из группы как ложноположительное или истинно-положительное и применить эту оценку для всей группы сразу. Кластеризация разделяется авторами на прямую и косвенную. Различие заключается в том, что для прямой кластеризации гарантируется зависимость или связи между похожими или связанными предупреждениями, сгруппированными вместе. Например, в работе Ли и др. [19] кластеризация предупреждений о дефектах производится на основе арифметической связи между индексными переменными при доступе к буферу. При косвенной кластеризации похожие предупреждения группируются по синтаксической или структурной схожести кода или характеристик предупреждения, вычисленных самим инструментом статического анализа кода или независимо от него. Так в работе Фрая и других [20] предлагается подход к вычислению метрики близости на основе различных характеристик предупреждения о дефекте, таких как: название функции, контекст строки, в которой обнаружен потенциальный дефект, и другие.

4.2 Ранжирование

Целью ранжирования является выстраивание предупреждений в выводе статического анализатора таким образом, чтобы те предупреждения, которые имеют большую вероятность быть истинно-положительными, оказывались наверху списка предупреждений, а с меньшей – внизу списка.

Статистическое ранжирование – наиболее распространенная техника приоритезации предупреждений. Например, в работе Кременека и Энглера [21] применена простая статистическая модель ранжирования дефектов. Она основана на наблюдении, что в коде, содержащем много успешных проверок и малое количество предупреждений, как правило содержится ошибка. С точки зрения работы статического анализатора, проводящего анализ потока данных в программе, успешная проверка останавливает распространение факта об инициализации ошибочной ситуации или, при обратном проходе, – факта о потенциально опасной операции, на пути исполнения программы.

Ранжирование по истории изменений. В этой категории дефекты ранжируются с использованием истории исправлений дефектов. Ким и Эрнст

[22, 23] дают предупреждению более высокий приоритет, если оно относится к категории дефектов, которые быстро исправляются программистами, то есть считаются более важными.

Самоадаптирующееся ранжирование, основанное на обратной связи. В этой категории дефекты ранжируются на основе обратной связи от пользователя. Например, Шен и др. [24] впервые предложили назначать каждому шаблону дефекта предопределенную вероятность того, что предупреждение о наличии дефекта истинно и ранжировать дефекты в зависимости от этой вероятности. А в последствии изначальное ранжирование изменяется на основе реакции пользователя в процессе инспекции предупреждений о дефектах.

Из других методов стоит выделить ранжирование предупреждений на основе статического вычисления вероятности исполнения мест, для которых предупреждения даны [25], и ранжирование предупреждений на основе слияния результатов нескольких инструментов статического анализа [26, 27].

4.3 Отсечение или классификация предупреждений

Подходы, отнесенные к этой категории, предлагают классифицировать дефекты при помощи двоичной классификации как имеющие практическое значение (actionable) или нет. При этом стоит отметить, что данный подход в результате может привести к сокрытию предупреждений о реальных дефектах в программе.

Классификация на основе машинного обучения. В работе Ханама и др. [28] классификация достигается за счёт нахождения похожих шаблонов в коде программы, окружающем место, для которого получено предупреждение о возможной ошибке. Юскел и Созер [29] оценили 34 алгоритма машинного обучения в их исследовании, используя 10 различных характеристик предупреждения.

Идентификация дельты предупреждений. Метод заключается в идентификации различия предупреждений в результате работы инструмента статического анализа через сравнение результатов анализа кода на предыдущей и текущей версии программного обеспечения. Например, Спакко и др. [30] идентифицировали новые предупреждения в сравнении с предыдущей версией, сравнивая предупреждения двумя способами: *составление пар* и *подписей предупреждений*. Чимдялвар и Кумар [31] отсекали вновь появляющиеся ложноположительные предупреждения о дефектах в развивающихся программных системах путем анализа влияния внесенных изменений в текущей системе и подавляя те предупреждения, которые не зависят от внесенных изменений.

Также стоит отметить работу Муске и др. [32], в которой авторы применяют *логистическую регрессию* для определения вероятности каждого предупреждения быть ложным или нет с точки зрения необходимости исправления ошибки в коде. Авторы выделили несколько групп факторов, влияющих на вероятность отнесения предупреждения к истинно-

положительному или ложноположительному, а также на вероятность того, что обнаруженная ситуация в коде должна быть исправлена.

4.4 Избавление от ложноположительных предупреждений

В рамках этого подхода используются более точные техники, такие как проверка моделей и символьное исполнение, для идентификации и устранения ложноположительных предупреждений инструментов статического анализа.

Достижение масштабируемости. В своей работе Пост и др. [33] предлагают подход к инкрементальному расширению контекста для верификации начиная с минимального контекста одной функции с постепенным расширением его на функции, вызывающие данную функцию и далее, по мере необходимости.

Улучшение производительности и эффективности. В другой работе [34] Муске и другие отмечают низкую производительность методов проверки моделей. В связи с этим авторы предлагают техники предсказания результата проверки модели, в связи с чем добиваются лучшей производительности избавления от ложноположительных предупреждений за счёт уменьшения количества вызовов для проверки моделей.

Стоит отметить, что существует ряд работ [35], [36], [37] о комбинации статического анализа и проверки моделей, где описываются подходы к тесной интеграции этих двух методов, в процессе которой они итеративно обмениваются информацией.

4.5 Создание легковесных инструментов статического анализа

В эту категорию попали инструменты легковесного статического анализа, которые способны с увеличенной производительностью анализировать очень большие программные системы. Тем не менее данные инструменты не гарантируют нахождение всех дефектов определенного типа.

Ховемейер и Пуг [38] разработали автоматические обнаружители различных шаблонов ошибок в Java-программах. Получившийся инструмент FindBugs получил одобрение как академическим сообществом, так и индустрией, несмотря на то, что проводит неглубокий внутрипроцедурный анализ.

Splint [39] – пример другого инструмента, использующего легковесный статический анализ для обнаружения возможных уязвимостей. Анализ, проводимый инструментом Splint, подобен анализу, проводимому компилятором, он эффективен и масштабируем и при этом обнаруживает широкий диапазон реализационных изъянов.

4.6 Комбинация статического и динамического анализа программ

В основном в эту группу попали инструменты, объединяющие подходы статического и динамического анализа программ.

В работе [40] описывается инструмент Check'N'Crash, который объединяет инструмент статического анализа ESC/Java [41] и JCrasher [42] – генератор тестов, путем извлечения конкретных значений свойств, специфических для абстрактных условий ошибок, найденных статическим анализатором, при помощи решения системы ограничений, с последующей генерацией тестов для воспроизведения дефекта.

Как развитие данного подхода в работе [43] авторы описывают инструмент DSD-Crasher, в котором используется трехступенчатый подход, состоящий из: исполнения программы с целью динамического обнаружения заложенного поведения программы для ограничения множества входных значений программы; статического анализа с целью обнаружения потенциальных ошибок; и динамической верификации обнаруженных потенциальных ошибок для проверки их достижимости.

Слайсинг программ также может использоваться для эффективного объединения статического и динамического анализа. В работе [44] описывается инструмент SANTE, который объединяет подход статического анализа для обнаружения дефектов в программах и генерацию тестов для подтверждения найденных дефектов. В связи с тем, что генерация тестов для больших программ может занимать значительное время, авторы инструмента предлагают применять слайсинг программ с целью уменьшения исходного кода, а также исключения незначительных деталей программы с точки зрения генерации теста, до начала самого процесса генерации тестов.

Ли и другие в работе [45] предлагают подход «остаточного исследования», в котором используется динамический анализ, как сервис для статического анализа с целью подтверждения во время исполнения является ли ситуация, найденная статическим анализатором истинной ошибкой, или нет.

В дополнение к инструментам и подходам, рассмотренным в работе [19], стоит отметить обзор [46], в котором авторы сконцентрировались на инструментах и подходах, так или иначе использующих объединение техник контроля качества программного обеспечения, объединяющих статический анализ программ и динамический анализ программ.

В работе [47] предлагается подход к объединению статического анализа с целью обнаружения подозрительных мест в программе и генерации тестов для подтверждения обнаруженных в процессе статического анализа возможных уязвимостей при помощи динамического анализа. Подход основан на статическом анализаторе SVR [48], построенном на базе промышленного компилятора GCC, который начиная с версии 4, включает в себя среду Tree-SSA [49], созданную для облегчения построения статических анализаторов на базе универсального промежуточного представления GIMPLE. Статический анализатор по исходному коду программы строит модель для инструмента Moped [50], на основе которой проверяются свойства безопасности программы. В качестве результата статического анализатора получается набор путей в программе, которые приводят к потенциальному дефекту. Далее для

этих путей строятся входные данные и проверяется достижимость определенных состояний в программе в процессе её запуска.

В работе [51] рассматривается подход, названный авторами как обобщенный алгоритм анализа программы. В основе подхода лежит идея объединения статического и динамического анализа в рамках одного инструмента JNuke, который абстрагирует алгоритм работы анализатора от метода анализа, и предоставляет возможность анализировать различные свойства программы, вычисляемые статически или обнаруживаемые динамически через единый интерфейс.

В работе [52] описывается идея, лежащая в основе ранее упоминавшегося инструмента SANTE, который развивает идеи, заложенные в инструменте PathCrawler [53], и объединяет подход статического анализа исходного кода программ на языке Си Frama-C [54, 55] и динамическое символьное исполнение программ с целью построения тестовых наборов, на которых воспроизводятся найденные дефекты.

Инструмент Yogi [56], разработанный в научном подразделении Microsoft Research для проекта тестирования драйверов операционной системы Windows, представляет комбинацию статического анализатора свойств безопасности программы, описанных на языке Slic [57] и интерпретатора промежуточного представления программы для Yogi, который способен проводить как символьное выполнение Yogi-представления программы, так и симуляцию выполнения программы на базе Yogi-представления с целью подтверждения достижимости ошибочной ситуации описанной на Slic.

В работе [58] авторами предлагается подход к объединению результатов работы статического и динамического анализа программ на языке Си с целью обнаружения уязвимости выхода за границы буфера в памяти. В процессе статического анализа вычисляются последовательности помеченных зависимостей между входными данными и уязвимыми операциями, которые в последствии используются в процессе вычисления входных данных при помощи генетического алгоритма, использующего фитнес-функцию, основанную на концепции частотно-спектрального анализа помеченных зависимостей программы, представленной в работе [59].

В работе [60] описывается инструмент ConDroid, предназначенный для анализа программ операционной системы Android. Основной особенностью этого инструмента является использование статического анализа кода программы с целью систематического обнаружения критических, с точки зрения безопасности, регионов программы, путей достижения этих регионов и итеративного динамического анализа. Итеративный динамический анализ в этом инструменте реализует идею кокретно-символьного (*англ.* concolic от concrete and symbolic) исполнения, в процессе которого программа исполняется на конкретных входных данных с целью сбора трассы исполнившихся операций и вычисления новых входных данных на основе символьного решения ограничения пути, собранного в процессе исполнения

программы. Итеративный динамический анализ на каждой итерации вычисляет выходные данные для программы таким образом, чтобы в конечном итоге провести исполнение программы по пути, вычисленному на этапе статического анализа.

Похожий подход используется в инструменте DyTa [61], предназначенном для анализа программ, написанных на языке C#. Инструмент использует статический анализ для определения мест потенциальных дефектов с возможными предусловиями. Далее проводится инструментация кода программы с целью добавления операций проверки утверждений о предусловиях в определенных точках программы. После этого программа запускается на исполнение под управлением инструмента Pex [62], проводящего генерацию тестовых наборов методом прозрачного ящика, и в процессе работы проверяет утверждения, вставленные на этапе инструментации.

Инструмент IntelliDroid [63] использует комбинацию статического и динамического анализа с целью обнаружения уязвимостей безопасности в программах операционной системы Android. Статический анализатор проверяет код программы в формате байткода DEX [64] с целью обнаружения использования вызовов к функциям стороннего API. Динамический анализатор использует информацию о месте использования стороннего API вместе с путями достижения точек вызова и генерирует входные данные программы при помощи решателя формул в булёвых ограничениях Z3 [65] с целью обнаружения использования стороннего API в процессе исполнения программы.

Инструмент STAR [66] ещё один инструмент, созданный для генерации тестовых сценариев, воспроизводящих критические ошибки в Java-программах. Используя информацию об необработанной исключительной ситуации в Java-программе, инструмент пытается по сообщению об ошибке получить информацию о трассировке стека, далее пытается восстановить предусловия для ошибки при помощи символьного исполнения и затем вычислить ограничения пути с целью генерации входных данных для тестового сценария воспроизведения ошибки.

В работе, посвященной инструменту AEG [67] описывается подход генерации входных данных для эксплуатации уязвимости переполнения буфера или некорректного использования форматной строки. Подход построен на совмещении техники статическом анализа исходного кода, статического анализа бинарного кода, представленного в формате LLVM и динамического анализа исполняемого кода с целью получения конкретной информации об адресах функций и буферов для построения эксплойта, эксплуатирующего уязвимость, найденную на стадии статического анализа.

В инструменте [68] предлагается подход для классификации дефектов утечки памяти при помощи совмещения статического анализа исходного программы инструментом Fortify [69] с дальнейшим конкретно-символьным (*англ.*

concolic) анализом программы на основе модифицированного инструмента CREST [70] для генерации тестовых наборов для программ на языке C.

В работе [71] рассматривается подход для автоматической генерации тестов для дефектов связанных с выходом за границы буфера в памяти. При помощи статического анализа находятся точки потенциальных дефектов, которые в дальнейшем подтверждаются при помощи динамического символьного исполнения, результатом которого являются наборы входных данных, на которых воспроизводится дефект.

Классифицируя данные работы можно вывести несколько основных направлений, используемых для объединения подходов статического и динамического анализа программ:

- статический анализ и генерация тестов [40, 43, 45, 58];
- статический анализ и проверка моделей [47];
- статический анализ и верификация в процессе исполнения [52];
- статический анализ и симуляция исполнения [56];
- статический анализ и символьное исполнение [44, 60, 61, 63, 67, 68, 71].

5. Заключение

В связи с наличием известных фундаментальных и технологических ограничений на методы статического анализа программ проблема уменьшения ложноположительных предупреждений об ошибках статических анализаторов остаётся актуальной. При наличии инструментов статического анализа промышленного качества, таких как Klocwork и Coverity, а также сред динамического анализа программ, таких как Valgrind [72] и QEMU [73] и инструментов Memcheck [74] и S^2E [75], построенных на их основе, на данный момент не существует промышленных или широко используемых в индустрии инструментов, совмещающих статический и динамический анализ программ. Несмотря на обширную классификацию подходов к улучшению качества результатов работы инструментов статического анализа, наиболее перспективным направлением решения данной проблемы представляется разработка и применение методов совмещения статического и динамического анализа, которые позволят избежать с одной стороны неточности, присущей методам статического анализа, а с другой стороны – высокой вычислительной сложности, присущей методам динамического анализа программ.

Список литературы

[1]. Sketch of The Analytical Engine Invented by Charles Babbage by L. F. Menabea from the Bibliotheque Universalle de Geneve, October, 1942, No. 82. With notes upon the Memoir by the translator Ada Augusta, countess of Lovelace. https://www.fourmilab.ch/babbage/sketch.html, , дата обращения 05.05.2017

[2]. Per Runeson, Carian Andersson, Thomas Thelin, Anneliese Andrews, Tomas Berling. What Do We Know about Defect Detection Methods? *IEEE Software May/June 2006*

[3]. IEEE 1044-2009 Standard Classification for Software Anomalies. *IEEE. 3 Park Avenue, New Yourk, NY 10016-5997, USA, 7 January 2010, ISBN 978-0-7381-6114-3*

[4]. Gerald J. Holzmann. The Power of 10: Roles for Developing Safety-Critical Code. *Computer/ 2006, vol. 39, no. 6, pp 95-97*

[5]. MISRA C: 2004 Guidelines for the use of the C language in critical systems. *First published October 2004, by MIRA Limited, Watling Street, Nuneaton, Warkwickshire CV10 0TU UK, ISBN 978-0-9524156-4-0*

[6]. E. J. Weyuker, T. J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on software engineering, 6(3):236-246.* May 1980.

[7]. E. W. Dijkstra. On the reliability of the programs. https://www.cs.utexas.edu/users/EWD/ewd03xx/EWD303.PDF, дата обращения 05.05.2017

[8]. Dennis M. Ritchie. The development of the C language. *Proceedings of HOPL-II The second ACM SIGPLAN conference on History of programming languages. Cambridge, MA, USA – April 20-23, 1993, pp. 201-208*

[9]. S. C. Johnson. A Portable Compiler: Theory and Practice. *Proceedings of 5th ACM POPL Symposium*, January 1978

[10]. S. C. Johnson. Lint, a Program Checker. *Unix Programmer's manual*, Seventh Edition, Vol. 2B, M.D. McIlroy and B.W. Kernigan, eds. AT&T Bell Laboratories: Murray Hill, NJ, 1979.

[11]. Benjamine Chelf, Andy Chou. The next generation of Static Analysis. *Coverity*, March 18, 2008. http://www.coverity.com/library/pdf/Coverity_White_Paper-SAT-Next_Generation_Static_Analysis.pdf, , дата обращения 05.05.2017

[12]. Pär Emanuelsson, Ulf Nilsson, A Comparative Study of Industrial Static Analysis Tools. *Technical report. Department of Computer and Information Science, Linköping University.* Linköping, Sweden, 2008.

[13]. Dawson Engler, Benjamin Chelf, Andy Chou, Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. OSDI'00 *Proceedings of the 4th conference on Symposium on Operating System Design and Implementation, Volume 4, Article No. 1*. San Diego, California – October 22-25, 2000

[14]. Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, Robert Bowdidge. Why don't software developers use static analysis tools to find bugs?. *ICSE'13 Proceedings of the 2013 International conference on Software Engineering.* San Francisco, CA, USE, May 18-26, 2013

[15]. John Franco, John Martin. A history of Satisfiability. *Handbook of Satisfiability.* IOS Press, 2009 doi:10.3233/978-1-58603-929-5-3

[16]. Coverity Scan: 2012 Open Source Report. http://wpcme.coverity.com/wp-content/uploads/2012-Coverity-Scan-Report.pdf, дата обращения 05.05.2017

[17]. Coverity Scan. Project Spotlight: Python. http://wpcme.coverity.com/wp-content/uploads/2013-Coverity-Scan-Spotlight-Python.pdf, дата обращения 05.05.2017

[18]. Tukaram Muske, Alexander Serebrenik. Survey of Approaches for Handling Static Analysis Alarms. *Proceedings of IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM).* Raleigh, NC, USA. October 2-3, 2016

[19]. Woosuk Lee, Wonchan Lee, Kwengkeun Yi. Sound non-statistical clustering of static analysis alarms. *VMCAI'12 Proceedings of the 13th international conference on*

verification, model checking and abstract verification interpretation. Philadelphia, PA, USA. January 22-24, 2012.

[20]. Zachary P. Fry, Westley Weimer. Clustering static analysis defect reports to Reduce maintenance costs. *WCRE'13 Proceeding of 30th working conference on reverse engineering.* Koblenz, Germany. October 14-17, 2013.

[21]. Ted Kremenek, Dawson Engler. Z-ranking: using statistical analysis to counter the impact of static analysis approximations. *SAS'03 Proceedings of the 10th International conference on static analysis.* San Diego, CA, USA. June 11, 2003.

[22]. Sunghun Kim, Michael D. Ernst. Prioritizing Warning Categories by Analyzing Software History. *MSR'07 Proceedings of the Fourth International Workshop on Mining Software Repositories.* Minneapolis, MN, USA. May 20-26, 2007.

[23]. Sunghun Kim, Michael D. Ernst. Which warnings should I fix first? *ESEC-FSE'07 Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering.* Dubrovnik, Croatia. September 03-07, 2007

[24]. Haihao Shen, Jianhong Fang, Jianjun Zhao. EFindBugs: Effective Error Ranking for FindBugs. *ICST'11 Proceedings of the 2011 Fourth IEEE International conference on Software Testing, Verification and Validation.* Berlin, Germany. March 21-25, 2011

[25]. Sunghun Kim, Michael D. Ernst. Prioritizing Software Inspection Results using Static Profiling. *SCAM'06 Source code analysis and manipulation.* Philadelphia, PA, USA. December 11, 2006.

[26]. Deguang Kong, Quan Zheng, Chao Chen, Jianmei Shuai, Ming Zhu. ISA: A source code static vulnerability detection system based on data fusion. *InfoScale'07 Proceedings of the 2nd international conference on Scalable information systems.* Suzhou, China. June 06-08, 2007

[27]. Na Meng, Qianxiang Wang, Qian Wu, Hong Mei. An approach to merge results of multiple static analysis tools. *QSIC'08 Proceedings of the 2008 the eighth international conference on quality software.* Oxford, UK. August 12-13, 2008

[28]. Quinn Hanam, Lin Tan, Reid Holmes, Patrick Lam. Finding patters in static analysis alerts. Improving actionable alert ranking. *MSR 2014 Proceedings of the 11th working conference on mining software repositories.* Hyderabad, India. May 31 – June 01, 2014

[29]. Ulas Yüskel, Hasan Sözer. Automated classification of static code analysis alerts: a case study. *ICSM'13 Proceedings of the 2013 IEEE international conference on software maintenance.* Eindhoven, Netherlands. September 24-26, 2013

[30]. Jaime Spacco, David Hovermeyer, William Pugh. Tracking defect warnings across versions. *MSR'06 Proceedings of the 2006 international workshop on mining software repositories.* Shanghai, China. May 22-23, 2006

[31]. Brahti Chimdyalwar, Shrawan Kumar. Effective false positive filtering for evolving software. *ISEC'11 Proceedings of the 4th India software engineering conference.* Thiruvananthapuram, Kerala, India. February 24-27, 2011

[32]. J. R. Rithruff, J. Penix, J. D. Morgenthaler, S. Elbaum, G. Rothermel. Predicting accurate and actionable static analysis warnings: and experimental approach. *ICSE'08 Proceedings of the 30th international conference on software engineering.* Leipzig, Germany. May 10-18, 2008

[33]. H. Post, C. Sinz, A. Kaiser, T. Gorges. Reducing false positives by combining abstract interpretation and bounded model checking. *ASE'08 Proceedings of the 2008 23rd IEEE/ACM international conference on automated software engineering.* L'Aquila, Italy. September 15-19, 2008

[34]. Tukaram Muske, Advaita Datar, Mayur Khanzode, Kumar Madhukar. Efficient elimination of false positives using bounded model checking. *ISSRE'15 Proceedings of the 2015 IEEE 26th international symposium on software reliability engineering.* Gaithersburg, MD, USA. November 2-5, 2015

[35]. M. Junker, R. Huuck, A. Fehnker, A. Knapp. SMT-based false positive elimination in static program analysis". *ICFEM'12 Proceedings of the 14th international conference on formal engineering methods: formal mehods and software engineering.* Kyoto, Japan. November 12-16, 2012

[36]. G. Brat, W. Visser. Combining static analysis and model checking for software analysis tools. *ACE'01 Proceedings of the 16th IEEE international conference on automated software engineering.* San Diego, CA, USA. November 26-29, 2001

[37]. A. Fenker, R. Huuck. Model checking driven static analysis for the real world: designing and tuning large scale bug detection. *Journal Innovations in systems and software engineering. Volume 9, Issue 1, March 2013*

[38]. D. Hovemeyer, W. Pugh. Finding bugs easily. ACM *SIGPLAN notices. Volume 39, issue 12, December 2004*

[39]. D. Evans, D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software. Volume 19, Issue 1, 2002*

[40]. C. Csallner, Y. Smaragdakis. Check'N'Crash: combining static checking and testing. *ICSE'05 Proceedings of the 27th international conference on software engineering.* St. Luis, MO, USA. May 15-21, 2005

[41]. C. Flanagan, K Rustan, M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata. Extended static checking for Java. *PLDI'02 Proceedings of the ACM SIGPLAN 2002 conference on programming language design and implementation.* Berlin, Germany. June 17-19, 2002

[42]. C. Csallner, Y. Smaragdakis. JCrasher: an automatic robustness testing tester for Java. *Software – Practice & Experience. Volume 34, Issue 11.* September 2004.

[43]. C. Csallner, Y. Smaragdakis. DSD-Crasher: a hybrid analysis tool for bug finding. *ISSTA'06 Proceedings of the 2006 international symposium on software testing and analysis.* Portland, Maide, USA July 17-20, 2006

[44]. O. Chebaro, N Kosmatov, A. Giorgetti, J. Julliand. Programs slicing enhances a verification technique combining static and dynamic analysis. *SAC'12 Proceedings of the 27th annual ACM symposium of applied computing.* Trento, Italy. March 26-30, 2012

[45]. K. Li, C Reichenbach, C. Csallner, Y. Smaragdakis. Residual investigation: predictive and precise bug detection. *ISSTA'2012 Proceedings of the 2012 international symposium on software testing and analysis.* Minneanapolis, MN, USA. July 15-20, 2012

[46]. F. Elberzhager, J. Münch, V.T. Ngoc Nha. A systematic study on the combination of static and dynamic quality assurance techniques. *Infromation and siftware technology. Vol 54, Issue1. January, 2012*

[47]. A. Hanna, H. Z. Ling, X Yang, M. Debbabi. A synergy between static and dynamic analysis for the detection of software security vulnerabilities. *OTM'09 Proceedings of the confederated international congress, CoopIS, DOA, IS and ADBASE 2009 on on the move to meaningful internet systems: part II.* Vilamoura, Protugal. November 01-06, 2009

[48]. R. Hadjidj, X. Yang, S. Tlili, M. Debabi. Model-checking for software vulnerabilities detection with multi-language support. *PST'08 Proceedings of the 2008 sixth annual conference on privacy, security and trust.* Fredericton, NB, Canada. October 01-03, 2008.

[49]. D. Novillo. Tree SSA: a new optimization infrastructure for GCC. *Proceedings of the GCC developers summit.* Ottawa, ON, Canada. May 25-27, 2003

[50]. S. Schwoon. Model-checking pushdown systems. *PhD thesis.* Technischen Universität München. 2002

[51]. C. Artho, A. Biere. Combined static and dynamic analysis. Technical Report 466, ETH Zürich, Zürich, Switzerland, 2005.

[52]. O. Chebaro, N. Kostomarov, A. Giorgetti, J. Julliand. Combining static analysis and test generation for C program debugging. *TAP'10 Proceedings of the 4th international conference on tests and proofs.* Málaga, Spain. July 01-02, 2010

[53]. N. Williams, B. Marre, P. Mouy, M. Roger. PathCrawler: automatic generation of tests by combining static and dynamic analysis. *EDCC'05 Proceedings of the 5th European conference on dependable computing.* Budapest, Hungary. April 20-22, 2005

[54]. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, B. Yakobowski. Frama-C: a software analysis perspective. *SEFM'12 Proceedings of the 10th international conference on software engineering and formal methods.* Thesaloniki, Grece. October 01-05, 2012

[55]. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, B. Yakobowski. Frama-C: a software analysis perspective. *Formal aspects of computing. Volume 27, issue 3. May, 2015*

[56]. A. V. Nori, S. K. Rajamani, S. Tetali, A. V. Thakur. The Yogi ptoject: software property checking via static analysis and testing. *TACAS'09 Proceedings of the 15th international conference on tools and algorithms for the construction and analysis of systems: held as part of the ETAPS'09 joint European conferences on theory and practice of software.* York, UK. March 22-29, 2009.

[57]. T. Ball, S. K. Rajamani. Slic: a specification language for interface checking (of C). *Technical report MSR-TR2001-21, Microsoft Research.* Redmond, WA, USA. January, 10. 2002

[58]. S. Rawat, D. Ceara, L. Mounier, M.-L. Potet. Combining static and dynamic analysis of vulnerability detection. *Cornell University Library arXiv:1305.3883.* May 16, 2013

[59]. T. Ball. The concept of dynamic analysis. *ESEC/FSE-7 Proceedings of the 7th European software engineering conference held jointly with 7th ACM SIGSOFT international symposium on foundations of software engineering.* Toulouse, France. September 06-10, 1999

[60]. J. Schütte, R. Fedler, D. Titze. ConDroid: targeted dynamic analysis of Android applications. *AINA'15 Proceedings of IEEE 29th international conference on advanced information networking and applications.* Gwangui, South Korea. March 24-27, 2015

[61]. X. Ge, K. Taneja, T. Xie, N. Tillmann. DyTa: dynamic symbolic execution guided with static verification results. *ICSE'11 Proceedings of the 33th international conference on software engineering.* Waikiki, Honolulu, HI, USA. May 21-28, 2011

[62]. N. Tillmann, J. de Hallex. Pex – white box test generation for .NET. *TAP'08 Proceedings of the 2nd international conference on tests and proofs.* Prato, Italy. April 09-11, 2008

[63]. M. Y. Wong, D. Lie. IntelliDroid: a targeted input generator for the dynamic analysis of android malware. *NDSS'16 The network and distributed system security symposium 2016.* San Diego, CA, USA. February 21-24, 2016

[64]. H. Gunadi. Formal certification of non-interferent Android bytecode (DEX bytecode). *ICECCS'15 Proceedings of the 2015 20th international conference on engineering and complex computer systems.* Gold Coast, Australia. December 9-12, 2015

[65]. L. De Moura, N. Bjørner. Z3: an efficient SMT solver. *TACAS'08/ETAPS'08 Proceedings of the theory and practice of software, 14th international conference on tools and algorithms for the constructions and analysis of systems.* Budapest, Hungary. March 29 – April 06, 2009

[66]. Chen N., Kim S. STAR: stack trace based automatic crash reproduction via symbolic execution. *IEEE transactions on software engineering. 2015, volume 41, issue 2.*

[67]. T. Avgerinos, S. Kil Cha, A. Rebert, E. J. Schwartz, M. Woo, D. Brumley. Automatic exploit generation. *Communications of the ACM, volume 57, issue 2, February 2014.*

[68]. M. Li, Y. Chen, L. Wang, G. Xu. Dynamically validating static memory leak warnings. *ISSTA'13 Proceedings of the 2013 international symposium on software testing and analysis.* Lugano, Switzerland. July 15-20, 2013.

[69]. HP Fortify. https://saas.hpe.com/en-us/software/sca, дата обращения 05.05.2017:

[70]. CREST – automatic test generation tool for C. https://github.com/jburnim/crest, дата обращения 05.05.2017

[71]. D. Babić, L. Martignoni, S. McCamant, S. Song. Statically-directed dynamic automated test generation. *ISSTA'11 Proceedings of the 2011 international symposium on software testing and analysis.* Toronto, Ontario, Canada. July 17-21, 2011

[72]. N. Nethercote, J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *PLDI'07 Proceedings of the 28th ACM SIGPLAN Conference on programming languages design and implementation.* San Diego, CA, USA. June 10-13, 2007

[73]. F. Bellard. QEMU, a fast and portable dynamic translator. *ATEC'05 Proceedings of the annual conference on USENIX annual technical conference.* Anaheim, CA, USA. April 10-15, 2005

[74]. J. Seward, N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. *ATEC'05 Proceedings of the annual conference on USENIX annual technical conference.* Anaheim, CA, USA. April 10-15, 2005

[75]. V. Chipounov, V. Kuznetsov, G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. ASPLOS XVI Proceedings of the sixteenth international conference on architectural support for programming languages and operating systems. Newport Beach, CA, USA. March 05-11, 2011

Survey on static program analysis results refinement approaches

A.Y. Gerasimov <agerasimov@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. In the present day, software development industry for different classes of computing devices grows at an extremely high speed. Continuously growing power of computational systems presents new opportunities to create powerful, often parallel, programs and software systems. This leads to the growth of complexity of software intended to manage these computational systems. The process of quality assurance calls for new approaches and methods both to check correctness and satisfiability of requirements for software as well as for check software for critical runtime defects and security vulnerabilities.

Program analysis is one of the methods intended to assure software quality. The static and dynamic analysis tool industry has been evolving aggressively since the first decade of 2000th. Nowadays there are many academic research and industrial tools for program analysis. But, due to fundamental limitations and engineering compromises for the sake of performance and scalability, static analysis tools cannot avoid false positive alarms. At the same time, reviewing static analysis tool alarms can take significant time of an experienced software engineer or a software quality assurance specialist. Hence, the task of automating refinement of static analysis tool results becomes more important. This survey covers approaches for static analysis tools result refinement. Approaches, which combine static and dynamic analysis of programs form the principal concern of this paper.

Keywords: static program analysis; dynamic program analysis; combined program analysis.

DOI: 10.15514/ISPRAS-2017-29(3)-6

For citation: Gerasimov A.Y. Survey on static program analysis results refinement approaches. *Trudy ISP RAN/Proc. ISP RAS*, vol. 29, issue 3, 2017, pp. 75-98 (in Russian). DOI: 10.15514/ISPRAS-2017-29(3)-6

References

[1]. Sketch of The Analytical Engine Invented by Charles Babbage by L. F. Menabea from the Bibliotheque Universalle de Geneve, October, 1942, No. 82. With notes upon the Memoir by the translator Ada Augusta, countess of Lovelace. https://www.fourmilab.ch/babbage/sketch.html, accessed 05.05.2017

[2]. Per Runeson, Carian Andersson, Thomas Thelin, Anneliese Andrews, Tomas Berling. What Do We Know about Defect Detection Methods? *IEEE Software May/June 2006*

[3]. IEEE 1044-2009 Standard Classification for Software Anomalies. *IEEE. 3 Park Avenue, New Yourk, NY 10016-5997, USA, 7 January 2010, ISBN 978-0-7381-6114-3*

[4]. Gerald J. Holzmann. The Power of 10: Roles for Developing Safety-Critical Code. *Computer/ 2006, vol. 39, no. 6, pp 95-97*

[5]. MISRA C: 2004 Guidelines for the use of the C language in critical systems. *First published October 2004, by MIRA Limited, Watling Street, Nuneaton, Warkwickshire CV10 0TU UK, ISBN 978-0-9524156-4-0*

[6]. E. J. Weyuker, T. J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on software engineering, 6(3):236-246.* May 1980.

[7]. E. W. Dijkstra. On the reliability of the programs. https://www.cs.utexas.edu/users/EWD/ewd03xx/EWD303.PDF, accessed 05.05.2017

[8]. Dennis M. Ritchie. The development of the C language. *Proceedings of HOPL-II The second ACM SIGPLAN conference on History of programming languages. Cambridge, MA, USA – April 20-23, 1993, pp. 201-208*

[9]. S. C. Johnson. A Portable Compiler: Theory and Practice. *Proceedings of 5th ACM POPL Symposium*, January 1978

[10]. S. C. Johnson. Lint, a Program Checker. *Unix Programmer's manual*, Seventh Edition, Vol. 2B, M.D. McIlroy and B.W. Kernigan, eds. AT&T Bell Laboratories: Murray Hill, NJ, 1979.

[11]. Benjamine Chelf, Andy Chou. The next generation of Static Analysis. *Coverity*, March 18, 2008. http://www.coverity.com/library/pdf/Coverity_White_Paper-SAT-Next_Generation_Static_Analysis.pdf, accessed 05.05.2017

[12]. Pär Emanuelsson, Ulf Nilsson, A Comparative Study of Industrial Static Analysis Tools. *Technical report. Department of Computer and Information Science, Linköping University.* Linköping, Sweden, 2008.

[13]. Dawson Engler, Benjamin Chelf, Andy Chou, Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. OSDI'00 *Proceedings of the 4th conference on Symposium on Operating System Design and Implementation, Volume 4, Article No. 1*. San Diego, California – October 22-25, 2000

[14]. Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, Robert Bowdidge. Why don't software developers use static analysis tools to find bugs?. *ICSE'13 Proceedings of the 2013 International conference on Software Engineering.* San Francisco, CA, USE, May 18-26, 2013

[15]. John Franco, John Martin. A history of Satisfiability. *Handbook of Satisfiability.* IOS Press, 2009. doi:10.3233/978-1-58603-929-5-3

[16]. Coverity Scan: 2012 Open Source Report. http://wpcme.coverity.com/wp-content/uploads/2012-Coverity-Scan-Report.pdf, accessed 05.05.2017

[17]. Coverity Scan. Project Spotlight: Python. http://wpcme.coverity.com/wp-content/uploads/2013-Coverity-Scan-Spotlight-Python.pdf, accessed 05.05.2017

[18]. Tukaram Muske, Alexander Serebrenik. Survey of Approaches for Handling Static Analysis Alarms. *Proceedings of IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM).* Raleigh, NC, USA. October 2-3, 2016

[19]. Woosuk Lee, Wonchan Lee, Kwengkeun Yi. Sound non-statistical clustering of static analysis alarms. *VMCAI'12 Proceedings of the 13th international conference on verification, model checking and abstract verification interpretation.* Philadelphia, PA, USA. January 22-24, 2012.

[20]. Zachary P. Fry, Westley Weimer. Clustering static analysis defect reports to Reduce maintenance costs. *WCRE'13 Proceeding of 30th working conference on reverse engineering.* Koblenz, Germany. October 14-17, 2013.

[21]. Ted Kremenek, Dawson Engler. Z-ranking: using statistical analysis to counter the impact of static analysis approximations. *SAS'03 Proceedings of the 10th International conference on static analysis.* San Diego, CA, USA. June 11, 2003.

[22]. Sunghun Kim, Michael D. Ernst. Prioritizing Warning Categories by Analyzing Software History. *MSR'07 Proceedings of the Fourth International Workshop on Mining Software Repositories.* Minneapolis, MN, USA. May 20-26, 2007.

[23]. Sunghun Kim, Michael D. Ernst. Which warnings should I fix first? *ESEC-FSE'07 Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering.* Dubrovnik, Croatia. September 03-07, 2007

[24]. Haihao Shen, Jianhong Fang, Jianjun Zhao. EFindBugs: Effective Error Ranking for FindBugs. *ICST'11 Proceedings of the 2011 Fourth IEEE International conference on Software Testing, Verification and Validation.* Berlin, Germany. March 21-25, 2011

[25]. Sunghun Kim, Michael D. Ernst. Prioritizing Software Inspection Results using Static Profiling. *SCAM'06 Source code analysis and manipulation.* Philadelphia, PA, USA. December 11, 2006.

[26]. Deguang Kong, Quan Zheng, Chao Chen, Jianmei Shuai, Ming Zhu. ISA: A source code static vulnerability detection system based on data fusion. *InfoScale'07 Proceedings of*

the 2nd international conference on Scalable information systems. Suzhou, China. June 06-08, 2007

[27]. Na Meng, Qianxiang Wang, Qian Wu, Hong Mei. An approach to merge results of multiple static analysis tools. *QSIC'08 Proceedings of the 2008 the eighth international conference on quality software.* Oxford, UK. August 12-13, 2008

[28]. Quinn Hanam, Lin Tan, Reid Holmes, Patrick Lam. Finding patters in static analysis alerts. Improving actionable alert ranking. *MSR 2014 Proceedings of the 11th working conference on mining software repositories.* Hyderabad, India. May 31 – June 01, 2014

[29]. Ulas Yüskel, Hasan Sözer. Automated classification of static code analysis alerts: a case study. *ICSM'13 Proceedings of the 2013 IEEE international conference on software maintenance.* Eindhoven, Netherlands. September 24-26, 2013

[30]. Jaime Spacco, David Hovermeyer, William Pugh. Tracking defect warnings across versions. *MSR'06 Proceedings of the 2006 international workshop on mining software repositories.* Shanghai, China. May 22-23, 2006

[31]. Brahti Chimdyalwar, Shrawan Kumar. Effective false positive filtering for evolving software. *ISEC'11 Proceedings of the 4th India software engineering conference.* Thiruvananthapuram, Kerala, India. February 24-27, 2011

[32]. J. R. Rithruff, J. Penix, J. D. Morgenthaler, S. Elbaum, G. Rothermel. Predicting accurate and actionable static analysis warnings: and experimental approach. *ICSE'08 Proceedings of the 30th international conference on software engineering.* Leipzig, Germany. May 10-18, 2008

[33]. H. Post, C. Sinz, A. Kaiser, T. Gorges. Reducing false positives by combining abstract interpretation and bounded model checking. *ASE'08 Proceedings of the 2008 23rd IEEE/ACM international conference on automated software engineering.* L'Aquila, Italy. September 15-19, 2008

[34]. Tukaram Muske, Advaita Datar, Mayur Khanzode, Kumar Madhukar. Efficient elimination of false positives using bounded model checking. *ISSRE'15 Proceedings of the 2015 IEEE 26th international symposium on software reliability engineering.* Gaithersburg, MD, USA. November 2-5, 2015

[35]. M. Junker, R. Huuck, A. Fehnker, A. Knapp. SMT-based false positive elimination in static program analysis". *ICFEM'12 Proceedings of the 14th international conference on formal engineering methods: formal mehods and software engineering.* Kyoto, Japan. November 12-16, 2012

[36]. G. Brat, W. Visser. Combining static analysis and model checking for software analysis tools. *ACE'01 Proceedings of the 16th IEEE international conference on automated software engineering.* San Diego, CA, USA. November 26-29, 2001

[37]. A. Fenker, R. Huuck. Model checking driven static analysis for the real world: designing and tuning large scale bug detection. *Journal Innovations in systems and software engineering. Volume 9, Issue 1, March 2013*

[38]. D. Hovemeyer, W. Pugh. Finding bugs easily. ACM *SIGPLAN notices. Volume 39, issue 12, December 2004*

[39]. D. Evans, D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software. Volume 19, Issue 1, 2002*

[40]. C. Csallner, Y. Smaragdakis. Check'N'Crash: combining static checking and testing. *ICSE'05 Proceedings of the 27th international conference on software engineering.* St. Luis, MO, USA. May 15-21, 2005

[41]. C. Flanagan, K Rustan, M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata. Extended static checking for Java. *PLDI'02 Proceedings of the ACM SIGPLAN 2002*

conference on programming language design and implementation. Berlin, Germany. June 17-19, 2002

[42]. C. Csallner, Y. Smaragdakis. JCrasher: an automatic robustness testing tester for Java. *Software – Practice & Experience. Volume 34, Issue 11*. September 2004.

[43]. C. Csallner, Y. Smaragdakis. DSD-Crasher: a hybrid analysis tool for bug finding. *ISSTA'06 Proceedings of the 2006 international symposium on software testing and analysis.* Portland, Maide, USA July 17-20, 2006

[44]. O. Chebaro, N Kosmatov, A. Giorgetti, J. Julliand. Programs slicing enhances a verification technique combining static and dynamic analysis. *SAC'12 Proceedings of the 27th annual ACM symposium of applied computing*. Trento, Italy. March 26-30, 2012

[45]. K. Li, C Reichenbach, C. Csallner, Y. Smaragdakis. Residual investigation: predictive and precise bug detection. *ISSTA'2012 Proceedings of the 2012 international symposium on software testing and analysis.* Minneanapolis, MN, USA. July 15-20, 2012

[46]. F. Elberzhager, J. Münch, V.T. Ngoc Nha. A systematic study on the combination of static and dynamic quality assurance techniques. *Infromation and siftware technology. Vol 54, Issue1. January, 2012*

[47]. A. Hanna, H. Z. Ling, X Yang, M. Debbabi. A synergy between static and dynamic analysis for the detection of software security vulnerabilities. *OTM'09 Proceedings of the confederated international congress, CoopIS, DOA, IS and ADBASE 2009 on on the move to meaningful internet systems: part II*. Vilamoura, Protugal. November 01-06, 2009

[48]. R. Hadjidj, X. Yang, S. Tlili, M. Debabi. Model-checking for software vulnerabilities detection with multi-language support. *PST'08 Proceedings of the 2008 sixth annual conference on privacy, security and trust.* Fredericton, NB, Canada. October 01-03, 2008.

[49]. D. Novillo. Tree SSA: a new optimization infrastructure for GCC. *Proceedings of the GCC developers summit.* Ottawa, ON, Canada. May 25-27, 2003

[50]. S. Schwoon. Model-checking pushdown systems. *PhD thesis.* Technischen Universität München. 2002

[51]. C. Artho, A. Biere. Combined static and dynamic analysis. Technical Report 466, ETH Zürich, Zürich, Switzerland, 2005.

[52]. O. Chebaro, N. Kostomarov, A. Giorgetti, J. Julliand. Combining static analysis and test generation for C program debugging. *TAP'10 Proceedings of the 4th international conference on tests and proofs.* Málaga, Spain. July 01-02, 2010

[53]. N. Williams, B. Marre, P. Mouy, M. Roger. PathCrawler: automatic generation of tests by combining static and dynamic analysis. *EDCC'05 Proceedings of the 5th European conference on dependable computing.* Budapest, Hungary. April 20-22, 2005

[54]. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, B. Yakobowski. Frama-C: a software analysis perspective. *SEFM'12 Proceedings of the 10th international conference on software engineering and formal methods.* Thesaloniki, Grece. October 01-05, 2012

[55]. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, B. Yakobowski. Frama-C: a software analysis perspective. *Formal aspects of computing. Volume 27, issue 3. May, 2015*

[56]. A. V. Nori, S. K. Rajamani, S. Tetali, A. V. Thakur. The Yogi ptoject: software property checking via static analysis and testing. *TACAS'09 Proceedings of the 15th international conference on tools and algorithms for the construction and analysis of systems: held as*

part of the ETAPS'09 joint European conferences on theory and practice of software. York, UK. March 22-29, 2009.

[57]. T. Ball, S. K. Rajamani. Slic: a specification language for interface checking (of C). *Technical report MSR-TR2001-21, Microsoft Research.* Redmond, WA, USA. January, 10. 2002

[58]. S. Rawat, D. Ceara, L. Mounier, M.-L. Potet. Combining static and dynamic analysis of vulnerability detection. *Cornell University Library arXiv:1305.3883.* May 16, 2013

[59]. T. Ball. The concept of dynamic analysis. *ESEC/FSE-7 Proceedings of the 7th European software engineering conference held jointly with 7th ACM SIGSOFT international symposium on foundations of software engineering.* Toulouse, France. September 06-10, 1999

[60]. J. Schütte, R. Fedler, D. Titze. ConDroid: targeted dynamic analysis of Android applications. *AINA'15 Proceedings of IEEE 29th international conference on advanced information networking and applications.* Gwangui, South Korea. March 24-27, 2015

[61]. X. Ge, K. Taneja, T. Xie, N. Tillmann. DyTa: dynamic symbolic execution guided with static verification results. *ICSE'11 Proceedings of the 33th international conference on software engineering.* Waikiki, Honolulu, HI, USA. May 21-28, 2011

[62]. N. Tillmann, J. de Hallex. Pex – white box test generation for .NET. *TAP'08 Proceedings of the 2nd international conference on tests and proofs.* Prato, Italy. April 09-11, 2008

[63]. M. Y. Wong, D. Lie. IntelliDroid: a targeted input generator for the dynamic analysis of android malware. *NDSS'16 The network and distributed system security symposium 2016.* San Diego, CA, USA. February 21-24, 2016

[64]. H. Gunadi. Formal certification of non-interferent Android bytecode (DEX bytecode). *ICECCS'15 Proceedings of the 2015 20th international conference on engineering and complex computer systems.* Gold Coast, Australia. December 9-12, 2015

[65]. L. De Moura, N. Bjørner. Z3: an efficient SMT solver. *TACAS'08/ETAPS'08 Proceedings of the theory and practice of software, 14th international conference on tools and algorithms for the constructions and analysis of systems.* Budapest, Hungary. March 29 – April 06, 2009

[66]. Chen N., Kim S. STAR: stack trace based automatic crash reproduction via symbolic execution. *IEEE transactions on software engineering. 2015, volume 41, issue 2.*

[67]. T. Avgerinos, S. Kil Cha, A. Rebert, E. J. Schwartz, M. Woo, D. Brumley. Automatic exploit generation. *Communications of the ACM, volume 57, issue 2, February 2014.*

[68]. M. Li, Y. Chen, L. Wang, G. Xu. Dynamically validating static memory leak warnings. *ISSTA'13 Proceedings of the 2013 international symposium on software testing and analysis.* Lugano, Switzerland. July 15-20, 2013.

[69]. HP Fortify. https://saas.hpe.com/en-us/software/sca, accessed 05.05.2017

[70]. CREST – automatic test generation tool for C. https://github.com/jburnim/crest, accessed 05.05.2017

[71]. D. Babić, L. Martignoni, S. McCamant, S. Song. Statically-directed dynamic automated test generation. *ISSTA'11 Proceedings of the 2011 international symposium on software testing and analysis.* Toronto, Ontario, Canada. July 17-21, 2011

[72]. N. Nethercote, J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *PLDI'07 Proceedings of the 28th ACM SIGPLAN Conference on programming languages design and implementation.* San Diego, CA, USA. June 10-13, 2007

[73]. F. Bellard. QEMU, a fast and portable dynamic translator. *ATEC'05 Proceedings of the annual conference on USENIX annual technical conference.* Anaheim, CA, USA. April 10-15, 2005

[74]. J. Seward, N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. *ATEC'05 Proceedings of the annual conference on USENIX annual technical conference.* Anaheim, CA, USA. April 10-15, 2005

[75]. V. Chipounov, V. Kuznetsov, G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. ASPLOS XVI Proceedings of the sixteenth international conference on architectural support for programming languages and operating systems. Newport Beach, CA, USA. March 05-11, 2011

Сравнительный анализ двух подходов к статическому анализу помеченных данных

[1] *М.В. Беляев <mbelyaev@ispras.ru>*
[1] *Н.В. Шимчик <shimnik@ispras.ru>*
[1] *В.Н. Игнатьев <valery.ignatyev@ispras.ru>*
[1,2] *А.А. Белеванцев <abel@ispras.ru>*
[1] *Институт системного программирования РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2] *Московский государственный университет имени М.В. Ломоносова, 119991, Россия, Москва, Ленинские горы, д. 1*

Аннотация. В настоящее время одним из наиболее эффективных средств поиска проблем безопасности ПО является анализ помеченных данных. Он может быть реализован на основе статического анализа и успешно обнаруживать ошибки, приводящие к уязвимостям, таким как внедрение кода или утечка непубличных данных. Возможны несколько существенно различных подходов к реализации алгоритма распространения пометок по внутреннему представлению программы: на основе анализа потоков данных или на базе символьного исполнения. В данной работе описаны особенности реализации обоих подходов в рамках существующей инфраструктуры статического анализатора для поиска ошибок в программах на C#, а также проведено сравнение этих подходов в различных аспектах: область применения, качество результатов, производительность и требовательность к ресурсам. Поскольку оба подхода используют единую инфраструктуру доступа к информации о программе и реализованы одной командой разработчиков, результаты сравнения близки к объективным и могут быть использованы при выборе оптимального варианта в контексте поставленной задачи.

Ключевые слова: taint analysis; static analysis; IFDS; symbolic execution

DOI: 10.15514/ISPRAS-2017-29(3)-7

Для цитирования: Беляев М.В., Шимчик Н.В., Игнатьев В.Н., Белеванцев А.А. Сравнительный анализ двух подходов к статическому анализу помеченных данных. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 99-116. DOI: 10.15514/ISPRAS-2017-29(3)-7

1. Введение

В настоящее время статический анализ все чаще используется для поиска серьёзных уязвимостей в ПО, таких как внедрение вредоносного кода или утечка непубличных данных. Помимо очевидного преимущества раннего обнаружения проблемы, статический анализ указывает точное место возникновения и проявления дефекта, а также предоставляет последовательность диагностических точек, содержащих путь распространения проблемы. С помощью тестирования сложно проверить все нетривиальные сценарии использования ПО, где часто содержатся уязвимости. Детекторы на основе статического анализа покрывают практически все возможные пути выполнения программы, что является существенным преимуществом, особенно в случае проблем безопасности.

Одним из наиболее эффективных методов статического анализа программ с целью поиска проблем безопасности является анализ помеченных данных. Полученные в результате работы анализатора предупреждения могут быть также использованы автоматизированными средствами проверки на основе динамического анализа или для генерации входных данных, вызывающих уязвимость.

В основе анализа помеченных данных лежит идея продвижения пометок по всем возможным путям распространения «интересных» данных в программе от некоторого истока (например, функции чтения пользовательских данных из формы) до стока (например, функции исполнения SQL запроса), тем самым вычисляя, как злоумышленник может внедрить свой код или какие секретные данные могут оказаться доступны (в примере – выполнить внедрение SQL кода). Анализ помеченных данных может быть реализован на основе как статического, так и динамического анализа. Помимо перечисленных ранее преимуществ поиска ошибок в статике, возможны ситуации, когда утечка пользовательских данных специально реализована автором программы и маскируется, если распознает запуск внутри динамического анализатора. Данная проблема особенно актуальна для широко распространённых в настоящее время мобильных приложений и должна быть обнаружена до начала распространения вредоносной программы.

В работе рассматриваются два подхода к статическому анализу помеченных данных. Первый подход основан на решении IFDS-задачи и называется алгоритмом табуляции [1]. Данный метод решает задачу анализа потоков данных, сводя её к задаче достижимости на расширенном межпроцедурном графе потока управления и имеет сложность $O(ED^3)$, а для локально разделимых задач — за $O(ED)$, где E — количество рёбер (нерасширенного) межпроцедурного ГПУ, а D — количество фактов анализа данных. Данный подход реализован в широко известном инструменте FlowDroid [4,5].

Второй подход основан на символьном выполнении и производит однократный обход графа развёртки функции, симулируя эффект выполнения каждой встреченной инструкции в графе. Обход функций выполняется в

топологическом порядке по графу вызовов, чтобы анализ всех вызываемых функций был завершён к началу рассмотрения вызывающей.

При решении задачи анализа потоков данных, помимо выбора схемы распространения пометок, существенное влияние оказывают методы моделирования окружения выполнения программы (моделирование библиотечных функций), построение модели графа вызовов программы для симуляции активности пользователя, анализ виртуальных вызовов, анализ исходных текстов на языках описания интерфейсов и другой семантики программы (например, XAML, для сбора доступных пользователю интерфейсов). Оба рассмотренных в данной работе подхода используют единую инфраструктуру анализа, в том числе представления окружения, поэтому в рамках данной статьи перечисленные проблемы не рассматриваются.

Работа состоит из следующих частей. Во второй части рассмотрена общая инфраструктура анализатора SharpChecker, необходимая для реализации обоих методов. В частях 3 и 4 описаны модель работы и особенности реализации анализа помеченных данных с помощью IFDS и символьного выполнения соответственно. В части 5 собраны результаты работы обоих алгоритмов как на специальном проекте WebGoat.NET, моделирующем распространённые уязвимости, так и на наборе проектов с открытым кодом на языке C#. Шестая часть содержит обобщение полученных результатов и рассматривает основные направления дальнейших исследований для их улучшения.

2. Инфраструктура анализатора SharpChecker

Сравнение алгоритмов анализа помеченных данных будет проводиться на основе SharpChecker [2,3] — статического анализатора для поиска ошибок в программах на языке C#. Он способен находить более 100 различных типов ошибок, производя поиск на основе синтаксического анализа (сигнатурный), анализа потоков данных и чувствительного к контексту вызова и путям выполнения символьного выполнения. Общая схема работы анализатора представлена на рисунке 1.

На первых этапах SharpChecker строит статический граф вызовов анализируемой программы, который в дальнейшем уточняется за счёт анализа виртуальных методов, вызовов через интерфейсы и делегаты. Подсистема анализа неявных вызовов предоставляет возможность рассматривать все методы, возможные для вызова в данной инструкции, позволяя детекторам или другим подсистемам анализа выбирать способ использования данной информации. Например, детектор разыменования null последовательно применяет все резюме кандидатов, а другие детекторы могут эвристически выбрать только один, основываясь, например, на ресурсоёмкости требуемых операций. Поскольку большинство атак производятся с помощью пользовательских интерфейсов, которые, в свою очередь, на C# чаще всего

реализованы на основе интерфейсов, качество алгоритмов девиртуализации оказывает очень существенное влияние на результаты анализа помеченных данных.

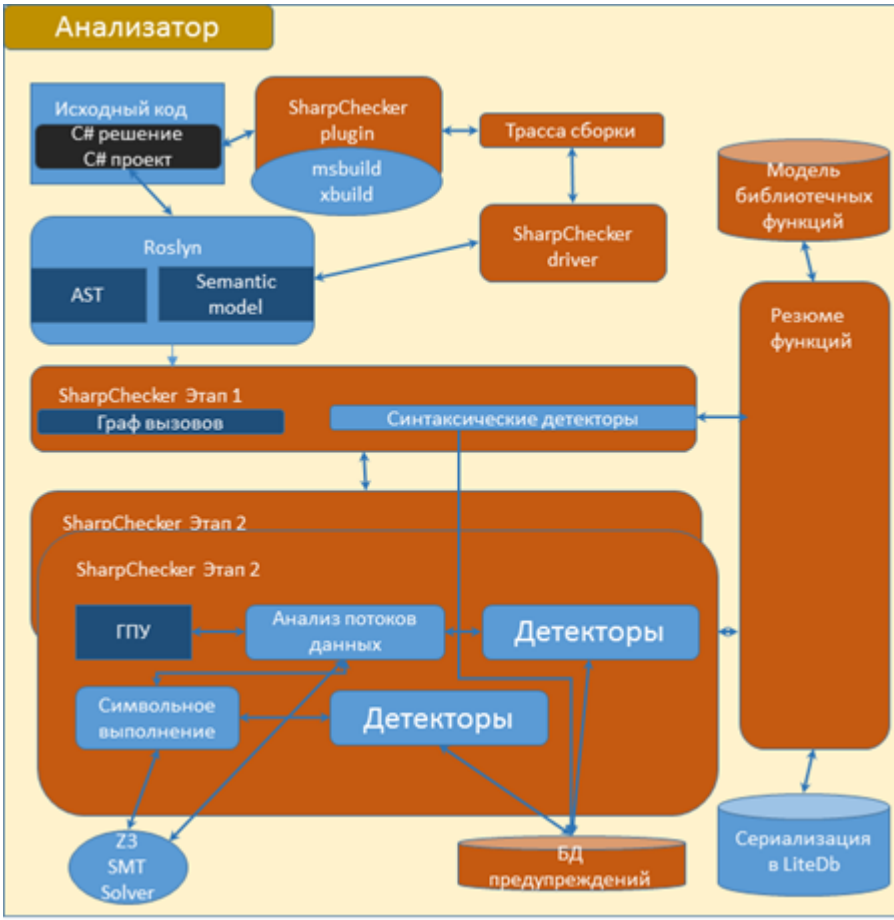


Рис. 1. Схема работы анализатора SharpChecker
Fig. 1. Schematic diagram of the SharpChecker analyzer

Символьное выполнение основано на обходе графа вызовов в порядке от вызываемых функций к вызывающим. Это позволяет свести межпроцедурный анализ к внутрипроцедурному за счёт построения резюме каждого метода. Резюме содержит всю необходимую информацию об эффектах вызова данной функции с учётом контекста вызова. Это достигается благодаря

использованию предусловий, вычисляемых с помощью SMT решателя в случае необходимости.

Во время внутрипроцедурного анализа для каждого метода строится граф потока управления, вершинами которого являются базовые блоки - последовательности подряд идущих инструкций. Инструкциями являются вершины абстрактного синтаксического дерева, семантически соответствующие отдельным операторам. Рёбра в графе потока управления являются ориентированными и показывают, в какие базовые блоки может быть совершён переход после завершения выполнения данного. Особыми (пустыми) базовыми блоками являются точка входа в метод и точка выхода из метода, в которых сходятся рёбра от всех базовых блоков, которые могут завершить выполнение тела метода (например, содержащие инструкцию return или возможное исключение). Следует отметить, что граф потока управления может содержать циклы, а значит, в нём может быть бесконечное число различных путей от точки входа в метод до точки выхода. Чтобы ограничить количество рассматриваемых путей, вместо графа потока управления строится его ациклическая развёртка (граф развёртки), в которой переходы по обратным рёбрам заменяются на переходы в отдельные компоненты, копирующие часть графа потока управления. Поскольку развёртка графа строится для фиксированной глубины и в ней нет обратных рёбер, то максимальная длина и суммарное количество различных путей от точки входа до точки выхода ограничены.

Символьное выполнение предполагает обход графа развёртки, при котором считается, что формальные параметры метода, а также все поля данного класса и статические поля других классов имеют произвольные значения. По этой причине в точке входа в метод формальные параметры и начальное состояние памяти параметризуются набором символьных переменных, тип которых совпадает с типом исходных полей и переменных, а конкретное значение никак не задаётся. Инструкциям в программе задаётся новая семантика, которая позволяет работать не с конкретными значениями, а с символьными выражениями: формулами над символьными переменными и константами. Так, если выполняется операция сложения двух целочисленных символьных переменных V1 и V2, то результатом символьного выполнения будет считаться формула V1 + V2. Аналогичные действия выполняются для унарных операторов и условного оператора. Значения, которые не являются константными и не могут быть выражены через существующие символьные переменные, моделируются с помощью введения новых символьных переменных. Вызовы методов моделируются с использованием информации, сохранённой в резюме вызываемого метода. Для сокращения количества интерпретируемых путей также применяется техника объединения состояний. В точке слияния путей выполнения ГПУ создаётся новый контекст анализа, строящийся на основе слияния (по заданным для каждого типа правилам) соответствующих пар фактов, пришедших с входящих рёбер.

Для моделирования окружения в анализаторе реализовано несколько подсистем. Первая основана на хранении множества «интересных» свойств библиотечных функций, например, что в результате её работы создаётся объект, требующий освобождения, или что первый параметр не может быть равен null. Для анализа помеченных данных данная подсистема может быть расширена с целью поддержки библиотечных передаточных функций, а также множеств истоков и стоков.

Другой метод моделирования окружения позволяет задавать поведение библиотечных функций на языке C#. Это особенно актуально для реализации коллекций, имеющих внутреннее состояние, которое необходимо учитывать для качественного анализа. Этот механизм также был доработан для поддержки продвижения пометок через вызовы библиотечных функций.

Таким образом, существующая инфраструктура анализатора SharpChecker позволяет единообразно решать проблему моделирования окружения, построения внутреннего представления и сбора данных, необходимых для реализации обоих подходов анализа помеченных данных.

3. Алгоритм анализа помеченных данных на основе IFDS

В основе алгоритма лежит распространение информации о помеченных данных по базовым блокам и рёбрам межпроцедурного графа потока управления. При этом пометки могут продвигаться от источников к стокам и наоборот. Будем говорить, что *прямой* анализ начинается от источников, *обратный* — от стоков. Правила распространения помеченных данных задаются *передаточными функциями*. Анализ для каждого источника (стока – в обратном) проводится независимо.

3.1 Задача анализа помеченных данных как IFDS-задача

Введём необходимые определения.

Путь доступа — это список, состоящий из начального объекта и полей. Начальным объектом пути доступа может являться литерал, локальная переменная, параметр, возвращаемое значение метода, ссылка `this`, статическое поле класса, выражение (например, вызов метода или оператора) или временный начальный объект, используемый в реализации передаточных функций и не представляющий никакого реального объекта, содержащегося в программе. Полями могут являться нестатические поля и общий элемент массива. Примеры путей доступа: `x`, `x.y.z`, `x.[...].w`, `this.f`, `a.f()`, `"password"`, `return`, `(a + b).p`. *Длиной* непустого пути доступа называется количество полей в нём + 1. Пути доступа хранятся в виде префиксного дерева, корнем которого является специальный пустой путь доступа `{root}`, имеющий длину 0.

Факт анализа данных — это информация о том, что путь доступа помечен. Факт содержит помеченный путь доступа и предшествующий факт, используемый для восстановления пути распространения помеченных данных.

Точка анализа данных — это информация о том, что в данной точке программы путь доступа, содержащийся в факте, помечен. Точка содержит факт и базовый блок, в котором этот факт рассматривается.

Алгоритм анализа работает с точками анализа данных, а передаточные функции — непосредственно с фактами анализа данных и путями доступа. Различные дефекты безопасности имеют различное соотношение количества источников и стоков, поэтому для возможности реализации детекторов дефектов был реализован как прямой (от источника к стоку), так и обратный (от стока к источнику) анализ.

Каждый факт распространяется независимо от других. Это позволяет создавать резюме, содержащие результаты анализа метода, и повторно использовать их, если входная точка ещё раз встретится в программе. Резюме для входной точки содержит выходные точки, полученные в результате анализа, включая пути распространения данных. Использование резюме значительно ускоряет анализ, так как это избавляет от необходимости повторно рассматривать не только метод, для которого построено резюме, но и все вызываемые им (в т. ч. транзитивно) методы.

Основная функция (`Solve`, листинг 1) осуществляет обработку большинства инструкций базового блока и вызывается для каждой начальной точки анализа (точки источника для прямого анализа и точки стока для обратного), а также для точек, входящих в вызываемый метод. Эта функция управляет очередью обработки точек, а также применением резюме. Если для входной точки существует резюме, применяются точки из резюме и результат возвращается. Применение точек заключается в замене предшествующего факта у факта входной точки резюме на факт входной точки функции `Solve`, чтобы восстановить путь распространения данных. Если же резюме не существует, то входная точка добавляется в очередь, и запускается основной цикл обработки. В этом цикле из очереди последовательно извлекаются точки, для них вызывается функция `SolveOnePoint`, выполняющая распространение помеченных данных, и результат помещается в очередь. Цикл выполняется, пока в очереди есть точки. В результат записываются точки, достигшие конца метода.

```
1: function Solve(point) → {Point}
2:   if ∃ a summary for point
3:     return Apply(summary points for point)
4:   worklist ← {point}, outPoints ← ∅
5:   while worklist ≠ ∅
6:     pt ← worklist.Get()
```

```
7:   if pt ∈ visited
8:     continue
9:   visited ← visited ∪ {pt}
10:  worklist.PutAll(SolveOnePoint(pt))
11:  if BasicBlock(pt) is exit block
12:     outPoints ← outPoints ∪ {pt}
13:  store outPoints as a summary for point
14:  return outPoints
```

Листинг 1. Псевдокод функции Solve для прямого анализа
Listing 1. Pseudo-code of the Solve function for direct analysis

Функция `SolveOnePoint` распространяет помеченные данные, используя передаточные функции `NormalTF`, `CallTF`, `RetTF`, `Call2RetTF`. В строках 10–12 оригинальный алгоритм расширяется для поддержки несбалансированных возвратов, что позволяет проводить анализ, начиная из источников, а не из метода `main`, которого может не быть, и просматривать в ходе анализа только те методы, в которые попадают помеченные данные. Возврат из метода называется сбалансированным, если метод был достигнут во время анализа по цепочке вызовов. Для сбалансированного вызова известно, в какую точку программы произойдёт возврат. Когда анализ возвращается в метод, с которого он начался, возврат из этого метода является несбалансированным, поэтому требуется перебирать все точки вызова этого метода, чтобы продолжить анализ. Примеры сбалансированных и несбалансированных возвратов приведены на рисунке 2.

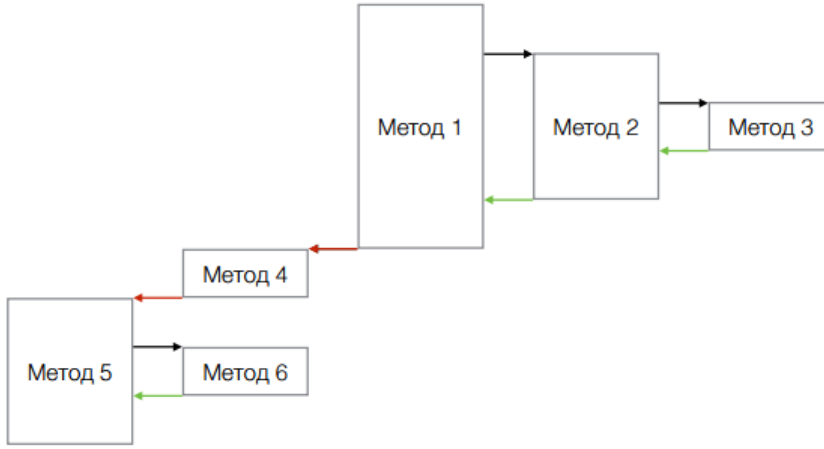


Рис. 2. Сбалансированные и несбалансированные возвраты
Fig. 2. Balanced and unbalanced returns

Здесь Метод 1 — метод, с которого начинается анализ, чёрные стрелки обозначают вызовы, зелёные — сбалансированные возвраты, красные — несбалансированные возвраты.

```
1: function SolveOnePoint(point) → {Point}
2:   points ← NormalTF(point)
3:   if last statement of BasicBlock(point) is call
4:     points_old ← points, points ← ∅
5:     foreach p in points_old
6:       points_call ← CallTF(p, call)
7:       points_callee ← ⋃_{p'∈points_call} Solve(p',call)
8:       points_ret ← ⋃_{p''∈points_callee} ReTF(p'',call)
9:       points ← points ∪ points_ret ∪
                          Call2RetTF(p, call)
10:  if BasicBlock(point) is exit block ∧ point
wasn't created by processing a call
11:    foreach call site call of current function
12:      points ← points ∪ ⋃_{p'∈RetTF(point,call)} Solve(p'))
13:  return points
```

Листинг 2 Псевдокод функции SolveOnePoint для прямого анализа
Listing 1. Pseudo-code of the SolveOnePoint function for direct analysis

В строках 7–9 применяются резюме для внутренних точек, строки 2–3 предотвращают постоянное пересоздание резюме для одних и тех же точек в строке 16.

Поскольку при обратном анализе частым случаем является передача «управления» в середину функции при возврате из вызова, то для частей таких методов также строится резюме. Оно может быть повторно использовано в дальнейшем для ускорения анализа. Это и является наиболее существенным отличием реализации метода `Solve` обратного анализа.

Практические реализации функций Solve, SolveOnePoint и передаточных функций содержат дополнения для вычисления не только результирующих точек, но и диагностических точек — точек анализа данных, в которых детектированы дефекты.

3.2 Передаточные функции для анализа помеченных данных

Анализ использует четыре вида передаточных функций — внутрипроцедурная передаточная функция, передаточная функция вызова, передаточная функция возврата и передаточная функция от вызова до возврата. Каждая из них принимает на вход одну точку анализа данных и необходимую дополнительную информацию и возвращает множество точек анализа данных, а внутри себя работает с фактами, а не точками.

Для определения передаточных функций введём следующие обозначения:

T — множество помеченных путей доступа, x, y, z — некоторые пути доступа, $\diamondsuit$ — унарный оператор, $\circ$ — бинарный оператор, $x.[f]$ — произвольный путь доступа, полученный из пути доступа x добавлением списка полей $[f] = .f_1.f_2 \ldots f_p$, в т. ч. пустого, $"t_0{a_1}...{a_n}t_n" — выражение интерполяции строк в языке C#, где t_i — фрагменты текста ($0 \le i \le n$), а a_i — аргументы ($1 \le i \le n$).

Передаточные функции очевидно различаются для прямого и обратного анализа, однако для понимания достаточно рассмотреть только один случай прямого анализа.

Обычная передаточная функция `NormalTF`:

Функция осуществляет внутрипроцедурное распространение помеченных данных внутри базового блока, последовательно обрабатывая его инструкции в прямом порядке. Выходные точки строятся по множеству T, используя ГПУ для определения следующих базовых блоков.

$$\texttt{var x = y} : T \to T \cup \{x.[f] : y.[f] \in T\};$$
$$\texttt{x = y} : T \to T \cup \{x.[f] : y.[f] \in T\} \setminus \{x.[f] : y.[f] \notin T\};$$
$$\diamondsuit\texttt{x} : T \to T \cup \{(\diamondsuit x).[f] : x.[f] \in T\};$$
$$\texttt{x} \circ \texttt{y} : T \to T \cup \{(x \circ y).[f] : x.[f] \in T \vee y.[f] \in T\};$$
$$\texttt{x} \circ\texttt{= y} : T \to T \cup \{x.[f] : y.[f] \in T\};$$
$$\texttt{\$"t0\{a1\}...\{an\}tn"} : T \to T \cup \{(\$"t0\{a1\}...\{an\}tn").[f] : \exists i;\ 1 \le i \le n \wedge arg_i.[f] \in T\};$$
$$\texttt{return x} : T \to T \cup \{return.[f] : x.[f] \in T\}.$$

Передаточная функция вызова `CallTF`:

Функция осуществляет распространение помеченных данных из текущего метода в вызываемый метод, сопоставляя фактические и формальные параметры, включая неявный параметр $this$, а также оставляя без изменений статические поля. Всё остальное вызываемому методу недоступно.

Объявление метода: `ReturnType f(param1, param2, ..., paramN)`.

Вызов метода: `a.f(arg1, arg2, ..., argN)`.

В результирующих точках указывается базовый блок $Entry$ вызываемого метода.

$$T \to \{param1.[f] : arg1.[f] \in T\} \cup \{this.[f] : a.[f] \in T\} \cup$$
$$\cup \{x.[f] : IsStatic(x) \wedge x.[f] \in T\}.$$

Передаточная функция возврата `RetTF`:

Функция распространяет помеченные данные из текущего метода в вызвавший его метод, сопоставляя формальные и фактические параметры вызова, включая неявный параметр $this$, а также оставляя без изменений статические поля. Всё остальное вызывающему методу недоступно. Заметим,

что пути доступа длины 1, соответствующие формальным параметрам, не распространяются, поскольку их изменения в языке C# локальны для текущего метода, если не указано ключевое слово **`ref`** или **`out`**. Возврат происходит в базовый блок, следующий за базовым блоком вызова.

$$T \to \{argI.x.[f]:paramI.x.[f] \in T\} \cup$$
$$\cup \{argI:IsOutParameter(paramI) \wedge paramI \in T\} \cup$$
$$\cup \{a.x.[f]:this.x.[f] \in T\} \cup \{x.[f]:IsStatic(x) \wedge x.[f] \in T\} \cup$$
$$\cup \{a.f(arg1, arg2, \ldots, argN).[f]:return.[f] \in T\}.$$

Передаточная функция от вызова до возврата `Call2RetTF`:

Эта передаточная функция внутрипроцедурно распространяет пометки из базового блока вызова в блок возврата для тех путей доступа, которые недоступны вызываемому методу. Такими путями доступа являются нестатические поля, переменные и параметры, которые не были переданы в вызываемую функцию как аргументы. В результирующих точках указывается базовый блок возврата, как и в `RetTF`.

$$T \to \{x.[f] : \neg IsStatic(x) \wedge a \neq x \wedge \ !\exists\, I : argI = x \wedge x.[f] \in T\} \cup$$
$$\cup \{x : \neg IsStatic(x) \wedge (a = x \vee \exists I : argI = x) \wedge x \in T\}.$$

3.3 Результаты тестирования

Тестирование проблем безопасности в активно используемом ПО затруднено тем, что как правило большинство реальных уязвимостей уже исправлено, поэтому количество сообщений анализатора очень мало. В связи с этим, сравнение подходов удобно проводить на специальном проекте для обучения информационной безопасности WebGoat.NET, который намеренно содержит ошибки. Однако приводятся также результаты запуска на других открытых проектах, содержащих искомые ошибки.

Испытания проводились на двух одинаковых компьютерах с процессором Intel Core i7-6700, 32 ГБ оперативной памяти, ОС Windows 10 x64. Проведённые испытания показали следующие результаты:

Табл. 1. Результаты работы метода анализа IFDS
Table. 1. Results of the work of the IFDS analysis method

Тип дефекта	Проект	SNAP	WebGoat		OmniDB + Spartacus		netMQ	ShareX	shadowsoc	OpenRA	banshee	cassandra	Всего
LDAP INJECTION	TP	8											8
	FP												
SQL INJECTION	TP		15	(29)	35	(630)							50
	Специально				2	(36)							2

	FP												
COMMAND INJECTION	**TP**												
	FP				1								1
CONSTANT CREDENTIALS	**TP**				75		1	1					77
	Тесты	1							6		1	10	18
	Инициализация				4								4
	Пустой пароль							3	1	12			16
	FP				1								1
INFORMATION_ EXPOSURE	**TP**												
	Tests												
	FP						64						
RESOURCE INJECTION	**TP**		1										1
	FP				2								2

Результаты работы анализатора были проанализированы вручную с целью оценки соотношения истинных срабатываний. Было установлено, что часть ошибок обнаружено в тестах и при инициализации переменных, а также некоторые предупреждения оказались ложными, поскольку указали на предусмотренные сценарии использования программы.

4. Анализ помеченных данных на основе символьного выполнения

В отличие от подхода IFDS, в данном методе помечаются символьные значения. Множества истоков, стоков и передаточных функций могут состоять из:

- вызовов методов, с отмеченными входными и/или выходными параметрами;
- обращений к полям класса на запись или чтение;
- формальных параметров метода.

Базовый алгоритм можно описать следующим образом. Пусть состояние программы в ходе символьного выполнения задаётся текущей инструкцией I, предикатом текущего пути P, множеством символьных значений V и отображением множества переменных на множество символьных выражений.

Дополним это состояние множеством помеченных значений $T \in V$.

В ходе символьного выполнения для каждого состояния выполняются следующие проверки:

- если I входит в множество истоков, то символьные значения, соответствующие её выходным параметрам, добавляются в множество T (то есть становятся помеченными);
- если I входит в множество передаточных методов и хотя бы один её входной параметр является помеченным, то все выходные параметры

также становятся помеченными;

- если I входит в множество стоков и хотя бы один входной параметр является помеченным, то алгоритм обнаружил дефект в программе.

Если используется символьное выполнение с объединением состояний, то при объединении множество помеченных символьных значений получается объединением двух исходных множеств: $T = T_1 \cup T_2$.

Приведённый алгоритм может констатировать факт достижения помеченными данными стока, однако для его практического использования, например, проверки истинности выданного предупреждения, необходимо построение множества диагностических точек, демонстрирующих пользователю анализатора путь в программе от истока до стока. Для этого каждому элементу из множества T требуется сопоставить структуру данных TInfo, сохраняющую и накапливающую информацию об истоке помеченности данного значения и пройденных передаточных методах. Путь распространения пометки будем называть трассой. Следует отметить, что эта структура может хранить больше одной трассы ввиду наличия передаточных методов с несколькими входными параметрами (например, операция соединения строк сохраняет помеченность обоих своих аргументов), а также в случае объединения символьных состояний (когда по разным путям одна переменная была помечена из разных источников)

Кроме того, рассмотренный алгоритм является внутрипроцедурным. Чтобы была возможность применять его для обнаружения межпроцедурных путей, можно использовать резюме методов - структуру данных, сохраняющую информацию о свойствах метода, выявленных в результате внутрипроцедурного анализа. Следует отметить, что в этом случае становится важным порядок анализа методов: он должен проводиться в обратном топологическом порядке (начиная с листовых вершин) по графу вызовов, чтобы вызывающие методы имели информацию о свойствах вызываемых методов.

В символьное состояние программы следует добавить множество PT потенциально помеченных значений, полностью аналогичное ранее введённому множеству T помеченных значений, с тем исключением, что:

- изначально в множество PT добавляются символьные значения, соответствующие формальным параметрам исследуемого метода, а в соответствующем TInfo сохраняется информация о данной точке входа в метод;
- инструкции из множества истоков не добавляют элементы в множество PT;
- при достижении потенциально помеченным значением инструкции из множества стоков в резюме метода добавляется информация о том, что метод становится межпроцедурным стоком с указанием точки входа в метод и участком трассы;

- аналогичным образом при выходе из тела метода в резюме добавляется информация о том, что метод становится межпроцедурным истоком (для значений из множества T) и/или передаточным методом (для значений из множества PT), с указанием выходных символьных значений и участком трассы.

Межпроцедурные истоки, стоки и передаточные методы обрабатываются таким же образом, как и обычные, за тем исключением, что связанный с ними участок трассы прибавляется к внутрипроцедурной трассе, формируя межпроцедурную трассу распространения помеченности данных.

Для поиска различных типов ошибок необходимо использование различных категорий пометок. В связи с этим некоторые из них приходится анализировать раздельно, тем самым повторно выполняя анализ одних и тех же путей передачи помеченных данных (так как множества передаточных методов, как правило, не отличаются). Одним из способов решения данной проблемы является введение понятия тега. Тег - это наследуемый от истока идентификатор, который приписывается точке входа в структуре TInfo. Теги прозрачны для передаточных методов и проверяются по достижении стока: если теги помеченных данных и стока не совпадают, то никакие действия не выполняются.

Для борьбы с ложными срабатываниями, возникающими из-за наличия в программе недостижимого кода, в структуре данных, содержащий информацию о помеченности, предусмотрено хранение предикатов пути и используются следующие правила объединения состояний A с предикатом пути P_A и B с предикатом P_B:

- если значение переменной v является помеченным в состоянии A, но не помечено в состоянии B значение переменной v будет считаться помеченным с предикатом P_A;
- если значение переменной v является помеченным и в состоянии A и в состоянии B, при этом у них общий исток значение переменной v считается помеченным с предикатом $(P_A \wedge P_B)$;
- если значение переменной v является помеченным и в состоянии A и в состоянии B, но у них различные истоки в TInfo сохраняются обе трассы, каждая со своим предикатом.

При достижении стока помеченным значением с предикатом P_t в состоянии с предикатом пути P строится формула, соответствующая предикату $P_t \wedge P$, которая передаётся SMT-решателю. Если решатель приходит к выводу, что формула не является выполнимой, сообщение о дефекте не выдаётся.

5. Результаты тестирования

Тестирование обоих методов проводилось на одинаковых машинах, использовался один и тот же набор проектов с открытым исходным кодом. У рассматриваемых методов анализа помеченных данных есть только один

общий детектор – поиск внедрения SQL кода, и оба детектора выдают одинаковое множество предупреждений на всём наборе тестов. Таким образом, вне зависимости от способа реализации, возможно достигнуть достаточной полноты анализа.

Тестирование производительности рассмотренных методов показывает следующие результаты:

Табл. 2. Сравнение производительности методов анализа
Table. 2. Comparison of the performance of analysis methods

	WebGoat.NET	OmniDB	CodeContracts
IFDS	10 сек.	117 сек.	377 сек.
Символьное выполнение	7 сек.	182 сек.	573 сек.
Символьное выполнение без построения резюме передаточных функций	7 сек.	129 сек.	512 сек.

Данные результаты соответствуют запуску анализатора с только одним включённым детектором SQL_INJECTION. Результирующий набор выданных предупреждений одинаков для всех запусков. Таким образом, можно отметить, что указанные методы имеют сравнимую производительность, однако основанный на IFDS, при условии работы только одного детектора, оказывается быстрее. В случае символьного исполнения необходимо вычислять существенно большее количество информации, необходимое для работы остальных детекторов, которое невозможно отключить. Кроме того, этот метод вычисляет информацию о возможных распространениях меток для всех функций, даже если такая информация не потребуется при дальнейшем анализе. Тестирование показало, что для указанных проектов эта информация несущественна, и отключение её вычисления существенно ускоряет работу метода, основанного на символьном выполнении, что показано в третьей строке.

Основные отличия в производительности подходов возникают при масштабировании. Тестирование показало, что существенное увеличение количества истоков оказывает неравномерное влияние. Это происходит при добавлении новых детекторов. Так, например, если часто используемую функцию ToString() объявить истоком, то время работы на OmniDB у IFDS подхода составит 278 секунд вместо 117 (замедление в 2.4 раза), а у второго - 223 сек. вместо 182 (замедление всего в 1.22 раза). Таким образом, при разработке новых детекторов необходимо принимать во внимание количество источников помеченных данных.

Заключение

В работе рассмотрены два подхода к реализации анализа помеченных данных с целью поиска ошибок в программах на языке C#: на основе IFDS и символьного выполнения. Оба метода были реализованы в рамках единой

инфраструктуры статического анализатора SharpChecker и протестированы на одинаковом наборе проектов с использованием одинакового аппаратного обеспечения. Исследования результатов и производительности показали, что возможно достижение сравнимой полноты анализа вне зависимости от использованного подхода. Метод IFDS имеет значимые преимущества в производительности при условии небольшого количества истоков, однако плохо масштабируется при их увеличении, что допустимо для реализации большинства детекторов.

Практика разработки детекторов для ошибок безопасности на основе анализа помеченных данных также показывает важность полноты множеств истоков, стоков и передаточных функций, а также точности моделирования окружения (библиотечных функций). Кроме того, наличие двух методов позволяет существенно повысить качество и производительность детекторов за счёт сравнения результатов их работы.

Дальнейшие исследования будут посвящены исследованию методов моделирования окружения, в том числе построения модели работы программы, которое заключается в дополнении графа вызовов рёбрами, позволяющими симулировать поведение пользователя. Для подхода IFDS актуальной является задача анализа псевдонимов, которая может повысить полноту результатов.

Список литературы

[1] Reps T., Horwitz S., Sagiv M. Precise Interprocedural Dataflow Analysis via Graph Reachability. Proceedings of the 22Nd ACM SIGPLANSIGACT Symposium on Principles of Programming Languages (POPL ’95), San Francisco, California, USA, ACM, 1995, pp. 49–61

[2] Кошелев В.К., Игнатьев В.Н., Борзилов А.И. Инфраструктура статического анализа программ на языке C#. Труды ИСП РАН, том 28, вып. 1, 2016 г., стр. 21–40. DOI: 10.15514/ISPRAS-2016-28(1)-2

[3] Кошелев В.К. Дудина И.А. Игнатьев В.Н. Борзилов А.И. Чувствительный к путям поиск дефектов в программах на языке C# на примере разыменования нулевого указателя. Труды ИСП РАН, том 27, вып. 5, 2015 г., стр. 59–86. DOI: 10.15514/ISPRAS-2015-27(5)-5

[4] FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycleaware Taint Analysis for Android Apps. Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’14), Edinburgh, United Kingdom, ACM, 2014, pp. 259–269

[5] Christian Fritz et al. Highly Precise Taint Analysis for Android Applications. Tech. rep. EC SPRIDE, May 2013, 14 p. http://www.bodden.de/pubs/TUD-CS-2013-0113.pdf, дата обращения 20.06.2017

Comparative analysis of two approaches to the static taint analysis

[1]*M.V. Belyaev <mbelyaev@ispras.ru>*
[1]*N.V. Shimchik <shimnik@ispras.ru>*
[1]*V.N.Ignatyev <valery.ignatyev@ispras.ru>*
[1,2] *A.A.Belevantsev <abel@ispras.ru>*
[1] *Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia*
[2] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia*

Abstract. Currently, one of the most efficient ways to find software security problems is taint analysis. It can be based on static analysis and successfully detect errors that lead to vulnerabilities, such as code injection or leaks of private information. Several different ways exist for the implementation of the algorithm for the taint data propagation through the program intermediate representation: based on the dataflow analysis (IFDS) or symbolic execution. In this paper, we describe how to implement both approaches within the existing static analyzer infrastructure to find errors in C# programs, and compare these approaches in different aspects: the scope of application, practical completeness, results quality, performance and scalability. Since both approaches use a common infrastructure for accessing information about the program and are implemented by a single development team, the results of the comparison are significant and can be used to select the best option in the context of the task. Our experiments show that it's possible to achieve the same completeness regardless of chosen approach. IFDS-based implementation has higher performance comparing with symbolic execution for detectors with small amount of taint data sources. In the case of multiple detectors and a large amount of sources the scalability of IFDS approach is worse than the scalability of symbolic execution.

Keywords: taint analysis; static analysis; IFDS; symbolic execution

DOI: 10.15514/ISPRAS-2017-29(4)-7

For citation: Belyaev M.V., Shimchik N.V., Ignatyev V.N., Belevantsev A.A. Comparative analysis of two approaches to the static taint analysis. *Trudy ISP RAN/Proc. ISP* RAS, vol. 29, issue 3, 2017, pp. 99-116 (in Russian). 10.15514/ISPRAS-2017-29(3)-7

References

[1]. Reps T., Horwitz S., Sagiv M. Precise Interprocedural Dataflow Analysis via Graph Reachability. Proceedings of the 22Nd ACM SIGPLANSIGACT Symposium on Principles of Programming Languages (POPL '95), San Francisco, California, USA, ACM, 1995, pp. 49–61

[2]. Koshelev V.K., Ignatyev V.N., Borzilov A.I. C# static analysis framework. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 1, 2016, pp. 21-40 (in Russian). DOI:10.15514/ISPRAS-2016-28(1)-2

[3]. V. Koshelev, I. Dudina, V. Ignatyev, A. Borzilov. Path-sensitive bug detection analysis of C# program illustrated by null pointer dereference. Trudy ISP RAN / Proc ISP RAS, vol. 27, issue 5, 2015, pp.59–86 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)-5

[4]. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycleaware Taint Analysis for Android Apps. Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14), Edinburgh, United Kingdom, ACM, 2014, pp. 259–269

[5]. Christian Fritz et al. Highly Precise Taint Analysis for Android Applications. Tech. rep. EC SPRIDE, May 2013, 14 p. http://www.bodden.de/pubs/TUD-CS-2013-0113.pdf, accessed 20.06.2017

Обзор задач и методов их решения в области классификации сетевого трафика★

А. И. Гетьман <thorin@ispras.ru>
Ю. В. Маркин <ustas@ispras.ru>
Д. О. Обыденков <obydenkov@ispras.ru>
Е. Ф. Евстропов < john0606@yandex.ru>
ИСП РАН, 109004, Россия, г. Москва, ул. А. Солженицына, дом 25

Аннотация. В статье рассматривается задача классификации сетевого трафика: характеристики, используемые для её решения, существующие подходы и области их применимости. Перечисляются прикладные задачи, требующие привлечения компонента классификации и дополнительные требования, проистекающие из особенности основной задачи. Анализируются свойства сетевого трафика, обусловленные особенностями среды передачи,а также применяемых технологий, так или иначе влияющие на процесс классификации. Рассматриваются актуальные направления в современных подходах к анализу и причины их развития.

Ключевые слова. Анализ сетевого трафика, сетевая безопасность, классификация сетевого трафика, машинное обучение, DPI

DOI: 10.15514/ISPRAS-2017-29(3)-8

Для цитирования: Гетьман А.И., Маркин Ю.В., Евстропов Е.Ф., Обыденков Д.О. Обзор задач и методов их решения в области классификации сетевого трафика. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 117-150. DOI: 10.15514/ISPRAS-2017-29(3)-8

1. Введение

В общем виде задача классификации сетевого трафика может быть сформулирована следующим образом: получение на вход некоторых характеристик сетевого трафика с выдачей на выходе класса, к которому данный вид трафика относится. В качестве входных характеристик могут выступать как данные пакетов, так и различные частотные характеристики, а в качестве выходных, как идентификатор конкретного приложения, ответственного за генерацию этого трафика, так и идентификатор вида

★ Работа поддержана грантом РФФИ 14-07-00606 А

трафика, например VoIP-трафик. Данная задача является одной из центральных тем в области организации сетевого взаимодействия. Исторически эта задача была наиболее актуальна в области управления трафиком для повышения эффективности использования существующих каналов связи и качества предоставляемых услуг для конечных пользователей. Однако на данный момент актуальность данной задачи значительно возросла, в связи с расширением её области применения, в которую на данный момент входят как системы применения политик, так и сфера информационной безопасности. Практически любая система анализа трафика в том или ином виде включает в себя компонент классификации.

Задача классификации трафика исследуется достаточно давно: её анализу и поиску эффективных решений в различных условиях и ограничениях посвящено значительное количество исследовательских работ, в том числе и за последние годы. Это связано, в том числе из тем, что сетевой ландшафт быстро меняется и методы, и алгоритмы, ещё недавно показывавшие хороший результат, в новых условиях значительно теряют свою эффективность или становятся вовсе неприменимыми. Среди условий, которые значительно влияют на применимость различных методов, можно выделить быстрый рост количества передаваемого трафика и пропускных способностей каналов связи – это приводит к необходимости поиска алгоритмов с пониженной вычислительной сложностью. Ещё одной тенденцией является значительное увеличение доли шифрованного трафика, что приводит к неприменимости подходов на основе анализа содержимого. Кроме того, в условиях распространения средств анализа и фильтрации многие разработчики сетевых приложений развивают механизмы, противодействующие идентификации используемых протоколов, что также усложняет анализ. В качестве примера такой тенденции можно привести историю развития P2P протоколов, которые начали активно фильтроваться со стороны интернет-провайдеров из-за того, что они слишком сильно нагружали существующие каналы связи, ухудшая качество сервиса для других пользователей этих же каналов. Это в свою очередь привело к ответной реакции от разработчиков P2P-клиентов в виде обфускации используемых протоколов для усложнения их идентификации.

В прикладной области данная задача также широко представлена – существует большое количество как коммерческих, так и свободно распространяемых систем, важнейшие компоненты которых отвечают за её решение.

Спектр предлагаемых решений достаточно широк - известны программные, программно-аппаратные, и полностью аппаратные реализации. Отчасти это связано с тем, что решение данной задачи имеет больше количество практических приложений, среди которых можно выделить:

- системы сбора статистики;
- системы управления трафиком, например, обеспечивающие качество связи (QoS, QoE) и оптимизирующие пропускную способность канала

(Wan Optimization);

- защитные системы: межсетевые экраны (NGFW), системы обнаружения и предотвращения вторжений (IDS/IPS), системы блокировки спама;
- системы применения политик к сетевому трафику (PCEF, PCRF, NAC).

2. Систематизация алгоритмов классификации

Для решения задачи классификации предложено большое количество алгоритмов, которые, в свою очередь можно классифицировать по используемым в них подходам.

С ростом числа подходов возникла потребность в их классификации. Один из вариантов классификации подходов, использующийся в компании Cisco [1], приведён на рис. 1.

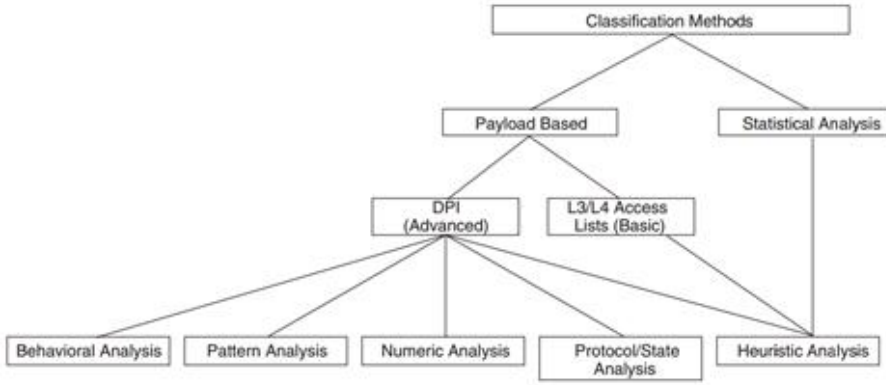


Рис. 1. Подходы к решению задачи классификации
Fig. 1. Approaches to solving the classification problem

Два основных направления – анализ содержимого передаваемых пакетов (payload-based) и статистический анализ характеристик передачи (statistical analysis): последовательность размеров пакетов, временные интервалы между пакетами и т. д.

Два наиболее распространённых метода применяемых для анализа содержимого передаваемых пакетов это поиск сигнатур (Pattern Analysis на рис. 1) и применение разборщиков данных различных протоколов (Protocol/State analysis). Сигнатуры обычно формулируются в терминах регулярных выражений. Хотя эта языковая конструкция является недостаточно выразительной и многие особенности протоколов не могут быть описаны в её терминах, преимуществом данного подхода является его скорость и масштабируемость по количеству сигнатур. В случае тривиальных

строковых сигнатур применяется группа алгоритмов массового поиска строк – типа Ахо-Корасик и турбо Бойер-Мур. В случае регулярных выражений применяются подходы на основе детерминированных и недетерминированных автоматов и гибридные варианты. Подробнее эти подходы рассмотрены в работе [2]. Подход на основе применения разборщиков, с другой стороны, имеет ряд недостатков:

- сложность разработки полноценного разборщика сообщений по сравнению с относительно простыми сигнатурами;
- более низкую скорость работы, которая зависит от применяемых алгоритмов разбора;
- плохую масштабируемость по количеству поддерживаемых протоколов – в худшем случае линейную, так как для гарантированного распознавания может потребоваться полный перебор всех разборщиков на отдельном пакете.

Последний недостаток обычно пытаются компенсировать, улучшая сложность в среднем, за счёт интеллектуального выбора последовательности применения разборщиков. Такие подходы будут подробнее рассмотрены ниже.

Основным преимуществом данного подхода является его высокая точность, так как проверяется полное соответствие структуры сообщения некоторому формату. Это свойство позволяет использовать данный подход для верификации работы других алгоритмов.

Статистические методы, в свою очередь, можно разделить на группы, в зависимости от того, характеристики какого объекта используются для классификации:

- характеристики отдельных пакетов в рамках отдельного потока (packet based);
- характеристики потоков в целом (flow based);
- характеристики нескольких потоков одного сетевого узла (host based);
- характеристики графов потоков (graph-based), применяющиеся в основном для детектирования P2P-протоколов.

Выбор конкретного метода определяется такими факторами как:

- Необходимая пропускная способность – у алгоритмов packet based она минимальна, так как выше объём обрабатываемых данных, у graph-based – максимальна по тем же причинам.
- Необходимая скорость принятия решения о классификации, т.е. количество информации которое надо собрать перед принятием решения. Packet based подходы, как правило позволяют классифицировать поток во время его активности, в то же время для graph-based подходов требуется длительное время наблюдения для сбора статистики.

- Возможные точки в топологии сети для получения информации о трафике, т.е. точки подключения системы классификации или её компонентов ответственных за сбор данных. Так для graph-based требуется возможность получения информации о значительной доли трафика всей сети.

3. Проблемы, возникающие при реализации новых подходов

Однако, несмотря на достаточно активное развитие области классификации сетевого трафика во многих работах отмечается ряд объективных факторов, сдерживающих это развитие [3]. Одним из таких факторов является отсутствие открытого набора данных для тестирования, в качестве которых обычно выступают сохранённые и размеченные сетевые трассы. Вследствие этого затруднено как тестирование качества разрабатываемого алгоритма, так и его сравнение с другими алгоритмами. В частности, это приводит к необходимости решения двух проблем в процессе разработки каждого нового алгоритма.

- Получение собственной сетевой трассы на внутренней сети, от партнёров по исследованию или из публичных источников. Осложняющим фактором является проблема приватности и возникающих рисков информационной безопасности. Для нивелирования этих факторов получаемые трассы, как правило, предварительно подвергаются процедуре анонимизации [4]. Это, в свою очередь, приводит к неприменимости подходов на основе анализа содержимого, так как основным методом анонимизации, помимо прочего, является удаление содержимого пакета уровня приложения.
- Эталонная разметка трассы по протоколам и приложениям, для последующего контроля качества разрабатываемого алгоритма, которая может выполняться несколькими основными способами, в зависимости от того, контролируется ли процесс снятия сетевой трассы или трасса получена из внешнего источника:
- Трасса из внешнего источника:
- Разметка вручную, что является очень ресурсоёмким процессом, с высокой вероятностью ошибок.
- Автоматически с помощью доступных средств классификации трафика. Данный подход приводит к проблеме качества классификации используемого эталонного средства, известной как «ground truth problem» [5].
- Контролируемое получение трассы:
- Ручная разметка по приложениям, которые в момент снятия трассы

выполнялись в системе.

- Использование автоматических средств разметки в процессе снятия трассы, например, [6].

В результате, в большинстве исследовательских работ используются разные трассы, полученные в разных точках разных сетей, на разных сценариях, в разные по длительности промежутки времени.

С другой стороны, требование приватности приводит к более активному развитию статистического направления классификации. Это происходит вследствие того, что для этой группы не требуется доступ к данным пакетов, а достаточно только общих характеристик, таких как размер и метка времени[7,8]. Таким образом, в качестве входных данных подходит большое количество открытых сетевых трасс, прошедших процедуру анонимизации.

4. Системы, использующие классификацию трафика

Помимо вопроса используемого подхода другим важным фактором является прикладная задача, решаемая конкретной системой, в рамках которой реализуется, компонент классификацию. В зависимости от этого, например, может заметно отличаться приемлемый уровень точности результатов классификации. Кроме того, может значительно отличаться и набор групп, на которые разбивается множество классифицируемых объектов. Наиболее грубая классификация используется, как правило, в системах управления трафиком, основной задачей которых является эффективное использование доступной полосы пропускания. Например, провайдер интернета может выделять три основные группы трафика.

- Чувствительный – вид трафика, который чувствителен к задержкам и требует скорейшей доставки. К нему можно отнести VoIP, потоковое видео, трафик онлайн игр.
- Нежелательный – спам и вредоносные типы трафика.
- Остальной – трафик, которому выделяется полоса, оставшаяся после обслуживания потоков чувствительных данных.

Защитные системы и системы применения политик подразумевают, как правило, значительно более точную классификацию – требуется определить конкретное приложение, генерирующее соответствующий трафик. В некоторых случаях необходимо проводить полный разбор трафика с выделением передаваемых команд и высокоуровневых объектов, таких как веб-страницы и другие виды файлов. Это может требоваться, например, для обнаружения потенциально опасного содержимого. Для оценки грубости конкретного подхода используется термин «гранулярность».

Важной характеристикой алгоритма классификации является то, в какой момент времени от начала поступления данных некоторого сетевого потока принимается решение о его принадлежности к тому или иному классу. Для

описания этой характеристики в работе [9] используется термин «ранняя классификация», подразумевающий, что результат классификации появляется вскоре после получения первых пакетов потока (в работе – от 1 до 4 пакетов), что позволяет использовать его, например, в процессе маршрутизации для присвоения приоритета на основе вида трафика. Эта характеристика влияет и на то, в каком классе систем, из приведённых выше, данный алгоритм может использоваться. Например, если алгоритм классификации принимает решение в момент завершения сетевого соединения, то это вполне приемлемо для систем сбора статистики, но неприемлемо для защитных систем.

Фактором, влияющим на оценку подхода, является скорость обработки, т.е. пропускная способность алгоритма. Данная характеристика складывается из двух – количества данных, которые алгоритм должен обработать для получения результата и сложности алгоритма относительно длины входа. Эта характеристика наиболее актуальна для DPI-подходов, которые используют максимальное количеств данных для обработки – всё содержимое отдельных пакетов. Данный вопрос подробно исследуется в большом числе работ, в основном в контексте выбора типа автомата для поиска сигнатур различных протоколов: детерминированный, недетерминированный или некоторый гибридный вариант [10-15].

Важной комплексной характеристикой системы классификации трафика является область ей применимости. В неё входит, как способность системы обрабатывать отдельные виды трафика (шифрованный, p2p и т.д.), так и то, в каких условиях данная система может функционировать и к каким особенностям передачи трафика она устойчива (потери и перестановки пакетов, асимметрия и т.д.). Указанные особенности трафика и их влияние на применимость, точность и скорость работы различных подходов будут рассмотрены ниже.

4.1 Оценка систем классификации

С учётом вышесказанного в работе [16] вводятся параметры конкретных реализации системы классификации, и приводится оценка подходов к классификации по этим параметрам. Эти оценки приведены на рис. 2.

- Точность – общая характеристика, отражающая долю правильно идентифицированного трафика от общего количества проанализированного трафика. Точность результатов определяется в основном тем, насколько хорошо выбраны признаки, по которым осуществляется классификация и качеством применяемой эвристики.
- Время реакции – время от момента получения первого пакета некоторого сетевого потока до момента его классификации. Является критичным для систем, работающих «на потоке», в частности защитных и систем управления трафиком. В это понятие также входит общая производительность алгоритма.

- Надёжность – отражает область применимости системы (например, возможность анализировать зашифрованный трафик) и устойчивость к возникающим в процессе передачи эффектам, таким как потери пакетов, асимметрия и т.д.

Classification methods	Port-based	Payload-based	Statistical classification	Host behavior based
Accuracy	Low	Low	Higher	Higher
Real-time	High	Middle	Higher	Higher
Robustness	Low	Low	Higher	Higher
Advantages	Simple,small computational overhead	No	Robustness, accuracy, fine-grained	Simple,small computational overhead
Disadvantages	Low accuracy, cannot be used alone	Almost useless, privacy risk	Large computational overhead,a lot of training,not stable when traffic changes	Coarseclassification,useless when transport layer encrypted, degradation in case of NAT
Status	Not in use	Not in use	Under test	Under test

Рис. 2. Оценка различных подходов к классификации трафика
Fig. 2. Evaluation of different approaches to traffic classification

Также в работе [16] приведены формулы, использующиеся для оценки точности алгоритма (в приведённых выше терминах). Для этого используется ряд метрик, основанных на доле истинных и ложных результатов классификациитrue/falsepositives/negatives (TP, TN, FN, FP). Наиболее часто используемыми метриками являются следующие [17].

- Правильность: Accuracy = (TP + TN)/(TP + TN + FP + FN).
- Точность: Precision = TP/(TP + FP).
- Полнота: Recall = TP/(TP + FN).
- F-мера. Так как максимальная точность и полнота недостижимы одновременно, то приходится искать некий баланс, который может оцениваться с помощью гармонического среднего между точностью и полнотой:
    - F=2*(Precision * Recall) / (Precision + Recall), в случае одинакового веса точности и полноты.В работе [18] был выбран другой набор параметров для оценки алгоритмов классификации. В него вошли: используемые для классификации данные, гранулярность результата, время принятия решения и вычислительная сложность подхода. Результаты сравнения подходов по этим параметрам

приведены на рис. 3.

Approach	Properties exploited	Granularity	Timeliness	Comput. Cost
Port-based	Transport-layer port	Fine grained	First Packet	Lightweight
Deep Packet Inspection	Signatures in payload	Fine grained	First payload	Moderate, access to packet payload
Stochastic Packet Inspection	Statistical properties of payload	Fine grained	After a few packets	High, eventual access to payload of many packets
Statistical	Flow-level properties	Coarse grained	After flow termination	Lightweight
	Packet-level properties	Fine grained	After few packets	Lightweight
Behavioral	Host-level properties	Coarse grained	After flow termination	Lightweight
	Endpoint rate	Fine grained	After a few seconds	Lightweight

Рис. 3. Сравнение подходов к классификации сетевого трафика
Fig. 3. Comparison of approaches to classification of network traffic

Совокупность оценок подходов, приведённых на рис. 2 и 3 могут быть использованы для определения применимости конкретного подхода в заданных ограничениях. Формулы для количественных оценок могут использоваться для сравнения нескольких реализаций алгоритмов классификации в рамках выбранного подхода.

5. Особенности сетевого трафика, влияющие на скорость и точность классификации

Помимо особенностей выбранного подхода и качества его реализации на скорость, точность и применимость конкретного алгоритма значительное влияние могут оказывать особенности передачи данных по сети. Следует отметить, что большинство описываемых далее особенностей не касаются напрямую задачи классификации, а являются предпосылками, на основе которых могут формулироваться требования, предъявляемые к системе, использующей компонент классификации. По сути, большая часть особенностей формирует набор необходимых видов предобработки, которые необходимо применять к данным, передаваемым посредством сетевых пакетов, прежде чем передавать их на вход алгоритму классификации. Отсутствие соответствующих видов предобработки может приводить либо к сужению области применимости, либо к потенциальному снижению точности некоторых подходов, либо к падению пропускной способности алгоритма.

5.1 Асимметрия

В зависимости от топологии сети и месте размещения компонента классификации может возникать ситуация, при которой не все передаваемые по сети пакеты будут проходить через компонент классификации – возникает асимметрия трафика. Подобная ситуация, является весьма распространённым явлением в корпоративных сетях [19], а также на магистральных каналах [20]. В работе [21] приведена типичная схема сети (рис. 4), приводящая к возникновению асимметрии. Там же приводится классификация видов асимметрии.

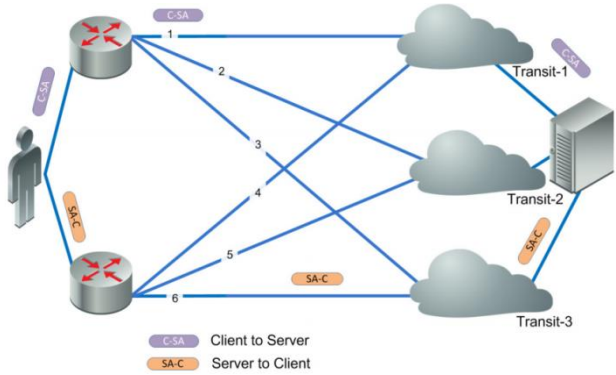


Рис. 4. Пример маршрутизации EQMP, при которой возникает асимметрия
Fig. 4. An example of EQMP routing, in which asymmetry arises

- Потоковая асимметрия.
  - Полная потоковая асимметрия, при которой любой пакет в обоих направлениях может проходить через любой шлюз.
  - Частичная консистентная потоковая асимметрия, при которой все пакеты одного потока заданного направления проходят через один шлюз, а все пакеты обратного направления этого же потока – через другой.
- IP-асимметрия.
  - Полная IP-асимметрия, при которой все потоки с одного IP-адреса могут проходить через любые шлюзы.
  - Частичная IP-асимметрия, при которой все пакеты одного потока для конкретного IP-адреса в обоих направлениях проходят через один шлюз, но разные потоки этого IP-адреса могут проходить через разные шлюзы.
  - Частичная консистентная IP-асимметрия, при которой при которой все пакеты одного IP-адреса заданного направления проходят через один шлюз, а все пакеты обратного

направления этого же IP-адреса – через другой.

При организации спутниковых каналов связи, асимметрия трафика является неотъемлемым свойством передачи данных [22]. Асимметрия может значительно влиять на качество классификации. Её наличие может приводить к:

- усложнению TCP-нормализация в отсутствие ACK-пакетов,
- невозможности IP-дефрагментации из-за отсутствующих пакетов,
- невозможности осуществить декодирование потокового сжатия,
- нарушению временных и размерных характеристик трафика.

В работе [21] рассматриваются различные способы решения проблемы асимметрии – от замыкания всех потоков в одной точки, до реализации распределённой системы классификации.

5.2 Группировка пакетов в потоки и сеансы

В процессе отдельного сеанса связи последовательность пакетов передаётся по сети в рамках некоторого потока, обладающего характеристиками, общими для соответствующей группы пакетов. Существует ряд используемых определений потока, наиболее распространённые из которых приведены на ресурсе [23]. В данной работе, при упоминании*потока* будет подразумеваться «односторонний поток транспортного уровня» – последовательность пакетов передающихся с заданного IP-адреса и TCP/UDP порта на данный IP-адрес и TCP/UDP порт, с указанием протокола транспортного уровня (TCP/UDP). Таким образом, поток задаётся пятёркой <srcIP, srcPort, dstIP, dstPort, protocol>. Так как пакеты, относящиеся к одному потоку, в некоторый промежуток времени, как правило, генерируются одним и тем же приложением, то обычно объектом классификации является не отдельный пакет, а поток целиком. Это позволяет оптимизировать задачу классификации, уменьшив объём обрабатываемых данных. Для этого используется принцип «до первого срабатывания», подразумевающий последовательный анализ пакетов одного потока, до момента его идентификации – последующие пакеты могут игнорироваться. Применение этого принципа может приводить к ошибкам (или неточностям) классификации, вследствие возможной вложенности протоколов. В частности, многие приложения, такие как Skype используют протокол HTTP в качестве транспортного для передачи своих данных.

Важность группировки возрастает в случае использования поточных алгоритмов сжатия – невозможно осуществить декодирование пакета, недекодировав перед этим предыдущие пакеты потока. Примерами таких алгоритмов являются Deflate и Gzip, которые будут подробнее рассмотрены ниже. Обобщением этой особенности является необходимость хранения некоторой информации на протяжении жизни потока – его «состояния». Помимо состояния декодера, примером такой информации является состояние протокола, для протоколов с состоянием (stateful). Причём для одного потока

состояний может быть несколько – по одному на каждый stateful протокол из используемого сетевого стека. Например, для stateful протокола QUIC – состояние будет одно, т.к. транспортный протокол UDP, поверх которого он реализован, состояния не имеет, а для протокола FTP – состояний будет два – одно собственно для FTP, второе – для TCP, поверх которого он реализован. Примером средства, архитектура которого предполагает поддержку stateful анализа на всех уровнях является инструмент GAPA [24], реализация которого, однако недоступна.

Потоки, полученные в результате группировки пакетов, в ряде случаев могут быть сгруппированы в сессии – группы связанных потоков, отвечающих за предоставление некоторого сетевого сервиса. Одним из ярких примеров сессии является пара потоков протоколов SIP и RTP, первый из которых отвечает за передачу команд, а второй – данных, при организации сеанса VoIP связи. Понятие *сессия* используется, например, в описании компонента классификации NBAR от компании Cisco [25]. Выделение сессий в некоторых случаях необходимо для корректной классификации. Примером может служить классификация потока данных FTP для FTP-сервера, работающего в пассивном режиме на основе анализа командного потока: в самом потоке данных отсутствуют какие либо заголовки и сигнатуры, так как данные передаются в «сыром» виде в ответ на соответствующую команду, передающуюся в другом потоке.

Возможна и обратная ситуация, когда несколько сессий (в смысле взаимодействий, соответствующих некоторому сетевому сервису) могут быть упакованы в один поток транспортного уровня. Простым примером такой ситуации является режим постоянного HTTP-соединения (keepalive), появившийся в версии 1.1[26] протокола - использование одного TCP-соединения для отправки и получения многократных HTTP-запросов и ответов вместо открытия нового соединения для каждой пары запрос-ответ. Развитием этой идеи является мультиплексирование, появившееся в версии 2.0[27], которое позволяет одновременные многократные запросы/ответы в одном соединении. Более сложным примером может служить протокол CitrixIndependentComputingArchitecture (ICA) [28], который в одном из режимов работы позволяет нескольким приложениям одновременно использовать одно TCP соединение. Этот пример также демонстрирует, что правило «один поток – одно приложение» не является абсолютным.

Следует отметить, что многие средства классификации, такие как nDPI [29], снимают с себя задачу группировки пакетов в потоки, предполагая, что данные, передаваемые на классификацию, уже сгруппированы в потоки некоторым внешним компонентом.

5.3 Изменчивость

С течением времени характеристики сетевого трафика меняются – в существующие протоколы вносятся изменения, появляются новые протоколы.

Это приводит к необходимости поддержания компонента классификации в актуальном состоянии. В общем случае для этого необходимо решать две задачи – выявление новых классифицирующих признаков или уточнение существующих, а также внедрение новых признаков в компонент классификации, работающий «на потоке». В случае port-based подходов это означает необходимость регулярного обновления таблицы соответствия пар <тип транспортного протокола, номер порта> протоколам прикладного уровня [30]. В случае DPI-подходов это означает необходимость автоматического [31] или ручного определения новых «сигнатур» и их добавления в компонент классификации. Среди алгоритмов автоматического поиска сигнатур можно выделить подходы на основе поиска повторяющихся шаблонов [32] и подходы на основе алгоритмов биоинформатики [33]. Альтернативный вариант получения новых сигнатур – заказ у стороннего поставщика соответствующей услуги [34]. Для их получения в этом случае обычно используется специальный программный инструментарий поддержки, примерами которого могут служить [35, 36]. Для статистических подходов, которые, как правило, основаны на алгоритмах машинного обучения это означает необходимость периодического переобучения [37], иначе их точность значительно деградирует [38]. Дополнительной трудностью для статистических алгоритмов является необходимость дополнительного класса «неизвестного трафика», в который требуется относить трафик, плохо подходящий к другим классам по своим характеристикам [39]. Независимо от подхода, возникает задача аналогичная ситуации с уязвимостью «нулевого дня» в информационной безопасности – минимизация времени от момента обнаружения неизвестного трафика до момента его корректного распознавания на потоке. В случае DPI-подходов это время суммируется из двух компонент – время на создание новой сигнатуры и время на перестроение представления, использующегося для распознавания (например, детерминированного автомата). В случае алгоритмов машинного обучения – это время на переобучение, обеспечивающее качественное распознавание нового протокола. В обоих случаях возникает дополнительная подзадача – получение достаточного количества материала для анализа и его предварительного разбиения на классы, в соответствии с которыми будет выделяться сигнатура или осуществляться переобучение. Это приводит к необходимости присутствия компонента «дополнительной классификации» трафика (возможно включающего ручной анализ), с которым не справился основной компонент классификации. В рамках этого компонента должны сохраняться образцы нераспознанного трафика, уточняться сигнатуры и эвристики, проводится переобучение (в случае алгоритмов машинного обучения) перед обновлением основного компонента классификации.

Общий вид архитектуры системы классификации, с учётом описанных подзадач имеет вид, приведённый на рис. 5, из работы [40].

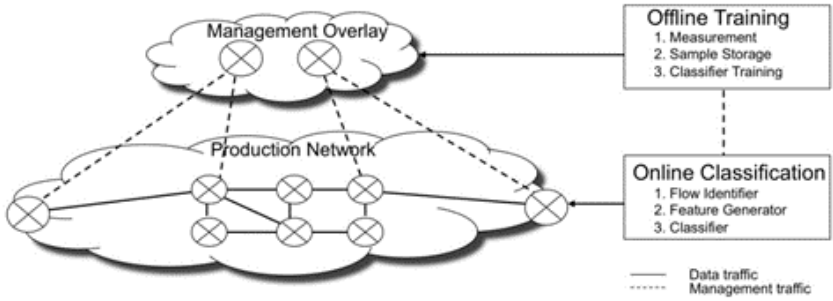


Рис. 5. Общая архитектура системы классификации трафика
Fig. 5. General architecture of the traffic classification system

5.4 Шифрование

Один из наиболее распространённых протоколов, использующихся для шифрования передаваемых данных – SSL, в частности обеспечивающий слой шифрования в HTTPS. Особенностью шифрованных данных является то, что к ним неприменимы алгоритмы классификации, использующие данные пакетов, уровня приложения (DPI на рис. 1). Таким образом, шифрование снижает применимость этого, достаточно обширного, класса алгоритмов. Кроме того, попытка применять данные подходы на шифрованном трафике, не отделяя его предварительно от остального потока, существенно снижает пропускную способность компонента классификации, так как приводит к частому проявлению «худшего» случая – просмотру всех данных пакета при отсутствии решения по его принадлежности к некоторому классу [41]. Для преодоления этого недостатка могут использоваться подходы, определяющие наличие факта сжатия или шифрования на основе измерения энтропии [41, 42]. Для классификации шифрованного трафика разрабатываются специализированные подходы. Некоторые из них основаны на анализе первых нескольких пакетов соединения – т.н. «рукопожатие», при котором стороны договариваются об используемом алгоритме шифрования, его параметрах и т.д. [43]. Однако наиболее перспективным выглядит применение подходов на основе машинного обучения, т.к. именно этому подходу посвящено большинство работ по классификации шифрованного трафика [44, 45]

В настоящее время, общая доля шифрованного трафика по данным Sandvine [44] за 2015 год составляет порядка 30% и имеет выраженную тенденцию к увеличению.Этому, в частности, способствует два основных фактора:

- решение крупного поставщика видеоконтента Netflix (наряду с YouTube) перейти на шифрованную передачу данных по HTTPS;

- инициативы Let'sEncrypt группы InternetSecurityResearchGroup (ISRG) и HTTPSEverywhere от ElectronicFrontierFoundation по автоматическому предоставлению свободных и бесплатных сертификатов всем желающими принудительного использования шифрованного соединения.

Несмотря на то, что напрямую DPI подход к шифрованному трафику не применим, во многих программных продуктах, предназначенных для классификации, в отдельных частных случаях классификация такого трафика всё же осуществляется. Это возможно, если сервис, с которым осуществляется обмен использует расширение Server Name Identification (SNI) протокола TLS для явного указания имени хоста, с которым будет осуществляться обмен. Соответственно классификатор может поддерживать базу имён хостов для наиболее часто используемых сервисов и на основании этого имени идентифицировать трафик, например к хосту google.com. Такая база, по сути, является аналогом базы портов IANA [30] с тем отличием, что идентифицирующим протокол признаком является не номер порта, извлекаемый из заголовка транспортного уровня, а строковое имя хоста, извлекаемое из заголовка TLS. Изначально это расширение было реализовано для того, чтобы была возможность предоставлять различные сертификаты для сайтов, использующих один и тот же IP-адрес и TCP-порт, что в настоящее время достаточно распространено.

5.5 Туннелирование

Туннелирование – это процесс, в ходе которого создается логическое соединение между двумя конечными точками посредством инкапсуляции различных протоколов. Может применяться для создания защищённых соединений (VPN) или для организации взаимодействия сетей, использующих разные протоколы (IPv6 и IPv4). Например, протокол GRE применяется для инкапсуляции пакетов сетевого уровня (OSI) в IP-пакеты и используется, в частности, для доступа в интернет с мобильных устройств.

От обычных многоуровневых сетевых моделей (OSI, TCP/IP) туннелирование отличается тем, что инкапсулируемый протокол относится к тому же или более низкому уровню сетевого стека, чем используемый в качестве туннеля.

В процессе туннелирования принимают участие следующие типы протоколов:

- транспортируемый – протокол объединяемых сетей;
- несущий – протокол транзитной сети;
- инкапсулирующий – помещает пакеты транспортируемого протокола в поле данных пакетов несущего протокола.

Использование туннелирования потенциально может вызывать трудности у всех подходов, но по разным причинам. В случае подходов на основе анализа содержимого (payload-based на рис. 1), требуют дополнительных усилий по разбору дополнительных заголовков, относящихся к организации туннеля,

чтобы получить данные приложения (в случае DPI) или последний заголовок транспортного уровня (в случае port-based подходов, L3/L4 Access Lists на рис.1). В случае применения статистических подходов требуется адаптация к конкретному туннелю, так как его наличие вносит искажение как в размеры пакетов (за счёт добавления заголовков), так и во временные характеристики (за счёт необходимости дополнительной обработки).

5.6 Сжатие

Используется многими протоколами для уменьшения объёма передаваемых данных, примерами могут служить DNS, где вместо повторов строк ставятся ссылки на первое вхождение и HTTP в котором применяются алгоритмы Deflate и GZIP. Влияние данной особенности с одной стороны сходно с влиянием туннелирования – могут меняться временные характеристики и размеры пакетов, а с другой – с влиянием шифрования на подходы DPI: отсутствие предварительного декодирования снижает точность и приводит к замедлению. Основное отличие от зашифрованного трафика – возможность декодирования «на лету» без знания сторонней информации (ключей шифрования). Однако само декодирование может быть ресурсоёмкой задачей, требующей дополнительных вычислительных ресурсов. Близость сжатых и шифрованных данных по их влиянию на DPI подходы подчёркивается общим термином «непрозрачные»(opaque) данные, использующимся для обозначения этих видов данных в работе [41]. Некоторые дополнительные аспекты, связанные с анализом непрозрачного трафика приведены в работе [44].

5.7 Фрагментация данных передаваемых по сети

Необходимость фрагментации связана с физическим ограничением на максимальный размер данных, которые могут быть переданы в одном пакете с использованием конкретной среды передачи (физический уровень модели OSI), а также с особенностями реализации конкретных протоколов (количеством байт в полях, хранящих размеры пакета). В частности, фрагментация имеет место на сетевом уровне (уровень IP TCP/IP стека), а её аналог - сегментация на транспортном уровне (уровень TCP TCP/IP стека). Фрагментация может иметь место и на уровне приложений – примером может служить режим chunkedtransferencoding, появившийся в версии 1.1 протокола HTTP. Фрагментация может влиять как на размеры пакетов, так и на то, что шаблон, используемый для классификации, может быть разделён между несколькими пакетами. Это особенно актуально для DPI-подходов, однако, в случае специально сгенерированных пакетов, это может повлиять даже на port-based подходы. Такие пакеты, в частности, используются для обхода межсетевых экранов при организации сетевых атак [45, 46].

6. Актуальные направления развития

С учетом факторов, затрудняющих классификацию с использованием разработанных ранее подходов, можно выделить несколько направлений для преодоления отдельных групп ограничений. Так для обработки зашифрованного трафика наряду с статистическими методами, точность которых пока относительно невысока применяется подход на основе идентификации сервиса, которые также комплементарны алгоритмам классификации и в ряде случаев позволяют повысить их точность. Для повышения скорости работы алгоритмов DPIпри сохранении относительно высокой точности применяются различные гибридные подходы, позволяющие уменьшить количество обрабатываемых данных и число проверяемых сигнатур. Для адекватного сравнения различных инструментов в условиях отсутствия открытой базы сетевых трасс и проблем с приватностью создаются модульные системы классификации. В рамках таких систем, отдельные инструменты и алгоритмы реализуются в виде независимых модулей, что позволят их оценивать и сравнивать на одних и тех же доступных конкретному исследователю трассах. Эти направления подробно рассматриваются в следующих разделах.

Для анализа P2P приложений, а также ситуаций, когда нет доступа к данным пакетов (наличие шифрования или только данных о потоках) применяются методы анализа поведения отдельных хостов и характеристик групп потоков на уровне графов сетевых взаимодействий.

6.1 Подход на основе идентификации сервиса

Одной из важных проблем для многих алгоритмов классификации является большой объём данных для обработки. Наиболее остро эта проблема стоит для DPIподходов, в которых требуется обрабатывать все данные пакетов. Для преодоления этого недостатка, независимо от подхода, используемого в работе [47], была предложена схема классификации на основе идентификации сервиса (service-basedclassification). Данная схема, по сути, является развитием схемы «до первого срабатывания», когда определив протокол по нескольким первым пакетам автоматически предполагается, что все пакеты данного потока также относятся к определённому протоколу. Идея заключается в том, что во многих сетевых взаимодействиях можно выделить серверную сторону, предоставляющую некоторый сервис по фиксированному IP-адресу и порту в течение длительного времени с использованием фиксированного протокола. Выделив серверную сторону и идентифицировав протокол для некоторого отдельного потока, можно с высокой долей вероятности утверждать, что в других сетевых потоках, одной из сторон которых является данный сервер (пара IP-адрес и порт) будет использовать тот же протокол. Ранее данная идея использовалась для построения базы знаний о доступных сетевых сервисах, с последующей их валидацией [48]; в частности, этот подход использовался для выделения P2Pтрафика [49]. Ранее рассмотренный подход классификации

шифрованного трафика на основе имени сервиса в SNI-расширении протокола TLS является частным случаем данного подхода. Среди проблем данного подхода можно указать:

- Высокую цену ошибки классификации, так как в этом случае ошибка распространится на большое число соединений с некоторым сервисом.
- Ограничение применимости к протоколам с динамическими портами, выделяемыми в рамках управляющего потока (FTP, SIP).
- Ограничение применимости к прокси-серверам (SOCKS), которые хоть и предоставляют фиксированный сервис на фиксированных портах, но этот сервис не привязан ни к какому протоколу прикладного уровня и может использоваться различными приложениями.
- Подход не применим к трафику, шифрование которого осуществляется на IP-уровне (IPSec).
- Применимость только к TCP-трафику, так как для идентификации сервера в контексте каждого отдельного соединения используется пара SYN-ACKфлагов в процессе рукопожатия при установке соединения.

6.2 Комбинации подходов и оптимизация алгоритмов

Основным преимуществом использования DPI-систем классификации является то, что помимо решения задачи классификации эти системы, как правило, применимы для полного разбора сетевых протоколов уровня приложения, извлечения передаваемых данных (сайтов, файлов, медиа-потоков и т.д.) и применение к ним высокоуровневых правил фильтрации и различных политик. Среди коммерческих систем данного класса можно указать Qosmos Intelligence Engine [50], ipoque PACE [51], Windriver Content Inspection Engine [52], Procera PacketLogic Content Intelligence [53]. Так как данные системы относятся к классу корпоративных, проблема шифрованного трафика (к которому DPI-подходы напрямую неприменимы) решается с помощью регистрации на всех машинах сети доверенного корневого сертификата, соответствующего используемому DPI-решению и реализацию схемы аналогичной MITM-атаке с распаковкой, анализом и последующей запаковкой передаваемого трафика. Примером реализации такой схемы является технология DPI-SSLot SonicWALL [54].

Среди других ограничений DPI-подхода можно указать уже упоминавшийся большой объём данных для анализа, а также рост сложности анализа с добавлением поддержки новых протоколов. Для преодоления этих недостатков используются гибридные схемы, некоторые из которых будут рассмотрены далее.

Исследование количества анализируемых байт для классификации

Целью исследования авторов работы [55] была попытка анализа известных инструментов DPI (в работе использовался инструмент с открытым исходным кодом L7[56]) на предмет:

- Количества анализируемых пакетов в одном потоке
- Наиболее частого смещения в пакетах сигнатур, поиск которых осуществляют инструменты DPI.

Для анализа количества анализируемых байт был построен график зависимости доли классифицированных потоков от величины смещения сигнатуры, по которой он был классифицирован – см. рис. 5.

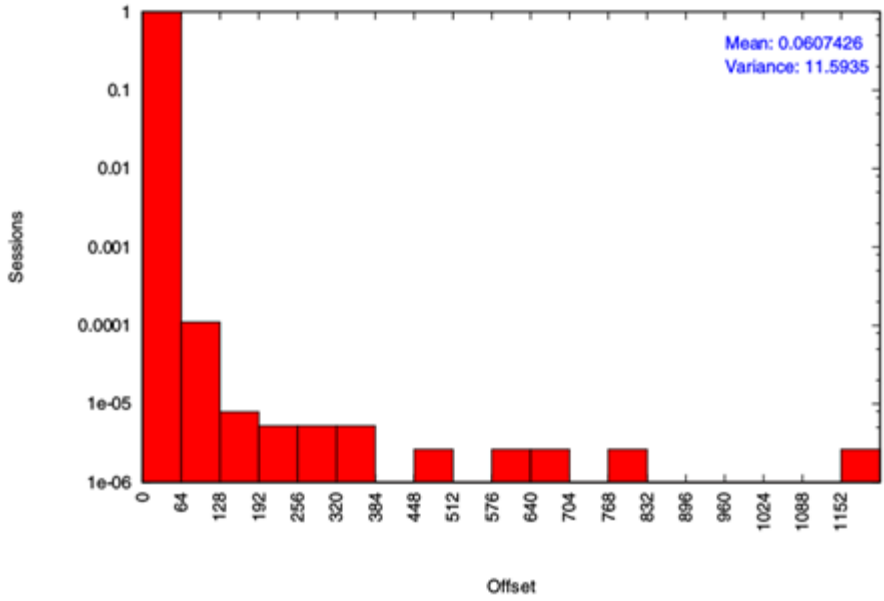


Рис. 5. График зависимости доли классифицированных потоков от смещения классифицирующей сигнатуры для инструмента L7
Fig. 5. Graph of the dependence of the fraction of classified flows on the shift of the classifying signature for the tool L7

График показывает, что в абсолютном большинстве случаев сигнатура расположена в пределах первых 32 байт первого пакета потока. Для проверки точности подхода на основании анализа только первых 32 байт-методом сравнения битовых строк, авторами был разработан инструмент PortLoad, который применял сигнатуры протоколов к первым 32 байтам по аналогии с тем, как применяется сопоставление в подходе на основе портов к 2 байтам номера порта. Было показано, что такой подход позволяет получить точность гораздо более высокую (74%), чем подход на основе анализа портов (24%) и

ненамного более низкую, чем подходна основе полноценного DPI (97%). В то же время скорость анализа была близкой к скорости анализа port-based подхода и значительно превышала скорость DPIанализа, как за счёт уменьшения количества обрабатываемых данных, так и за счёт использования более быстрых алгоритмов сравнения фиксированных строк, а не поиска регулярных выражений. Так, на исследованной трассе время работы реализованного алгоритма всего в 2.8 раза превышало время работы port-basedалгоритма и было в 30 раз меньше времени работы инструмента L7.

Легковесный анализ пакетов

Целью авторов работы [57] являлось создание инструмента классификации сравнимого по точности с известными решениями на основе DPI-подхода, однако использующим для анализа меньше данных, что делало бы его менее ресурсоёмким и более быстрым.

Одной из проблем, возникающих при реализации подхода на основе DPI – ограниченность данных для анализа, вследствие проблем с приватностью: большинство свободно-распространяемых трасс, в частности организацией CAIDA [23] содержит «обрезанные» пакеты. Ограничение длины входных данных, необходимых для анализа позволило бы использовать такие трассы, решая данную проблему.

Свой подход авторы определили, как легковесный анализ пакетов (LightweightPacketInspection, LPI), и реализовали в виде библиотеки с открытым исходным кодом. Идея заключается в использовании совокупности подходов на основе анализа портов, статистического анализа и DPI для нивелирования ограничений при сохранении точности. В качестве сигнатуры используется совокупность значений:

- IP-адресов и портов обоих участников обмена (на основе портов)
- Длины пакетов в обоих направлениях (статистические подходы)
- Первые 4 байта пакетов уровня приложения в обоих направлениях (DPI)

Обоснованием уменьшения размеров анализируемых данных пакетов уровня приложения являются следующие факты:

- в работе [55] было показано, что для классификации большинства приложений используются сигнатуры для префикса пакетов ограниченной длины (в работе использовался префикс длины 32)
- большинство свободно-доступных трасс используют стандартную анонимизацию, которая заключается в удалении содержимого пакетов уровня приложения, за исключением первых 4х байт.

При вызове функции идентификации последовательно вызываются независимые модули распознавания, соответствующие различным протоколам (на данный момент поддерживается около 200 протоколов). Порядок вызова модулей соответствует их приоритетам (задаются вручную), которые

выставляются в соответствии с точностью и простотой функции идентификации и популярностью данного протокола.

Функция идентификации может включать четыре вида правил:

- Правила на содержимое – может включать символ *, соответствующий произвольному символу
- Правила на размер пакета, использующиеся для протоколов не содержащих явных сигнатур в первых 4х байтах
- Правила на номера портов, применяющиеся только для портов с фиксированными протоколами
- Правила на IP-адреса, использующиеся для идентификации специфических сервисов

Оценки качества классификации показывают, что хотя точность анализа и ниже, чем у других открытых инструментов, в ряде случаев она может быть достаточна для практического применения.

Извлечение метаданных

Авторы инструмента с открытым исходным кодом nDP [15] рассматривают решение задачу классификации не как некоторую самоцель, но в комплексе, на примере того какие задачи на её основе решаются в коммерческих инструментах, таких как iPoque [51] и Qosmos [50]. Они приходят к выводу, что помимо определения протокола уровня приложения требуется извлекать метаданные, т.е. значения отдельных полей высокоуровневых протоколов, что в свою очередь требует разработки полноценных разборщиков этих протоколов. Наличие таких разборщиков, в свою очередь означает, что факт их применимости к некоторым данным может служить гарантией того, что данные относятся к протоколу, которому соответствует разборщик. Эта же идея лежит в основе инструмента Wireshark [58]. В этом случае ключевым вопросом производительности является порядок применения разборщиков к данным, чтобы избежать линейного роста сложности по мере увеличения числа поддерживаемых протоколов. Для улучшения скорости анализа в среднем использовались следующие предположения:

- Повышенная вероятность работы приложений с учётом транспортного протокола на закреплённых за ними портах [30].
- Уменьшение количества применяемых разборщиков по мере анализа последовательности пакетов в потоке с отбрасыванием тех, которые должны были сработать на некотором начальном префиксе данных потока.

Также в работе [15] было проведено исследование на предмет минимального количества пакетов, необходимого для классификации потоков различных протоколов. На основании исследования был сделан вывод о том, что эта величина сильно протоколо-зависима и хотя для многих протоколов составляет 1(DNS, NetFlow, SNMP) для специфических протоколов, таких как

BitTorrent она равна 8. Этот порог и был выбран в качестве ограничения для количества анализируемых пакетов в потоке.

Благодаря возможности извлечения метаданных в инструмент был добавлен подход на основе идентификации сервиса по имени хоста (HTTP) или используемым сертификатам в случае шифрованного потока (SNI). Для сопоставления имён с базой известных сервисов использовалась реализация алгоритма Ахо-Корасик. Ещё один побочный положительный эффект от извлечения метаданных – возможность классификации связанных потоков, например, автоматическая классификация потока данных RTP, на основе разбора управляющего потока SIP. Ещё более актуально это для протокола FTP, в случае работы сервера в пассивном режиме – в этом случае поток данных в принципе не содержит специфических сигнатур и заголовков и может быть классифицирован только на основе анализа потока команд.

Сам инструмент nDPI разрабатывался на основе открытой кодовой базы проекта OpenDPI, которая была в значительной мере переработана, в частности, для оптимизации работы в многопоточном режиме.

6.3 Анализ графов взаимодействий

Одной из первых работ, в которой авторы исследовали поведения отдельного сетевого узла с точки зрения шаблонов его взаимодействия с другими узлами является работа [59]. Авторы выделяют несколько уровней шаблонов сетевых взаимодействий отдельного сетевого узла:

- Социальный – количество сетевых узлов (IP-адресов), с которыми данный узел взаимодействует (популярность узла) и их объединение в связанные сообщества.
- Функциональный - набор ролей, в которых выступает данный узел (клиент, сервер или оба варианта). На данном уровне учитываются взаимодействия по разным портам.
- Прикладной – анализ проводится на уровне транспортных потоков (пар IP-адресов и портов) для идентификации отдельных приложений по эвристическим шаблонам, включающим количество пакетов, байт и транспортный протокол передачи.

Авторы показывают, что такой подход позволяет детектировать p2pвзаимодействия, новые неизвестные протоколы, а также некоторые виды атак.

Развитие идея анализа графов для анализа сетей получила в работе [60]. В ней произошла формализация представления массива сетевых взаимодействий в виде графа дисперсии трафика (TrafficDispertionGraph, TDG). Данный граф описывается как пара множеств V-вершин и E-рёбер, где:

- v изV – вершина графа, точка (сетевой узел) в сети, с заданным IP-адресом;

- e из E – ребро, показывающее наличие сетевого потока между сетевыми узлами.

В работе было показано, что клиент-серверные взаимодействия (HTTP)значительно отличаются от взаимодействий типа P2Pпо плотности графа взаимодействий сетевого узла (2|E|/|V|). В случае P2Pэта плотность значительно выше. Всего для анализа использовались три характеристики:

- плотность;
- количество сетевых узлов с двусторонними соединениями (ведут себя и как клиент и как сервер);
- эффективный диаметр (из 90% выборки взять наибольший диаметр графа).

По совокупности этих параметров взаимодействия P2P отличались от HTTP-подобного трафика с практически 100% точностью. На основе этих фактов был предположен метод, основанный на классификации трафика по поведению сетевых узлов, который можно эффективно использовать для распознания P2P трафика. В процессе испытаний метода был выявлен существенный недостаток, который заключается в том, что данный метод плохо отличает P2Pтрафик от других стандартных протоколов типа DNSи SMTP. В процессе развития, для преодоления этого недостатка было найдено два подхода – усложнение анализа графа с учётом динамики его изменения со временем и комбинирование подхода с другими в рамках модульных систем классификации (в частности, подхода на основе анализа портов). Первый из указанных подходов рассматривается в работе [61], в котором показано, что процесс изменения TCG-графа со временем значительно отличается у DNSи P2Pи, таким образом, служит лучшей характеристикой используемого протокола. Для оценки временных изменений в работе были предложен ряд новых метрик, показывающих как изменение в структуре (без учёта конкретных рёбер и вершин), так и изменение в составе вершин и рёбер. Второй подход подробно рассматривается в следующем разделе.

6.4 Модульные системы классификации

В качестве основных аргументов для создания модульных систем классификации, в которых независимые модули реализуют различные подходы и алгоритмы классификации можно указать следующие:

- Необходимость инструмента для качественного сравнения разных подходов с целью выявления их сильных и слабых сторон на одних и тех же входных данных, что в обычных условиях невозможно из-за проблем с приватностью и отсутствия публичных тестовых наборов.
- Комбинация различных подходов с целью взаимного нивелирования их слабых сторон для повышения качества классификации.

Среди известных систем можно указать NetraMark [62]. В качестве базовых принципов разработки указываются:

- Совместимость реализуемых подходов – корректное сопоставление классов/приложений, в рамках которых осуществляется классификация в разных инструментах.
- Воспроизводимость – возможность верификации результатов, полученных разными исследовательскими группами с учётом особенностей входных данных.
- Расширяемость – простота добавления новых подходов в качестве отдельных модулей.
- Синергия – подбор комбинаций классификаторов для взаимного усиления.
- Гибкость настройки – простое переконфигурирование отдельных алгоритмов, быстрый подбор свойств, используемых для классификации.

Примерно эти же принципы были положены в основу разрабатываемой с 2008 года по настоящее время платформы TIE[63,64] (TrafficIdentificationEngine). В данной платформе больший упор сделан на перспективные методы машинного обучения, однако реализовано и большое количество подходов DPI и их оптимизированных версий, рассмотренных в предыдущих разделах. Кроме того, лучше проработаны вопросы архитектуры платформы с выделением отдельных функциональных компонент. Общая схема архитектуры приведена на рис. 6.

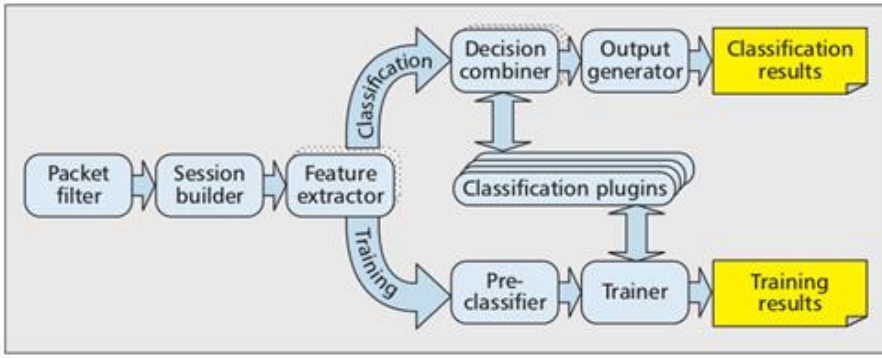


Рис 6. Общая схема архитектура платформы TIE
Fig. 6. General scheme of the architecture of the TIE platform

Исследованию вопросов совместного использования различных походов и алгоритмов классификации с точки зрения как уровней входных данных для анализа, так и осмысленности обмена результатами классификации (с целью

кросс-валидации), посвящена работа [65]. Данная работа содержит перечисление большинства известных подходов с анализом их сильных и слабых сторон, на основании которого строится схема их оптимального применения к входным данным, с учётом взаимного обмена результатами классификации. Полученная схема приведена на рис. 7.

7. Заключение

На основании проведённого обзора подходов к классификации трафика, можно сделать следующие выводы.

- Существует большое количество алгоритмов и подходов с различными достоинствами, недостатками, отличающихся по скорости обработки, области применимости и точности получаемых результатов.
- Сравнение различных алгоритмов значительно затруднено из-за отсутствия общедоступной базы полноценных размеченных сетевых трасс, на которых было бы возможно проводить сравнения. Отсутствие такой базы вызвано объективными причинами, такими как необходимость обеспечения информационной безопасности и приватности пользователей сети. Доступные наборы трасс, например, в базе организации CAIDA являются «анонимизированными», т. е. не содержат данных уровня приложения в пакетах. Это позволяет применять к ним статистические подходы и подходы, использующие заголовки 3 и 4 уровней, но исключает применение подходов на основе DPI.
- Одним из наиболее активно развивающихся на данный момент направлений является применение различных алгоритмов машинного обучения, графового и статистического анализа, по причине их применимости, в том числе к шифрованному трафику (в отличие от DPI-подходов), доля которого быстро растёт. Данное направление, однако, также не избавлено от недостатков. В частности, точность алгоритмов может снижаться, если в анализируемом потоке присутствует трафик приложений, неиспользовавшихся в процессе обучения. Другой проблемой является необходимость периодического переобучения при изменении характеристик известных протоколов и появлении новых.
- Другим развивающимся направлением является разработка комбинированных подходов и систем классификации. Одной из причин для развития является попытка преодоления недостатков отдельных подходов (например, невысокая точность или скорость обработки) и использование их преимуществ. В качестве примеров таких подходов можно привести инструменты Libprotoident и PortLoad. Создание систем классификации обусловлено

необходимостью открытой инфраструктуры, в рамках которой можно реализовать любой новый подход, что позволит сравнить его с другими на одних и тех же собственных данных, не раскрывая их сторонним лицам. Примерами таких систем могут служить NeTraMark, TIE и система из работы [65].

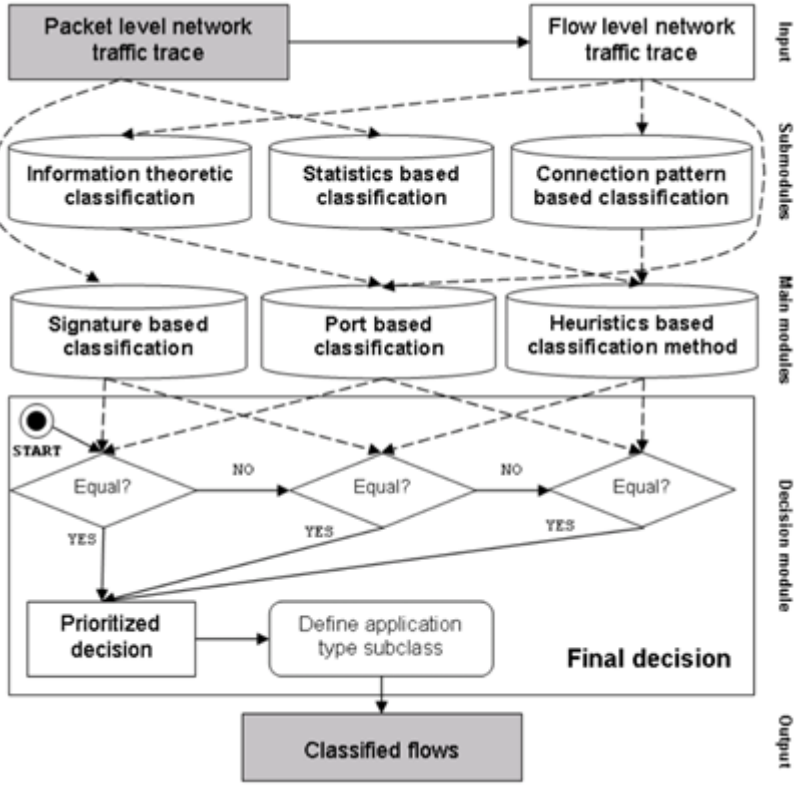


Рис 7. Схема взаимодействия разных компонент классификации трафика
Figure 7. Scheme of interaction between different components of traffic classification

Список литературы

[1]. Cisco WAN and Application Optimization Solution Guide. http://www.cisco.com/c/en/us/td/docs/nsite/enterprise/wan/wan_optimization/wan_opt_sg/chap05.html, дата обращения 01.12.2015

[2]. А.И. Гетьман, Е.Ф. Евстропов, Ю.В. Маркин. Анализ сетевого трафика в режиме реального времени: обзор прикладных задач, подходов и решений. Препринт ИСП РАН, 28, 2015 г., стр. 1-52.

[3]. M.Mellia, A. Pescapè, L. Salgarelli. Traffic classification and its applications to modern networks. Elsevier Computer Networks, Dec. 2008

[4]. T. Farah, L. Trajkovic. Anonym: A tool for anonymization of the Internet traffic. In IEEE 2013 International Conference on Cybernetics (CYBCONF), 2013, pp. 261-266.

[5]. V. Carela-Español, T. Bujlow, P. Barlet-Ros. Is Our Ground-Truth for Traffic Classification Reliable? In Proceedings of the 15th International Conference on Passive and Active Measurement - Vol. 8362. Springer-Verlag New York Inc., New York, NY, USA, 2014, pp. 98-108.

[6]. F. Gringoli, L. Salgarelli, M. Dusi, N. Cascarano, F. Risso, and K. C. Claffy. GT: picking up the truth from the ground for internet traffic //SIGCOMM Computer Communication Review, Volume 39, Issue 5, October 2009, pp. 12-18.

[7]. J. Erman, M. Arlitt, and A. Mahanti. TrafficClassificaton Using Clustering Algorithms. In ACM SIGCOMM MineNet Workshop, September 2006.

[8]. N. Williams, S. Zander, and G. Armitage. Apreliminary performance comparison of five machinelearning algorithms for practical ip traffic flowclassification. In ACM SIGCOMM CCR, Vol. 36, No. 5, pp.7-15, October 2006.

[9]. A. Dainotti, A. Pescapé, C. Sansone. Early classification of network traffic through multi-classification. In Proceedings of the Third international conference on Traffic monitoring and analysis (TMA'11), 2011. Springer-Verlag, Berlin, Heidelberg, pp. 122-135.

[10]. Cascarano N, Ciminiera L, Risso F. Optimizing deep packet inspection for high-speed traffic analysis. Network System Manager. 2011 19(1), pp. 7–31.

[11]. S. Kumar and P. Crowley. Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection. In Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '06), 2006, New York, USA, pp. 339-350.

[12]. D. Ficara, S. Giordano, G. Procissi, F.Vitucci, G.Antichi, A. Di Pietro. An Improved DFA for Fast Regular Expression Matching. SIGCOMM Comput. Commun. Rev. 38, 5 (September 2008), pp. 29-40.

[13]. F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection. In Proceedings of the ACM/IEEE symposium on Architecture for networking and communications systems (ANCS '06). 2006, New York, USA, pp. 93-102.

[14]. S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese. Curing Regular Expressions Matching Algorithms From Insomnia. In Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems (ANCS '07). 2007,New York, USA, pp. 155-164

[15]. R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the Big Bang: Fast and Scalable Deep Packet Inspection with Extended Finite Automata. In Proceedings of the ACM SIGCOMM conference on Data communication (SIGCOMM '08). 2008, New York, USA, pp. 207-218.

[16]. Cao Z., Cao S., Xiong G., Guo L.Progress in Study of Encrypted Traffic Classification. In Proceedings of International standard conference on trustworthy computing and services, 2012, Beijing, China, pp. 78-86

[17]. M. Sokolova, N. Japkowicz, S. Szpakowicz. Beyond accuracy, f-score and ROC: a family of discriminant measures for performance evaluation //In Proceedings of the 19th Australian joint conference on Artificial Intelligence: advances in Artificial Intelligence (AI'06), Berlin, Heidelberg, 2006, pp. 1015-1021.

[18]. S. Valenti, D. Rossi, A. Dainotti, A. Pescapè, A. Finamore, M. Mellia. Reviewing traffic classification. In DataTraffic Monitoring and Analysis, Springer-Verlag, Berlin, Heidelberg, 2013, pp. 123-147.

[19]. D. Maurizio. Observing routing asymmetry in Internet traffic. https://www.caida.org/research/traffic-analysis/asymmetry, дата обращения 01.12.2015
[20]. K. Fukuda. Difficulties of identifying application type in backbone traffic, 2010 International Conference on Network and Service Management, Niagara Falls, ON, 2010, pp. 358-361
[21]. H. Balakrishnan and V. Padmanabhan. How network asymmetry affects TCP. IEEE Communications Magazine,Vol. 39, pp. 60 -67, April 2001.
[22]. Applying Network Policy Control to Asymmetric Traffic: Considerations and Solutions. https://www.sandvine.com/downloads/general/whitepapers/applying-network-policy-control-to-asymmetric-traffic.pdf, дата обращения 01.12.2015
[23]. CAIDAFlowTypes. https://www.caida.org/research/traffic-analysis/flowtypes/, дата обращения 01.12.2015.
[24]. N. Borisov, D.J. Brumley, H.J. Wang, J. Dunagan, P. Joshi, C. Guo. A Generic Application-Level Protocol Analyzer and Its Language. In Proceedings of 14th Annual Network and Distributed System Security Symposium, 2007.
[25]. CiscoNBAR. http://www.cisco.com/c/en/us/products/ios-nx-os-software/network-based-application-recognition-nbar/index.html, дата обращения 01.12.2015.
[26]. RFC 2616. Hypertext Transfer Protocol -- HTTP/1.1. https://www.ietf.org/rfc/rfc2616.txt, дата обращения 01.12.2015.
[27]. RFC 7540. Hypertext Transfer Protocol Version 2 (HTTP/2). https://tools.ietf.org/html/rfc7540, дата обращения 01.12.2015.
[28]. Administering Cisco QoS in IP Networks. Including CallManager 3.0, QoS, and uOne. 1st Edition, Syngress 2001, eBook ISBN: 9780080481890, pp. 561
[29]. L. Deri, M. Martinelli, T. Bujlow, and A. Cardigliano, "ndpi: Opensource high-speed deep packet inspection," in Wireless Communications and Mobile Computing Conference (IWCMC), 2014 International. IEEE, 2014, pp. 617–622.
[30]. Service Name and Transport Protocol Port Number Registry. http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml, дата обращения 01.12.2015
[31]. P. Haffner, S. Sen, O. Spatscheck, D. Wang. ACAS: automated construction of application signatures // In Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data (MineNet '05), ACM, New York, NY, USA, 2005, pp. 197-202.
[32]. Y. Wang, Y. Xiang, W. Zhou, S. Yu. Generating regular expression signatures for network traffic classification in trusted network management, Journal of Network and Computer Applications. Volume 35, Issue 3, May 2012, pp. 992-1000
[33]. G. Szabó, Z.Turányi, L. Toka, S. Molnár, A. Santos. 2011. Automatic protocol signature generation framework for deep packet inspection // In Proceedings of the 5th International ICST Conference on Performance Evaluation Methodologies and Tools, Brussels, Belgium, Belgium, 2011, pp. 291-299.
[34]. Перспективный мониторинг. http://amonitoring.ru/service/snort/, дата обращения 01.12.2015
[35]. G. Bossert, F. Guihéry, G. Hiet. Towards automated protocol reverse engineering using semantic information. In Proceedings of the 9th ACM symposium on Information, computer and communications security (ASIA CCS '14). ACM, New York, NY, USA, 2014, pp. 51-62.
[36]. Гетьман А.И., Маркин Ю.В., Обыденков Д.О., Падарян В.А., Тихонов А.Ю. Подходы к представлению результатов анализа сетевого трафика. Труды ИСП РАН, том 28, вып. 6, 2016, стр. 103-110. DOI: 10.15514/ISPRAS-2016-28(6)-7

[37]. O. Mula-Valls. A practical retraining mechanism for network traffic classification in operational environments // Master Thesis Universitat Poliecnica de Catalunya, 2011.

[38]. R. Wang, L. Shi, B. Jennings. Ensemble Classifier for Traffic in Presence of Changing Distributions. In Proceedings of the Symposium on Computers and Communications (ISCC 2013), Split, Croatia, 7-10 July, 2013, pp. 629-635

[39]. J. Zhang, C. Chen, Y. Xiang, .W. Zhou. Robust network traffic identification with unknown applications. In Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security (ASIA CCS '13), 2013, ACM, New York, NY, USA, pp. 405-414.

[40]. R. Wang. Advances in Machine-Learning-Based Traffic Classifiers. https://labs.ripe.net/Members/rwang/advances-in-machine-learning-based-traffic-classifiers, дата обращения 01.12.2015

[41]. A. White, S. Krishnan, M. Bailey, F. Monrose, P. Porras. Clear and Present Data: Opaque Traffic and its Security Implications for the Future. NDSS, 2013.

[42]. J. Olivain, J. Goubault-Larrecq. Detecting subverted cryptographicprotocols by entropy checking. Technical report, Laboratoire Spcificationet Verification, June 2006.

[43]. L.Bernaille, R. Teixeira. Early recognition of encrypted applications. In Proceedings of the 8th international conference on Passive and active network measurement (PAM'07), 2007, Springer-Verlag, Berlin, Heidelberg, 165-175.

[44]. Global Internet Phenomena Spotlight: Encrypted Internet Traffic. https://www.sandvine.com/downloads/general/global-internet-phenomena/2015/encrypted-internet-traffic.pdf, дата обращения 01.12.2015

[45]. IP Fragmentation Attacks on Checkpoint Firewalls. https://www.giac.org/paper/gsec/589/ip-fragmentation-attacks-checkpoint-firewalls/101350, дата обращения 01.12.2015

[46]. M. Handley, V. Paxson, C. Kreibich. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In Proceedings of the 10th conference on USENIX Security Symposium, vol. 10. USENIX Association, Berkeley, CA, USA, 2001, pp. 9-25.

[47]. M. Baldi, A. Baldini, N. Cascarano, F. Risso. Service-based traffic classification: Principles and validation. In Proceedings of the IEEE Sarnoff Symposium (SARNOFF'09), 2009. IEEE Press, Piscataway, NJ, pp. 115–120.

[48]. W. Moore, K. Papagiannaki. Toward the AccurateIdentification of Network Applications. International Workshop on Passive and Active Network Measurement (PAM 2005), 2005, Boston MA, USA, vol. 3431, pp. 41-54.

[49]. T. Karagiannis, A. Broido, M. Faloutsos, Kc. Claffy. Transport layer identification of P2P traffic. In Proceedings of 4th ACM SIGCOMM conference on Internet measurement, 2004, pp. 121 – 134.

[50]. QosmosixEngine. http://www.qosmos.com/products/deep-packet-inspection-engine/, дата обращения 01.12.2015

[51]. Ipoque PACE. https://www.ipoque.com/products/pace, дата обращения 01.12.2015

[52]. Windriver Content Inspection Engine. http://www.windriver.com/products/product-overviews/PO_Wind-River-Content-Inspection-Engine.pdf, дата обращения 01.12.2015

[53]. Procera PacketLogic Content Intelligence. https://www.proceranetworks.com/content-intelligence.html, дата обращения 01.12.2015

[54]. DPI-SSL. https://www.sonicwall.com/ssl-decryption-and-inspection/, дата обращения 01.12.2015

[55]. G. Aceto, A. Dainotti, W. de Donato, A. Pescap. PortLoad: Taking the Best of Two Worlds in Traffic Classification," in IEEE INFOCOM 2010 – WIP Track, 2010.
[56]. L7-filter. http://l7-filter.sourceforge.net/, дата обращения 01.12.2015.
[57]. S. Alcock, R. Nelson, Libprotoident: Traffic Classification Using Lightweight Packet Inspection, Technical report, University of Waikato, 2013. http://www.wand.net.nz/publications/lpireport, дата обращения 01.12.2015
[58]. Wireshark. https://www.wireshark.org/, дата обращения 01.12.2015.
[59]. T.Karagiannis, K.Papagiannaki, M. Faloutsos. BLINC: multilevel traffic classification in the dark. In Proceedings of the SIGCOMM '05. 2005, ACM, New York, NY, USA, pp.229-240.
[60]. M. Iliofotou, H. Kim, M. Faloutsos, M.Mitzenmacher, P. Pappu, G. Varghese. Graph-based P2P traffic classification at the internet backbone. In Proceedings of the INFOCOM'09. 2009, IEEE Press, Piscataway, NJ, USA, pp. 37-42.
[61]. M.Iliofotou, M. Faloutsos, M.Mitzenmacher. Exploiting dynamicity in graph-based traffic analysis: techniques and applications. In Proceedings of the CoNEXT '09. 2009, ACM, New York, NY, USA, pp. 241-252.
[62]. S. Lee, H. Kim, D. Barman, S. Lee, C. Kim, T. Kwon, Y. Choi. NeTraMark: a network traffic classification benchmark. SIGCOMM Comput. Commun. Rev. 41, 1 (January 2011), pp. 22-30
[63]. A. Dainotti, W. Donato, A.Pescapé. TIE: A Community-Oriented Traffic Classification Platform. In Proceedings of the First International Workshop on Traffic Monitoring and Analysis (TMA '09), 2009, Springer-Verlag, Berlin, Heidelberg, pp. 64-74.
[64]. W. Donato, A. Pescape, A. Dainotti. Traffic identification engine: an open platform for traffic classification.In IEEE Network, vol. 28, no. 2, pp. 56-64, March-April 2014.
[65]. G. Szabo, I. Szabo, D. Orincsay. Accurate Traffic Classification. IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks, Espoo, Finland, 2007, pp. 1-8.

A survey of problems and solution methods in network traffic classification

A. I. Get'man <thorin@ispras.ru>
Yu. V. Markin <ustas@ispras.ru>
D. O. Obidenkov <obydenkov@ispras.ru>
E. F. Evstropov < john0606@yandex.ru>
Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Annotation. The paper discusses the problem of network traffic classification: the characteristics that are used to solve it, existing approaches and their limitations. Applied tasks that require classification are listed, as well as additional requirements that arise from the main problem. Properties of network traffic that root in communication medium specifics are analyzed as well as the technology being used where they influence the classification process. Relevant directions in current approaches to analysis and the reasons for their development are discussed.

Keywords. Network traffic analysis, network security, network traffic classification, machine learning, DPI

DOI: 10.15514/ISPRAS-2017-29(3)-8

For citation: Ge'tman A.I., Markin Yu.V, Evstropov E.F. Obydenkov D.O. A survey of problems and solution methods in network traffic classification classification. *Trudy ISP RAN/Proc. ISP* RAS, vol. 29, issue 3, 2017, pp. 117-150 (in Russian). 10.15514/ISPRAS-2017-29(3)-8

References

[1]. Cisco WAN and Application Optimization Solution Guide. http://www.cisco.com/c/en/us/td/docs/nsite/enterprise/wan/wan_optimization/wan_opt_sg/chap05.html, accessed 01.12.2015

[2]. A.I Get'man, E.F Evstropov, Yu. V. Markin, Wirespeed network traffic analysis: survey of applied problems, approaches and solutions. Preprint ISP RAS, 28, 2015, pp. 1-52 (in Russian)

[3]. M.Mellia, A. Pescapè, L. Salgarelli. Traffic classification and its applications to modern networks. Elsevier Computer Networks, Dec. 2008

[4]. T. Farah, L. Trajkovic. Anonym: A tool for anonymization of the Internet traffic. In IEEE 2013 International Conference on Cybernetics (CYBCONF), 2013, pp. 261-266.

[5]. V. Carela-Español, T. Bujlow, P. Barlet-Ros. Is Our Ground-Truth for Traffic Classification Reliable? In Proceedings of the 15th International Conference on Passive and Active Measurement - Vol. 8362. Springer-Verlag New York Inc., New York, NY, USA, 2014, pp. 98-108.

[6]. F. Gringoli, L. Salgarelli, M. Dusi, N. Cascarano, F. Risso, and K. C. Claffy. GT: picking up the truth from the ground for internet traffic //SIGCOMM Computer Communication Review, Volume 39, Issue 5, October 2009, pp. 12-18.

[7]. J. Erman, M. Arlitt, and A. Mahanti. TrafficClassificaton Using Clustering Algorithms. In ACM SIGCOMM MineNet Workshop, September 2006.

[8]. N. Williams, S. Zander, and G. Armitage. Apreliminary performance comparison of five machinelearning algorithms for practical ip traffic flowclassification. In ACM SIGCOMM CCR, Vol. 36, No. 5, pp.7-15, October 2006.

[9]. A. Dainotti, A. Pescapé, C. Sansone. Early classification of network traffic through multi-classification. In Proceedings of the Third international conference on Traffic monitoring and analysis (TMA'11), 2011. Springer-Verlag, Berlin, Heidelberg, pp. 122-135.

[10]. Cascarano N, Ciminiera L, Risso F. Optimizing deep packet inspection for high-speed traffic analysis. Network System Manager. 2011 19(1), pp. 7–31.

[11]. S. Kumar and P. Crowley. Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection. In Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '06), 2006, New York, USA, pp. 339-350.

[12]. D. Ficara, S. Giordano, G. Procissi, F.Vitucci, G.Antichi, A. Di Pietro. An Improved DFA for Fast Regular Expression Matching. SIGCOMM Comput. Commun. Rev. 38, 5 (September 2008), pp. 29-40.

[13]. F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection. In Proceedings of the

ACM/IEEE symposium on Architecture for networking and communications systems (ANCS '06). 2006, New York, USA, pp. 93-102.

[14]. S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese. Curing Regular Expressions Matching Algorithms From Insomnia. In Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems (ANCS '07). 2007,New York, USA, pp. 155-164

[15]. R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the Big Bang: Fast and Scalable Deep Packet Inspection with Extended Finite Automata. In Proceedings of the ACM SIGCOMM conference on Data communication (SIGCOMM '08). 2008, New York, USA, pp. 207-218.

[16]. Cao Z., Cao S., Xiong G., Guo L.Progress in Study of Encrypted Traffic Classification. In Proceedings of International standard conference on trustworthy computing and services, 2012, Beijing, China, pp. 78-86

[17]. M. Sokolova, N. Japkowicz, S. Szpakowicz. Beyond accuracy, f-score and ROC: a family of discriminant measures for performance evaluation //In Proceedings of the 19th Australian joint conference on Artificial Intelligence: advances in Artificial Intelligence (AI'06), Berlin, Heidelberg, 2006, pp. 1015-1021.

[18]. S. Valenti, D. Rossi, A. Dainotti, A. Pescapè, A. Finamore, M. Mellia. Reviewing traffic classification. In DataTraffic Monitoring and Analysis, Springer-Verlag, Berlin, Heidelberg, 2013, pp. 123-147.

[19]. D. Maurizio. Observing routing asymmetry in Internet traffic. https://www.caida.org/research/traffic-analysis/asymmetry, accessed 01.12.2015

[20]. K. Fukuda. Difficulties of identifying application type in backbone traffic, 2010 International Conference on Network and Service Management, Niagara Falls, ON, 2010, pp. 358-361

[21]. H. Balakrishnan and V. Padmanabhan. How network asymmetry affects TCP // IEEE Communications Magazine,Vol. 39, pp. 60 -67, April 2001.

[22]. Applying Network Policy Control to Asymmetric Traffic: Considerations and Solutions. https://www.sandvine.com/downloads/general/whitepapers/applying-network-policy-control-to-asymmetric-traffic.pdf, accessed 01.12.2015

[23]. CAIDAFlowTypes. https://www.caida.org/research/traffic-analysis/flowtypes/, accessed 01.12.2015.

[24]. N. Borisov, D.J. Brumley, H.J. Wang, J. Dunagan, P. Joshi, C. Guo. A Generic Application-Level Protocol Analyzer and Its Language. In Proceedings of 14th Annual Network and Distributed System Security Symposium, 2007.

[25]. CiscoNBAR. http://www.cisco.com/c/en/us/products/ios-nx-os-software/network-based-application-recognition-nbar/index.html, accessed 01.12.2015.

[26]. RFC 2616. Hypertext Transfer Protocol -- HTTP/1.1. https://www.ietf.org/rfc/rfc2616.txt, accessed 01.12.2015.

[27]. RFC 7540. Hypertext Transfer Protocol Version 2 (HTTP/2). https://tools.ietf.org/html/rfc7540, accessed 01.12.2015.

[28]. Administering Cisco QoS in IP Networks. Including CallManager 3.0, QoS, and uOne. 1st Edition, Syngress 2001, eBook ISBN: 9780080481890, pp. 561

[29]. L. Deri, M. Martinelli, T. Bujlow, and A. Cardigliano, "ndpi: Opensource high-speed deep packet inspection," in Wireless Communications and Mobile Computing Conference (IWCMC), 2014 International. IEEE, 2014, pp. 617–622.

[30]. Service Name and Transport Protocol Port Number Registry. http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml, accessed 01.12.2015

[31]. P. Haffner, S. Sen, O. Spatscheck, D. Wang. ACAS: automated construction of application signatures. In Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data (MineNet '05), ACM, New York, NY, USA, 2005, pp. 197-202.

[32]. Y. Wang, Y. Xiang, W. Zhou, S. Yu. Generating regular expression signatures for network traffic classification in trusted network management, Journal of Network and Computer Applications. Volume 35, Issue 3, May 2012, pp. 992-1000

[33]. G. Szabó, Z.Turányi, L. Toka, S. Molnár, A. Santos. 2011. Automatic protocol signature generation framework for deep packet inspection. In Proceedings of the 5th International ICST Conference on Performance Evaluation Methodologies and Tools, Brussels, Belgium, Belgium, 2011, pp. 291-299.

[34]. Perspective monitoring. http://amonitoring.ru/service/snort/, accessed 01.12.2015.

[35]. G. Bossert, F. Guihéry, G. Hiet. Towards automated protocol reverse engineering using semantic information. In Proceedings of the 9th ACM symposium on Information, computer and communications security (ASIA CCS '14). ACM, New York, NY, USA, 2014, pp. 51-62.

[36]. Get'man A. I., Markin Yu. V., Obydenkov D. O., Padaryan V. A., Tikhonov A. Yu. Methods of presenting the results of network traffic analysis. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 6, 2016, pp. 103-110 (in Russian). DOI: 10.15514/ISPRAS-2016-28(6)-7

[37]. O. Mula-Valls. A practical retraining mechanism for network traffic classification in operational environments // Master Thesis Universitat Poliecnica de Catalunya, 2011.

[38]. R. Wang, L. Shi, B. Jennings. Ensemble Classifier for Traffic in Presence of Changing Distributions // In Proceedings of the Symposium on Computers and Communications (ISCC 2013), Split, Croatia, 7-10 July, 2013, pp. 629-635

[39]. J. Zhang, C. Chen, Y. Xiang, .W. Zhou. Robust network traffic identification with unknown applications. In Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security (ASIA CCS '13), 2013, ACM, New York, NY, USA, pp. 405-414.

[40]. R. Wang. Advances in Machine-Learning-Based Traffic Classifiers. https://labs.ripe.net/Members/rwang/advances-in-machine-learning-based-traffic-classifiers, accessed 01.12.2015

[41]. A. White, S. Krishnan, M. Bailey, F. Monrose, P. Porras. Clear and Present Data: Opaque Traffic and its Security Implications for the Future. NDSS, 2013.

[42]. J. Olivain, J. Goubault-Larrecq. Detecting subverted cryptographicprotocols by entropy checking. Technical report, Laboratoire Spcificationet Verification, June 2006.

[43]. L.Bernaille, R. Teixeira. Early recognition of encrypted applications. In Proceedings of the 8th international conference on Passive and active network measurement (PAM'07), 2007, Springer-Verlag, Berlin, Heidelberg, 165-175.

[44]. Global Internet Phenomena Spotlight: Encrypted Internet Traffic. https://www.sandvine.com/downloads/general/global-internet-phenomena/2015/encrypted-internet-traffic.pdf, accessed 01.12.2015

[45]. IP Fragmentation Attacks on Checkpoint Firewalls. https://www.giac.org/paper/gsec/589/ip-fragmentation-attacks-checkpoint-firewalls/101350, accessed 01.12.2015

[46]. M. Handley, V. Paxson, C. Kreibich. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In Proceedings of the 10th conference on USENIX Security Symposium, Vol. 10. USENIX Association, Berkeley, CA, USA, 2001, pp. 9-25.

[47]. M. Baldi, A. Baldini, N. Cascarano, F. Risso. Service-based traffic classification: Principles and validation. In Proceedings of the IEEE Sarnoff Symposium (SARNOFF'09), 2009. IEEE Press, Piscataway, NJ, pp. 115–120.

[48]. W. Moore, K. Papagiannaki. Toward the AccurateIdentification of Network Applications. InternationalWorkshop on Passive and Active Network Measurement (PAM 2005), 2005, Boston MA, USA, vol. 3431, pp. 41-54.

[49]. T. Karagiannis, A. Broido, M. Faloutsos, Kc. Claffy. Transport layer identification of P2P traffic. In Proceedings of 4th ACM SIGCOMM conference on Internet measurement, 2004, pp. 121 – 134.

[50]. QosmosixEngine. http://www.qosmos.com/products/deep-packet-inspection-engine/, accessed 01.12.2015

[51]. Ipoque PACE. https://www.ipoque.com/products/pace, accessed 01.12.2015

[52]. Windriver Content Inspection Engine. http://www.windriver.com/products/product-overviews/PO_Wind-River-Content-Inspection-Engine.pdf, accessed 01.12.2015

[53]. Procera PacketLogic Content Intelligence. https://www.proceranetworks.com/content-intelligence.html, accessed 01.12.2015

[54]. DPI-SSL. https://www.sonicwall.com/ssl-decryption-and-inspection/, accessed 01.12.2015

[55]. G. Aceto, A. Dainotti, W. de Donato, A. Pescap. PortLoad: Taking the Best of Two Worlds in Traffic Classification," in IEEE INFOCOM 2010 – WIP Track, 2010.

[56]. L7-filter. http://l7-filter.sourceforge.net/, accessed 01.12.2015.

[57]. S. Alcock, R. Nelson, Libprotoident: Traffic Classification Using Lightweight Packet Inspection, Technical report, University of Waikato, 2013. http://www.wand.net.nz/publications/lpireport, accessed 01.12.2015

[58]. Wireshark. https://www.wireshark.org/, accessed 01.12.2015.

[59]. T.Karagiannis, K.Papagiannaki, M. Faloutsos. BLINC: multilevel traffic classification in the dark. In Proceedings of the SIGCOMM '05. 2005, ACM, New York, NY, USA, pp.229-240.

[60]. M. Iliofotou, H. Kim, M. Faloutsos, M.Mitzenmacher, P. Pappu, G. Varghese. Graph-based P2P traffic classification at the internet backbone. In Proceedings of the INFOCOM'09. 2009, IEEE Press, Piscataway, NJ, USA, pp. 37-42.

[61]. M.Iliofotou, M. Faloutsos, M.Mitzenmacher. Exploiting dynamicity in graph-based traffic analysis: techniques and applications. In Proceedings of the CoNEXT '09. 2009, ACM, New York, NY, USA, pp. 241-252.

[62]. S. Lee, H. Kim, D. Barman, S. Lee, C. Kim, T. Kwon, Y. Choi. NeTraMark: a network traffic classification benchmark. SIGCOMM Comput. Commun. Rev. 41, 1 (January 2011), pp. 22-30

[63]. A. Dainotti, W. Donato, A.Pescapé. TIE: A Community-Oriented Traffic Classification Platform. In Proceedings of the First International Workshop on Traffic Monitoring and Analysis (TMA '09), 2009, Springer-Verlag, Berlin, Heidelberg, pp. 64-74.

[64]. W. Donato, A. Pescape, A. Dainotti. Traffic identification engine: an open platform for traffic classification. In IEEE Network, vol. 28, no. 2, pp. 56-64, March-April 2014.

[65]. G. Szabo, I. Szabo, D. Orincsay. Accurate Traffic Classification. IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks, Espoo, Finland, 2007, pp. 1-8.

Комбинация методов статической верификации композиции требований

В.О. Мордань <mordan@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, 25

Аннотация. Статическая верификация программного обеспечения доказывает выполнение требований в программах, однако для этого необходимо большое количество вычислительных ресурсов, кроме того, соответствующая задача не всегда может быть решена. На данный момент не существует универсальный метод статической верификации, который мог бы эффективно проверять произвольные программы, поэтому на практике необходимо выбирать более подходящий метод и настраивать его под конкретную задачу. В данной статье предлагается комбинировать различные методы для повышения производительности и улучшения результата верификации, что можно рассматривать как первый шаг в создании универсального метода статической верификации. Предложенные методы были реализованы для комбинации активно развивающихся в настоящее время методов верификации композиции требований. Апробация реализованных методов на модулях ядра операционной системы Linux продемонстрировала их преимущества относительно отдельного применения методов верификации композиции требований.

Ключевые слова: статическая верификация программного обеспечения; уточнение абстракции по контрпримерам; задача достижимости; композиция требований.

DOI: 10.15514/ISPRAS-2017-29(3)-9

Для цитирования: Мордань В.О. Комбинация методов статической верификации композиции требований. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 151-170. DOI: 10.15514/ISPRAS-2017-29(3)-9

1. Введение

Статическая верификация программного обеспечения предназначена для проверки исходного кода программы без его выполнения, при этом рассматриваются все возможные пути выполнения программы. Главное достоинство статической верификации заключается в том, что она нацелена на доказательство отсутствия ошибок, являющихся нарушениями проверяемого требования. Однако в общем случае статическая верификация является

неразрешимой задачей, поскольку она сводится к проблеме останова [1]. Кроме того, на практике статическая верификация требует большого количества вычислительных ресурсов (таких, как процессорное время и оперативная память), что существенно затрудняет ее применение на практике. Поэтому статическую верификацию имеет смысл использовать только для проверки программного обеспечения с высокими требованиями надежности.

В качестве примера подобного программного обеспечения можно рассмотреть ядро операционной системы Linux, которое состоит из более 20 миллионов строк кода на языке программирования C, новая версия которого выходит каждые 2-3 месяца [2]. При этом существуют сотни требований на корректное использование интерфейсов сердцевины ядра [3], которым должен удовлетворять код каждого модуля ядра Linux, число которых составляет порядка 5 тысяч. Если в модуле нарушается одно из подобных требований, то это может привести к отказу всей операционной системы. Статическая верификация способна доказывать, что в модулях нет нарушений проверяемых требований. При проведении статической верификации необходимо выполнить проверку каждого подобного требования в каждом модуле каждой новой версии ядра, поэтому задачи эффективной проверки выполнения композиции требований и нахождения большего числа нарушений требований являются важными и актуальными.

Для решения данных задач были предложены различные методы статической верификации композиции требований [4-7], каждый из которых имеет как преимущества, так и недостатки, а также свою область применимости. Поэтому на практике необходимо правильно выбрать наиболее подходящий метод и множество параметров для него с учетом решаемых задач, что позволит как снизить потребляемые ресурсы, так и успешно решить большее количество задач. В то же самое время известны примеры, в которых комбинация различных методов позволяла улучшить те или иные характеристики верификации [8]. В данной статье будут рассмотрены основные идеи комбинации методов на примере верификации композиции требований для повышения двух основных характеристик – производительности верификации и числа успешно решенных задач.

Следующий раздел содержит краткое описание существующих методов статической верификации композиции требований.

2. Методы верификации композиции требований

С помощью статической верификации проверяются требования к программам, которые сводятся к проблеме достижимости. Процесс верификации состоит из двух этапов – подготовки и решения задачи достижимости. Задача достижимости формулируется следующим образом – требуется доказать недостижимость некоторых точек программы, которые соответствуют нарушениям проверок выполнения требований, из точки входа в программу. Поскольку в общем случае задача достижимости является неразрешимой, то

на практике она решается с ограниченными ресурсами. Результатом верификации является либо доказательство отсутствия нарушений требования (вердикт *safe*), либо пример нарушения требования в виде трассы ошибки (вердикт *unsafe*), либо невозможность решить задачу по тем или иным причинам (вердикт *unknown*), например, из-за исчерпания выделенных на решение задачи вычислительных ресурсов.

Для постановки задач достижимости существует два подхода – это инструментирование исходного кода программы [9] (то есть добавление проверок выполнения требования непосредственно в код) и наблюдательные автоматы [10] (то есть передача модели требования верификатору в его внутреннем представлении независимо от кода). Метод инструментирования предоставляет более широкие возможности по формализации требований, поскольку он использует возможности языка программирования исходной программы. Однако при этом усложняется исходный код программы за счет добавленных проверок, что особенно негативно сказывается при проверке композиции требований.

Для решения задач достижимости наиболее масштабируемым на данный момент является подход уточнения абстракции по контрпримерам (counterexample guided abstraction refinement, или CEGAR [11]). Инструменты, реализующие данный подход, являются одними из лидеров в международных мероприятиях по верификации Competition on Software Verification [12]. Помимо этого, подход CEGAR успешно используется и на практике: в инструментах BLAST [13] и CPAchecker [14] для верификации модулей ядра операционной системы Linux (в рамках системы верификации Linux Driver Verification Tools, или LDV Tools [15]) и в инструменте SLAM [16] для верификации драйверов операционной системы Windows (в рамках проекта Static Driver Verification). Именно поэтому в данной статье в качестве базового подхода решения задач достижимости будет рассматриваться CEGAR.

Для верификации композиции требований необходимо разрешить две проблемы – это чрезмерный рост числа состояний, вызванный тем, что хотя бы одно из проверяемых требований не может быть верифицировано с выделенными ограничениями на ресурсы, и остановка верификации после нахождения первого нарушения требования.

2.1. Метод обнаружения всех однотипных нарушений

Для решения проблемы остановки верификации после нахождения первого нарушения требования был предложен метод *обнаружения всех однотипных нарушений* (ОВН) [4], который продолжает верификацию после нахождения нарушения требования и производит анализ результата, то есть найденных трасс ошибок. Основная задача анализа результата состоит в исключении так называемых «наведенных» трасс ошибок (то есть соответствующих одному и тому же нарушению требования). Например, имеется утечка памяти в функции, которая может вызываться произвольное количество раз, и для

каждого случая выдается отдельная трасса ошибки. Для того чтобы не пропустить новые нарушения требования, предлагается автоматически фильтровать трассы ошибок по относительно простым критериям (например, на полное совпадение). Далее трассы анализирует человек, помечая и исправляя причину нарушения требования. При этом все трассы ошибок, которые содержат помеченный человеком фрагмент, исключаются из дальнейшего анализа, то есть человек рассматривает ровно столько трасс, сколько уникальных нарушений требований найдено. По сравнению с базовым методом статической верификации метод ОВН способен находить примерно в 1.5 раза больше нарушений требований, при этом время проведения ручного анализа результата возрастает пропорционально числу нарушений требований (то есть также примерно в 1.5 раза) [6, 17].

2.2. Метод условной многоаспектной верификации

Метод *условной многоаспектной верификации* (УМАВ) был предложен для верификации композиции требований [5]. Для предотвращения чрезмерного роста числа состояний время проверки каждого требования ограничивается при проверке композиции требований, то есть выделенные на решение задачи ресурсы распределяются равномерным образом между проверкой каждого требования. Для этого используется предположение о том, что единовременно проверяется только одно требование, которое достаточно точно может быть определено из найденного контрпримера в подходе CEGAR. После нахождения нарушения некоторого требования верификация продолжается, но без нарушенного требования, что позволит получить результат для остальных требований без необходимости проведения ручной фильтрации. За счет переиспользования знаний о верификации между проверкой тех требований, которые не ведут к чрезмерному росту числа состояний, данный метод позволяет повысить производительность верификации композиции требований в несколько раз при незначительных потерях результата (порядка 1-2%) [5, 17], которые вызваны тем, что используемое в методе предположение не всегда работает.

2.3. Метод автоматных спецификаций

Предложенный метод условной многоаспектной верификации обладает следующим недостатком – инструментирование исходного кода усложняет задачи достижимости, поскольку дополнительные проверки добавляются в исходный код программы, и чем больше проверяемых требований, тем больше будут усложняться задачи. Кроме того, при инструментировании проверки требований находятся в исходном коде и нет возможности их удалить во время верификации, например, если было найдено нарушение требования и больше его не нужно проверять. Для решения данной проблемы был предложен метод *автоматных спецификаций* (АС) [7], который формализует

требование независимо от исходного кода во внутреннем представлении верификатора, расширяя наблюдательные автоматы за счет добавления произвольных конструкций языка С во внутреннее представление программы во время верификации, что делает возможности данного метода по представлению требований эквивалентными инструментированию. Метод АС демонстрирует сопоставимые результаты с методом инструментирования [17], при этом больше подходит для верификации композиции требований.

2.4. Метод декомпозиции автоматной спецификации

Поставленную в данной работе цель можно переформулировать следующим образом – дана спецификация, состоящая из набора требований, и необходимо найти ее разбиение на группы требований для совместной верификации. Так, в методе УМАВ все требования помещаются в одну группу, а при последовательной верификации – в каждую группу помещается ровно одно требование. Метод *декомпозиции автоматной спецификации* (ДАС) [7] проверяет отдельно (то есть в разных группах) те требования, которые могут приводить к чрезмерному росту числа состояний, и совместно (в одной группе) все остальные требования. Верификация каждой группы требований производится с помощью метода АС. Все идеи метода УМАВ, которые не относятся к подходу CEGAR, например, продолжение верификации после нахождения нарушения требования, но без нарушенного требования, также используются и в методе ДАС. На практике данный метод требует несколько больше ресурсов на решение задач достижимости, чем метод УМАВ, но при этом демонстрирует меньший процент потерь результата за счет того, что в нем не используются эвристики [7, 17].

2.5. Простейшие комбинации методов

Классификация всех приведенных методов верификации композиции требований приведена в табл. 1. Заметим, что для решения определенных задач можно применять комбинацию методов. Так, например, для проверки выполнения композиции требований с возможностью нахождения нескольких нарушений требований можно использовать методы УМАВ и ОВН совместно [6], так как анализ результата в методе ОВН производится уже после решения задач достижимости методом УМАВ. Комбинация методов УМАВ и ОВН позволяет получить результат метода ОВН с незначительными потерями, при этом требуется практически то же количество ресурсов, что и в методе УМАВ [6, 17].

В следующем разделе будет дана основная идея комбинации методов верификации композиции требований в общем виде.

Табл. 1. Методы верификации композиции требований
Table 1. Verification methods for checking requirements composition

Метод	Постановка задач достижимости	Верификация композиции требований	Остановка после нахождения первого нарушения
Базовый	Инструментирование	Нет	Нет
ОВН	Инструментирование	Нет	Да
УМАВ	Инструментирование	Да	Нет
УМАВ с ОВН	Инструментирование	Да	Да
АС	Автоматы	Нет	Нет
АС с ОВН	Автоматы	Нет	Да
ДАС	Автоматы	Да	Нет
ДАС с ОВН	Автоматы	Да	Да

3. Основная идея комбинации методов

Пусть для верификации дана спецификация $\Omega = \{\omega_1, ..., \omega_n\}$, состоящая из *n* различных требований, и выделено *T* единиц ресурсов (например, процессорных секунд). Под композицией требований будем понимать непустое подмножество всех требований (группу): $\xi \subseteq \Omega, \xi \neq \varnothing$. Формально метод верификации композиции требований представляет собой оператор вида $\psi(\xi, T)$: $\xi \to V$, который для каждого требования возвращает вердикт $v \in V = \{safe, unsafe, unknown\}$. Тогда *комбинацией* методов $\Psi = \{\psi_1(\xi_1, t_1), ..., \psi_k(\xi_k, t_k)\}$ верификации композиции требований Ω называется следующий оператор:

$$C(\Psi, \Omega): \quad \Omega \to V.$$

Иными словами, комбинация методов должна получить вердикт для каждого из проверяемых требований, используя для этого заданные методы верификации. Основная цель комбинации методов заключается в том, чтобы либо получить результат быстрее, либо успешно решить больше задач достижимости.

Комбинация методов может выполняться как параллельно, так и последовательно.

3.1. Параллельная комбинация

Наиболее простой является *параллельная* (или *статическая*) комбинация, схема которой приведена на рис. 1.

Основная идея параллельной комбинации состоит в том, что до начала верификации с помощью оператора *разбиения* создаются группы требований:

$$\{\xi_1,...,\xi_k\},\quad \xi_i \subset \Omega,\quad \xi_i \cap \xi_j = \varnothing,\quad \forall i \neq j,\quad i,j \in \overline{1,k},\quad \cup_{i=1}^{k} \xi_i = \Omega,$$

каждой из которых также назначается один из методов для верификации. Далее методы производят верификацию соответствующих групп (возможно, параллельно), после чего результат их работы объединяется. При этом предполагается, что ресурсы, выделенные на верификацию всех требований, распределяются для каждого метода пропорционально числу верифицируемых требований, то есть на верификацию группы ξ_i будет выделено $t_i = T\dfrac{|\xi_i|}{n}$ единиц ресурсов.

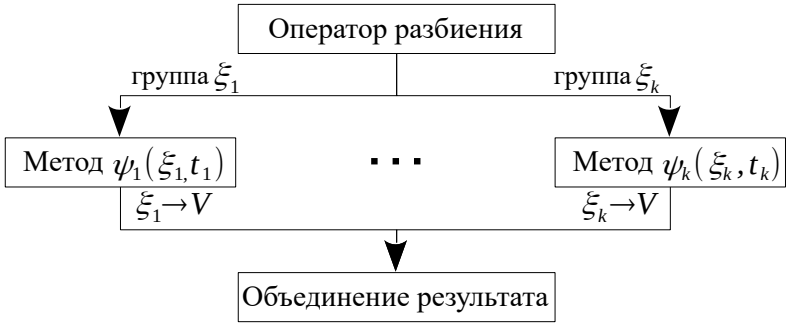


Рис. 1. Параллельная комбинация
Fig 1. Parallel combination

Параллельная комбинация во многом схожа с методами параллельной верификации, которые за счет параллельного решения различных задач достижимости на нескольких компьютерах или ядрах процессора позволяют сократить астрономическое время верификации. Одним из примеров параллельной верификации является проверка выполнения каждого требования на отдельном компьютере с помощью базового метода. Основное отличие параллельной комбинации от обычных методов параллельной верификации состоит в том, что главной целью является снижение требуемых ресурсов и улучшение результата, а не только сокращение астрономического времени верификации.

Для достижения поставленной цели оператор разбиения должен помещать в одну группу требования, совместная верификация которых более эффективна. Например, разные требования к одним и тем же программным интерфейсам

имеет смысл верифицировать вместе, а требование, которое ведет к чрезмерному росту числа состояний значительно чаще остальных, скорее, всего, следует верифицировать отдельно. Для выполнения подобного разбиения может использоваться информация о проверяемом коде.

Стоит заметить, что подобную технику можно реализовать внутри рассмотренного метода ДАС, поскольку он также автоматически разбивает спецификацию на группы требований. Однако комбинация методов имеет ряд преимуществ. Во-первых, решаемая задача достижимости не будет усложняться за счет требований, которые не нужно проверять. Например, в методе УМАВ в код не будут добавлены проверки подобных требований, что сократит накладные расходы на верификацию. Во-вторых, чрезмерный рост числа состояний, вызванный хотя бы одним из требований, потенциально может привести к исчерпанию всей доступной оперативной памяти и невозможности получить результат для остальных требований. Если же данное требование будет верифицироваться отдельно, то результат не будет получен только для данного требования. В-третьих, комбинация методов более универсальна, поскольку может использовать не только метод ДАС, а потенциально те же идеи могут быть применимы и не только для верификации композиции требований. Поэтому параллельная комбинация может как предоставить более точный результат, так и затратить на его получение меньше ресурсов.

Однако параллельная комбинация имеет достаточно ограниченные возможности, поскольку методы верификации не взаимодействуют между собой. Например, один метод не может справиться с решение задачи и тратит все выделенные ресурсы впустую, в то время как она относительно просто решается с помощью другого. Проблема заключается в том, что группы требований получаются статически и не могут меняться во время верификации, что может негативно сказываться на результате. Кроме того, чрезмерное увеличение верифицируемых групп требований будет вести к увеличению числа однотипных действий, что приведет к снижению производительности. Следствием данных проблем является отсутствие уверенности в том, что в общем случае параллельная комбинация методов будет более эффективной или более точной, чем применение каждого из методов в отдельности.

Таким образом, параллельная комбинация простейшим способом объединяет методы верификации с целью повышения производительности и улучшения результата, однако в общем случае не предоставляет гарантий достижения поставленной цели.

3.2. Последовательная комбинация

Последовательная (или *динамическая*) комбинация предполагает итеративное применение методов верификации в некоторой последовательности, при этом каждый последующий метод будет верифицировать требования из числа тех, с

которыми не справились предыдущие методы. Схема последовательной комбинации приведена на рис. 2.

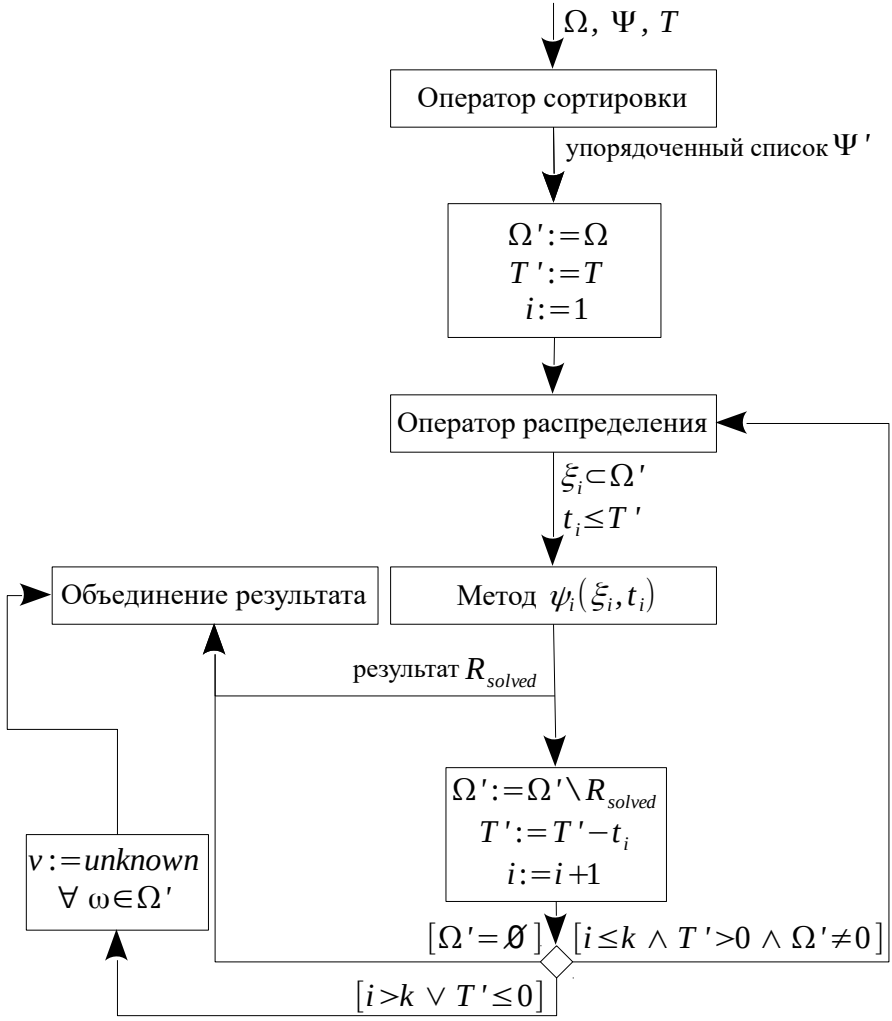


Рис. 2. Последовательная комбинация
Fig 2. Sequential combination

Последовательная комбинация вначале упорядочивает все методы с помощью оператора *сортировки*, а затем по очереди их применяет. Для текущего используемого метода $\psi_i(\xi_i, t_i)$ оператор *распределения* выделяет группу требований $\xi_i \subseteq \Omega'$ и вычислительные ресурсы $t_i \le T'$. Результат работы

каждого метода разбивается на два множества: успешно верифицированные требования $\left(R_{solved} = \left\{\xi_i \to v \mid v \in \left\{safe, unsafe\right\}\right\}\right)$ и остальные. Результат R_{solved} запоминается, и соответствующие требования больше не проверяются (то есть удаляются из множества проверяемых требований Ω'). Данная процедура повторяется до тех пор, пока множество проверяемых требований не пусто $\left(\Omega' \neq \varnothing\right)$, список методов верификации не исчерпан $\left(i \leq k\right)$ и ресурсы не исчерпаны $\left(T' > 0\right)$. Если же после завершения работы комбинации методов не все требования получили вердикт (то есть $\Omega' \neq \varnothing$), то они искусственно получают вердикт *unknown*.

Основное преимущество последовательной комбинации перед параллельной состоит в том, что имеется возможность учитывать результат работы предыдущих методов для формирования групп требований и выделения ресурсов для их верификации (с помощью оператора распределения). Кроме того, можно отметить, что последовательная комбинация является более общим случаем параллельной комбинации. Действительно, если определить оператор распределения таким образом, что он для каждого метода выдает заранее определенный набор групп, среди которых нет пересечений, и равномерным образом распределяет ресурсы, то он будет работать аналогично оператору разбиения параллельной комбинации.

Последовательная комбинация базируется на идее условной проверки моделей [8]: имеется несколько методов верификации, которые последовательно применяются для решения поставленной задачи, если она не была решена одним из методов, то выдается условие, описывающее причину неудачного завершения (например, какое именно требование ведет к исчерпанию выделенной оперативной памяти), которое поможет разбить требования на группы и правильно распределить ресурсы для оставшихся методов. Однако в отличие от условной проверки моделей последовательная комбинация решает разные задачи достижимости, в которых проверяются различные требования. Помимо этого, последовательная комбинация предполагает, что в общем случае методы могут быть реализованы разными инструментами, то есть для каждого метода будет производиться отдельный запуск инструмента верификации. Поэтому последовательную комбинацию можно рассматривать как обобщение метода условной проверки моделей, поскольку она также может быть использована не только для верификации композиции требований.

Последовательная комбинация предоставляет универсальный способ для улучшения выбранной характеристики. Например, если требуется повысить производительность, то имеет смысл сначала использовать более быстрые методы с небольшими ограничениями на ресурсы, что позволит успешно решить большую часть задач. Если же требуется максимально снизить число

потерь, то стоит перепроверять известные ситуации, ведущие к потерям результата в используемых методах. В данном случае преимуществом последовательной комбинации является универсальность, поскольку требуемая конфигурация для достижения поставленной цели будет автоматически подобрана во время верификации для каждой из решаемых задач.

С другой стороны, подобная универсальность может приводить к тому, что в частных случаях последовательная комбинация будет уступать применению отдельных методов верификации. Например, задача может быть успешно решена только методом, который помещен в конец списка. В данном случае результат все равно будет получен, однако будут затрачены ресурсы на попытку решить задачу предыдущими методами.

Таким образом, последовательная комбинация методов предназначена для использования преимуществ нескольких методов и минимизацию их недостатков с целью повышения производительности верификации и улучшения результата.

4. Варианты реализации комбинации методов

Различные варианты комбинации методов верификации композиции требований были реализованы в рамках открытой системы верификации Linux Driver Verification Tools [15] (LDV Tools), которая предназначена для верификации модулей ядра операционной системы Linux относительно требований на корректное использование программных интерфейсов сердцевины ядра. Система LDV Tools готовит задачи достижимости на основе модулей ядра, которые решаются с помощью статического верификатора CPAchecker [14]. Данный верификатор реализует все описанные в разделе 2 методы верификации композиции требований.

Помимо предложенных возможны и другие варианты реализации комбинации методов в других системах верификации.

4.1. Регрессионная условная многоаспектная верификация

Каждый из рассмотренных методов верификации композиции требований может терять эффективность и получаемый результат с ростом числа требований: в методе УМАВ происходит усложнение кода из-за инструментирования, а в методе ДАС имеются накладные расходы на использование нескольких автоматов. В методе УМАВ данная проблема проявляется наиболее заметно, что приводит к относительно большим потерям результата, среди которых встречаются и реальные ошибки. Одним из возможных решений этой проблемы является разбиение композиции требований на группы в рамках параллельной комбинации с использованием одного метода УМАВ.

Для выполнения разбиения вводится функция сложности $f(\omega)$, которая задает некоторый критерий сложности проверки каждого из требований. Например, функция сложности может возвращать среднее число нерешенных задач для каждого требования. Будем считать, что все требования в Ψ упорядочены в соответствии с выбранной функцией сложности. Для оператора разбиения дополнительно задается ожидаемое число групп в разбиении $2 \le m \le \lfloor \log_2 n \rfloor$. Операция разбиения выполняется по следующей схеме. Вначале для последовательности значений функции сложности $f(\omega_1),...,f(\omega_n)$ вычисляются среднее арифметическое и медиана. Максимальное из полученных двух значений *M* разделяет требования на две группы – «простую», для требований из которой значение функции сложности не превосходит *M,* и «сложную», содержащую остальные требования. Данная процедура разбиения повторяется для требований из «сложной» группы, если число требований в ней больше 1, или для требований из «простой» группы в противном случае. После выполнения данной процедуры *(m – 1)* раз будет получено искомое разбиение на *m* групп.

Основная идея данного разбиения состоит в том, чтобы верифицировать отдельно наиболее сложные требования в небольших группах или отдельно, если их сложность значительно выше остальных, и создать одну группу, сложность требований из которой существенно ниже. Для получения функции сложности может использоваться информация о верификации данных требований в предыдущих версиях программного обеспечения (по аналогии с регрессионной верификацией [18]).

4.2. Последовательная комбинация «точность»

Для получения более точного результата была реализована следующая версия последовательной комбинации. На первом шаге все требования верифицируются с помощью метода УМАВ с ограничением по времени t_1, что позволит более эффективно решить большую часть задач за счет эвристик данного метода. На втором шаге все требования, получившие вердикт *unknown* на предыдущем шаге из-за эвристик, проверяются отдельно с помощью метода АС с ограничением по времени *t* на проверку каждого требования. В данном случае предполагается, что подобные требования наиболее сложные для верификации совместно с другими, поэтому для получения более точного результата их следует проверять отдельно. На третьем шаге все оставшиеся требования проверяются более ресурсоемким и более точным методом ДАС. Число оставшихся требований на этом шаге соизмеримо с числом всех требований, при этом более оптимизированным методом УМАВ задачу уже решить не удалось, поэтому и применяется метод ДАС.

Данная реализация объединяет преимущества использования методов верификации композиции требований УМАВ и ДАС и с помощью метода АС устраняет недостатки для предотвращения потерь.

4.3. Последовательная комбинация «производительность»

Для повышения производительности был предложен упрощенный вариант предыдущей реализации. Вначале задача решается методом УМАВ, если же она не была решена, то применяется метод ДАС.

5. Эксперименты

Для сопоставления предложенных комбинаций с существующими методами сравнивалось процессорное время (как решения задач достижимости, так и всего процесса верификации) и потери результата (то есть задачи, не решенные с теми же ограничениями на ресурсы, что и в базовом методе). Эксперименты проводились с помощью системы верификации LDV Tools [15] (ветка *composition_of_reachability_tasks*), задачи достижимости решались с помощью статического верификатора CPAchecker [14]. Проверялись все модули ядра операционной системы Linux версии *4.0-rc1* в конфигурации *allmodconfig*, для которых система LDV Tools успешно готовит задачи достижимости (то есть 4 041 модуль), относительно 30 имеющихся в системе LDV Tools требований. В качестве решения данной задачи от системы верификации ожидались 121 230 вердиктов. В совокупности проверяемые модули содержат около 9 млн. строк кода. Для метода УМАВ использовалась версия 23 131 инструмента CPAchecker ветки *cmav*, для метода ДАС – версия 20 125 инструмента CPAchecker ветки *muauto*. В обоих случаях CPAchecker использовался с конфигурацией, которая включает предикатную абстракцию [19] и анализ явных значений [20]. Для экспериментов использовались компьютеры (узлы кластера) со следующими характеристиками: процессор Intel Xeon E312xx (Sandy Bridge) 2.6 GHz (8 ядер), 64 GB оперативной памяти, операционная система Ubuntu 14.04 (64-bit) с ядром Linux 3.13, Java версии 1.7.0_101 (вычислительный кластер ИСП РАН).

Базовое ограничение на используемые ресурсы было выбрано в соответствии с международными мероприятиями по верификации Competition on Software Verification [12]: 15 минут процессорного времени и 15 GB оперативной памяти на проверку одного требования в одной задаче достижимости. Дополнительно использовалось ограничение по памяти на размер кучи для виртуальной машины Java в 13 GB (13/15 от базового ограничения на оперативную память), что необходимо для корректной работы верификатора CPAchecker. При этом на проверку нескольких требований в одной задаче достижимости ограничение на процессорное время увеличивалось пропорционально числу требований (то есть проверка 30 требований

ограничивалась 27 000 процессорных секунд). Ограничение на оперативную память не изменялось, чтобы подготавливаемые задачи могли решаться на тех же компьютерах. При этом на каждом узле одновременно могло решаться четыре задачи достижимости (по одному ядру процессора и 15 GB оперативной памяти на каждую), оставшиеся ресурсы (4 ядра процессора и 4 GB оперативной памяти) использовались системой LDV Tools для подготовки задач достижимости, запуска комбинации методов и обработки результата.

5.1. Оценка вариантов параллельной комбинации

В данном эксперименте функция сложности возвращала среднее значение релевантных требованию модулей, полученное при верификации предыдущих версий ядра Linux (то есть число модулей, в которых верификация завершалась успешно и требование потенциально могло быть нарушено). Значения данной функции сложности после сортировки образовали следующую последовательность: 1, 2, 4, 11, 11, 17, 18, 25, 30, 36, 55, 60, 63, 68, 83, 91, 111, 113, 118, 131, 134, 155, 178, 214, 302, 356, 783, 970, 1048, 1048. Разбиение производилось на 2 и на 3 группы. Нетрудно заметить, что при применении алгоритма, описанного в разделе 4.1, данные требования в первом случае разбиваются на 23 и 7, а во втором – на 23, 4 и 3. Помимо этого, было рассмотрено случайное разбиение требований на две равные группы (то есть по 15 требований в каждой группе). Результаты эксперимента представлены в табл. 2. Процессорное время и общее ускорении в таблице представлено отдельно для решения задач достижимости верификатором CPAchecker (столбец «Решение задач») и для всего процесса верификации системой LDV Tools (столбец «Всего»), поскольку нужно учитывать, что при верификации m групп требований готовится и решается m задач достижимости для каждого модуля.

Параллельной комбинации удалось ускорить решение задач достижимости в сравнении с методом УМАВ и снизить число потерь результата. Параллельная комбинация с разбиением требований на 2 группы согласно предложенному алгоритму требует примерно на 20% меньше ресурсов на решение задач достижимости и примерно на 10% меньше ресурсов на весь процесс верификации в сравнении с методом УМАВ, при этом число потерь снижается до 1%, а количество потерянных реальных ошибок сокращается с 9 до 2. При увеличении числа групп наблюдается возрастание как времени подготовки задач достижимости примерно на 40% (с 317 000 секунд до 443 000) за счет того, что ставится большее число задач, так и времени решения задач достижимости (что, в частности, выражается снижением среднего ускорения решения задач с 5.83 до 4.68), поскольку выполняется большее число однотипных действий. В результате разбиение на 3 группы (и больше) уже не приводит к снижению суммарного времени верификации относительно метода УМАВ, хотя способствует незначительному сокращению процента потерь.

Случайное разбиение требований на две группы в общем случае демонстрирует более низкие показатели, так как не учитываются особенности решаемых задач.

Табл. 2. Оценка параллельной комбинации
Table 2. Evaluation of parallel combination

Метод	Результат			Процессорное время (с) / общее ускорение	
	Safe	*Unsafe*	Потери	Решение задач	Всего
Базовый	118 703	667	0 (0.00%)	3 889 000 1.00	6 742 000 1.00
УМАВ(30)	117 162	634	1 648 (1.36%)	1 289 000 3.02	1 514 000 4.45
УМАВ(23) || УМАВ(7)	117 556	654	1 201 (0.99%)	1 080 000 3.60	1 397 000 4.83
УМАВ(23) || УМАВ(4) || УМАВ(3)	117 603	658	1 147 (0.95%)	1 141 000 3.41	1 584 000 4.26
УМАВ(15) || УМАВ(15) (случайное разбиение)	117 310	641	1 465 (1.21%)	1 176 000 3.31	1 526 000 4.42

Таким образом, параллельная комбинация демонстрирует как повышение производительности, так и улучшение результата при разбиении на небольшое число групп. Однако для этого требуется использование вспомогательных данных о решаемых задачах, с помощью которых выполняется разбиение требований на группы.

5.2. Оценка вариантов последовательной комбинации

В последовательной комбинации метод УМАВ в обоих случаях был ограничен 1 200 секундами процессорного времени, для каждого запуска метода АС выделялось 900 секунд (базовое ограничение). Результаты эксперимента представлены в табл. 3.

Последовательная комбинация «точность» позволила максимально сократить количество потерь до 0.26%, что сопоставимо со статистической погрешностью. Помимо этого, не было потеряно ни одной реальной ошибки в отличие от остальных методов (метод УМАВ теряет 9 реальных ошибок, а метод ДАС – 2), что крайне важно на практике. При этом требуется разумное количество ресурсов для верификации – меньше метода ДАС, но больше метода УМАВ. Последовательная комбинация «производительность»

сократила ресурсы как на решение задач достижимости, так и на весь процесс верификации по сравнению с методами УМАВ и ДАС, при этом процент потерь оказался незначительно больше метода ДАС.

Таким образом, последовательные комбинации методов позволяют улучшать характеристики методов верификации без дополнительной информации о решаемых задачах.

Табл. 3. Оценка последовательной комбинации
Table 3. Evaluation of sequential combination.

Метод	Результат			Процессорное время (с) / общее ускорение	
	Safe	*Unsafe*	Потери	Решение задач	Всего
Базовый	118 703	667	0 (0.00%)	3 889 000 1.00	6 742 000 1.00
АС	118 929	680	182 (0.15%)	3 434 000 1.13	6 107 000 1.10
УМАВ	117 162	634	1 648 (1.36%)	1 289 000 3.02	1 514 000 4.45
ДАС	118 386	673	548 (0.45%)	1 373 000 2.83	1 550 000 4.35
УМАВ → АС → ДАС («точность»)	118 679	695	311 (0.26%)	1 367 000 2.84	1 592 000 4.23
УМАВ → ДАС («производительность»)	118 320	637	630 (0.52%)	1 196 000 3.25	1 426 000 4.73

6. Заключение

В данной статье были предложены различные варианты комбинации методов статической верификации для повышения производительности и улучшения результата с использованием существующих методов верификации. Реализация предложенных методов для верификации композиции требований продемонстрировала улучшение характеристик при проверке модулей ядра операционной системы Linux.

Параллельная комбинация методов рекомендуется для использования совместно с регрессионной верификацией, что позволит получать требуемую информацию о верификации для разбиения требований на группы. Помимо этого, параллельная комбинация для верификации одной программы может

использовать несколько компьютеров (ядер процессора), что позволит сократить и астрономическое время.

Последовательная комбинация не требует дополнительной информации о решаемых задачах и является более универсальным методом для проверки произвольного программного обеспечения. Различные ее вариации могут улучшать главным образом ту или иную характеристику верификации.

В будущем планируется использовать комбинацию методов для большего числа методов верификации с целью решения произвольных задач достижимости.

Список литературы

[1]. Turing A. M. On Computable numbers, with an application to the Entscheidungsproblem. Proceedings of the London Mathematical Society, pp. 230-265, 1936.

[2]. Corbet J., Kroah-Hartman G., McPherson A. Linux kernel development. How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It. https://www.linux.com/publications/linux-kernel-development-how-fast-it-going-who-doing-it-what-they-are-doing-and-who-5, дата обращения 10.06.2017.

[3]. Мутилин В. С., Новиков Е. М., Хорошилов А. В. Анализ типовых ошибок в драйверах операционной системы Linux. Труды ИСП РАН, т. 22, с. 349-374, 2012. DOI: 10.15514/ISPRAS-2012-22-19.

[4]. Mordan V., Novikov E. Minimizing the number of static verifier traces to reduce time for finding bugs in Linux kernel modules. Proceedings of 8th Spring/Summer Young Researchers Colloquium on Software Engineering, vol. 1, 2014. DOI: 10.15514/syrcose-2014-8-5.

[5]. Mordan V., Mutilin V. Checking several requirements at once by CEGAR. LNCS, vol. 9609, pp. 218-232, 2016. DOI: 10.1007/978-3-319-41579-6_17.

[6]. Мордань В. О., Мутилин В. С. Проверка нескольких требований за один запуск инструмента статической верификации с помощью CEGAR. Программирование, т. 4. с. 225-238, 2016.

[7]. Apel S., Beyer D., Mordan V., Mutilin V., Stahlbauer A. On-The-Fly Decomposition of Specifications in Software Model Checking. Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 349-361, 2016. DOI: 10.1145/2950290.2950349.

[8]. Beyer D., Henzinger T. A., Keremoglu M. E., Wendler P. Conditional model checking: a technique to pass information between verifiers. Proceedings ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE 2012), article 57, 2012. DOI: 10.1145/2393596.2393664.

[9]. Новиков Е. М. Развитие метода контрактных спецификаций для верификации модулей ядра операционной системы Linux. Диссертация на соискание ученой степени кандидата физико-математических наук. ИСП РАН, 2013.

[10]. Beyer D., Chlipala A., Henzinger T. A., Jhala R., Majumdar R. The BLAST query language for software verification. Proceedings of SAS. LNCS, vol. 3148, pp. 2-18, 2004. DOI: 10.1007/978-3-540-27864-1_2.

[11]. Clarke E., Grumberg O., Jha S., Lu Y., Veith H. Counterexample-guided abstraction refinement. Proceedings of CAV, LNCS, vol. 1855, pp. 154-169, 2000. DOI: 10.1007/10722167_15.

[12]. Beyer D. Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016). Proceedings of TACAS. LNCS, vol. 9636, pp. 887-904, 2016. DOI: 10.1007/978-3-662-49674-9_55.

[13]. Beyer D., Henzinger T., Jhala R., Majumdar R. The software model checker BLAST. International Journal on Software Tools for Technology Transfer, vol. 9, issue 5, pp. 505-525, 2007. DOI: 10.1007/s10009-007-0044-z.

[14]. Beyer D., Keremoglu M. CPAchecker: A tool for configurable software verification. Proceedings of Computer Aided Verification. LNCS, vol. 6806, pp. 184–190, 2011. DOI: 10.1007/978-3-642-22110-1_16.

[15]. Khoroshilov A., Mutilin V., Novikov E., Shved P., Strakh A. Towards an open framework for C verification tools benchmarking. Perspectives of Systems Informatics. LNCS, vol. 7162, pp. 179-192, 2012. DOI: 10.1007/978-3-642-29709-0_17.

[16]. Ball T., Rajamani S. K. The SLAM project. Debugging system software via static analysis. Proceedings of Symposium on Principles of Programming Languages (POPL), pp. 1–3, 2002. DOI: 10.1145/565816.503274.

[17]. Мордань В. О. Методы верификации программ на основе композиции задач достижимости. Диссертация на соискание ученой степени кандидата физико-математических наук. ИСП РАН, 2017.

[18]. Beyer D., Wendler P. Reuse of verification results. Proceedings of the 20th International Workshop on Model Checking Software (SPIN 2013). LNCS, vol. 7976, pp. 1-17, 2013. DOI: 10.1007/978-3-642-39176-7_1.

[19]. Beyer D., Keremoglu M., Wendler P. Predicate abstraction with adjustable-block encoding. Proceedings of Formal Methods in Computer-Aided Design, pp. 189-198, 2010.

[20]. Beyer D., Löwe S. Explicit-state software model checking based on CEGAR and interpolation. Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE 2013). LNCS, vol. 7793, pp. 146-162, 2013. DOI: 10.1007/978-3-642-37057-1_11.

Combination of static verification methods for checking requirements composition

V.O. Mordan <mordan@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. Static verification proves correctness of the software against checked requirements, but it requires a lot of resources for that and its task is undecidable in general case. At present there is no universal static verification method, which could efficiently check any software. That is why one should choose more appropriate method and set its parameters for checking correctness of the given requirements in a given program. This paper suggests to combine different static verification methods in order to increase efficiency and effectiveness of verification, which is the first step in creating universal method for static verification. The suggested methods were implemented as combination of actively developing static verification methods for checking requirements composition. Implementation of the suggested methods showed their advantages on Linux kernel modules in comparison with using of each verification method separately.

Keywords: software model checking; counterexample guided abstraction refinement; reachability task; requirements composition.

DOI: 10.15514/ISPRAS-2017-29(3)-9

For citation: Mordan V.O. Combination of static verification methods for checking requirements composition. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 3, pp. 151-170 (in Russian). DOI: 10.15514/ISPRAS-2017-29(3)-9

References

[1]. Turing A. M. On Computable numbers, with an application to the Entscheidungsproblem. Proceedings of the London Mathematical Society, 1936, pp. 230-265.

[2]. Corbet J., Kroah-Hartman G., McPherson A. Linux kernel development. How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It. https://www.linux.com/publications/linux-kernel-development-how-fast-it-going-who-doing-it-what-they-are-doing-and-who-5, свободный, accessed 29.06.2017.

[3]. Mutilin V.S., Novikov E.M., Khoroshilov A.V. Analysis of typical faults in Linux operating system drivers. Trudy ISP RAN / Proc. ISP RAS, vol. 22, 2012, pp. 349–374 (in Russian). DOI: 10.15514/ISPRAS-2012-22-19.

[4]. Mordan V., Novikov E. Minimizing the number of static verifier traces to reduce time for finding bugs in Linux kernel modules. Proceedings of 8th Spring/Summer Young Researchers Colloquium on Software Engineering, vol. 1, 2014. DOI: 10.15514/syrcose-2014-8-5.

[5]. Mordan V., Mutilin V. Checking several requirements at once by CEGAR. LNCS, vol. 9609, pp. 218-232, 2016. DOI: 10.1007/978-3-319-41579-6_17.
[6]. Mordan V. O., Mutilin V. S. Checking several requirements at once by CEGAR. Programming and Computer Software, vol. 42, no. 4, pp. 225–238, 2016. DOI: 10.1134/S0361768816040058.
[7]. Apel S., Beyer D., Mordan V., Mutilin V., Stahlbauer A. On-The-Fly Decomposition of Specifications in Software Model Checking. Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 349-361, 2016. DOI: 10.1145/2950290.2950349.
[8]. Beyer D., Henzinger T. A., Keremoglu M. E., Wendler P. Conditional model checking: a technique to pass information between verifiers. Proceedings ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE 2012), article 57, 2012. DOI: 10.1145/2393596.2393664.
[9]. Novikov E. M. Development of a contract specification method for verification of Linux kernel modules. PhD thesis. ISP RAS, 2013 (in Russian).
[10]. Beyer D., Chlipala A., Henzinger T. A., Jhala R., Majumdar R. The BLAST query language for software verification. Proceedings of SAS. LNCS, vol. 3148, pp. 2-18, 2004. DOI: 10.1007/978-3-540-27864-1_2.
[11]. Clarke E., Grumberg O., Jha S., Lu Y., Veith H. Counterexample-guided abstraction refinement. Proceedings of CAV, LNCS, vol. 1855, pp. 154-169, 2000. DOI: 10.1007/10722167_15.
[12]. Beyer D. Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016). Proceedings of TACAS. LNCS, vol. 9636, pp. 887-904, 2016. DOI: 10.1007/978-3-662-49674-9_55.
[13]. Beyer D., Henzinger T., Jhala R., Majumdar R. The software model checker BLAST. International Journal on Software Tools for Technology Transfer, vol. 9, issue 5, pp. 505-525, 2007. DOI: 10.1007/s10009-007-0044-z.
[14]. Beyer D., Keremoglu M. CPAchecker: A tool for configurable software verification. Proceedings of Computer Aided Verification. LNCS, vol. 6806, pp. 184–190, 2011. DOI: 10.1007/978-3-642-22110-1_16.
[15]. Khoroshilov A., Mutilin V., Novikov E., Shved P., Strakh A. Towards an open framework for C verification tools benchmarking. Perspectives of Systems Informatics. LNCS, vol. 7162, pp. 179-192, 2012. DOI: 10.1007/978-3-642-29709-0_17.
[16]. Ball T., Rajamani S. K. The SLAM project. Debugging system software via static analysis. Proceedings of Symposium on Principles of Programming Languages (POPL), pp. 1–3, 2002. DOI: 10.1145/565816.503274.
[17]. Mordan V. O. Methods of software verification based on composition of reachability tasks. PhD thesis. ISP RAS, 2017 (in Russian).
[18]. Beyer D., Wendler P. Reuse of verification results. Proceedings of the 20th International Workshop on Model Checking Software (SPIN 2013). LNCS, vol. 7976, pp. 1-17, 2013. DOI: 10.1007/978-3-642-39176-7_1.
[19]. Beyer D., Keremoglu M., Wendler P. Predicate abstraction with adjustable-block encoding. Proceedings of Formal Methods in Computer-Aided Design, pp. 189-198, 2010.
[20]. Beyer D., Löwe S. Explicit-state software model checking based on CEGAR and interpolation. Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE 2013). LNCS, vol. 7793, pp. 146-162, 2013. DOI: 10.1007/978-3-642-37057-1_11.

Сертифицируемая бортовая операционная система реального времени JetOS для российских проектов воздушных судов

Ю.А. Солоделов <yasolodelov@2100.gosniias.ru>
Н.К. Горелиц <nkgorelits@2100.gosniias.ru >
Государственный научно-исследовательский институт авиационных систем, 125319, Россия, г. Москва, ул. Викторенко, д. 7

Аннотация. JetOS - перспективная бортовая операционная система реального времени (ОСРВ), разработка которой в настоящее время ведется в рамках научно-исследовательской работы ГосНИИАС. В статье указаны предпосылки появления JetOS и рассмотрены работы, ведущиеся по направлению создания ОСРВ. ОСРВ разрабатывается в соответствии с DO-178C и ARINC 653, учтена возможность работы с OpenGL SC. В статье приведены аппаратные аспекты создания ОСРВ – поддержка многоядерности, платформонезависимость и другие. Одной из важнейших задач при разработке ОСРВ является получение сертификационного пакета, соответствующего DO-178C, благодаря чему JetOS можно будет применять при создании и модернизации авионики для гражданской авиации.

Ключевые слова: операционная система реального времени; ОСРВ; интегрированная модульная авионика; ИМА; сертификация; DO-178C; КТ-178C; ARINC 653; авионика.

DOI: 10.15514/ISPRAS-2017-29(3)-10

Для цитирования: Солоделов Ю.А., Горелиц Н.К. Сертифицируемая бортовая операционная система реального времени JetOS для российских проектов воздушных судов. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 171-178. DOI: 10.15514/ISPRAS-2017-29(3)-10

1. Введение

Современные комплексы бортового оборудования (КБО) проектируются в соответствии с идеологией, известной как «Интегрированная модульная авионика» (ИМА). [1], [2]. Одной из ключевых особенностей ИМА является возможность исполнения нескольких функциональных приложений (реализующих программную часть той или иной самолетной системы) на одном вычислителе. Необходимым условием при этом является разделение

приложений по времени исполнения и доступным ресурсам (т.е. ограничение вытесняющей многозадачности и контроль доступа к памяти для нескольких приложений).

Такой режим работы приложений обеспечивается операционной системой реального времени (ОСРВ), что делает ОСРВ неотъемлемой и важнейшей частью любого современного вычислительного модуля и комплекса бортового оборудования в целом. Каждое разрабатываемое гражданское воздушное судно (ВС) или комплектующее изделие из состава ВС должно проходить сертификацию в уполномоченной организации. В общем случае объектом сертификации является весь комплекс программных и аппаратных средств, входящих в состав КБО воздушного судна. Чтобы ОСРВ можно было сертифицировать в составе комплектующего изделия или борта гражданской авиации, она должна разрабатываться в соответствии с требованиями DO-178C (в русскоязычной редакции – КТ-178С [3]).

На протяжении длительного времени как в отечественной программе ИМА, так и на разрабатываемых воздушных судах применялись зарубежные ОСРВ (например, VxWorks 653 или Thales MACS2). Однако имеется прецедент, при котором один из разработчиков ОСРВ в одностороннем порядке прекратил сотрудничество с рядом отечественных компаний, что осложнило проводимые работы и поставило вопрос о замене зарубежного продукта на отечественный аналог [4].

В настоящее время ОСРВ, удовлетворяющих требованиям КТ-178С, в нашей стране нет [4]. В такой ситуации задача создания отечественной сертифицируемой ОСРВ стала очень актуальной.

Для решения этой задачи в ГосНИИАС была организована одногодичная научно-исследовательская работа (НИР), в рамках которой был создан задел по основным направлениям, необходимым для создания бортовой ОСРВ. По итогам этой работы была заложена трехлетняя НИР (2017-19), целью которой является разработка основных компонентов ОСРВ под рабочим названием “JetOS”.

2. Разработка ОСРВ

Основная задача, поставленная в рамках НИР – создание работоспособной высокопроизводительной бортовой ОСРВ с сертификационным пакетом, который впоследствии должен быть включен в общий набор сертификационных данных при разработке КБО. К работам привлекается ряд соисполнителей, в частности, ИСП РАН и ДС БАРС.

Необходимо пояснить такую постановку задачи как «создание сертификационного пакета». По действующим в настоящее время международным регламентам сертификации подлежит не ПО, а система (комплектующее изделие), в состав которого упомянутое ПО входит [5]. Поэтому говорить о сертификации ПО в отрыве от разрабатываемого воздушного судна или комплектующего изделия было бы некорректно.

Поскольку данная работа ведется независимо от создания ВС или КИ, ее целью должна являться подготовка всех необходимых материалов (т.е. сертификационного пакета) для последующего включения их в процесс разработки и сертификации комплектующего изделия.

В состав сертификационного пакета входят, помимо прочего, следующие материалы:

1. планы и стандарты, по которым разрабатывается ПО (КТ-178С, гл.11.1 -11.8) [3],
2. требования к разрабатываемому ПО (КТ-178С, гл. 11.9) [3],
3. проект ПО(КТ-178С, гл. 11.10) [3],
4. исходный код самого ПО и тестов (КТ-178С, гл. 11.11, 11.13) [3],
5. широчайший спектр результатов верификации (КТ-178С, гл. 11.14) [3] – от результатов инспекции планов, стандартов, требований и т.п. до результатов прогона тестов, анализа трассируемости между данными жизненного цикла и т.д.

Полный перечень данных жизненного цикла приведен в КТ-178С, гл. 11 [3].

Первоначально задача постановки процессов, удовлетворяющих КТ-178С, ставилась очень широко; планировалась постановка процессов для создания нескольких разнородных компонентов (ядро ОСРВ, графическая система, файловая система, модуль информационной безопасности и т.д.) при участии нескольких компаний-разработчиков. Усугублялась задача тем, что для ее решения планировалось объединить несколько инструментов жизненного цикла линейки IBM Rational, т.е. провести работы по их интеграции и разрешению возникающих проблем для нескольких коллективов сразу. Для решения задачи планировалось применять разработанный ранее в рамках НИР ГосНИИАС инструмент ИСУТ (информационная система управления требованиями), расширяющий возможности продуктов IBM Rational, но работы над ОСРВ показали необходимость расширения требований к данному инструменту.

В результате было принято решение уменьшить объем решаемых задач, и в настоящее время постановка процессов КТ-178С продолжается для двух компаний: ГосНИИАС и ИСП РАН. Забегая вперед, отметим, что объем компонентов, подлежащих разработке в рамках сертификационного процесса, также был сокращен. При более детальном рассмотрении основной акцент в работе перенесен на ядро ОСРВ и основные системные компоненты (например, стандартная библиотека языка С или библиотека, предоставляющая сервисы ARINC 653 [6]).

Параллельно с постановкой процессов велись работы по прототипированию ОСРВ, т.е. разработке пробного кода с целью проверки решений, которые должны лечь в основу проекта ПО. Стоит отметить, что требованиями КТ-178С такой работы не предусмотрено; там код должен разрабатываться исключительно при наличии архитектуры и требований низкого уровня, т.е.

проекта ПО; в свою очередь, проект ПО должен разрабатываться только при наличии требований. Однако практика показала, что разрабатывать архитектуру комплексного ПО, не проведя предварительную проверку решений на практике, очень затруднительно. В связи с этим (и не вступая в противоречие с КТ-178С) был разработан т.н. прототип ОСРВ – работающий код системы, предназначенный для ранней проверки принимаемых архитектурных решений.

Как прототип, так и основной код ОСРВ разрабатывается на языке С. Это обусловлено простотой его синтаксиса сравнительно с более поздними разработками (например, C++) и широкой распространенностью (компиляторы языка С существуют для подавляющего большинства аппаратных платформ в мире). В соответствии с КТ-178С процесс кодирования регламентируется стандартом на кодирование, разработанным в рамках данного проекта и входящим в сертификационный пакет. Стандарт на кодирование в значительной степени базируется на документе MISRA C и синтаксисе языка C99. Стоит отметить, что при создании бортового ПО крайне большое значение уделяется верификации исходного (а зачастую и исполняемого) кода, а широкие синтаксические возможности и развитые парадигмы программирования более современных языков очень усложняют верификацию кода или делают ее вовсе невозможной. Так, в настоящее время применение объектно-ориентированных языков при разработке бортового ПО допускается лишь с существенными ограничениями (см. [7]).

Одним из важных требований к системному бортовому ПО является возможность работы с графической библиотекой OpenGL (как правило, разновидности Safety Critical – OpenGL SC). OpenGL SC является подмножеством стандарта OpenGL для применения в составе критических систем. В настоящее время OpenGL SC представлен версиями 1.0.1 и 2.0. Бортовые функциональные приложения используют сервисы OpenGL для отображения как двухмерных изображений (мнемокадры, таблицы и пр.), так и трехмерных – например, при моделировании рельефа местности.

В работы по ОСРВ с самого начала было заложено графическое направление: создание собственной библиотеки OpenGL SC (причем полностью программной, сертифицируемой по DO-178C и независимой от аппаратуры) и создание графического менеджера, аналогичного компоненту DRM из состава ядра Linux [8]. Для выполнения этих работ были налажены контакты с научными коллективами ИПМ им. М.В. Келдыша РАН и МГТУ им. Н.Э. Баумана; в настоящее время рассматривается вопрос о выделении этих работ в отдельное направление, не привязанное непосредственно к созданию ОСРВ.

Говоря про создание ОСРВ для авионики, нельзя не упомянуть про стандарт ARINC 653 [6]. Данный стандарт, определяющий программный интерфейс и режимы работы бортового функционального ПО, на протяжении ряда лет является общепринятым во всем мире. При разработке JetOS заложена

реализация ARINC 653 наиболее актуальной версии (2015 года), причем не только основных, но и дополнительных сервисов.

Создание ОСРВ как полноценного продукта подразумевает также разработку целого спектра инструментов – в первую очередь, интегрированной среды разработки функционального ПО, а также компонентов ОСРВ, необходимых для отладки, мониторинга и трассировки разрабатываемых приложений.

3. Аппаратные аспекты создания ОСРВ

Специфика архитектуры ИМА (возможность применения разнородных модулей) и современное состояние аппаратных платформ (постоянная модернизация и быстрое развитие аппаратных платформ) диктуют такие требования к ОСРВ, как поддержка многоядерности и легкая переносимость между различными аппаратными платформами с различными архитектурами процессоров.

Вопрос переносимости решается способом, типовым для различных ОСРВ. В архитектуре закладывается разделение на платформо-зависимые и платформо-независимые компоненты, при этом большинство компонентов (и в первую очередь – ядро ОСРВ) должны являться платформо-независимыми, т.е. разрабатываться на языке С и не иметь ассемблерных вставок.

При переносе на новую аппаратную платформу платформо-независимый код ОСРВ компилируется с помощью инструментов, предоставляемых разработчиком данной платформы, и объединяется с низкоуровневым кодом, специально написанным для конкретной аппаратуры. Такой подход позволяет сохранить основную кодовую базу неизменной и одновременно осуществлять поддержку широкого спектра аппаратных решений. При этом сертификационный пакет, полученный в ходе разработки ОСРВ, в значительной части сохраняет свою актуальность (за исключением информации, относящейся к конкретной аппаратной платформе).

Программный продукт может соответствовать требованиям КТ-178С только для определенных аппаратных платформ; при переходе на новую аппаратуру должна проводиться повторная верификация всей портируемой кодовой базы, а весь новый исходный код должен быть разработан в соответствии с процессами КТ-178С. Поэтому в рамках данной НИР была выбрана так называемая основная платформа, для которой будут собираться верификационные данные; ей стал вычислительный модуль МУПД2G на базе процессора P3041 (PowerPC), разработанный компанией НКБ ВС в рамках отечественной программы ИМА, проводившейся под руководством ГосНИИАС. Помимо МУПД2G имеющийся код ОСРВ портирован на платформу на базе P1010 (PowerPC), а также на i.MX6 (на базе архитектуры ARM). Вопрос многоядерности является комплексным. Как указано в CAST-32 (см. [9]), документ DO-178C разрабатывался в то время, когда применение многоядерных процессоров еще не было повсеместным, и в связи с этим специфика применения данного типа платформ в нем (и, соответственно, в

КТ-178С) не отражена. Многоядерность отражается на всех данных жизненного цикла – от планов и стандартов до результатов тестирования, и для ОСРВ данный вопрос находится на стадии проработки. Необходимо отметить также, что вышеупомянутый процессор P3041 является четырехъядерным.

4. Заключение

В настоящее время проект находится на стадии постановки процессов DO-178C и параллельной подготовки артефактов первой версии. Успешно проведено прототипирование на широком спектре компонентов - как системного, так и прикладного уровня (включая графический менеджер, сетевой стек, файловую систему и библиотеку OpenGL). В рамках прототипирования была разработана базовая версия требований высокого уровня и проекта ПО, которая сейчас активно развивается и перерабатывается; параллельно ведутся работы по рефакторингу кодовой базы и созданию инфраструктуры тестирования.

Опыт применения бортовых ОСРВ в мире показывает, что продукт, соответствующий DO-178C может быть сертифицирован и для применения в других отраслях промышленности, что достигается за счет заложенных в стандарте ограничений и жестких требований [10]. Поэтому вопрос адаптации JetOS для индустриальной техники, космоса, транспорта, медицины также является актуальным и активно прорабатывается; при этом основной задачей НИР остается создание работоспособной высокопроизводительной ОСРВ с сертификационным пакетом, который впоследствии можно будет использовать при создании КБО для гражданских самолетов.

Список литературы

[1]. Федосов Е.А. Проект создания нового поколения интегрированной модульной авионики с открытой архитектурой. Полет, №8, 2008 г., стр. 15-22.

[2]. Федосов Е.А. Косьянчук В.В., Сельвесюк Н.И. Интегрированная модульная авионика. Радиоэлектронные технологии, №1, 2015 г., стр. 66-71.

[3]. Квалификационные требования часть 178С, АР МАК, 2014

[4]. Федосов Е.А., Ковернинский И.В., Кан А.В., Волков В.Б., Солоделов Ю.А. Применение операционных систем реального времени в интегрированной модульной авионике. Труды ГосНИИАС: вопросы авионики, №4(24), 2015 г.

[5]. Руководство Р4754 по процессам сертификации высокоинтегрированных сложных бортовых систем воздушных судов гражданской авиации. АР МАК, 2010

[6]. Avionics application software standard interface (ARINC 653). SAE-ITC, 2015

[7]. DO-332: Object-Oriented Technology and Related Techniques. RTCA, December 13, 2011

[8]. Ragav Gopalan. Inside Linux graphics.Intel Embedded, April 2011

[9]. Multi-core Processors (CAST-32A). Certification Authorities Software Team, 2016

[10]. Sven Nordhoff. Successful multicore certification with software-partitiong. SYSGO AG, 2016

Certifiable onboard real-time operation system JetOS for Russian aircrafts design

Yu.A. Solodelov <yasolodelov@2100.gosniias.ru>
N.K. Gorelits <nkgorelits@2100.gosniias.ru >
State Research Institute of Aviation Systems,
125319, Russia, Moscow, Viktorenko Str, 7

Abstract. JetOS is a prospective onboard real-time operating system (RTOS). Nowadays GosNIIAS develops JetOS in the scope of the research and development project. One of the most important tasks during JetOS development is to create the DO-178C certification kit, which will allow JetOS to be used for development and modification of avionics for civil aircraft. Today there is no operating system certified in accordance with DO-178C in Russia, therefore the JetOS creation is the matter of current importance. Using DO-178C requires the developer to have very strict development processes. The arrangement of processes that satisfy the DO-178C requirements is a very responsible and demanding task because of high expectations in the fields of safety and security. JetOS is being developed primarily for onboard equipment based on the integrated modular avionics (IMA). One of the key features of IMA is the ability to execute several functional applications on one target onboard module. The obvious consequence of this feature is a necessity to have a time and resource partitioning of applications. In avionics field application partition along with a host of other features is defined in ARINC 653 international standard, so its support is the significant requirement for JetOS. ARINC 653 defines application programming interface (API) and modes of operation for onboard functional software. JetOS supports the up-to-date version of ARINC 653 (2015) with supplementary services. JetOS also supports the safety-critical graphical library – OpenGL SC; the special implementation of the OpenGL SC library is being developed along with JetOS itself. OpenGL SC services are used to draw two-dimensional and three-dimensional pictures by onboard functional software. JetOS is a certifiable modular cyber-safe real-time operating system, which is designed in order to support several hardware architectures and to be easily adopted for different hardware boards. The scope of the JetOS project also includes creation of the tools necessary for functional software development, especially aircraft systems.

Keywords: real-time operation system; RTOS; integrated modular avionics; IMA; certification; DO-178C; ARINC 653; civil avioncs.

DOI: 10.15514/ISPRAS-2017-29(3)-10

For citation: Solodelov Yu.A., Gorelits N.K. Certifiable onboard real-time operation system JetOS for Russian aircrafts design. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 3, 2017. pp. 171-178 (in Russian). DOI: 10.15514/ISPRAS-2017-29(3)-10

References

[1]. Fedosov E.A. The new generation open architecture IMA project. Polet, №8, 2008 , pp. 15-22 (in Russian).

[2]. Fedosov E.A., Kosyanchuk V.V., Selvesyuk N.I. Integrated Modular Avionics. Radioelektronnye tehnologii, №1, 2015, pp. 66-71 (in Russian).
[3]. Qualification requirements part 178C. IAC, 2014 (in Russian).
[4]. Fedosov E.A., Koverninsky I.V., Kan A.V., Volkov V.B., Solodelov Yu.A. Use of real-time operating systems in the integrated modular avionics. GosNIIAS Proceedings: avionics, №4(24), 2015 г (in Russian).
[5]. R4754. IAC, 2010 (in Russian).
[6]. Avionics application software standard interface (ARINC 653). SAE-ITC, 2015
[7]. DO-332: Object-Oriented Technology and Related Techniques. RTCA, December 13, 2011
[8]. Ragav Gopalan. Inside Linux graphics.Intel Embedded, April 2011
[9]. Multi-core Processors (CAST-32A). Certification Authorities Software Team, 2016
[10]. Sven Nordhoff. Successful multicore certification with software-partitiong. SYSGO AG, 2016

Обзор методов динамической компиляции запросов

[1,2] *Е. Ю. Шарыгин <eush@ispras.ru>*
[1] *Р. А. Бучацкий <ruben@ispras.ru>*
[1] *Институт системного программирования РАН,*
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.
[2] *Московский государственный университет имени М.В. Ломоносова,*
119991, Россия, Москва, Ленинские горы, д. 1

Аннотация. Эффективное использование процессора является решающим фактором производительности аналитических систем, особенно с увеличением размеров обрабатываемых данных. В то же время возрастающие объёмы доступной основной памяти позволяют значительно сократить количество обращений к медленным дисковым хранилищам и тем самым отводят традиционные для большинства систем обработки данных оптимизации подсистемы ввода–вывода на второй план. Одним из наиболее эффективных способов повышения эффективности использования процессора и сокращения накладных расходов, прежде всего проявляющихся в затратах на интерпретацию планов запросов, является компиляция запросов в исполняемый код во время выполнения (динамическая компиляция). В последнее время наблюдается рост интереса к методам динамической компиляции запросов как в академических, так и в прикладных разработках. Данная статья является обзором литературы в области динамической компиляции запросов, в основном для реляционных СУБД. Представлены работы последних лет, описаны архитектурные особенности методов, сделана классификация работ, приведены основные результаты.

Ключевые слова: динамическая компиляция; JIT-компиляция; языки запросов; SQL; push-модель; специализация кода.

DOI: 10.15514/ISPRAS-2017-29(3)-11

Для цитирования: Шарыгин Е. Ю., Бучацкий Р. А. Обзор методов динамической компиляции запросов.Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 179-224. DOI: 10.15514/ISPRAS-2017-29(3)-11

1. Введение

Одной из основных функций современных СУБД [1] является предоставление интерфейса для выполнения операций по определению, изменению или выборке данных. Чаще всего такой интерфейс реализуется посредством специализированного языка запросов, например, SQL, что позволяет максимально изолировать задачи, решаемые СУБД, от других частей информационной системы.

Процесс обработки запросов в большинстве СУБД, таким образом, включает в себя несколько этапов:

1. синтаксический анализ, восстановление структуры запроса из текстового представления;
2. семантический анализ, определение используемых таблиц, атрибутов, типов данных, функций и т. д.;
3. составление и оптимизация плана выполнения запроса;
4. выполнение (интерпретация) плана запроса.

Можно заметить, что только затраты на выполнение этапа 4 (выполнения плана) зависят не только от сложности самого запроса, но и от размера данных, обрабатываемых СУБД, и поэтому доминируют над затратами на выполнение этапов 1–3 с увеличением размера данных.

Системы, работающие по приведённой выше схеме обработки запросов, в рамках данной статьи будем называть *классическими*. Недостатки классической схемы проявляются в следующих сценариях использования:

- Если информационная система отправляет много запросов, большинство из которых принадлежит одному из сравнительно небольшого числа различных классов, то этапы 1–3 в классической схеме для каждого класса запросов выполняются многократно. Современные СУБД решают эту проблему при помощи поддержки параметризованных запросов (prepared statements). Параметризованный запрос анализируется, транслируется в план выполнения и оптимизируется только один раз во время определения, и во время выполнения оптимизированный план переиспользуется для всех экземпляров запроса. В данной статье параметризованные запросы в дальнейшем не рассматриваются.
- Если большинство запросов, отправляемых информационной системой, принадлежит ограниченному числу запросов, известных во время разработки системы, то в классической схеме все перечисленные этапы выполняются многократно, несмотря на то, что как план выполнения, так и параметры запроса (если они есть) остаются на протяжении работы системы неизменными. Параметризация запросов позволяет избежать выполнения этапов 1–3, но не этапа 4 (собственно выполнения), алгоритмическая сложность которого доминирует над остальными этапами. Накладных расходов на интерпретацию можно избежать в этом случае при помощи (статической) компиляции запросов в машинный код [2–4] при сборке информационной системы. Статическая компиляция в дальнейшем также не рассматривается.
- Если запросы, отправляемые к СУБД, достаточно сложны и выполняются достаточно долго, накладные расходы на интерпретацию плана в классической схеме (этап 4) могут составлять значительную часть в общей времени обработки запроса. Решением этой проблемы является динамическая компиляция, методам реализации которой и посвящена данная статья.

Методы динамической компиляции предполагают замену в общем времени

обработки запроса времени интерпретации $t_I(N)$ на суммарное время компиляции и выполнения скомпилированного кода $t_C + t_E(N)$, где N — размер данных, обрабатываемых запросом, t_C — время компиляции. Учёт структуры и константных данных запроса и проводимые во время компиляции оптимизации делают скомпилированный код более эффективным: $t_E(N) < t_I(N)$, но чтобы динамическая компиляция имела смысл, необходимо, чтобы $t_C + t_E(N) < t_I(N)$, то есть чтобы размер данных был достаточно велик: $N > N_0$.

Существующие методы динамической компиляции — и СУБД, их использующие, — можно разделить на несколько классов:

- компиляция выражений (раздел 2) — компиляция таких частей запросов, как арифметические и логические выражения и доступ к атрибутам, при сохранении интерпретации плана запроса;
- компиляция запросов (раздел 3) — компиляция запросов целиком, выполнение без интерпретации;
- автоматическая специализация (раздел 4) — специализация интерпретатора запросов во время выполнения к данным и структуре запроса.

2. Компиляция выражений и горячих участков кода

Динамическая компиляция выражений позволяет избежать накладных расходов на исполнение обобщённого кода СУБД благодаря компиляции некоторых «горячих» функций СУБД в машинный код во время выполнения с учётом конкретного запроса. Множество рассматриваемых функций-кандидатов включает в себя:

1. Функции, осуществляющие вычисление арифметических выражений. Так как выражения в запросе становятся известны только во время выполнения, вычисление выражений, как правило, реализуется в классических СУБД при помощи интерпретации, накладные расходы, связанные с которой, динамическая компиляция позволяет избежать.
2. Пользовательские функции. Многие системы позволяют использовать в запросах функции, определённые пользователем или на процедурных языках, поддержка которых встроена непосредственно в язык запросов, или через API (например, на C или C++). Вызов пользовательской функции во время выполнения запроса сопряжён с накладными расходами на поддержку абстракции: интерфейс, позволяющий пользовательской функции получать значения и типы параметров и устанавливать возвращаемое значение как правило реализован обобщённо. Кроме того, тело функции также обычно неизвестно до начала выполнения запроса. Динамическая компиляция позволяет встроить тело пользовательской функции непосредственно в машинный код вызывающей функции и удалить

накладные расходы за счёт оптимизации функции в месте вызова.

3. Функции доступа к атрибутам. В СУБД, использующих построчное хранение кортежей (N-ary Storage Model [1, 5]), атрибуты каждого конкретного кортежа располагаются в памяти последовательно. Смещение заданного атрибута в кортеже определяется его порядковым номером и типами предшествующих атрибутов (которые, в свою очередь, в случае реляционной модели данных определяются заголовком отношения) и потому вычисляется во время выполнения запроса. Динамическая компиляция позволяет предвычислить или существенно оптимизировать (в случаях, когда смещение атрибута также зависит и от *значений* предшествующих атрибутов [6]) вычисление смещений атрибутов, используя информацию о таблицах, к которым производится обращение.
4. Функции, использующие константные данные времени выполнения. Динамическая компиляция позволяет встроить в код значения не изменяемых во время выполнения глобальных переменных и полей структур, например, параметров работы СУБД или номера выполняемой транзакции (снимка таблицы) при использовании механизма многоверсионности (MVCC). Можно заметить, что в эту категорию входят также внутренние структуры интерпретатора выражений и заголовки таблиц, что делает описываемый здесь класс функций обобщением классов функций, описываемых выше.

При компиляции выражений сохраняется общий механизм интерпретации плана запроса, компиляции которого посвящена раздел 3.

2.1 Impala (2014)

Impala [7, 8] — это распределённая аналитическая система обработки данных для Apache Hadoop [9], использующая в качестве хранилища данных распределённую файловую систему Apache HDFS [9] или распределённую нереляционную СУБД Apache HBase [10].

Impala предоставляет интерфейс запросов на языке SQL и использует LLVM [11] для компиляции частей запроса в машинный код во время выполнения.

В качестве примера приводится функция доступа к атрибутам кортежа в памяти: динамическая компиляция позволяет специализировать общий код доступа к атрибутам, реализующий поддержку многих типов данных, с использованием доступной во время выполнения информации о том, к каким таблицам в данном запросе производится обращение. Поскольку эта функция вызывается для каждого обрабатываемого в цикле кортежа, даже удаление небольшого числа инструкций приводит к значительному приросту производительности.

Среди оптимизаций, которые удалось применить во время динамической компиляции, авторы выделяют:

- подстановку констант времени выполнения;
- удаление условных переходов;
- удаление операций загрузки из памяти;
- разворачивание циклов;
- встраивание виртуальных вызовов.

В частности, арифметические и логические выражения в Impala представляются в виде иерархии классов с перегруженным методом вычисления значения, и встраивание вызовов виртуальных функций позволяет значительно сократить связанные с этим накладные расходы (см. рис. 1).

```
IntVal my_func(const IntVal& v1, const IntVal& v2) {
  return IntVal(v1.val * 7 / v2.val);
}

SELECT my_func(col1 + 10, col2) FROM ...
```

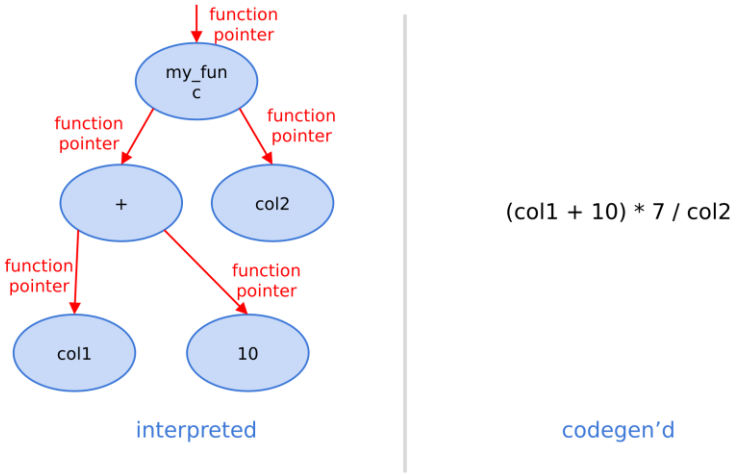


Рис. 1. Компиляция выражений в Impala
Fig. 1. Expression compilation in Impala

В [8] также описывается подход к разработке динамического компилятора с использованием инфраструктуры LLVM, при котором оптимизируемые функции реализуются на языке высокого уровня (C++) и компилируются статически в LLVM IR и в объектный код, что позволяет переиспользовать один и тот же исходный код функций как в компиляторе, в котором производится связывание полученного кода на LLVM IR с генерируемым модулем (единицей трансляции) LLVM во время выполнения, так и в интерпретаторе, с которым полученный код связывается статически. Этот же метод также применяется и для оптимизации пользовательских функций на C++, которые связываются с использующим их кодом при использовании интерпретатора динамически через динамически подгружаемую библиотеку, а

при использовании компилятора — статически во время выполнения.
Динамическая компиляция выражений в Impala позволяет получить ускорение до 5.7 раз на запросе Q1 (который входит в состав бенчмарка TPC-H [12]), при этом количество выполняемых инструкций и ветвлений сокращается в 4.29 и 3.76 раз, соответственно.

2.2 Spark SQL (2015)

Apache Spark [13] — это фреймворк для реализации распределённой обработки слабоструктурированных и неструктурированных данных для Apache Hadoop. Основными областями применения Spark являются потоковая обработка данных, обработка графовых данных и машинное обучение.
Spark SQL [14] — это модуль Apache Spark, реализующий поддержку реляционной модели обработки данных и декларативного языка запросов SQL.
В Spark SQL применяется динамическая компиляция арифметических и логических выражений. Для генерации промежуточного представления (абстрактного синтаксического дерева Scala) и компиляции его в байткод виртуальной машины Java используются встроенные средства языка Scala (quasiquotes).
На простых запросах динамическая компиляция выражений в Spark SQL позволяет получить ускорение до 3.5 раз по сравнению с интерпретацией.

2.3 Компиляция выражений в PostgreSQL

PostgreSQL [15] — высокопроизводительная объектно-реляционная СУБД, поддерживающая язык запросов SQL и расширяемая пользовательскими функциями, операторами, индексами, типами данных и процедурными языками. PostgreSQL использует построчное хранение кортежей (N-ary Storage Model [1, 5]).
В число СУБД, основанных на PostgreSQL [16], входят нереляционная СУБД ToroDB [17], колоночная СУБД Vertica [18], графовая СУБД AgensGraph [19], аналитическая СУБД VitesseDB [20], распределённые аналитические СУБД ParAccel [21], Redshift [22], Greenplum [23] и DeepgreenDB [24] и многие другие. Динамическая компиляция на уровне выражений или запросов используется по крайней мере в VitesseDB, ParAccel, Redshift, Greenplum и DeepgreenDB.
В этом подразделе рассмотрим некоторые существующие исследования по разработке и реализации динамической компиляции выражений для PostgreSQL.

2.3.1 Микроспециализация (2012)

Метод микроспециализации [25–27] заключается в замене некоторых

критических для производительности функций или участков кода СУБД на версии, специализированные под конкретную таблицу, запрос или кортеж на основе специально подготовленных шаблонов кода. [26] приводит три других способа специализации кода в СУБД:

- Архитектурная специализация, заключающаяся в адаптировании архитектуры СУБД под конкретный класс приложений. Примером является использование колоночного хранения или потоковой обработки данных.
- Компонентная специализация заключается в разработке специализированных компонентов СУБД, ориентированных под конкретные типы данных или сценарии использования. Примерами являются новые типы операторов и индексов.
- Пользовательская специализация, которая состоит в предоставлении пользователю средств специализации выполнения запросов за счёт пользовательских функций.

Исследователи рассматривают цикл обработки запросов в СУБД и отмечают, что разные переменные получают значение на разных этапах его выполнения. Например, переменные, определяемые данными таблицы, могут становиться известны только после чтения соответствующих буферов базы данных во время обработки запроса, переменные, связанные со структурой запроса, известны после получения и анализа запроса, в то время как переменные, описывающие заголовки таблиц и конфигурационные параметры СУБД, известны уже после начала работы СУБД и редко меняются.

В предлагаемой схеме разработчик определяет функции-кандидаты для специализации, удовлетворяющие следующим требованиям:

1. Функция-кандидат должна вызываться в цикле обработки запросов.
2. Функция-кандидат должна занимать существенную долю в общем времени выполнения запроса.
3. Функция-кандидат должна содержать обращения к переменным, значения которых инвариантны на всём протяжении выполнения тела цикла обработки запроса.
4. Функция-кандидат должна существенно зависеть от этих переменных и допускать эффективную специализацию при фиксированных значениях переменных.

Примерами функций-кандидатов являются функции вычисления выражений, функции доступа к атрибутам, функции, реализующие конкретные операторы плана выполнения [27]. Исследователи использовали профилировщик Callgrind [28] для выявления функций-кандидатов, удовлетворяющих условию 2.

В зависимости от того, на каком этапе выполнения СУБД становятся известны значения переменных, используемых в функции-кандидате, подразделяются три вида шаблонов специализированных функций (“bees” в терминологии [25–27]):

- relation bee (переменные определяются схемой таблицы),

- query bee (переменные определяются запросом) и
- tuple bee (переменные определяются значениями конкретных атрибутов в кортеже).

Например, на рис. 2 и рис. 3 представлен пример relation bee — функции доступа к атрибутам в СУБД PostgreSQL — до и после специализации, при этом цветом отмечены инвариантные переменные.

```
1 void slot_deform_tuple(TupleTableSlot *slot, int natts) {
2   ...
3   tp = (char *) tup + tup->t_hoff;
4   for (; attnum < natts; attnum++) {
5     Form_pg_attribute thisatt = att[attnum];
6     if (hasnulls && att_isnull(attnum, bp)) {
7       values[attnum] = (Datum) 0;
8       isnull[attnum] = true;
9       slow = true;
10      continue;
11    }
12    isnull[attnum] = false;
13    if (!slow && thisatt->attcacheoff >= 0) {
14      off = thisatt->attcacheoff;
15    } else if (thisatt->attlen == -1) {
16      if (!slow && off == att_align_nominal(off, thisatt->attalign)) {
17        thisatt->attcacheoff = off;
18      } else {
19        if (!slow && off == att_align_nominal(off, thisatt->attalign)) {
20          thisatt->attcacheoff = off;
21        } else {
22          off = att_align_pointer(off, thisatt->attalign, -1, tp + off);
23          slow = true;
24        }
25      } else {
26        off = att_align_nominal(off, thisatt->attalign);
27        if (!slow)
28            thisatt->attcacheoff = off;
29      }
30      values[attnum] = fetchatt(thisatt, tp + off);
31      off = att_addlength_pointer(off, thisatt->attlen, tp + off);
32      if (thisatt->attlen <= 0)
33          slow = true;
34    }
35    ...
36  }
37 }
```

Рис. 2. Функция доступа к атрибутам в PostgreSQL
Fig. 2. Attribute access function in PostgreSQL

Шаблоны могут инстанциироваться или заранее, если возможные значения переменных, связанных, например, с конкретными атрибутами кортежей или параметрами отношений или запросов, принадлежат ограниченному диапазону, — в этом случае для каждой комбинации возможных значений переменных, участвующих в шаблоне, инстанциируется по отдельной специализированной функции — или непосредственно после получения переменными своих значений во время выполнения. Для relation bees это время определения схемы таблицы, для query bees — время построения плана выполнения запроса, и для tuple bees — время вставки или изменения данных таблицы.

Во время инстанциирования происходит заполнение «дыр» в шаблонах конкретными значениями переменных и дальнейшая оптимизация — свёртка и продвижение констант, удаление ветвлений.

```
1 void GetColumnsToLongs(char bee_id, int address, char* data, int* start_att,
2                        int* offset, bool* isnull, Datum* values) {
3   *(long*)isnull = 0;
4   isnull[8] = 0;
5   values[0] = *(int*)data;
6   values[1] = *(int*)(data + 4);
7   values[2] = (long)(address + bee_id * 32 + 1000);
8   *start_att = 3;
9   if (end_att < 4) return;
10  *offset = 8;
11  if (*offset != (((long)(*offset) + 3) & ~((long)3)))
12    if (!(*(char*)(data + *offset)))
13        *offset = (long)(*offset + 3) & ~(long)3;
14  values[3] = (long)(data + *offset);
15  *offset += VARSIZE_ANY(data + *offset);
16  *offset = ((long)(*offset) + 3) & ~((long)3);
17  values[4] = (*(long*)(data + *offset)) & 0xffffffff;
18  *offset += 4;
19  values[5] = (long)(address + bee_id * 32 + 1001);
20  *start_att = 6;
21  if (end_att < 7) return;
22  if (!(*(char*)(data + *offset)))
23      *offset = (long)(*offset + 3) & ~(long)3;
24  values[6] = (long)(data + *offset);
25  *offset += VARSIZE_ANY(data + *offset);
26  values[7] = *(int*)(address + bee_id * 32 + 1002);
27  if (!(*(char*)(data + *offset)))
28      *offset = (long)(*offset + 3) & ~(long)3;
29  values[8] = (long)(data + *offset);
30  *start_att = 9;
31 }
```

Рис. 3. Функция доступа к атрибутам после специализации
Fig. 3. Specialized attribute access function

В [25, 26] описывается архитектура системы HIVE, которая служит для построения, компиляции, инстанциирования шаблонов кода, а также кеширования и удаления инстанциированных шаблонов при удалении соответствующих объектов (схем, таблиц) из базы данных.

В [27] описывается механизм горячей замены специализированных функций, который позволяет совмещать выполнение нескольких специализированных функций во время выполнения одного запроса. Для этого при изменении значений переменных, используемых в специализированной функции, происходит изменение кода вызывающей функции для замены адресов функций в инструкциях вызова. Механизм позволяет эффективно специализировать код операторов с несколькими внутренними состояниями: например, оператор JOIN, состояния которого определяют, из какого дочернего оператора происходит считывание кортежей.

Подход реализован для СУБД PostgreSQL и на бенчмарке TPC-H показывает средний прирост производительности в 12.4%, при этом на разных запросах прирост производительности составил от 1.4% до 32.8%. На бенчмарке TPC-C [29] средний прирост производительности превысил 11%.

2.3.2 Butterstein, Grust (2016)

В работе [30] приведены результаты профилирования на запросе TPC-H

Q1 [12], согласно которому вычисление арифметических и логических выражений занимает до 70% от суммарного времени обработки запроса.

В работе предлагается метод компиляции выражений в запросах в машинный код с использованием инфраструктуры LLVM [11]. Метод основан на «дырах» в генерируемом коде на LLVM IR, которые заполняются по мере генерации кода для всего выражения.

В работе предлагается оптимизация, направленная на минимизацию количества вызовов функции `slot_getattr`, предоставляющей в PostgreSQL доступ к значениям атрибутов кортежа, при помощи разделения «дыр» по множеству атрибутов, значения которых к моменту достижения «дыр» во время выполнения гарантировано извлечены, и дублированию кода под разные «дыры» с добавлением необходимых вызовов `slot_getattr` где необходимо.

Например, на рис. 4 приведён результат компиляции логического выражения $\big(p_1(A) \wedge p_2(B)\big) \vee p_3(A,B)$ с двумя дырами F_1 и F_2, которые соответствуют случаям, когда первый дизъюнкт принимает значение «ложь», и при этом загружены или только атрибут `A`, или и `A`, и `B`. Код, сгенерированный в эти «дыры» во время компиляции оператора `OR`, отличается наличием в случае F_1 загрузки атрибута `A`, в то время как в случае F_2 это не требуется.

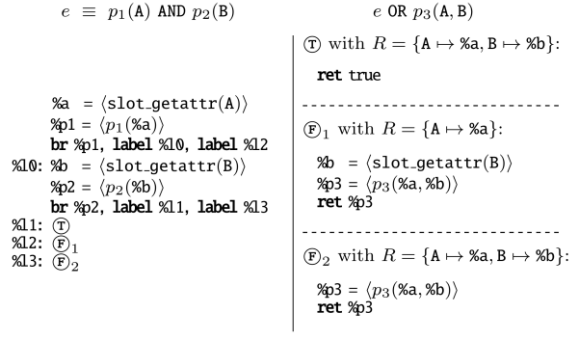


Рис. 4. Разделение «дыр» при компиляции логического выражения
Fig. 4. Expression compilation with hole splitting

Метод реализован в СУБД PostgreSQL и показывает ускорение до 37% на бенчмарке TPC-H.

2.3.3 Компиляция выражений в компиляторе, разрабатываемом в ИСП РАН (2016)

В ИСП РАН разрабатывается компилятор запросов для PostgreSQL с использованием LLVM, частью которого является компилятор выражений [6].

В работе приведены результаты профилирования выполнения запросов

из бенчмарка TPC-H [12], согласно которым на некоторых запросах вычисление предикатов оператора WHERE занимает более 50% от общего времени выполнения.

При компиляции выражения осуществляется обход представляющего его синтаксического дерева и генерация соответствующего кода на LLVM IR. Вопрос необходимости переписывания на LLVM API всего множества функций, доступных для вызова из выражений в запросах на варианте языка SQL, реализованном в PostgreSQL, решён при помощи метода предкомпиляции (рис. 5), который предполагает предварительное преобразование исходного кода PostgreSQL, реализующего встроенные функции, в вид, доступный кодогенератору. Преобразование осуществляется в два этапа:

1. Трансляция исходного кода PostgreSQL в биткод LLVM при помощи компилятора Clang [31].
2. Трансляция полученного биткода функций в последовательности вызовов LLVM API на языке C++ для воссоздания соответствующего промежуточного представления в памяти при помощи компиляции под встроенную в LLVM псевдоплатформу CPPBackend.

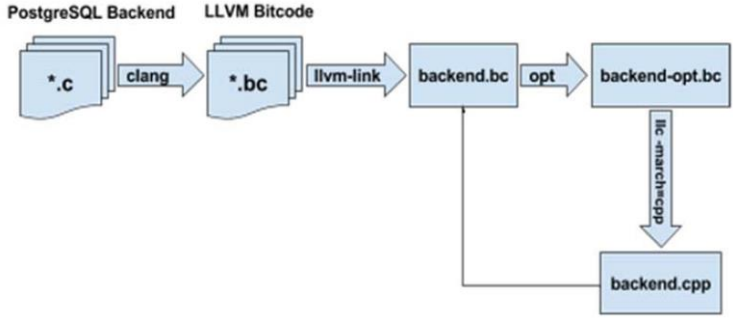


Рис. 5. Метод предкомпиляции выражений
Fig. 5. Expression precompilation method

В [6] также описана оптимизация доступа к атрибутам кортежа, основанная на предподсчёте разницы между смещениями извлекаемых атрибутов с использованием доступной в заголовке таблицы информации о типах атрибутов и установленных флагах `attnotnull` и `attlen`, определяющих, имеет ли атрибут фиксированный размер в памяти или нет. Последовательности неиспользуемых в выражении атрибутов фиксированной длины в скомпилированном коде пропускаются. Оптимизация позволяет уменьшить число извлекаемых атрибутов до $\{n \leq m(N) \vee n \in N \vee \neg attfixed(n)\} \vee$, где N — множество используемых атрибутов, по сравнению с обобщённой реализацией в интерпретаторе, которая требует извлечения всех $m(N)$ атрибутов с номерами в промежутке до максимального номера атрибута, используемого в выражении.

Представленный метод реализован в виде расширения к PostgreSQL и показывает ускорение до 4.7 раз на некоторых запросах.

2.3.4 Greenplum (2016)

В [23] описывается использование динамической компиляции запросов с использованием LLVM в СУБД Greenplum.

Исследователи выделяют два способа применения динамической компиляции в СУБД: компиляция выражений и компиляция запросов — и разделяют некоторые существующие СУБД в соответствии с этим на два класса. Динамическая компиляция в Greenplum используется на уровне выражений и горячих функций, что позволяет расширять множество компилируемых функций инкрементально в рамках существующей СУБД.

Результат профилирования на запросах TPC-H [12] выделил три множества функций-кандидатов для компиляции: функции доступа к атрибутам, функции интерпретатора выражений и агрегатные функции.

Процесс замены каждой отдельной функции состоит из следующих шагов (рис. 6):

1. Замена непосредственных вызовов функции вызовами по указателю, помещённому во внутренние структуры данных СУБД.
2. Компиляция функции-кандидата в LLVM IR и в машинный код во время инициализации запроса.
3. Замена указателей в структурах данных на указатели в результирующий машинный код.
4. Освобождение скомпилированного кода и сбрасывание указателей во время финализации запроса.

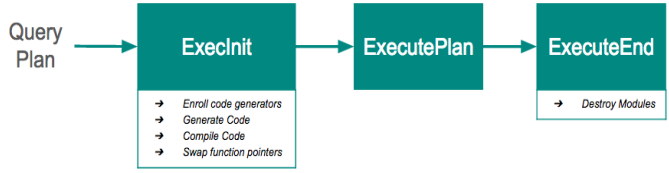


Рис. 6. Компиляция горячих функций в Greenplum
Fig. 6. Hotspot compilation in Greenplum

Метод позволяет компилировать только те обобщённые функции, выполнение которых связано с существенными накладными расходами в профиле работы СУБД, а также вводить определённые ограничения при компиляции каждой конкретной функции, при нарушении которых прерывать компиляцию до шага 3 (замены указателей), что приведёт к использованию при выполнении запроса обобщённой версии соответствующей функции. Недостатком является необходимость сохранения интерфейсов между обобщённым и компилируемым кодом.

На запросе TPC-H Q1 применение динамической компиляции позволило

получить двукратное ускорение, которое складывается в основном из ускорения функций интерпретатора выражений (1.25 раза) и агрегатных функций (1.35 раза).

3. Компиляция запросов

Классической моделью выполнения, используемой в интерпретаторах запросов, является модель Volcano [32], предложенной в одноимённой системе выполнения запросов.

В модели Volcano результатом выполнения каждого оператора в плане выполнения запроса является итератор, предоставляющий доступ к последовательности возвращаемых кортежей. Функцией каждого оператора является формирование последовательности возвращаемых кортежей из последовательностей кортежей, принимаемых на вход от дочерних операторов. Например, оператор агрегации, применённый к отсортированной последовательности входных кортежей, осуществляет группировку кортежей по значению одного или нескольких атрибутов и вычисление аргегатных функций, при этом результирующая последовательность состоит из одного кортежа на каждую группу кортежей во входной последовательности.

Интерфейс итератора состоит из трёх методов:

- метод `open` вызывается для инициализации внутреннего состояния оператора;
- метод `next` вызывается для вычисления и возврата *одного* следующего кортежа в результирующей последовательности оператора, при достижении конца результирующей последовательности `next` возвращает специальный символ конца последовательности;
- метод `close` вызывается для освобождения ресурсов и сброса внутреннего состояния оператора (при этом конец последовательности мог и не быть достигнут).

Модель Volcano упрощает разработку операторов за счёт применения принципа декомпозиции, позволяет распределять операторы по различным вычислительным узлам и естественным образом представлять бесконечные последовательности, поскольку вычисление каждого конкретного члена результирующей последовательности откладывается до того момента, как значения его атрибутов потребуются для вычисления очередного кортежа в родительском операторе.

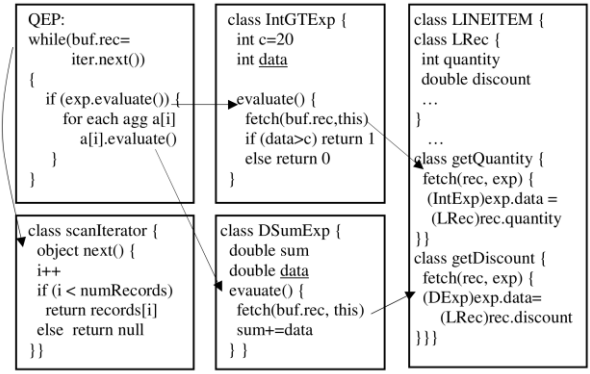
В типичном случае, однако, модель Volcano крайне неэффективна, потому что для получения каждого следующего кортежа требуется многократные сохранение и загрузка состояний операторов в соответствующем поддерева плана выполнения запроса.

Альтернативой является модель явных циклов (push-based модель [33]), в которой дочерний оператор возвращает результирующие кортежи родительскому оператору посредством вызова соответствующего

обработчика, принимающего очередной кортеж и продолжающего его обработку. При этом загрузка состояния дочернего оператора не требуется.
Push-based модель используется в компиляторах запросов, обзору которых посвящён этот раздел, при этом компиляция декомпозируется за счёт введения соответствующих абстракций времени компиляции.

3.1 JamDB (2006)

JamDB [34] — это реляционная СУБД в основной памяти, реализованная на Java. JamDB поддерживает построчное хранение кортежей (N-ary Storage Model) как Java-объектов в куче JVM, первичные и внешние ключи, трёхзначную логику, индексирование.
Для JamDB реализовано два движка запросов: интерпретатор и компилятор. Интерпретатор следует классической модели Volcano и состоит из трёх основных компонент: итераторы, соответствующие вершинам дерева плана, интерпретатор арифметических и логических выражений и интерпретатор агрегаций. Интерпретатор реализован обобщённо с использованием указателей на функции и данные типа `void*`, значения которых становятся известны только во время выполнения, потому что код интерпретатора используется для вычисления многих запросов, не известных во время сборки интерпретатора, и поэтому для каждого конкретного запроса реализация, предоставляемая интерпретатором, далека от оптимальной.



(a) Interpreted Plan

```
class Q1 {  class Q1Agg { double sum1; }
            execute(){
               Lrecords = (LRecord[])records;
               while (i < numRecords) {
                  rec = Lrecords[i];
                  if (rec.quantity > 20) q1agg.sum1+=rec.discount;
                  i++ ;
            }}}
```

(b) Compiled Plan

Рис. 7. Сравнение интерпретируемого и компилируемого планов JamDB
Fig. 7. Interpreted and compiled plans in JamDB

Компилятор запросов использует push-модель выполнения и реализован на основе виртуальной машины JVM. Код генерируется во время обхода дерева плана, начиная с листовых вершин. По завершении обхода фрагменты кода объединяются в одну функцию и компилируются и загружаются в программу как Java-класс. Использование JVM позволяет существенно упростить реализацию компилятора, в частности, генерацию кода и спекулятивную оптимизацию горячих участков, которая позволяет получить производительность, сравнимую с аналогичным кодом, реализованным на языке C. Динамическая компиляция позволяет специализировать код под конкретный SQL-запрос, провести межоператорные оптимизации и девиртуализировать и встроить вызовы, например, арифметических функций в выражениях запроса.

Сравнение интерпретатора запросов с кодом, генерируемым компилятором, приведено на рис. 7.

Исследователи отмечают, что в системах длительного хранения накладные расходы на интерпретацию проявляются не так чётко, как в системах в основной памяти из-за того, что затраты на доступ к данным часто доминируют над затратами на вычисления.

В [34] также отмечается реализация длительного хранения для JamDB, что

потребует структуры данных, позволяющей минимизировать затраты на загрузку и запись данных на диск за счёт разделения данных на страницы или буферы, и соответствующей системы управление буферами в памяти.

Исследователи также отмечают возможность упростить реализацию компилятора за счёт использования технологии Java Emitter Templates (JET) [35].

Экспериментальные результаты на бенчмарке TPC-H показывают, что производительность компилятора запросов, реализованного JamDB, в 2 раза превышает производительность интерпретатора, которая, в свою очередь, значительно превышает производительность коммерческой СУБД DB2 [36]. При этом производительность компилятора запросов превышает производительность планов, реализованных вручную на C/C++.

3.2 DBToaster (2009)

DBToaster [37] — это система потоковой обработки данных, основанная на вычислении и актуализации стационарных запросов и представлений к потоку изменений, добавлений и удалений кортежей в таблицах базы данных.

Исследователи отмечают, что традиционные интерпретаторы и компиляторы запросов не справляются с задачами потоковой обработки данных, поскольку вынуждены вычислять запросы заново при любом изменении данных и не могут использовать результаты предыдущих вычислений.

Подход, предлагаемый в DBToaster, основан на вычислении при помощи алгебраических преобразований для каждого стационарного запроса специального набора разностных запросов на каждую таблицу базы данных и на каждый тип изменения этой таблицы (*update, insert, delete*), которые будут срабатывать на соответствующих типах изменений и выполнять актуализацию результата стационарного запроса к изменению данных, что позволяет избежать повторных вычислений.

Обновление одного стационарного запроса может потребовать введения новых вспомогательных стационарных запросов, возможно параметризованных. В DBToaster автоматически вычисляется результирующее множество стационарных запросов, которые требуется вычислять и обновлять при изменениях таблиц, и множество разностных запросов, нужных для эффективного вычисления, после чего разностные запросы компилируются в машинный код при помощи JIT-компилятора LLVM.

В статье рассматриваются применения DBToaster в контексте алгоритмической торговли и аналитических запросов: потоковая обработка данных и инкрементальное вычисление запросов позволяет ускорить этап загрузки данных, необходимый в классических СУБД.

3.3 Компиляция запросов для LINQ

LINQ (Language Intergrated Query) [38] — это расширение языков платформы .NET, позволяющее встраивать выражения на декларативном языке запросов в программы на хостовом языке. Синтаксис LINQ-запросов во многом позаимствован из SQL, семантика определяется пользователем и реализуется на хостовом языке: источник данных реализует соответствующий интерфейс, в вызовы функций которого транслируются запросы LINQ. Интеграция с хостовым языком позволяет в LINQ-запросах использовать выражения (которые передаются источнику LINQ или в виде функций-объектов CLR, или в виде синтаксических деревьев) и систему типов хостового языка.

Источники данных реализуются в модели Volcano: реализуемый ими интерфейс `IEnumerable` состоит из метода `GetEnumerator`, который возвращает объект-итератор с методами `Current` для доступа к текущему объекту, на который указывает итератор, и `MoveNext` для продвижения итератора к следующему объекту.

LINQ позволяет использовать одни и те же декларативные запросы как для работы с реляционными данными или XML — соответствующие источники встроены в .NET — так и с распределёнными данными [39].

3.3.1 DryadLINQ (2008)

DryadLINQ [39] — это реализация LINQ на основе системы распределённой обработки данных Dryad [40]. Использование декларативного языка запросов упрощает распределённую и параллельную обработку данных и делает возможным применение высокоуровневых оптимизаций.

В DryadLINQ по входному LINQ-запросу строится граф выполнения (Execution Plan Graph) — неориентированный ациклический граф, вершинами которого являются операторы, а дугами — входы и выходы операторов. К графу выполнения применяются статические оптимизации, такие как конвейеризация операторов, удаление лишних вычислений, вставка операторов агрегации и оптимизация ввода–вывода — после чего вершины графа распределяются по вычислительным узлам и для каждого узла генерируются специализированный код сериализации данных для отправки между узлами и LINQ-выражение, осуществляющее обработку данных на самом узле, при этом вычисления также распределяются по доступным ядрам процессора при помощи библиотеки PLINQ [41]. Результирующий код обработки данных и сериализации компилируется во внутреннее представление .NET и передаётся по сети вместе с библиотеками, необходимыми для его работы.

В процессе вычисления запроса также применяются динамические оптимизации для перестроения графа выполнения во время работы исходя из статистической информации об обрабатываемых данных.

3.3.2 Steno (2011)

Steno [42] — это компилятор запросов для LINQ.

Исследователи отмечают, что накладные расходы на интерпретацию и поддержку абстракции времени выполнения (модели итераторов) замедляют (последовательное) выполнение LINQ-запросов в несколько раз по сравнению с эквивалентной реализацией вручную на императивном языке. Накладные расходы представлены в основном виртуальными вызовами функций, необходимыми как для передачи управления коду, реализующему вершины плана выполнения запроса, так и для вызова функций-параметров запроса: предикатов и проекций; и необходимостью сохранения и загрузки состояния операторов для эмуляции сопроцедур, соответствующих операторам запроса, на императивном языке реализации LINQ. Накладные расходы увеличиваются с увеличением уровня вложенности запросов.

Steno решает эту проблему за счёт динамической компиляции запросов в специализированный и оптимизированный императивный код в модели явных циклов.

Компилятор Steno реализован в виде отдельного источника данных, который транслирует абстрактное синтаксическое дерево (АСД) запроса в код на промежуточном представлении QUIL (Query Intermediate Language), на котором проводится оптимизация слияния итераторов (iterator fusion) и трансляция вложенных запросов во вложенные циклы. По коду на промежуточном представлении затем строится АСД языка C#, соответствующее классу с единственным методом, реализующим конкретный запрос, которое затем компилируется в динамическую библиотеку при помощи компилятора C# и загружается в программу. Объект загруженного класса инстанциируется при помощи механизма рефлексии и инициализируется ссылками на используемые в запросе объекты.

Промежуточное представление QUIL служит для сведения большого числа операторов LINQ к шести основным:

- `Src` представляет исходную коллекцию объектов.
- `Trans` выполняет поэлементное преобразование последовательности, параметризуется функцией преобразования.
- `Pred` выполняет фильтрацию последовательности, параметризуется функцией-предикатом.
- `Sink` выполняет материализацию последовательности во временной коллекции в памяти, параметризуется функциями `() → IEnumerable<U>` для инициализации коллекции и `IEnumerable<U> × T → IEnumerable<U>` для обновления коллекции.
- `Agg` выполняет агрегацию коллекции в скалярное значение, параметризуется функциями `() → U` для инициализации скалярного значения и `U × T → U` для обновления скалярного значения.

- `Ret` означает конец выполнения запроса.

Для запроса без подзапросов, корректные программы на QUIL распознаются конечным автоматов на рис. 8.

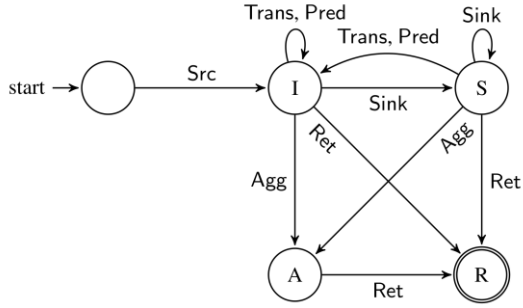


Рис. 8. Конечный автомат операторов QUIL в Steno
Fig. 8. Finite state machine of QUIL operators in Steno

Генерация кода в Steno также следует этому автомату. Каждый переход на рис. 8 связан с определённой процедурой генерации кода — зависящей, таким образом, и от оператора, и от состояния — при этом в каждом состоянии поддерживаются три точки вставки (см. рис. 9):

- пролог цикла (loop prelude),
- тело цикла (loop body),
- эпилог цикла (loop postlude).

Наличие подзапросов — вложенных пар операторов `Src`–`Ret` — делает язык QUIL контекстно-зависимым, поэтому к конечном автомату на рис. 8 добавляется стек (делая его автоматом с магазинной памятью). В стек добавляются тройки точек вставки (пролог, тело цикла, эпилог), соответствующие одному уровню вложенности запроса.

Интереса заслуживает переход из состояния *I* (Iterating) в состояние *R* (Returning) по оператору `Ret`. На самом внешнем уровне вложенности в этом случае генерируется оператор `yield return` языка C#, что является встроенным в язык способом определения итераторов. Если же этот переход является частью вложенного подзапроса, то вместо этого две тройки $(\alpha_i, \mu_i, \omega_i)$ и $(\alpha_{i+1}, \mu_{i+1}, \omega_{i+1})$ на вершине стека заменяются на совмещённую тройку $(\alpha_i, \mu_{i+1}, \omega_i)$ — таким образом происходит совмещение внешнего и вложенного циклов.

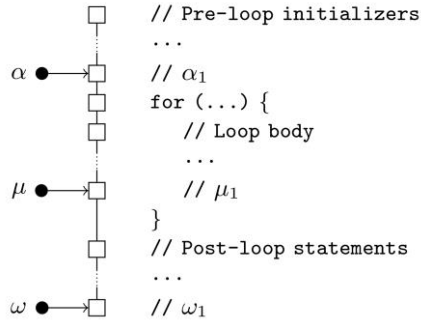


Рис. 9.Точки вставки кода при компиляции запроса в Steno
Fig. 9. Code insertion points in Steno

Метод имеет константные, но довольно большие накладные расходы, связанные с использованием механизмов динамической загрузки кода и рефлексии, которые могут быть сокращены при помощи кеширования загруженных объектов CLR, соответствующих скомпилированным LINQ-запросам, для повторного использования. Накладные расходы также могут быть сокращены при помощи статической компиляции, в частности за счёт совмещения с этапом кодогенерации DryadLINQ [39].

В статье также рассматривается вопрос распределения вычислений на несколько вычислительных узлов и интеграции с DryadLINQ. Метод основан на выделении последовательностей гомоморфных операторов — операторов, которые могут быть применены к элементам последовательности независимо. Гомоморфными операторами QUIL являются операторы `Trans`, `Pred`, а также вложенные запросы (`Src` и `Ret`).

На нескольких синтетических тестах Steno позволяет получить ускорение от 3.32 до 14.1 раз по сравнению со встроенной в .NET реализацией LINQ для массивов в памяти, при этом накладные расходы по сравнению с императивным кодом, оптимизированным вручную, составляют до 53%, но в среднем не превышают 3%. При тестировании с DryadLINQ исследователи получили ускорение до 1.9 раз, но отмечают существенную зависимость от особенностей задачи.

3.4 HIQUE (2010)

В HIQUE [43] рассматривается динамическая компиляция SQL-запросов на основе шаблонов кода.

Алгоритмы, используемые в традиционных СУБД, разработаны в первую очередь для оптимизации использования подсистемы ввода-вывода, что в современных условиях и при современных объёмах доступной памяти недостаточно. В случаях, когда вся база данных или значительная её часть

умещается в оперативной памяти, узким местом производительности является процессор. Исследователи отмечают, что изменение подсистемы хранения для упрощения обработки данных — одно из предложенных решений — слишком радикально меняет архитектуру существующих СУБД. Более ортогональным решением является компиляция запросов в машинный код, которая позволяет значительно сократить накладные расходы на исполнение плана запроса по сравнению с интерпретаторами, реализующими Volcano-модель и основанными на абстракции итератора.

Предлагаемый в статье подход *целостного выполнения запросов* (*holistic query evaluation*) основан на динамической компиляции плана выполнения запроса в программу на некотором языке программирования на основе шаблонов кода, разработанных для каждого оператора. Полученная программа оптимизируется как одно целое, при этом применяются как платформо-зависимые, так и межоператорные оптимизации, недоступные в интерпретаторе из-за существующих границ абстракции между различными операторами. Динамическая компиляция позволяет убрать эти границы абстракции, уменьшить число вызовов функций и увеличить локальность данных, в том числе за счёт более эффективного использования регистров процессора.

HIQUE использует построчное хранение кортежей (N-ary Storage Model). Архитектура подсистемы обработки запросов представлена на рис. 10.

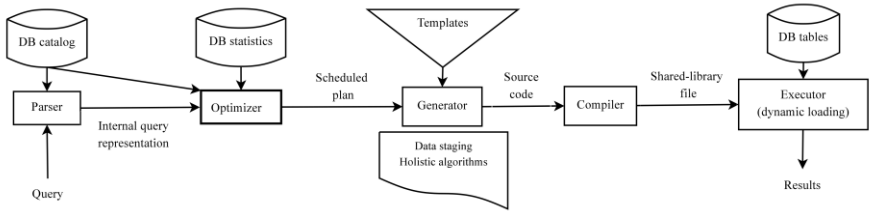


Рис. 10. Архитектура обработки запросов в HIQUE
Fig. 10. Query processing architecture of HIQUE

Результатом работы оптимизатора является топологически отсортированная последовательность операторов, в которой на вход оператор o_i принимает или выход оператора o_j, где $j < i$, или первичную таблицу в базе данных. Компиляция каждого оператора состоит из этапа предобработки, который заключается в определении используемых атрибутов и вычислении смещений, применении предикатов сканирования и вставки дополнительных операторов сортировки и секционирования, где необходимо, и следующего за ним этапа генерации кода, который заключается в инстанциировании соответствующего шаблона кода — в статье приведены примеры шаблонов для некоторых операторов. Промежуточные результаты работы каждого оператора сохраняются во временные таблицы: между каждой парой операторов неявно

вставляется точка материализации.

Результатом компиляции является функция на языке C, которая затем транслируется в объектный код и подгружается в основную программу для выполнения.

Экспериментальная оценка производительности на бенчмарке TPC-H показывает прирост производительности от 4 раз (в сравнении с колоночной СУБД MonetDB [44]) до 167 раз (в сравнении с PostgreSQL).

3.5 HyPer (2011)

HyPer [33, 45] — СУБД в основной памяти, основанная на модели явных циклов и динамической компиляции запросов в машинный код с использованием инфраструктуры LLVM.

Исследователи утверждают, что с ростом объёмов основной памяти производительность СУБД больше определяется эффективностью использования процессора и классическая Volcano-модель, несмотря на свою простоту и гибкость, не позволяет использовать процессор достаточно эффективно из-за нелокального доступа к памяти и ошибок в прогнозировании переходов.

Предлагаемый и реализованный в HyPer подход состоит в определении *границ конвейеризации* (*pipeline boundaries*) — операторов плана выполнения запроса, которые приводят к материализации кортежей, см. рис. 11 — и компиляции операторов в пределах границ конвейеризации в один цикл обработки данных (см. рис. 12), оставляя таким образом границы между операторами только там, где необходимо: материализация кортежей происходит только на границах между циклами, которые определяются планом выполнения запроса и используемыми в нём алгоритмами обработки данных, а на каждой итерации одного цикла происходит применение нескольких операторов к одному кортежу, что позволяет максимально эффективно использовать для хранения атрибутов кортежа регистры процессора.

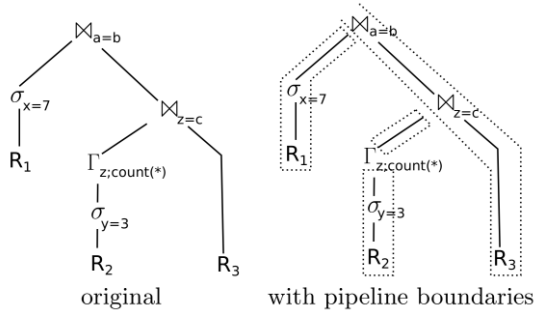


Рис. 11. Определение границ конвейеризации в HyPer
Fig. 11. Pipeline boundaries in HyPer

```
initialize memory of ⋈_{a=b}, ⋈_{c=z}, and Γ_z
for each tuple t in R_1
    if t.x = 7
        materialize t in hash table of ⋈_{a=b}
for each tuple t in R_2
    if t.y = 3
        aggregate t in hash table of Γ_z
for each tuple t in Γ_z
    materialize t in hash table of ⋈_{z=c}
for each tuple t_3 in R_3
    for each match t_2 in ⋈_{z=c}[t_3.c]
        for each match t_1 in ⋈_{a=b}[t_3.b]
            output t_1 ∘ t_2 ∘ t_3
```

Рис. 12. Результат компиляции границ конвейеризации в HyPer (псевдокод)
Fig. 12. Pipeline boundaries compiled to pseudocode

Генерация кода выполняется при помощи методов `produce` и `consume`, сопоставленных операторам плана выполнения запроса. Метод `produce` определён для всех операторов и служит для построения кода, состоящего из одного или нескольких (в количестве листовых вершин) циклов и реализующего функциональность соответствующего поддерева плана выполнения запроса. `produce` рекурсивно вызывается для потомков внутренних операторов плана выполнения запроса. Когда выполнение доходит до листового оператора, генерируется заголовок цикла и вызывается метод `consume` родительского оператора, который служит для генерации кода, обрабатывающего приём очередного кортежа от дочернего оператора. Методы `consume` определены только для внутренних операторов плана в количестве, равном общему числу дуг в дереве плана выполнения запроса. Генерация кода для всего запроса производится посредством вызова метода `produce` на самом внешнем операторе плана. Вызовы `produce` и `consume`, таким образом, производят обход дерева плана в глубину, по завершении которого самый внешний вызов `produce` возвращает код, реализующий в модели явных циклов весь исходный запрос.

Интерфейс, представленный функциями `produce` и `consume`, подобен интерфейсу итераторов в Volcano-модели и упрощает независимую разработку реляционных операторов, но существует только во время компиляции запроса — результат компиляции не содержит вызовов этих функций и представлен всего несколькими императивными циклами (см. рис. 12).

Для генерации машинного кода в HyPer используется инфраструктура LLVM [11] и промежуточное представление LLVM IR. Исследователи отмечают следующие преимущества над альтернативными технологиями:

- приемлемое время компиляции (по сравнению с языками высокого уровня);

- достаточную низкоуровневость и контроль над результирующим машинным кодом (например, в части использования флагов арифметического переполнения);
- наличие неограниченного множества виртуальных регистров;
- платформо-независимость;
- типобезопасность;
- наличие большого числа встроенных оптимизаций.

Для максимизации продуктивности, упрощения поддержки и уменьшения времени компиляции существенная часть внутренней логики операторов реализована на C++ и вызывается из сгенерированного кода. Тем не менее, для обеспечения максимальной производительности необходимо, чтобы наиболее горячие участки кода, в особенности части, ответственные за обработку кортежей в циклах в пределах границ конвейеризации, были реализованы на LLVM IR во избежание накладных расходов на вызовы функций на C++, в частности на сохранение и загрузку регистров согласно используемому соглашению о вызовах.

В [33] исследователи отмечают некоторые особенности генерации кода для оптимизации производительности, в частности:

- загрузку атрибутов из памяти немного раньше, чем необходимо, для сокрытия задержек работы с памятью;
- использование циклов с постусловием вместо циклов с предусловием для улучшения предсказания переходов.

В статье также описывается обработка в цикле нескольких кортежей одновременно с использованием SIMD-инструкций, доступных в современных процессорах.

[45] дополнительно описывает представление SQL-значений и операций в коде на LLVM IR, при этом как правило одному SQL-значению соответствует несколько LLVM-значений (например, число или указатель и флаг *null* или длина строки) и одной операции — несколько элементарных операций LLVM (например, сложение чисел и проверка переполнения); абстракцию времени компиляции для генерации графа потока управления (условий и циклов) и абстракцию времени компиляции для представления кортежей (в сжатом или дематериализованном виде). Использование абстракций времени компиляции существенно упрощает разработку и поддержку динамического компилятора и при этом не приводит к дополнительным затратам времени выполнения.

На синтетических запросах динамическая компиляция в HyPer позволяет получить ускорение до 2–8 раз в сравнении с интерпретацией в зависимости от запроса, в некоторых случаях до 3000 раз, а на бенчмарке TPC-H — до 3.7 раз в сравнении с колоночной СУБД VectorWise [46].

3.6 Hekaton (2013)

Hekaton [47, 48] — это расширение SQL Server [49], включающее в себя

таблицы и индексы для данных в основной памяти, неблокирующие структуры данных для эффективного многопоточного выполнения и MVCC и компилятор запросов.

Исследователи отмечают, что производительность СУБД при выполнении OLTP-запросов зависит от трёх основных параметров: количества выполняемых инструкций, количества циклов на инструкцию и коэффициента масштабируемости — причём последние два параметра суммарно могут дать только прирост производительности в 3–4 раза. Для ускорения СУБД в 10–100 раз необходимо, таким образом, существенно сократить количество выполняемых инструкций.

Компилятор запросов Hekaton принимает на вход структуры результат работы оптимизатора и генерирует код на промежуточном представлении PIT (Pure Imperative Tree), на основе которого после серии преобразований генерируется код на языке C, компилируемый и загружаемый в процесс SQL Server для выполнения.

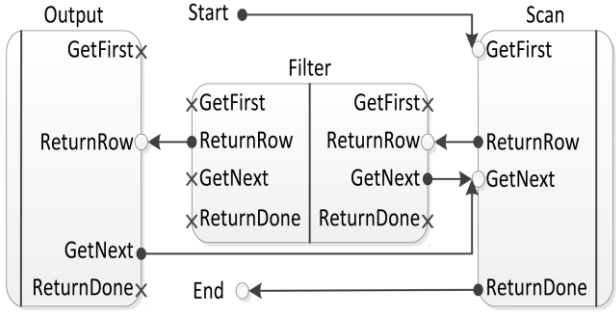


Рис. 13. Межоператорный поток управления в Hekaton
Fig. 13. Interoperator control flow in Hekaton

Операторы в Hekaton реализуют интерфейс, состоящий из функций `GetFirst`, `GetNext`, `ReturnRow` и `ReturnDone`, что позволяет комбинировать операторы произвольным образом согласно плану выполнения запроса.

В Hekaton реализована модель явных циклов при помощи объединения кода операторов в одну функцию и трансляции переходов между ними — вызовов функций `GetFirst`, `GetNext` и т. д. — в безусловные переходы между соответствующими операторам базовыми блоками (см. рис. 13).

Результирующий код (рис. 14) из соображений безопасности не содержит идентификаторов исходного запроса, что ещё больше затрудняет экспертный анализ, но проведённое исследование показало, что генерация кода в одну функцию позволяет минимизировать как число выполняемых инструкций, так и размер бинарного кода.

Сравнение с интерпретатором запросов показало сокращение количества выполняемых инструкций до 10–15 раз.

```
    /*Seek*/
l_17:; /*seek.GetFirst*/
    hr = (HkCursorHashGetFirst(
        cur_15 /*[dbo].[Customers].[Customers_pk]*/ ,
        (context->Transaction),
        0, 0, 1,
        ((struct HkRow const**)(&rec1_16 /*[dbo].[Customers].[Customers_pk]*/ ))));
    if ((FAILED(hr)))
    {
        goto l_2 /*exit*/ ;
    }
l_20:; /*seek.1*/
    if ((hr == 0))
    {
        goto l_14 /*filter.child.ReturnRow*/ ;
    }
    else
    {
        goto l_12 /*query.ReturnDone*/ ;
    }
l_21:; /*seek.GetNext*/
    hr = (HkCursorHashGetNext(
        cur_15 /*[dbo].[Customers].[Customers_pk]*/ ,
        (context->ErrorObject),
        ((struct HkRow const**)(&rec1_16 /*[dbo].[Customers].[Customers_pk]*/ ))));
    if ((FAILED(hr)))
    {
        goto l_2 /*exit*/ ;
    }
    goto l_20 /*seek.1*/ ;
    /*Filter*/
l_14:; /*filter.child.ReturnRow*/
    result_22 = ((rec1_16 /*[dbo].[Customers].[Customers_pk]*/ ->hkc_1 /*[Id]*/ ) ==
        ((long)((valueArray[1 /*@id*/ ]).SignedIntData)));
    if (result_22)
    {
        goto l_13 /*output.child.ReturnRow*/ ;
    }
    else
    {
        goto l_21 /*seek.GetNext*/ ;
    }
```

Рис. 14. Пример результирующего кода в Hekaton
Fig. 14. Compiled code in Hekaton

3.7 MemSQL (2016)

MemSQL [50] — это СУБД в основной памяти, в которой реализована компиляция запросов в машинный код при помощи инфраструктуры LLVM.
Исследователи выделяют три направления оптимизации производительности программных систем:

- оптимизация подсистемы ввода–вывода: планировка (scheduling), перемещение данных, распределение нагрузки;
- оптимизация потребления памяти вычислительных узлов;
- оптимизация использования ЦПУ для вычислений — в первую очередь сокращение количества исполняемых инструкций.

В дисковых СУБД оптимизация ввода–вывода имеет гораздо более важное значение, потому что ввод–вывод в дисковых СУБД является самым «узким местом». В СУБД в основной памяти, напротив, оптимизации памяти и

вычислений выходят на первый план и компиляция запросов в эффективный машинный код может дать существенный прирост производительности.

В MemSQL для компиляции используются высокоуровневое промежуточное представление MPL (MemSQL Plan Language) и промежуточное представление среднего уровня MBC (MemSQL Bit Code), которое затем транслируется в низкоуровневое промежуточное представление LLVM IR.

Исследователи отмечают, что компиляция запросов даёт как количественные, так и качественные преимущества для конечного пользователя: например, возможность использования систем визуализации и мониторинга реального времени.

3.8 Компиляция запросов в компиляторе, разрабатываемом в ИСП РАН (2016)

В ИСП РАН разрабатывается расширение [51, 52] к СУБД PostgreSQL, реализующее динамическую компиляцию запросов на основе инфраструктуры LLVM.

Модель итераторов, используемая в PostgreSQL, сопряжена с существенными накладными расходами, связанными с неявными вызовами функций `next` и сопутствующими ошибками предсказания переходов и необходимостью сохранения состояния операторов между вызовами функций `next`.

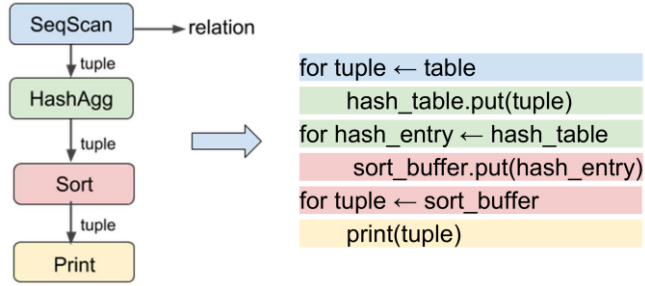


Рис. 15. Переход к модели явных циклов
Fig. 15. Push-based execution model

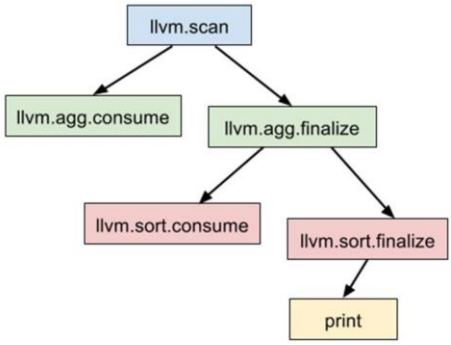
В статье описывается переход от модели итераторов к модели явных циклов (см. рис. 15) и алгоритм генерации кода в этой модели (см. рис. 16), основанный на сопоставленных каждому оператору функциях `consume` и `finalize`. Во время генерации кода совершается обход плана выполнения запроса, при котором каждая вершина получает на вход функции `consume` и `finalize` от родительского оператора и генерирует код, в котором функция `consume` вызывается для каждого очередного возвращаемого родительскому оператору кортежа, а функция `finalize` — после возврата последнего кортежа. Генерации кода для внутренних операторов состоит в генерации функций `consume` и `finalize` (содержащих вызовы родительских функций

`consume` и `finalize`) по одной на дочерний оператор и вызова генераторов дочерних операторов. В листовых операторах происходит генерация основных циклов обхода таблицы или индекса и вызов переданных функций `consume` и `finalize`. Процесс генерации начинается с передачи генератору самого внешнего оператора плана выполнения запроса функции `consume`, осуществляющей передачу очередного кортежа результата, и пустой функции `finalize`.

```
llvm.sort.consume = Sort.consume()
llvm.sort.finalize = Sort.finalize(print, null)
    llvm.agg.consume = HashAgg.consume()
    llvm.agg.finalize = HashAgg.finalize(llvm.sort.consume, llvm.sort.finalize)
        llvm.scan = SeqScan(llvm.agg.consume, llvm.agg.finalize)
```

Рис. 16. Генерация кода в модели явных циклов
Fig. 16. Code generation for push-based execution model

На рис. 17 показан результат компиляции. После встраивания получается код как на рис. 15 справа.




```
llvm.sort.consume(tuple) {
    sort_buffer.put(tuple)
}
llvm.sort.finalize() {
    for tuple ← sort_buffer
        print(tuple)
}
llvm.agg.consume(tuple) {
    hash_table.put(tuple)
}
llvm.agg.finalize() {
    for hash_entry ← hash_table
        llvm.sort.consume(tuple)
    llvm.sort.finalize()
}
llvm.scan() {
    for tuple ← table
        llvm.agg.consume(tuple)
    llvm.agg.finalize()
}
```

Рис. 17. Результирующий код в модели явных циклов
Fig. 17. Compiled code in push-based model

На бенчмарке TPC-H метод позволяет получить ускорение до 5.5 раз.

4. Методы, основанные на специализации кода

4.1 ToasterBooster (2013)

ToasterBooster [53] — это расширение DBToaster [37] для динамической компиляции запросов при помощи технологии многоуровневой компиляции, реализованной во фреймворке LMS (Lightweight Modular Staging) [54] для языка Scala.

Многоуровневая компиляция в LMS реализуется посредством

параметризованных типов Scala, LMS предоставляет параметризованный тип `Rep`. Если значения типа `T` относятся к уровню компиляции `N`, то значения типа `Rep[T]` относятся к уровню компиляции `N+1`. Например, отличие между типами `Rep[T] -> Rep[T]` и `Rep[T -> T]` состоит в том, что значениями первого типа являются функции на множестве объектов промежуточного представления, в то время как значениями второго типа являются функции *на языке* промежуточного представления. Во время выполнения операций над значениями типа `Rep[T]` осуществляется генерация и оптимизация кода на языке промежуточного представления, реализующего соответствующие операции над значениями типа `T`.

Явное разделение уровней выполнения позволяет провести автоматическую специализацию кода и данных следующего уровня к коду и данным текущего уровня: встраивание функций, разворачивание циклов, подстановка константных значений.

Архитектура системы представлена на рис. 18. После оптимизации на языках промежуточного представления M3 и K3 запрос попадает в генератор кода на языке Scala, в котором проводятся дополнительные оптимизации на языке промежуточного представления LMS и генерируется код на Scala или C++, реализующий разностные функции исходного запроса.

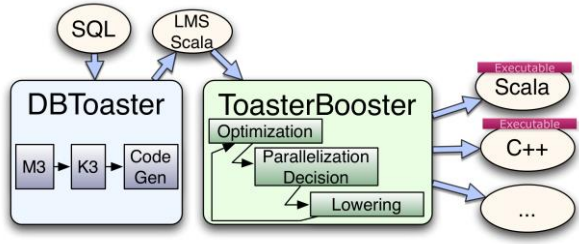


Рис. 18. Архитектура ToasterBooster
Fig. 18. ToasterBooster architecture

В статье также рассматривается реализация оптимизации дефорестации в LMS, которая заключается в удалении границ абстракции между коллекциями, используемыми в генерируемом коде, при помощи генераторов, предоставляющих интерфейс коллекции времени компиляции (что выражается в замене типов `Rep[Collection[T]]` типами `Generator[T]`) и позволяющих совместить различные операции над коллекцией и получить на выходе эффективный императивный код.

Экспериментальная оценка на синтетических запросах показывает ускорение до 5 раз относительно компилятора запросов, встроенного в DBToaster.

4.2 LegoBase (2014)

LegoBase [55] — компилятор запросов, реализованный на языке Scala.

Для оптимизации производительности в нём используется технология многоуровневой компиляции и фреймворк многоуровневой компиляции LMS [54].

В статье рассматриваются оптимизации, реализованные на уровне промежуточного представления LMS специально для LegoBase, такие как:

- оптимизация структур данных — частичное вычисление адресов и параметров обращений, замена ассоциативных структур данных (хеш-таблиц) линейными (массивами);
- изменение модели выполнение с Volcano-модели (pull-based) на push-based — исследователи отмечают, что реализация автоматического изменения модели выполнения оказалась возможна только благодаря достаточно высокому уровню абстракции реализации операторов на языке Scala;
- удаление лишней материализации кортежей на границе операторов;
- изменение схемы расположения данных таблиц: замена построчного хранения поколоночным.

В результате компиляции и оптимизации LMS генерирует код на языке Scala. Для достижения максимальной производительности в LegoBase реализована дополнительная трансляция результирующего кода в код на языке C. Исследователи отмечают, что трансляция в C потребовала решения двух проблем:

- Трансляции вызовов библиотечных функций и структур данных. Для решения этой проблемы в LegoBase используется библиотека GLib [56].
- Трансляции модели управления памятью. В языке Scala используется сборщик мусора, поэтому во время трансляции кода со Scala в C необходимо обеспечить своевременное освобождение неиспользуемой памяти во избежание утечек памяти. Для решения этой проблемы в LegoBase используется ручное управление памятью.

Полученный код на языке C компилируется при помощи Clang [31] и исполняется.

В статье также рассматривается возможность перекомпиляции кода во время выполнения при изменении параметров СУБД. Например, при изменении параметра, отвечающего за логирование этапов выполнения запроса, требуется перекомпилировать функции, в которых производится проверка этого параметра, с целью добавления или удаления соответствующих вызовов. Подход, используемый в LegoBase, позволяет реализовывать компиляторы языков запросов на языке высокого уровня без потерь в производительности. Многоуровневая компиляция позволяет автоматически удалять из результирующего кода на промежуточном представлении используемые при разработке абстракции. Исследователи отмечают следующие преимущества по сравнению с другими методами компиляции запросов:

- типобезопасность (обеспечивается использованием высокоуровневого языка);
- относительная простота реализации и поддержки;
- поддержка межоператорных оптимизаций (по сравнению с методами, основанными на шаблонах);
- поддержка высокоуровневых оптимизаций (обеспечивается более высоким уровнем промежуточного представления по сравнению с, например, промежуточным представлением LLVM).

Эксперименты с использованием бенчмарка TPC-H показывают, что LegoBase достигает большей производительности, чем СУБД HyPer [33], в которой для компиляции запросов используется LLVM [11]. Сравнение с HyPer указывает на упущенные оптимизационные возможности, вызванные слишком низким уровнем абстракции, предоставляемым LLVM IR, и реализованные при помощи использования более высокоуровневого промежуточного представления LMS.

4.3 DexterDB (2015)

В [57] представлен метод компиляции запросов на основе специализации исходного кода СУБД на уровне промежуточного представления LLVM, реализованный для DexterDB [58].

Исследователи выделяют два основных подхода к вычислению запросов в СУБД: классический подход с использованием интерпретатора запросов и динамическая компиляция на основе шаблонов. Проблемой классического подхода является производительность, а проблемой второго — сложность разработки, расширения и поддержки. Подход на основе специализации кода представляет собой расширение классического подхода и совмещает и простоту разработки, свойственную интерпретаторам запросов, и высокую степень специализации и оптимизации, свойственную динамическим компиляторам.

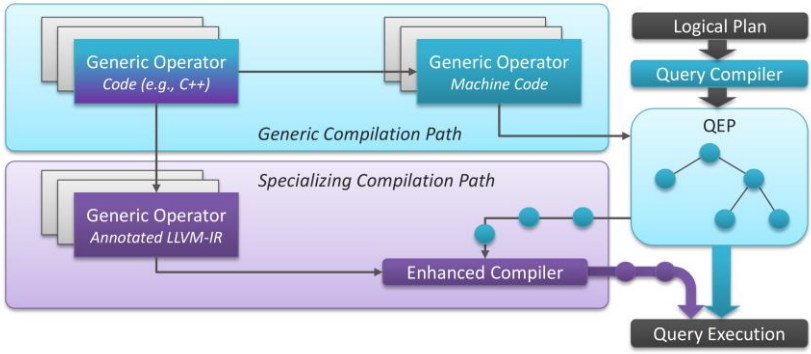


Рис. 19. Специализация запросов в DexterDB

Fig. 19. Generic and specialized compilation paths in DexterDB

В предлагаемом подходе операторы плана выполнения реализуются на высокоуровневом языке и могут быть использованы как непосредственно (см. *Generic Compilation Path* на рис. 19) — этот случай эквивалентен классическому подходу — так и с использованием специализатора. В последнем случае (*Specializing Compilation Path*) исходный код СУБД, являющийся обобщённой реализацией оператора, предварительно компилируется в LLVM IR (см. рис. 20) и во время выполнения автоматически специализируется к параметрам оператора, соответствующим конкретному запросу.

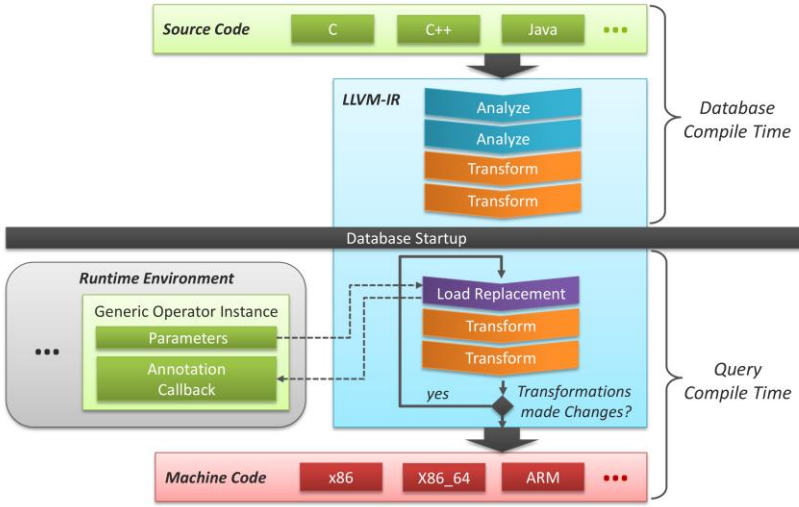


Рис. 20. Компиляция кода операторов в DexterDB
Fig. 20. Compilation pipeline in DexterDB

Специализация основана на частичном вычислении инструкций загрузки констант из памяти (Load Replacement), которое выполняется в цикле вместе со встроенными оптимизациями LLVM — в статье приведён перечень и последовательность применения встроенных оптимизаций — до достижения неподвижной точки.

Load replacement состоит в замене инструкций загрузки из памяти (`load`), соответствующих параметрам оператора, непосредственными значениями этих параметров. Особенности реализации операторов в DexterDB позволяют сократить поиск потенциальных инструкций для замены: в DexterDB каждый оператор моделируется классом, а его параметры — полями соответствующих объектов. Таким образом, доступ к параметрам оператора всегда происходит по смещениям относительно указателя `this` на объект класса, который в методах оператора является неявным параметром, — что в LLVM IR

соответствует определённым цепочкам инструкций `load`, `bitcast` и `getelementptr`. На каждой итерации во время специализации обнаруживаются все такие цепочки с известным смещением относительно `this`, соответствующие указателям на константные параметры, и заменяются на значения соответствующих параметров — значения по соответствующим указателям.

Определение того, соответствует ли адрес относительно значения `this` константному параметру, происходит в DexterDB также во время выполнения посредством вызова метода `bool memoryIsConstant(void *mem)`, который с байтовой точностью определяет множество виртуальных адресов констант в памяти процесса СУБД.

Оценка производительности на бенчмарке SSB [59] показывает ускорение в 1.2–1.6 раза по сравнению с классической схемой (*Generic Compilation Path*).

4.4 LB2-Spatial (2016)

LB2-Spatial [60] — это СУБД с поддержкой геопространственных данных на основе СУБД LB2 [61], которая, в свою очередь, является форком LegoBase [55].

Поддержка геопространственных данных требует специализированных вычислительно сложных алгоритмов и пространственных индексов, таких как R-деревья, k-d деревья и quad-деревья. В LB2-Spatial эти алгоритмы реализованы с использованием фреймворка многоуровневой компиляции LMS [54] для Scala, что позволяет для каждого запроса автоматически генерировать специализированные версии этих алгоритмов.

В статье также затрагивается вопрос расширения СУБД PostgreSQL и Spark / SparkSQL компиляцией геопространственных функций, а также распределения вычислений на несколько узлов.

На тестовом запросе LB2-Spatial позволяет получить ускорение до 14 раз относительно геопространственного расширения PostGIS [62] к PostgreSQL.

4.5 Flare (2017)

Flare [63] — это компилятор запросов для Apache Spark, совместимый на уровне интерфейса со Spark SQL [14], от встроенного компилятора которого он отличается в следующем:

- Во Flare запросы компилируются не в байткод JVM, а в машинный код, что позволяет избежать связанных с JVM накладных расходов.
- Во Flare запросы компилируются целиком, а не на уровне отдельных операторов или групп операторов, как в Spark SQL, что позволяет повысить эффективность использования вычислительных узлов за счёт сокращения накладных расходов на поддержку абстракции и передачи данных между узлами. Исследователи отмечают, что на практике вертикальное масштабирование в пределах нескольких

сильных узлов позволяет решить задачи проще и дешевле, чем горизонтальное масштабирование кластера на много слабых узлов.

- Flare поддерживает компиляцию и встраивание пользовательских функций на нескольких предметно-ориентированных языках, непрозрачных для оптимизатора и компилятора Spark SQL.

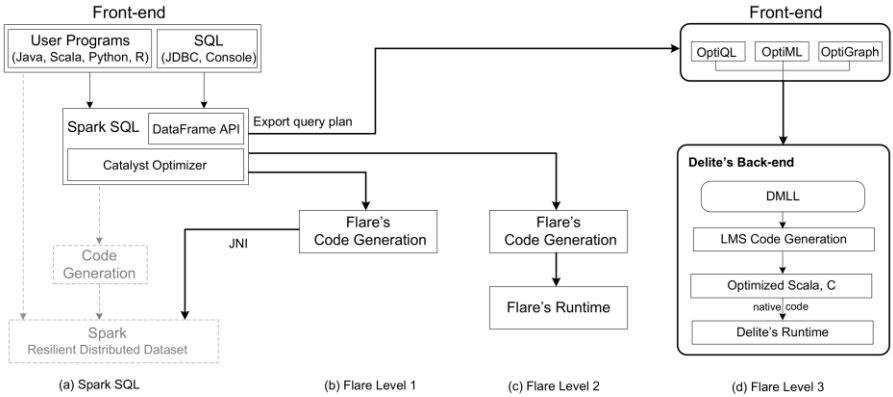


Рис. 21. Архитектура Flare
Fig. 21. Flare architecture

Архитектура Flare (рис. 21) состоит из нескольких уровней. Первый уровень заменяет генератор байткода JVM для операторов запроса на генератор, реализованный с использованием фреймворка LMS [54], промежуточное представление которого транслируется в C, компилируется в машинный код и подгружается в адресное пространство процесса Spark. Для вызова к среде выполнения Spark из результирующего кода используются механизм JNI (Java Native Interface).

Второй уровень содержит компилятор запросов в модели явных циклов и заменяет части среды выполнения Spark. Выполнение в несколько потоков поддерживается при помощи OpenMP [64]. Выполнение на нескольких вычислительных узлах не поддерживается и требует расширения Spark механизмами перехвата управления, которые позволили бы обеспечить запуск и координацию Flare на нескольких узлах. Flare также компилирует код загрузки и дематериализации данных таблиц с использованием информации о схеме таблиц и используемого алгоритма кодирования.

Третий уровень добавляет поддержку пользовательских функций на предметно-ориентированных языках OptiQL [65], OptiML [66], OptiGraph [65], OptiMesh [65], реализованных на Scala, что позволяет комбинировать различные парадигмы программирования в одном запросе. Для этого используется фреймворк Delite [67] и внутреннее представление DMLL [68], в которое генерируется и план выполнения, полученный от встроенного в Spark SQL оптимизатора запросов Catalyst, и

пользовательские функции, используемые в запросе.
Экспериментальная оценка на бенчмарке TPC-H показывает время работы, сравнимое с HyPer [33] и ускорение относительно Spark SQL до 89 раз.

5. Заключение

В статье рассмотрены работы в области динамической компиляции запросов, как для распределённых [1], так и для нераспределённых систем (см. табл. 1).
Можно заметить, что во всех рассмотренных динамических компиляторах запросов для распределённых систем, за исключением Flare и DryadLINQ, и некоторых компиляторах для PostgreSQL компиляция реализуется на уровне выражений и горячих функций. Здесь показателен подход Greenplum, основанный на замене указателей в структурах данных, изначально указывающих на обобщённый код, во время подготовки запроса к выполнению, и немного более общий подход микроспециализации и горячей замены [27], основанный на динамическом изменении указателей в коде программы в зависимости от значений определённых переменных.
Компилятор запросов Flare разработан для распределённой системы Spark, но поддерживает выполнение одного запроса строго на одном узле, поэтому принципиально не отличается от других рассмотренных компиляторов запросов для нераспределённых систем (в частности LegoBase).
DryadLINQ является единственным из рассмотренных компиляторов запросов, который транслирует один запрос в код для нескольких вычислительных узлов. Для этого после серии статических оптимизизаций запрос разделяется на части (LINQ-подвыражения), каждая из которых компилируется в промежуточное представление .NET и отправляется на множество узлов, которые во время выполнения будут отвечать за соответствующую часть запроса. Интересно, что само отображение подвыражений на вычислительные узлы в DryadLINQ также может меняться во время выполнения.
Можно выделить два основных подхода к реализации компиляторов запросов: реализация генератора на основе модели явных циклов (JamDB, HyPer, Hekaton, MemSQL, Steno, компилятор ИСП РАН) и реализация двухуровневого компилятора на основе фреймворка LMS (LegoBase, Flare, ToasterBooster, LB2-Spatial).
Преобразование в модель явных циклов (push-модель) явно сформулировано в [33]. Оно состоит в генерации по плану выполнения запроса императивного кода со вложенными циклами, в котором одна из листовых вершин плана выполнения (сканирование таблицы или индекса) является самым внешним циклом, а из самого внутреннего цикла вызывается код, отвечающий за приём кортежей самой внешней (корневой) вершиной плана. В разных компиляторах можно найти разные способы реализации этой модели (HyPer, Hekaton, Steno, компилятор ИСП РАН).
В HIQUE реализована более простая модель, основанная на генерации кода

операторов в топологическом порядке с материализацией во временные таблицы на каждом шаге. Тем не менее, даже такая модель позволила получить существенное ускорение относительно коммерческих СУБД.

Компиляторы, основанные на специализации кода, к разновидности которой можно отнести многоуровневую компиляцию в том виде, в котором она реализована в LMS, позволяют существенно упростить разработку и поддержку компиляторов за счёт того, что код ядра компилятора является по сути кодом обобщённого интерпретатора плана выполнения, при этом накладные расходы, связанные с интерпретацией, сокращаются используемым при компиляции фреймворком (LegoBase, Flare, ToasterBooster, LB2-Spatial) или набором оптимизаций (DexterDB). Можно отметить некоторое сходство этих подходов недавней линии исследований в области реализации виртуальных машин [69, 70].

Табл. 1. Сводная таблица компиляторов запросов
Table 1. Summary of just-in-time query compilers

Система обработки данных	**Распределённая**	**Компилятор запросов**	**Единица компиляции**
Impala	да	[7, 8]	Выражения
Greenplum	да	[23]	Выражения
Spark	да	Spark SQL [14]	Выражения
		Flare [60]	Запрос (LMS)
Dryad	да	DryadLINQ [38]	Распределённый запрос
PostgreSQL	нет	Микроспециализация [25–27]	Выражения
		Butterstein, Grust [30]	Выражения
		Компилятор ИСП РАН [6, 49, 50]	Запрос (push-модель)
JamDB	нет	[34]	Запрос (push-модель)
HyPer	нет	[33, 43]	Запрос (push-модель)
SQL Server	нет	Hekaton [45, 46]	Запрос (push-модель)
MemSQL	нет	[48]	Запрос

Система обработки данных	Распределённая	Компилятор запросов	Единица компиляции
			(push-модель)
LINQ-совместимые / .NET	—	Steno [41]	Запрос (push-модель)
HIQUE	нет	[42]	Запрос (шаблоны)
DBToaster	нет	[36]	Разностные запросы
		ToasterBooster [51]	Запрос (LMS)
DBX[1]	нет	LegoBase [53]	Запрос (LMS)
LB2	нет	LB2-Spatial [58]	Запрос (LMS)
DexterDB	нет	[55]	Запрос (специализация)

Список литературы

[1]. Кузнецов, С. Основы современных баз данных. http://citforum.ru/database/osbd/contents.shtml (дата обращения 18.05.2017).
[2]. Chamberlin, D.D., Astrahan, M.M., et al. 1981. A history and evaluation of System R. *Commun. ACM*. 24, 10 (1981), 632–646.
[3]. Wade, B.W. 2012. Compiling SQL into System/370 machine language. *IEEE Annals of the History of Computing*. 34, 4 (2012), 49–50.
[4]. Greer, R. 1999. Daytona and the fourth-generation language Cymbal. *SIGMOD 1999, proceedings ACM SIGMOD international conference on management of data* (Philadelphia, Pennsylvania, USA, 1999), 525–526.
[5]. Copeland, G.P., Khoshafian, S. 1985. A decomposition storage model. *Proceedings of the 1985 ACM SIGMOD international conference on management of data* (Austin, Texas, USA, 1985), 268–279.
[6]. Шарыгин, Е., Бучацкий, Р., Скворцов, Л., Жуйков, Р., Мельник, Д. 2016. Динамическая компиляция выражений в SQL-запросах для СУБД PostgreSQL. *Труды ИСП РАН*. 28, 4 (2016), 217–240.
[7]. Kornacker, M., Behm, A., et al. 2015. Impala: A modern, open-source SQL engine for Hadoop. *CIDR 2015, seventh biennial conference on innovative data systems research* (Asilomar, CA, USA, 2015).
[8]. Wanderman-Milne, S., Li, N. 2014. Runtime code generation in Cloudera Impala. *IEEE Data Eng. Bull.* 37, 1 (2014), 31–37.
[9]. Apache Hadoop, open-source software for reliable, scalable, distributed computing. The

[1] Неназванная коммерческая СУБД

Apache Software Foundation; http://hadoop.apache.org (дата обращения 19.06.2017).
[10]. Apache HBase, the Hadoop database, a distributed, scalable, big data store. The Apache Software Foundation; https://hbase.apache.org (дата обращения 19.06.2017).
[11]. Lattner, C., Adve, V.S. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. *2nd IEEE / ACM international symposium on code generation and optimization (CGO 2004)* (San Jose, CA, USA, 2004), 75–88.
[12]. TPC-H, an ad-hoc, decision support benchmark. Transaction Processing Performance Council; http://www.tpc.org/tpch (дата обращения 25.05.2017).
[13]. Apache Spark, a fast and general engine for large-scale data processing. The Apache Software Foundation; https://spark.apache.org (дата обращения 19.06.2017).
[14]. Armbrust, M., Xin, R.S., et al. 2015. Spark SQL: Relational data processing in Spark. *Proceedings of the 2015 ACM SIGMOD international conference on management of data* (Melbourne, Victoria, Australia, 2015), 1383–1394.
[15]. PostgreSQL, an open source object-relational database system. The PostgreSQL Global Development Group; https://www.postgresql.org (дата обращения 16.06.2017).
[16]. PostgreSQL derived databases. PostgreSQL wiki; https://wiki.postgresql.org/wiki/PostgreSQL_derived_databases (дата обращения 20.06.2017).
[17]. ToroDB Stampede, a database bridging NoSQL and SQL. 8Kdata; https://www.torodb.com (дата обращения 19.06.2017).
[18]. Vertica, a “shared nothing” distributed analytical database. Hewlett Packard Enterprise Development; https://www.vertica.com (дата обращения 19.06.2017).
[19]. AgensGraph, a highly optimized, multi-model graph database for the modern, complex connected data environment. Bitnine Global; http://www.agensgraph.com (дата обращения 19.06.2017).
[20]. Tan, C. 2015. Vitesse DB: 100% Postgres, 100X faster for analytics. Presented at the 2nd South Bay PostgreSQL Meetup; https://docs.google.com/presentation/d/1R0po7_Wa9fym5U9Y5qHXGlUi77nSda2LlZXPuAxtd-M/pub (дата обращения 20.06.2017).
[21]. ParAccel 2010. *The ParAccel analytic database: A technical overview*. ParAccel, Inc. https://marketplace.informatica.com/mpresources/docs/ParAccel-Technical-Overview-White-Paper%202011.pdf (дата обращения 20.06.2017).
[22]. Gupta, A., Agarwal, D., et al. 2015. Amazon Redshift and the case for simpler data warehouses. *Proceedings of the 2015 ACM SIGMOD international conference on management of data* (Melbourne, Victoria, Australia, 2015), 1917–1923.
[23]. Armenatzoglou, N., Rajaraman, K.J., et al. 2016. Improving query execution speed via code generation. *Pivotal Engineering Journal; http://engineering.pivotal.io/post/codegen-gpdb-qx* (дата обращения 20.06.2017). (2016).
[24]. DeepgreenDB, a scalable MPP data warehouse solution derived from the open source Greenplum database project. Vitesse Data; http://vitessedata.com/deepgreen-db (дата обращения 19.06.2017).
[25]. Zhang, R., Debray, S., Snodgrass, R.T. 2012. Micro-specialization: dynamic code specialization of database management systems. *10th annual IEEE/ACM international symposium on code generation and optimization, CGO 2012* (San Jose, CA, USA, 2012), 63–73.
[26]. Zhang, R., Snodgrass, R.T., Debray, S. 2012. Micro-specialization in DBMSes. *IEEE 28th international conference on data engineering (ICDE 2012)* (Washington, DC, USA (Arlington, Virginia), 2012), 690–701.

[27]. Zhang, R., Snodgrass, R.T., Debray, S. 2012. Application of micro-specialization to query evaluation operators. *Workshops proceedings of the IEEE 28th international conference on data engineering, ICDE 2012* (Arlington, VA, USA, 2012), 315–321.
[28]. *Callgrind: A call-graph generating cache and branch prediction profiler*. Valgrind Developers; http://valgrind.org/docs/manual/cl-manual.html (дата обращения 8.06.2017).
[29]. TPC-C, an on-line transaction processing benchmark. Transaction Processing Performance Council; http://www.tpc.org/tpcc (дата обращения 8.06.2017).
[30]. Butterstein, D., Grust, T. 2016. Precision performance surgery for PostgreSQL: LLVM-based expression compilation, just in time. *PVLDB*. 9, 13 (2016), 1517–1520.
[31]. Clang: A C language family frontend for LLVM. The LLVM Foundation; https://clang.llvm.org/ (дата обращения 1.06.2017).
[32]. Graefe, G. 1994. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.* 6, 1 (1994), 120–135.
[33]. Neumann, T. 2011. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550.
[34]. Rao, J., Pirahesh, H., Mohan, C., Lohman, G. 2006. Compiled query execution engine using jVM. *Proceedings of the 22Nd international conference on data engineering* (Washington, DC, USA, 2006), 23.
[35]. Java Emitter Templates, part of Eclipse Modeling Framework. Eclipse Foundation; http://www.eclipse.org/modeling/m2t/?project=jet (дата обращения 7.06.2017).
[36]. DB2, a relational database. IBM Corporation;
https://www.ibm.com/analytics/us/en/technology/db2 (дата обращения 21.06.2017).
[37]. Ahmad, Y., Koch, C. 2009. DBToaster: A SQL compiler for high-performance delta processing in main-memory databases. *PVLDB*. 2, 2 (2009), 1566–1569.
[38]. Box, D., Hejlsberg, A. 2007. LINQ: .NET language-integrated query. *Microsoft Developer Network; https://msdn.microsoft.com/en-us/library/bb308959.aspx (дата обращения 8.06.2017)*. (2007).
[39]. Yu, Y., Isard, M., et al. 2008. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. *8th USENIX symposium on operating systems design and implementation, OSDI 2008, proceedings* (San Diego, California, USA, 2008), 1–14.
[40]. Dryad data-parallel processing framework. Microsoft; https://www.microsoft.com/en-us/research/project/dryad (дата обращения 19.06.2017).
[41]. Duffy, J. 2007. A query language for data parallel programming: Invited talk. *Proceedings of the 2007 workshop on declarative aspects of multicore programming* (New York, NY, USA, 2007), 50.
[42]. Murray, D.G., Isard, M., Yu, Y. 2011. Steno: Automatic optimization of declarative queries. *Proceedings of the 32Nd aCM sIGPLAN conference on programming language design and implementation* (New York, NY, USA, 2011), 121–131.
[43]. Krikellas, K., Viglas, S., Cintra, M. 2010. Generating code for holistic query evaluation. *Proceedings of the 26th international conference on data engineering, ICDE 2010* (Long Beach, California, USA, 2010), 613–624.
[44]. MonetDB, an open source column-oriented database. MonetDB B.V. https://www.monetdb.org (дата обращения 21.06.2017).
[45]. Neumann, T., Leis, V. 2014. Compiling database queries into machine code. *IEEE Data Eng. Bull.* 37, 1 (2014), 3–11.
[46]. Actian Vector (former VectorWise), a relational vectorized columnar analytic database. Actian Corporation; https://www.actian.com/analytic-database/vector-smp-analytic-database

(дата обращения 19.06.2017).
[47]. Diaconu, C., Freedman, C., et al. 2013. Hekaton: SQL server's memory-optimized OLTP engine. *Proceedings of the ACM SIGMOD international conference on management of data, SIGMOD 2013* (New York, NY, USA, 2013), 1243–1254.
[48]. Freedman, C., Ismert, E., Larson, P. 2014. Compilation in the Microsoft SQL Server Hekaton engine. *IEEE Data Eng. Bull*. 37, 1 (2014), 22–30.
[49]. SQLServer, a relational database. Microsoft; https://www.microsoft.com/en-us/sql-server (дата обращения 19.06.2017).
[50]. Paroski, D. 2016. Code generation: The inner sanctum of database performance. *High Scalability; http://highscalability.com/blog/2016/9/7/code-generation-the-inner-sanctum-of-database-performance.html* (дата обращения 19.06.2017). (2016).
[51]. Бучацкий, Р., Шарыгин, Е., Скворцов, Л., Жуйков, Р., Мельник, Д., Баев, Р. 2016. Динамическая компиляция SQL-запросов для СУБД PostgreSQL. *Труды ИСП РАН*. 28, 6 (2016), 37–48.
[52]. Melnik, D., Buchatskiy, R., Zhuykov, R., Sharygin, E. 2017. JIT-compiling SQL queries in PostgreSQL using LLVM. Presented at PGCon 2017; https://www.pgcon.org/2017/schedule/attachments/467_PGCon%202017-05-26%2015-00%20ISPRAS%20Dynamic%20Compilation%20of%20SQL%20Queries%20in%20PostgreSQL%20Using%20LLVM%20JIT.pdf (дата обращения 19.06.2017).
[53]. Dashti, M., Abadi, R. 2013. Database query optimization using compilation techniques. (2013).
[54]. Rompf, T., Odersky, M. 2010. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. *Generative programming and component engineering, proceedings of the ninth international conference on generative programming and component engineering, GPCE 2010* (Eindhoven, The Netherlands, 2010), 127–136.
[55]. Klonatos, Y., Koch, C., Rompf, T., Chafi, H. 2014. Building efficient query engines in a high-level language. *PVLDB*. 7, 10 (2014), 853–864.
[56]. GLib, a general-purpose utility library. The GNOME Foundation; https://developer.gnome.org/glib/ (дата обращения 2.06.2017).
[57]. Hänsch, C., Kissinger, T., Habich, D., Lehner, W. 2015. Plan operator specialization using reflective compiler techniques. *Datenbanksysteme für business, technologie und web (BTW), 16. Fachtagung des GI-fachbereichs «datenbanken und informationssysteme» (DBIS), 4.-6.3.2015, proceedings* (Hamburg, Germany, 2015), 363–382.
[58]. Dexter: Dresden index for transactional access on emerging technologies. Dresden Database Systems Group; http://wwwdb.inf.tu-dresden.de/research-projects/projects/dexter/ (дата обращения 17.01.2013).
[59]. O'Neil, P., O'Neil, B., Chen, X. 2009. Star Schema Benchmark. (2009).
[60]. Tahboub, R.Y., Rompf, T. 2016. On supporting compilation in spatial query engines: (Vision paper). *Proceedings of the 24th ACM SIGSPATIAL international conference on advances in geographic information systems, GIS 2016* (Burlingame, California, USA, 2016), 9:1–9:4.
[61]. Rompf, T. LB2, a fork of LegoBase. https://github.com/TiarkRompf/legobase-micro (дата обращения 16.06.2017).
[62]. PostGIS, a spatial database extender for PostgreSQL. PostGIS Project Steering Committee; http://postgis.net (дата обращения 21.06.2017).
[63]. Essertel, G.M., Tahboub, R.Y., Decker, J.M., Brown, K.J., Olukotun, K., Rompf, T. 2017. Flare: Native compilation for heterogeneous workloads in Apache Spark. *CoRR*. abs/1703.08219, (2017).

[64]. OpenMP, an API specification for parallel programming. OpenMP Architecture Review Board; http://www.openmp.org (дата обращения 19.06.2017).
[65]. Sujeeth, A.K., Rompf, T., et al. 2013. Composition and reuse with compiled domain-specific languages. *ECOOP 2013 - object-oriented programming - 27th european conference, proceedings* (Montpellier, France, 2013), 52–78.
[66]. Sujeeth, A.K., Lee, H., et al. 2011. OptiML: An implicitly parallel domain-specific language for machine learning. *Proceedings of the 28th international conference on machine learning, ICML 2011* (Bellevue, Washington, USA, 2011), 609–616.
[67]. Brown, K.J., Sujeeth, A.K., et al. 2011. A heterogeneous parallel framework for domain-specific languages. *2011 international conference on parallel architectures and compilation techniques, PACT 2011* (Galveston, TX, USA, 2011), 89–100.
[68]. Brown, K.J., Lee, H., et al. 2016. Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. *Proceedings of the 2016 international symposium on code generation and optimization, CGO 2016* (Barcelona, Spain, 2016), 194–205.
[69]. Würthinger, T., Wöß, A., Stadler, L., Duboscq, G., Simon, D., Wimmer, C. 2012. Self-optimizing AST interpreters. *Proceedings of the 8th symposium on dynamic languages, DLS '12* (Tucson, AZ, USA, 2012), 73–82.
[70]. Würthinger, T., Wimmer, C., et al. 2013. One VM to rule them all. *ACM symposium on new ideas in programming and reflections on software, onward! 2013, part of SPLASH '13* (Indianapolis, IN, USA, 2013), 187–204.

Survey of Just-in-Time Query Compilation Methods

[1,2] *E. Y. Sharygin <eush@ispras.ru>*
[1] *R. A. Buchatskiy <ruben@ispras.ru>*
[1] *Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.*
[2] *Lomonosov Moscow State University, CMC Department bldg. 52, GSP-1, Leninskie Gory, Moscow, 119991, Russia*

Abstract. Data processing systems have been traditionally optimized for I/O, mainly because, until pretty recently, disk storage has been the most affordable type of storage and the most prevalent one. This is not necessarily the case today, particularly in the world of big data analytics. As the problems posed by data analytics become more commonplace, efficient CPU utilization becomes the new bottleneck. Just-in-time query compilation is a promising solution to this challenge that is currently being applied both in academic studies and across the industry. This paper is a survey of just-in-time query compilation methods sampled from the literature available on the subject. All methods are broadly categorized into expression compilation and hotspot methods, whole-query compilation methods, and specialization-based methods. A number of query processors are identified within confines of each category, various methods, architectures, and significant results are described. Finally, we conclude with an overview of most general approaches to query compilation that we identified.

Keywords: just-in-time compilation; query engines; query languages; expression compilation; hotspot compilation; holistic compilation; push-model; code specialization.

DOI: 10.15514/ISPRAS-2017-29(3)-11

For citation: Sharygin E. Y., Buchatskiy R. A. Survey of Just-in-Time Query Compilation Methods. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue. 3, 2017, pp. 179-224 (in Russian). DOI: 10.15514/ISPRAS-2017-29(3)-11

References

[1]. Kuztetsov, S. Foundations of Modern Database Systems. http://citforum.ru/database/osbd/contents.shtml, accessed 18.05.2017 (in Russian).
[2]. Chamberlin, D.D., Astrahan, M.M., et al. 1981. A history and evaluation of System R. *Commun. ACM*. 24, 10 (1981), 632–646.
[3]. Wade, B.W. 2012. Compiling SQL into System/370 machine language. *IEEE Annals of the History of Computing*. 34, 4 (2012), 49–50.
[4]. Greer, R. 1999. Daytona and the fourth-generation language Cymbal. *SIGMOD 1999, proceedings ACM SIGMOD international conference on management of data* (Philadelphia, Pennsylvania, USA, 1999), 525–526.
[5]. Copeland, G.P., Khoshafian, S. 1985. A decomposition storage model. *Proceedings of the 1985 ACM SIGMOD international conference on management of data* (Austin, Texas, USA, 1985), 268–279.
[6]. Sharygin E.Y., Buchatskiy R.A., Skvortsov L.V., Zhuykov R.A., Melnik D.M. Dynamic compilation of expressions in SQL queries for PostgreSQL. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 4, 2016. pp. 217-240 (in Russian). DOI: 10.15514/ISPRAS-2016-28(4)-13
[7]. Kornacker, M., Behm, A., et al. 2015. Impala: A modern, open-source SQL engine for Hadoop. *CIDR 2015, seventh biennial conference on innovative data systems research* (Asilomar, CA, USA, 2015).
[8]. Wanderman-Milne, S., Li, N. 2014. Runtime code generation in Cloudera Impala. *IEEE Data Eng. Bull*. 37, 1 (2014), 31–37.
[9]. Apache Hadoop, open-source software for reliable, scalable, distributed computing. The Apache Software Foundation; http://hadoop.apache.org (accessed 19.06.2017).
[10]. Apache HBase, the Hadoop database, a distributed, scalable, big data store. The Apache Software Foundation; https://hbase.apache.org (accessed 19.06.2017).
[11]. Lattner, C., Adve, V.S. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. *2nd IEEE / ACM international symposium on code generation and optimization (CGO 2004)* (San Jose, CA, USA, 2004), 75–88.
[12]. TPC-H, an ad-hoc, decision support benchmark. Transaction Processing Performance Council; http://www.tpc.org/tpch (accessed 25.05.2017).
[13]. Apache Spark, a fast and general engine for large-scale data processing. The Apache Software Foundation; https://spark.apache.org (accessed 19.06.2017).
[14]. Armbrust, M., Xin, R.S., et al. 2015. Spark SQL: Relational data processing in Spark. *Proceedings of the 2015 ACM SIGMOD international conference on management of data* (Melbourne, Victoria, Australia, 2015), 1383–1394.
[15]. PostgreSQL, an open source object-relational database system. The PostgreSQL Global Development Group; https://www.postgresql.org (accessed 16.06.2017).
[16]. PostgreSQL derived databases. PostgreSQL wiki; https://wiki.postgresql.org/wiki/PostgreSQL_derived_databases (accessed 20.06.2017).
[17]. ToroDB Stampede, a database bridging NoSQL and SQL. 8Kdata; https://www.torodb.com (accessed 19.06.2017).

[18]. Vertica, a “shared nothing” distributed analytical database. Hewlett Packard Enterprise Development; https://www.vertica.com (accessed 19.06.2017).
[19]. AgensGraph, a highly optimized, multi-model graph database for the modern, complex connected data environment. Bitnine Global; http://www.agensgraph.com (accessed 19.06.2017).
[20]. Tan, C. 2015. Vitesse DB: 100% Postgres, 100X faster for analytics. Presented at the 2nd South Bay PostgreSQL Meetup; https://docs.google.com/presentation/d/1R0po7_Wa9fym5U9Y5qHXGlUi77nSda2LlZXPuAxtd-M/pub (accessed 20.06.2017).
[21]. ParAccel 2010. *The ParAccel analytic database: A technical overview*. ParAccel, Inc. https://marketplace.informatica.com/mpresources/docs/ParAccel-Technical-Overview-White-Paper%202011.pdf (accessed 20.06.2017).
[22]. Gupta, A., Agarwal, D., et al. 2015. Amazon Redshift and the case for simpler data warehouses. *Proceedings of the 2015 ACM SIGMOD international conference on management of data* (Melbourne, Victoria, Australia, 2015), 1917–1923.
[23]. Armenatzoglou, N., Rajaraman, K.J., et al. 2016. Improving query execution speed via code generation. *Pivotal Engineering Journal; http://engineering.pivotal.io/post/codegen-gpdb-qx* (accessed 20.06.2017). (2016).
[24]. DeepgreenDB, a scalable MPP data warehouse solution derived from the open source Greenplum database project. Vitesse Data; http://vitessedata.com/deepgreen-db (accessed 19.06.2017).
[25]. Zhang, R., Debray, S., Snodgrass, R.T. 2012. Micro-specialization: dynamic code specialization of database management systems. *10th annual IEEE/ACM international symposium on code generation and optimization, CGO 2012* (San Jose, CA, USA, 2012), 63–73.
[26]. Zhang, R., Snodgrass, R.T., Debray, S. 2012. Micro-specialization in DBMSes. *IEEE 28th international conference on data engineering (ICDE 2012)* (Washington, DC, USA (Arlington, Virginia), 2012), 690–701.
[27]. Zhang, R., Snodgrass, R.T., Debray, S. 2012. Application of micro-specialization to query evaluation operators. *Workshops proceedings of the IEEE 28th international conference on data engineering, ICDE 2012* (Arlington, VA, USA, 2012), 315–321.
[28]. *Callgrind: A call-graph generating cache and branch prediction profiler*. Valgrind Developers; http://valgrind.org/docs/manual/cl-manual.html (accessed 8.06.2017).
[29]. TPC-C, an on-line transaction processing benchmark. Transaction Processing Performance Council; http://www.tpc.org/tpcc (accessed 8.06.2017).
[30]. Butterstein, D., Grust, T. 2016. Precision performance surgery for PostgreSQL: LLVM-based expression compilation, just in time. *PVLDB*. 9, 13 (2016), 1517–1520.
[31]. Clang: A C language family frontend for LLVM. The LLVM Foundation; https://clang.llvm.org/ (accessed 1.06.2017).
[32]. Graefe, G. 1994. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.* 6, 1 (1994), 120–135.
[33]. Neumann, T. 2011. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550.
[34]. Rao, J., Pirahesh, H., Mohan, C., Lohman, G. 2006. Compiled query execution engine using jVM. *Proceedings of the 22Nd international conference on data engineering* (Washington, DC, USA, 2006), 23.
[35]. Java Emitter Templates, part of Eclipse Modeling Framework. Eclipse Foundation; http://www.eclipse.org/modeling/m2t/?project=jet (accessed 7.06.2017).

[36]. DB2, a relational database. IBM Corporation; https://www.ibm.com/analytics/us/en/technology/db2 (accessed 21.06.2017).
[37]. Ahmad, Y., Koch, C. 2009. DBToaster: A SQL compiler for high-performance delta processing in main-memory databases. *PVLDB*. 2, 2 (2009), 1566–1569.
[38]. Box, D., Hejlsberg, A. 2007. LINQ: .NET language-integrated query. *Microsoft Developer Network; https://msdn.microsoft.com/en-us/library/bb308959.aspx* (accessed 8.06.2017). (2007).
[38]. Yu, Y., Isard, M., et al. 2008. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. *8th USENIX symposium on operating systems design and implementation, OSDI 2008, proceedings* (San Diego, California, USA, 2008), 1–14.
[40]. Dryad data-parallel processing framework. Microsoft; https://www.microsoft.com/en-us/research/project/dryad (accessed 19.06.2017).
[41]. Duffy, J. 2007. A query language for data parallel programming: Invited talk. *Proceedings of the 2007 workshop on declarative aspects of multicore programming* (New York, NY, USA, 2007), 50.
[42]. Murray, D.G., Isard, M., Yu, Y. 2011. Steno: Automatic optimization of declarative queries. *Proceedings of the 32Nd aCM sIGPLAN conference on programming language design and implementation* (New York, NY, USA, 2011), 121–131.
[43]. Krikellas, K., Viglas, S., Cintra, M. 2010. Generating code for holistic query evaluation. *Proceedings of the 26th international conference on data engineering, ICDE 2010* (Long Beach, California, USA, 2010), 613–624.
[44]. MonetDB, an open source column-oriented database. MonetDB B.V. https://www.monetdb.org (accessed 21.06.2017).
[45]. Neumann, T., Leis, V. 2014. Compiling database queries into machine code. *IEEE Data Eng. Bull.* 37, 1 (2014), 3–11.
[46]. Actian Vector (former VectorWise), a relational vectorized columnar analytic database. Actian Corporation; https://www.actian.com/analytic-database/vector-smp-analytic-database (accessed 19.06.2017).
[47]. Diaconu, C., Freedman, C., et al. 2013. Hekaton: SQL server's memory-optimized OLTP engine. *Proceedings of the ACM SIGMOD international conference on management of data, SIGMOD 2013* (New York, NY, USA, 2013), 1243–1254.
[48]. Freedman, C., Ismert, E., Larson, P. 2014. Compilation in the Microsoft SQL Server Hekaton engine. *IEEE Data Eng. Bull.* 37, 1 (2014), 22–30.
[49]. SQLServer, a relational database. Microsoft; https://www.microsoft.com/en-us/sql-server (accessed 19.06.2017).
[50]. Paroski, D. 2016. Code generation: The inner sanctum of database performance. *High Scalability; http://highscalability.com/blog/2016/9/7/code-generation-the-inner-sanctum-of-database-performance.html* (accessed 19.06.2017). (2016).
[51]. Buchatskiy R.A., Sharygin E.Y., Skvortsov L.V., Zhuykov R.A., Melnik D.M., Baev R.V. Dynamic compilation of SQL queries for PostgreSQL. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 6, 2016, pp. 37-48 (in Russian). DOI: 10.15514/ISPRAS-2016-28(6)-3.
[52]. Melnik, D., Buchatskiy, R., Zhuykov, R., Sharygin, E. 2017. JIT-compiling SQL queries in PostgreSQL using LLVM. Presented at PGCon 2017; https://www.pgcon.org/2017/schedule/attachments/467_PGCon%202017-05-26%2015-00%20ISPRAS%20Dynamic%20Compilation%20of%20SQL%20Queries%20in%20PostgreSQL%20Using%20LLVM%20JIT.pdf (accessed 19.06.2017).
[53]. Dashti, M., Abadi, R. 2013. Database query optimization using compilation techniques.

(2013).
[54]. Rompf, T., Odersky, M. 2010. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. *Generative programming and component engineering, proceedings of the ninth international conference on generative programming and component engineering, GPCE 2010* (Eindhoven, The Netherlands, 2010), 127–136.
[55]. Klonatos, Y., Koch, C., Rompf, T., Chafi, H. 2014. Building efficient query engines in a high-level language. *PVLDB*. 7, 10 (2014), 853–864.
[56]. GLib, a general-purpose utility library. The GNOME Foundation; https://developer.gnome.org/glib/ (accessed 2.06.2017).
[57]. Hänsch, C., Kissinger, T., Habich, D., Lehner, W. 2015. Plan operator specialization using reflective compiler techniques. *Datenbanksysteme für business, technologie und web (BTW), 16. Fachtagung des GI-fachbereichs «datenbanken und informationssysteme» (DBIS), 4.-6.3.2015, proceedings* (Hamburg, Germany, 2015), 363–382.
[58]. Dexter: Dresden index for transactional access on emerging technologies. Dresden Database Systems Group; http://wwwdb.inf.tu-dresden.de/research-projects/projects/dexter/ (accessed 17.01.2013).
[59]. O'Neil, P., O'Neil, B., Chen, X. 2009. Star Schema Benchmark. (2009).
[60]. Tahboub, R.Y., Rompf, T. 2016. On supporting compilation in spatial query engines: (Vision paper). *Proceedings of the 24th ACM SIGSPATIAL international conference on advances in geographic information systems, GIS 2016* (Burlingame, California, USA, 2016), 9:1–9:4.
[61]. Rompf, T. LB2, a fork of LegoBase. https://github.com/TiarkRompf/legobase-micro (accessed 16.06.2017).
[62]. PostGIS, a spatial database extender for PostgreSQL. PostGIS Project Steering Committee; http://postgis.net (accessed 21.06.2017).
[63]. Essertel, G.M., Tahboub, R.Y., Decker, J.M., Brown, K.J., Olukotun, K., Rompf, T. 2017. Flare: Native compilation for heterogeneous workloads in Apache Spark. *CoRR*. abs/1703.08219, (2017).
[64]. OpenMP, an API specification for parallel programming. OpenMP Architecture Review Board; http://www.openmp.org (accessed 19.06.2017).
[65]. Sujeeth, A.K., Rompf, T., et al. 2013. Composition and reuse with compiled domain-specific languages. *ECOOP 2013 - object-oriented programming - 27th european conference, proceedings* (Montpellier, France, 2013), 52–78.
[66]. Sujeeth, A.K., Lee, H., et al. 2011. OptiML: An implicitly parallel domain-specific language for machine learning. *Proceedings of the 28th international conference on machine learning, ICML 2011* (Bellevue, Washington, USA, 2011), 609–616.
[67]. Brown, K.J., Sujeeth, A.K., et al. 2011. A heterogeneous parallel framework for domain-specific languages. *2011 international conference on parallel architectures and compilation techniques, PACT 2011* (Galveston, TX, USA, 2011), 89–100.
[68]. Brown, K.J., Lee, H., et al. 2016. Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. *Proceedings of the 2016 international symposium on code generation and optimization, CGO 2016* (Barcelona, Spain, 2016), 194–205.
[69]. Würthinger, T., Wöß, A., Stadler, L., Duboscq, G., Simon, D., Wimmer, C. 2012. Self-optimizing AST interpreters. *Proceedings of the 8th symposium on dynamic languages, DLS '12* (Tucson, AZ, USA, 2012), 73–82.
[70]. Würthinger, T., Wimmer, C., et al. 2013. One VM to rule them all. *ACM symposium on new ideas in programming and reflections on software, onward! 2013, part of SPLASH '13*

(Indianapolis, IN, USA, 2013), 187–204.

О задаче приближенного нахождения максимальной двудольной клики

Н.Н. Кузюрин
Интитут системного программирования РАН,
109004, Москва, ул. А. Солженицына, 25

Аннотация. Задача о нахождении большой "спрятанной" клики в случайном графе и ее аналог для двудольных графов являются объектами рассмотрения в данной заметке.

Ключевые слова: случайный граф; большая спрятанная клика; сложность нахождения

DOI: 10.15514/ISPRAS-2017-29(3)-12

Для цитирования: Кузюрин Н.Н. О задаче приближенного нахождения максимальной двудольной клики. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 225-232. DOI: 10.15514/ISPRAS-2017-29(3)-12

1. Введение

Изучение свойств случайных дискретных структур является важным направлением дискретной математики, в последние годы привлекающим внимание все большего числа исследователей во всем мире. В целом, роль вероятностных методов в современном развитии дискретной математики трудно переоценить: эти методы используются как при изучении различных свойств случайных структур (графов, гиперграфов и т.д.), так и при доказательствах существования комбинаторных объектов с заданными свойствами и построении эффективных алгоритмов. Заметную роль здесь играют несколько трудных задач, которые привлекли внимание специалистов и не поддаются решению. Одной таких задач является задача о нахождении большой "спрятанной" клики в случайном графе. Именно эта задача и ее аналог для двудольных графов являются объектами рассмотрения в данной заметке.

2. Обзор известных результатов

Прежде чем говорить о сложности задач для случайных графов рассмотрим, что известно о трудности нахождения их приближенных решений при анализе по худшему случаю.

Определение. *Кликой в графе* G *называется множество вершин, любые две из которых соединены ребром. Максимально возможное число вершин в клике* G *обозначается* $w(G)$ *и называется размером максимальной клики.*

Сформулируем сейчас аналог задачи о максимальной клике для случая двудольных графов. Напомним, что граф $G = (V, E)$ называется двудольным, если его множество вершин можно разбить на два непустых подмножества V_1 и V_2 так, что $V = V_1 \cup V_2$ и в графе нет ребра, оба конца которого принадлежат одному из множеств V_1 или V_2.

Определение. *Двудольной кликой в двудольном графе* $G = (V_1, V_2, E)$ *называется полный двудольный подграф* (U, W) *графа* $G, U \subseteq V_1, W \subseteq V_2$

Известно несколько вариантов задачи о двудольной клике.

Сбалансированная двудольная клика. В этом случае $|V_1| = |V_2|$ и $|U| = |W|$. Известно, что задача нахождения максимальной двудольной сбалансированной клики (т.е. максимизации $|U| = |W|$) NP-полна [6].

Другой вариант задачи – это двудольная клика с максимальным числом ребер, т.е требуется максимизировать $|U| \cdot |W|$. Эта задача, как показано сравнительно недавно, также NP-полна [3].

Известно, что задача о максимальной клике в произвольном графе NP-полна [6]. В результате длительных исследований удалось доказать, что задача о максимальной клике очень плохо аппроксимируется. Напомним, что мультипликативной ошибкой алгоритма A называется максимум по всем входам данной длины отношения стоимости решения, найденного алгоритмом A, к стоимости оптимального решения. Наилучший известный полиномиальный алгоритм для нахождения максимальной клики гарантирует мультипликативную ошибку не более $O(n(\log\log n)^2/(\log n)^3)$ [4]. Отметим, что аппроксимация с ошибкой n тривиальна, так, что полученный результат не намного улучшает тривиальную оценку. Более того, Хостад [10] показал, что для задачи о максимальной клике не существует полиномиального приближенного алгоритма, имеющего ошибку менее $n^{1-\delta}$, для любого фиксированного $\delta > 0$ (в предположении $RP \neq NP$).

Известно, однако, что для ряда задач на графах их аналоги для двудольных графов решаются значительно проще. Возможно именно этим объясняется тот факт, что результаты о трудности аппроксимации для задачи СБАЛАНСИРОВАННАЯ ДВУДОЛЬНАЯ КЛИКА гораздо слабее результатов для задачи о клике в произвольном графе. Приведем сейчас известные

результаты о трудности аппроксимации задачи о сбалансированной двудольной клике.

1. В работе [1] доказано, что задача о сбалансированной двудольной клике не аппроксимируема в полиномиальное время с мультипликативной ошибкой r, для некоторой константы r>0 (в предположении $P \neq NP$).
2. В работе [2] доказано, что задача о сбалансированной двудольной клике не аппроксимируема в полиномиальное время с мультипликативной ошибкой $O(n^a)$, для некоторой константы $a > 0$ в предположении справедливости следующей гипотезы:

   Гипотеза. Пусть d – достаточно большая константа, не зависящая от n. Не существует полиномиального алгоритма, который отвергает почти все случайные 3-КНФ формулы с n булевыми переменными и dn скобками, причем никогда не отвергает выполнимую формулу (доля которых, как известно, стремится к нулю при достаточно большом d).
3. В работе [11] доказано, что если задача о сбалансированной двудольной клике аппроксимируема в полиномиальное время с мультипликативной ошибкой не более $2^{(\log n)^a}$, для любого $a > 0$, то задача 3-ВЫПОЛНИМОСТЬ может быть решена за время $2^{n^{3/4}+\varepsilon}$ для любого $\varepsilon > 0$.

3. Основной результат

В этой же работе доказано, что если задача о сбалансированной двудольной клике аппроксимируема в полиномиальное время с мультипликативной ошибкой не более некоторой константы, то задача о максимальной клике в графе может быть аппроксимирована в полиномиальное время с ошибкой не более $n/2^{c\sqrt{\log n}}$ для некоторой константы $c > 0$.

Нами доказана теорема о трудности аппроксимации задачи о сбалансированной двудольной клике в предположении о трудности нахождения "спрятанной большой клики" в случайном графе (см., например, [12, 13]).

Далее мы даем необходимые определения и формулируем результат.

Обозначим через $G_{n,1/2}$ случайный граф, в котором все ребра появляются независимо с вероятностью 1/2. Дадим формулировку задачи о спрятанной клике в случайном графе.

3.1 Спрятанная k-клика

Дан случайный граф $G \in G_{n,1/2}$, выбираем в нем случайное подмножество из k вершин и соединяем их ребрами, образуя полный подграф (клику). Требуется найти спрятанную клику.

Известно, что с вероятностью стремящейся к единице при $n \to \infty$ граф $G \in G_{n,1/2}$ не содержит клик размера больше $2\log n$, и максимальная клика имеет размер $(2 + o(1))\log n$. Однако, неизвестно полиномиального алгоритма нахождения клики размера $c\log n$ при $c > 1$. В [7] даже высказано предположение о том, что задача нахождения такой клики вычислительно трудна. Косвенное подтверждение получено в [9] для одного класса популярных алгоритмов (алгоритмов, построенных на эвристиках "моделирования отжига").

В настоящее время, несмотря на довольно интенсивные исследования, неизвестно полиномиального алгоритма решения задачи о спрятанной k-клике при $k = o(\sqrt{n})$ [13], что привело к тому, что была сформулирована гипотеза о трудности ее решения (чем больше параметр k, тем сильнее эта гипотеза). Отметим, что ряд результатов уже получен некоторыми исследователями в предположении справедливости этой гипотезы [12]. Более того, она рассматривается и как один из криптографических примитивов [5].

Рассмотрим по аналогии со спрятанной кликой задачу о спрятанной двудольной клике в двудольном случайном графе. Пусть $|V_1| = |V_2| = n$. Образуем случайный двудольный (n, n)-граф следующим образом: выберем каждое ребро между V_1 и V_2 с вероятностью 1/2 независимо от других ребер. Обозначим этот класс случайных графов через $GB(n, n, 1/2)$.

3.2 Спрятанная двудольная (*k*,*k*)-клика

Дан случайный граф $G \in GB(n, n, 1/2)$, выбираем в V_1 случайное подмножество из k вершин, затем в V_2 случайное подмножество из k вершин и соединяем их ребрами, образуя полный двудольный (k, k)-подграф (двудольную клику). Требуется найти спрятанную клику.

Нетрудно показать, что с вероятностью стремящейся к единице при $n \to \infty$ граф $G \in GB(n, n, 1/2)$ не содержит двудольной (k, k)-клики с $k > 2\log n$.

Справедлива следующая

Теорема. *Пусть* $k > c\log n$, $c > 4$ *– константа. Если существует полиномиальный вероятностный алгоритм нахождения двудольной* (k, k)*-клики, спрятанной в случайном графе* $G \in GB(n, n, 1/2)$*, то существует и*

полиномиальный вероятностный алгоритм нахождения встроенной $2k$-клики в случайном графе $G \in G_{2n,1/2}$.

Доказательство. Опишем простую сводимость задачи СПРЯТАННАЯ КЛИКА к задаче СПРЯТАННАЯ ДВУДОЛЬНАЯ КЛИКА. Итак, пусть нам дан граф $G \in G_{2n,1/2}$ содержащий клику из $2k$ вершин.

Разобъем вершины G на два подмножества V_1 и V_2, причем вершина попадает в каждый класс с вероятностью 1/2. Образуем двудольный граф $GB = (V_1, V_2, E)$, включив в E только ребра соединяющие V_1 и V_2 в G. Довольно очевидно, что по построению полученный двудольный граф является случайным (за исключением встроенной в него $2k$-клики) с вероятностью появления ребра 1/2.

При таком разбиении вершин посмотрим, как разделились вершины $2k$-клики. Оценим снизу вероятность P_k того, что они разделились поровну, т.е. и в V_1 и в V_2 попало ровно по k вершин и, кроме того, $|V_1| = |V_2| = n$. Имеем:

$$P_k = \frac{\binom{2k}{k} \cdot \binom{2n-2k}{n-k}}{2^{2n}}.$$

Воспользуемся неравенством:

$$\binom{2m}{m} \geq c \cdot \frac{2^{2m}}{\sqrt{m}}.$$

Получим:

$$P_k \geq c^2 \frac{2^{2k} \cdot 2^{2n-2k}}{2^{2n}\sqrt{4k(n-k)}} =$$

$$= \frac{c^2}{\sqrt{4k(n-k)}} \geq \frac{c^2}{n}.$$

Отсюда сразу вытекает следствие о неаппроксимируемости задачи о максимальной сбалансированной двудольной клике в двудольном графе.

Следствие. *Пусть задача СПРЯТАННАЯ k-КЛИКА не может быть решена никаким полиномиальным вероятностным алгоритмом при $k = \Omega(t(n))$. Тогда для задачи о максимальной сбалансированной двудольной клике не существует полиномиального приближенного алгоритма гарантирующего мультипликативную ошибку $O(t(n))$.*

В настоящее время в качестве $t(n)$ можно выбрать любую функцию $t(n) = o(\sqrt{n})$.

Список литературы

[1]. S. Khot. Improved inapproximability results for maxclique, chromatic number and approximate graph coloring, Proceedings of the 42th Annual Symposium on Foundations of Computer Science, 2001, pp. 600–609
[2]. U. Feige. Relations between average case complexity and approximation complexity, Proceedings of the 34th Annual Symposium on the Theory of Computing, 2002, pp. 534–543.
[3]. R. Peters. The maximum edge biclique problem is NP-complete, Research Memorandum 789, Faculty of Economics and Business Administration, Tilburg University, 2000.
[4]. U. Feige, R. Krauthhgamer. Finding and sertifying a large hidden clique in a semi-random graph, Random Structures and Algorithms, v. 13, 1998, pp. 457-466.
[5]. A. Juels, M. Peinado. Hiding Cliques for Cryptographic Security, Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms, 1998, pp. 678-684.
[6]. R. Karp. Reducibility among combinatorial problems, in The complexity of computer computations, Plenum Press, New York, 1972, pp. 85-103.
[7]. R. Karp. The probabilistic analysis of some combinatorial search algorithms, in Algorithms and Complexity: New directions and recent results, Academic Press, 1976, pp. 1-19.
[8]. L. Kucera. Expected complexity of graph partitioning problems, Discrete Applied Mathematics, v. 57, 1995,, pp. 193-212.
[9]. M. Jerrum. Large cliques elude the Metropolis process, Random Structures and Algorithms, v. 3, 1992, pp. 347-359.
[10]. J. Hastad. Clique is hard to approximate within $n^{1-\varepsilon}$, Proceedings of the 37th Annual IEEE Symposium on Foundations of Computing, 1997, pp. 627-636.
[11]. U. Feige, S. Kogan. Hardness of approximation of the balanced complete bipartite subgraph problem.
[12]. N. Alon, A. Andoni, T. Kaufman, K. Matulef, R. Rubinfeld, N. Xie. Testing k-wise and almost k-wise independence, Proc. Annual Symposium on the Theory of Computing, 2007, pp. 496–505.
[13]. N. Alon, M. Krivelevich, B. Sudakov. Finding a large hidden clique in a random graph, Random Structures and Algorithms, 1998, v. 13, pp. 457–466.

On the problem of finding approximation of bipatite cliques

Nikolay N. Kuzyurin
Institute for System Programming of RAS,
25, A. Solzhenitsyna str., Moscow, Russia, 109004

Abstract. In this paper, we consider the problem of finding large hidden clique in random graph and it's analog for bipartite graphs.

Keywords: random graph; large hidden clique; finding complexity

DOI: 10.15514/ISPRAS-2017-29(3)-12

For citation: Kuzyurin N.N. On the problem of finding approximation of bipatite cliques..

Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 3, 2017, pp. 225-232 (in Russian). DOI: 10.15514/ISPRAS-2017-29(3)-12

References

[1]. S. Khot. Improved inapproximability results for maxclique, chromatic number and approximate graph coloring, Proceedings of the 42th Annual Symposium on Foundations of Computer Science, 2001, pp. 600–609

[2]. U. Feige. Relations between average case complexity and approximation complexity, Proceedings of the 34th Annual Symposium on the Theory of Computing, 2002, pp. 534–543.

[3]. R. Peters. The maximum edge biclique problem is NP-complete, Research Memorandum 789, Faculty of Economics and Business Administration, Tilburg University, 2000.

[4]. U. Feige, R. Krauthhgamer. Finding and sertifying a large hidden clique in a semi-random graph, Random Structures and Algorithms, v. 13, 1998, pp. 457-466.

[5]. A. Juels, M. Peinado. Hiding Cliques for Cryptographic Security, Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms, 1998, pp. 678-684.

[6]. R. Karp. Reducibility among combinatorial problems, in The complexity of computer computations, Plenum Press, New York, 1972, pp. 85-103.

[7]. R. Karp. The probabilistic analysis of some combinatorial search algorithms, in Algorithms and Complexity: New directions and recent results, Academic Press, 1976, pp. 1-19.

[8]. L. Kucera. Expected complexity of graph partitioning problems, Discrete Applied Mathematics, v. 57, 1995,, pp. 193-212.

[9]. M. Jerrum. Large cliques elude the Metropolis process, Random Structures and Algorithms, v. 3, 1992, pp. 347-359.

[10]. J. Hastad. Clique is hard to approximate within $n^{1-\varepsilon}$, Proceedings of the 37th Annual IEEE Symposium on Foundations of Computing, 1997, pp. 627-636.

[11]. U. Feige, S. Kogan. Hardness of approximation of the balanced complete bipartite subgraph problem.

[12]. N. Alon, A. Andoni, T. Kaufman, K. Matulef, R. Rubinfeld, N. Xie. Testing k-wise and almost k-wise independence, Proc. Annual Symposium on the Theory of Computing, 2007, pp. 496–505.

[13]. N. Alon, M. Krivelevich, B. Sudakov. Finding a large hidden clique in a random graph, Random Structures and Algorithms, 1998, v. 13, pp. 457–466.

Experiments on Parallel Composition of Timed Finite State Machines[1]

A.P. Sotnikov <sotnikhtc@gmail.com>
N.V. Shabaldina <nataliamailbox@mail.ru>
M.Л, Gromov <maxim.leo.gromov@gmail.com>
Tomsk State University,
36, Lenin ave., Tomsk, 634050, Russia

Abstract. In this paper, we continue our work that is devoted to the parallel composition of Timed Finite State Machines (TFSMs). We consider the composition of TFSMs with timeouts and output delays. We held experiments in order to estimate how often parallel composition of nondeterministic TFSMs (with and without timeouts) has infinite sets of output delays. To conduct these experiments we have created two tools: the first one for converting TFSMs into automata (this tool is integrated into BALM-II), the second one for converting the global automaton of the composition into TFSM. As it was suggested in earlier works, we describe the infinite sets of output delays by linear functions, and it is important to know how often these sets of linear functions appear to justify the importance of future investigations of the TFSM parallel compositions (especially for deriving cascade composition). Results of the experiments show significant amount (around 50 %) of TFSMs with infinite number of output delays. We also estimate the size of the global automaton and the composed TFSM. In the experiments, we do not consider global automata with the huge number of states (more then 10000).

Keywords: Timed finite state machine; parallel composition; BALM-II.

DOI: 10.15514/ISPRAS-2017-29(3)-13

For citation: Sotnikov A.P., Shabaldina N.V., Gromov M.L. Experiments on Parallel Composition of Timed Finite State Machines. *Trudy ISP RAN/Proc. ISP RAS*, vol. 29, issue 3, 2017, pp. 233-246. DOI: 10.15514/ISPRAS-2017-29(3)-13

1. Introduction

Different systems, for example web-services, telecommunication protocols digital networks etc. are targeted on interaction with each other. To analyze and synthesize such systems one needs an adequate formal model. *The Finite State Machine* (FSM) has proven to be a classical model for description of *input-output reactive discrete*

[1] This work is supported by the grant for the basic research №16-49-03012 of Russian Scientific Foundation.

event systems [1]. Here, by "*input-output reactive*" we mean, that every input action is always followed by output reaction, and by "*discrete event*" we mean, that domains for input and output actions are finite (discrete) sets. In this case, when talking about interacting systems, main concept is *a composition* of FSMs. If there are two communicating systems and the behavior of each system is described by an FSM, then their common work can be described by the composition of those FSMs. Under appropriate assumptions [2,3] this composition will also be an FSM. In this work we consider so-called parallel composition [2]. In the parallel composition the interacting systems work asynchronously in the assumption of a slow environment and this is enough to guarantee the composition to be an FSM again. To build an FSM composition BALM-II (Berkeley Automata and Language Manipulation) can be used [3].

For more precise description of a system one should consider time aspects of its behavior as well. For that reason we need some model which would be appropriate for description of an input-output reactive timed discrete event system. Probably the most general way to describe timed discrete event system (not necessarily input-output reactive) is a timed automaton [6]. In works [4, 5] authors describe web-services, using language BPEL. In [4] authors tell how they translate web services into timed automata in order to verify them. In [5] the authors use more complicated model – so-called *Timed Extended Finite State Machine*. Then they convert a TEFSM into timed automaton. For further analysis, both, [4] and [5], use UPPAAL [16] as an instrument.

Although timed automata are more than enough to describe any input-output reactive timed discrete event system, they are not convenient for us. The reason is that we would like to keep some room for analysis of the composition. Namely, we would like to use the composition as a specification for a test generation. For the best of our knowledge, methods of a test generation with guaranteed fault coverage for (timed) automata are not well developed (frankly speaking we know only one paper [7], which describes such a method). In contrary, test generation methods for FSMs are well-developed and are still developing [8, 9]. And since parallel composition of FSMs guarantees, that the result is FSM again, we would like to consider some kind of *Timed Finite State Machine* as a model, which would presume this property of the parallel composition. One possible timed augmentation of the FSM was mentioned in [5]. But this model is quite complex and it is not clear, how to build parallel composition for it. Another option to introduce timed FSM is Timed FSM with time guards [10]. The theory of this model is highly developed [11], but it lacks an efficient method to build parallel composition as well as the precious model.

And at last, the model we use in this paper, is the *Timed Finite State Machine with output delays and timeouts* (TFSM) [12-15]. This model allows building parallel composition in the same manner as it is done for common FSM. Given two TFSMs we need to compose. First, the corresponding automata should be built [7], then we compose those automata, obtaining so called *the global automaton of the*

composition. And then we need to transform the global automaton into TFSM. In [12] very interesting effect of the parallel composition is shown. It turned out that composition of two TFSMs (with constant delays) can have infinite number of output delays for a some transitions and those delays can be described by a finite set of linear functions $\{b + k \cdot t \mid b, k \in \{0\} \cup \mathbb{N}\}$. The main objective of this paper is to investigate how often this effect occurs. This would justify further development of the theory of TFSM with infinite (countable) number of delays.

In some works [4, 5] authors use UPPAAL [16] as an instrument for manipulation with models. Although UPPAAL is very powerful tool of timed systems analysis it does not suit us, because it does not allow to build the composition explicitly. In [14] we compare some tools that can be used for deriving the parallel composition of TFSMs, and explain why we have chosen BALM-II.

BALM-II was designed to build parallel composition of two FSMs. To be able to use this tool for TFSMs we use well-known transformation of TFSM into FSM, and in this work we create a tool for converting TFSM into automaton and integrate it into BALM-II. After deriving two automata for the given two TFSMs we construct a global automaton (using BALM-II). In work [14] we suggest two approaches for getting output delays from the composition of corresponding automata: first deals with BALM-II once again, and the second is based on analyzing of time loops in the automaton. In this work we create tool for converting global automaton into TFSM based on the second approach.

Moreover, in works [14, 15] we consider TFSMs with output delays (without timeouts). In this work we consider TFSMs with output delays and timeouts.

We use implemented tools to hold experiments. Since we describe the infinite sets of output delays by linear functions, it is important to know how often these sets of linear functions appear. The experimental results show significant amount (around 50 %) of TFSMs with infinite number of output delays. We also estimate the size of the global automaton and the composed TFSM. Unfortunately the upper bound of the number of states in the global automaton is exponential due to fact that the automaton determinization is needed for composition. In order to get the results of the experiments in reasonable time, we throw away all examples for which the number of states in the global automaton is too huge (more than 10000 states).

We see the contribution of the paper as three points. First, the algorithm for deriving TFSM from the given global automaton (for the case, when TFSMs have both output delays and timeouts). Second, new tools that allow to derive the binary parallel composition of TFSMs automatically (taking in the mind that composition of two *automata* can be derived using BALM-II). And probably the main point, the experiments have shown, that the theory of TFSMs with linearly-countable output delays is worth to be developed.

The outline of the paper is as follows. In Section II some preliminaries are given. In Section III we describe the structure of the composition that we consider in our work, and how the components communicate with each other. Section IV is devoted to one of the implemented tools which allow to derive TFSM based on the global

automaton; we discuss extraction of output delay functions from the global automaton using an example, and propose an algorithm that is lied in the basis of the tool. Section V describes the experiments and experimental results. Section VI concludes the paper.

2. Preliminaries

A finite automaton S is a 5-tuple $(S, X, s_0, F, \lambda_S)$, where S is a finite nonempty set of states with s_0 as the initial state and $F \subseteq S$ as a set of final (accepting) states; X is an alphabet of actions; and $\lambda_S \subseteq S \times X \times S$ is a transition relation. In this work we consider only finite automata, so we will write simply "automaton" (meaning finite automaton). The transition relation defines all possible transitions of the automaton. The language L_S of automaton S is the set of all sequences α in alphabet X, such that in automaton S there is a sequence of transitions (marked by α) from the initial state to some final state. An FSM S is a 5-tuple $(S, I, O, s_0, \lambda_S)$, where S is a finite nonempty set of states with s_0 as the initial state; I and O are input and output alphabets; and $\lambda_S \subseteq S \times I \times O \times S$ is a transition relation. In FSM all states are final.

Let $\mathbb{N}$ be the set of natural numbers. Let $\mathcal{F} = \{ b + k \cdot t \mid b, k \in \{0\} \cup \mathbb{N} \}$ – the set of all possible linear functions. TFSM [12] is an FSM with timeouts and output delays $S = (S, I, O, s_0, \lambda_S, \Delta_S, \sigma_S)$, where 5-tuple $(S, I, O, s_0, \lambda_S)$ is underlying FSM, Δ_S: $S \to S \times (\mathbb{N} \cup \{\infty\})$ is a timeout function that determine maximal time of waiting for input symbol, σ_S: $\lambda_S \to (2^{\mathcal{F}} \setminus \varnothing)$ is an output delay function that determine for each transition time delay for producing output symbol (output timeout).

The semantics of Timed FSM is as follows. We describe the behavior of a system that has time aspects: timeouts and output delays. Timeouts describe the situation when the system comes from one state to another not under the input symbol, but in the case when no inputs are applied during some period of time. In practice it's the case of waiting for the password in internet-banking, etc. As for output delays, the meaning of them is that the output symbol is produced for the given input symbol not immediately but after some period of time. For example, a light can change not immediately after a button is pushed but after some time.

We suppose that there is a global clock (timed variable) and this clock is reset to zero when an input symbol (action) is applied, when an output symbol is produced and when the state of the system is changed (for example, in the case of transition under timeout).

3. Composition of Timed Finite State Machines

Parallel composition describes a dialog between two components. The structure of the composition is presented in Fig. 1.

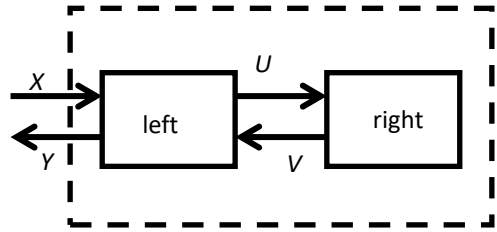


Рис. 1. Структура бинарной параллельной композиции
Fig. 1. Structure of binary parallel composition

We suppose that the system works in "slow environment" (it means that the next input can be applied to the composition only after it produces external output to the previous input), the alphabets of different channels don't intersect and there are no infinite dialogs under internal inputs (it means no livelocks). We also suppose that each component and the whole composition have timed variables. The values of these variables are increasing synchronously, and they reset when the system gets an input or when the state is changed.

In order to compose two TFSMs using BALM-II, we need, first of all, to derive the corresponding automaton for each TFSM [12, 13]. In this work we implement a tool for this step and integrate it into BALM-II as a new command TFSM2AutV1. This implementation requires, that MV description of TFSM (BALM-II format) contains special variable called *Time*. Domain of the variable *Time* contains only non-negative integers which are used to describe timeouts and delays. For example, if a table of transitions has head as follows

```
.table I Time O CS -> NS
```

and we would like to represent timeout transition $s_1 \overset{t}{\rightarrow} s_2$, then it will appear as

```
^ t ^ s1 s2
```

where *I* – is the variable for input action, *O* – is the variable for output action, *CS* – is the variable for the current state, *NS* – is the variable for the next state, *t* – is some non-negative integer from the domain of the variable *Time* and the symbol ^ represents the fact, that there is no action in corresponding channel. The ordinary transition with delay, like $s_1 \xrightarrow{i/o(d)} s_2$ is described as

```
i d o s1 s2
```

where *i* – is from the domain of *I*, *o* – is from the domain of *O*, and *d* – is from the domain of *Time*.

Then we derive parallel composition of two automatons using BALM-II (we describe how to do this in works [14, 15]). The resulting automaton is so-called global automaton and it describes the common behavior of two automata that are working together in a dialogue mode.

After deriving a global automaton that describes the common behavior of two given TFSMs we need to construct the corresponding TFSM. We also develop a tool for this step and describe the corresponding algorithm in the next section.

4. Deriving TFSM Based on the Global Automaton. Extracting Output Delays Functions

Let's consider an example of a global automaton (Fig. 2) and describe how to derive the corresponding TFSM (Fig. 3). One can see that after Request there can be output Deliver after $3 + 5t$ or $4 + 5t$ tick counts, where t is arbitrary non-negative integer number.

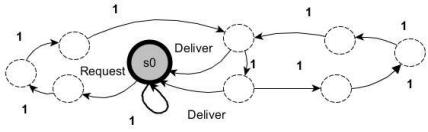


Рис. 2. Пример глобального полуавтомата

Fig. 2. An example of global automaton

Рис. 3. Соответствующий временной автомат
Fig. 3. Corresponding TFSM

In work [14] we propose a procedure for deriving TFSM based on the global automaton for the case when the given TFSMs have only output delays (no timeouts). In this work we propose more common algorithm that works also for the case when the given TFSMs have both output delays and timeouts. The idea of this algorithm is very simple. According to the theory the final states of the global automaton correspond to the states of TFSM. Every sequence, which starts and finishes at some final states of the automaton and goes through non-final states, corresponds to the transition of the TFSM. The sequence can start only with input action or with special action 1. If this sequence starts with special action 1, then every action of the sequence is 1 and corresponding transition is timeout transition (timeout is the number of 1^s needed to reach final state). If the first action of the sequence is input action, then the last action is output action and the intermediate actions are 1^s. In this case the corresponding TFSM transition is ordinary input-output transition with delay. The delay – is the number of 1^s in-between the input and the output actions of the sequence. We just need to keep in mind, that sequence of 1^s may form a loop. So we do some precautions to detect loops when traversing the automaton transitions. The number of 1^s before the loop gives us b for linear function and the length of the loop gives k for the function.

Algorithm 1. Deriving TFSM based on the global automaton.

Input. Global automaton $A = \langle A, I \cup O \cup \{1\}, a_0, F, \lambda_A \rangle$

Output. TFSM $T = \langle T, I, O, t_0, \lambda_T, \Delta_T, \sigma_T \rangle$ with the same behavior.

$t_0 \equiv a_0$; $T := \{t_0\}$;

FOREACH non-visited state t from T **DO**

IF $\exists \langle t, 1, t' \rangle \in \lambda_A$ **THEN DO**

ADD t' in T;

IF $t == t'$ **THEN ADD** $\langle t, \infty, t' \rangle$ in Δ_T;

ELSE ADD $\langle t, 1, t' \rangle$ in Δ_T;

DONE

FOREACH input action i such, that $\exists \langle t, i, t' \rangle \in \lambda_A$

DO

$b := 0$; $V := \varnothing$;

WHILE t' != NULL **AND** t' is **NOT** visited

DO

$t'.b := b$;

FOREACH output action o such, that $\exists \langle t', o, t'' \rangle \in \lambda_A$

DO

IF $t'' \in F$ **THEN DO**

ADD t'' in T;

ADD $\langle t', o, t'' \rangle$ in V;

DONE

DONE

mark t' as visited;

b++;

IF $\exists \langle t', 1, t'' \rangle \in \lambda_A$ **THEN** $t' := t''$;

ELSE $t' :=$ NULL;

DONE

IF $t' ==$ NULL **THEN** $k := 0$, $n_{\text{loop}} := \infty$;

ELSE $k := b - t'.b$, $n_{\text{loop}} := t'.b$;

FOREACH $\langle t', o, t'' \rangle$ in V **DO**

ADD $\langle t, i, o, t'' \rangle$ in λ_T;

IF $t'.b < n_{\text{loop}}$ THEN

ADD $\langle \langle t, i, o, t'' \rangle, t'.b \rangle$ in σ_T;

ELSE

ADD $\langle \langle t, i, o, t'' \rangle, t'.b + k*x \rangle$ in σ_T;

DONE

DONE

mark *t* as visited;

DONE

5. Experimental Results

We conduct the experiments according to the following steps:

Step 1. Generate two complete nondeterministic observable timed FSMs: `left.fsm` and `right.fsm`. At this step we use FSM generator from the tool [17].

Step 2. Convert generated TFSMs into AUT-format (BALM-II format). After this step we have two files in AUT-format: `left.aut` and `right.aut`.

The number of all states in the automaton is $S + S*I*D + S*T$, where S – is the number of states in original TFSM, I – number of input symbols, D – maximal delay, T – maximal finite timeout. The number of stable states is $S*T$.

If we have no timeouts then $T = 0$ and the number of states of the automaton is $S + S*I*D$.

Step 3. Convert files `left.aut` and `right.aut` with TFSMs into files `left_aut.aut` and `right_aut.aut` with corresponding automata. In order to do this, we created the tool and integrated it into BALM-II. We described the algorithm for this transformation in the work [14]. In this work we only add in that algorithm the transformation for timeout transitions.

Step 4. Derive the global automaton. We derive the global automaton using the same sequence of BALM-II commands as we described in work [14].

The number of final states in the product automaton is $S1*T1*S2*T2$, where $S1$ is the number of final states in the left component, $S2$ is the number of final states in the right component, $T1$ is the maximal finite timeout in the left component, $T2$ is the maximal finite timeout in the right component.

After the restriction we will have the global automaton with at most $2^{S1*T1*S2*T2} - 1$ states (since the `restriction` command includes determinization of the automaton).

If we have no timeouts then the number of final states in the product automaton is at most $S1*S2$ and after restriction we have at most $2^{S1*S2} - 1$.

Step 5. Derive TFSM based on the global automaton. For this step we created the tool based on the algorithm that was proposed in the previous section.

We generated one hundred pairs of TFSMs for each set of parameters values (number of states, maximal time delays and timeouts). In order to get results of the experiments in reasonable time we fixed the number of inputs and outputs for each channel to 2. We also need to mention that in experiments we did not consider global automata with the huge number of states (more than 10000). It means that we have thrown away such examples. The reason is that the upper bound of the number

of states in the global automaton is exponential because of determinization used during composition.

The experimental results are represented in Table 1. In the 3rd and 4th columns there are percentages of TFSMs with infinite number of output delays (we need to use linear functions for describing output delays). The difference is that for the 3rd column we calculated percentage of such TFSMs for the case when the components are TFSMs with timeouts and output delays, and for the 4th column – only output delays (no timeouts). First of all, we would like to comment on dashes ('-') in the 3rd and 4th columns. In those cases we could not conduct the experiments for the given parameters in the reasonable time and the reason is the exponential upper bound of the state's number in the global automaton. For example, let's consider the last row in the Table 1: we have 4 states in the left TFSM and 4 states in the right TFSM, the maximal finite timeout for the both is the same and it is equal to 7. According to our experiments' procedure, we first derive the corresponding automata for the given TFSMs. The number of final states in the automata is $S*T$, where S is the number of states in original TFSM, T – maximal finite timeout. So for our case the number of states in the automaton for the left component (let's denote it as $S1$) will be equal to the number of states in the automaton for the right component (let's denote it as $S2$) and $S1 = S2 = = 4*7 = 28$. So, each automaton will have 28 stable states. Then, we estimate the number of states in the product automaton as $S1*T1*S2*T2$, where $T1$ is the maximal finite timeout in the left component, $T2$ is the maximal finite timeout in the right component, so, for our case the number of states in the product automaton will be 28*7*28*7 = 38416. After the `restriction` we will have the global automaton with at most $2^{S1*T1*S2*T2} - 1$ states since the command restriction does determinization of the automaton, and for the last row in our table in the worst case it can be $2^{38416} - 1$ states and of cause it's too huge automaton to deal with.

Табл. 1. Экспериментальные результаты
Table 1. Experimental results

Number of states	***Maximal delay / timeout***	***Percent of TFSMs with infinite number of output delays (with timeouts)***	***Percent of TFSMs with infinite number of output delays (without timeouts)***
2	2	38	23
3	2	39	37
4	2	43	28
5	2	34	-
2	3	47	38
3	3	56	42
4	3	66	55

Number of states	***Maximal delay / timeout***	***Percent of TFSMs with infinite number of output delays (with timeouts)***	***Percent of TFSMs with infinite number of output delays (without timeouts)***
2	4	46	42
3	4	61	47
4	4	63	53
2	5	67	36
3	5	-	51
4	5	34	69
2	6	-	52
3	6	-	54
4	6	-	68
2	7	-	51
3	7	-	50
4	7	-	75

According to our experimental results, around 50 % of TFSMs, that describe the behavior of the composition, has the infinite number of output delays, so, further investigations of such compositions are needed. It is an actual task especially for the case of cascade composition [15], when each component is a TFSM with timeouts and final sets of output delays, and we first compose two internal components and then we need to compose the resulting TFSM with the remaining part of the system. However, according to our experimental results, this resulting TFSM has infinite number of output delays with high probability. So, more investigations of such compositions are needed.

6. Conclusions

This paper is devoted to parallel composition of Timed Finite State Machines (TFSMs). We consider the composition of TFSMs with transitions under timeouts and output delays. It is known that even for the case when output delays are the finite sets of nonnegative integers, the result of such composition can be a TFSM with infinite set of output delays, and we describe such infinite sets by linear functions. It is important to know how often these sets of linear functions appear in order to estimate the importance of future investigations such compositions (especially for deriving cascade composition). In order to conduct the experiments we created two tools: the first one for converting TFSM into automaton (we integrated it into BALM-II), the second one for converting the global automaton into TFSM. The experimental results show significant amount (around 50 %) of

TFSMs with infinite number of output delays, so, further investigations of such compositions are needed. We also estimate the size of global automaton and the composed TFSM. In experiments we do not consider global automata with the huge number of states (more then 10000). The reason is that the upper bound of the number of states in the global automaton is exponential because of determinization used during composition. We plan to propose another approach for deriving the composition of Timed Finite State Machines. It will be the part of our future work.

References

[1]. Gill A. Introduction to the theory of finite state machines, New-York, McGraw-Hill, 1962.

[2]. N. Yevtushenko, T. Villa, R. Brayton, A. Petrenko, and A. Sangiovanni-Vincentelli. Solution of parallel language equations for logic synthesis. In The Proceedings of the International Conference on Computer-Aided Design. 2001. pp. 103–110.

[3]. G. Castagnetti, M. Piccolo, T. Villa, N. Yevtushenko, A. Mishchenko, Robert K. Brayton. Solving Parallel Equations with BALM-II. Technical Report No. UCB/EECS-2012-181, Electrical Engineering and Computer Sciences University of California at Berkeley. 2012. [Electronic resource] http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-181.pdf (date of access: 21.04.2016).

[4]. Gregorio Diaz, Juan-Jos e Pardo, Mar a-Emilia Cambronero, Valent n Valero, and Fernando Cuartero. Automatic Translation of WS-CDL Choreographies to Timed Automata, volume 3670 of Lecture Notes in Computer Science, book section 17, pages 230{242. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-28701-8. doi: 10.1007/11549970 17

[5]. M. Lallali, F. Zaidi, and A. Cavalli. Timed modeling of web services composition for automatic testing. In Signal-Image. Technologies and Internet-Based System, 2007. Bibliography 102 SITIS '07. Third International IEEE Conference on, pages 417- 426, Dec 2007. DOI: 10.1109/SITIS.2007.110.

[6]. R. Alur and D. L. Dill. A theory of timed automata. Theoretical computer science. 1994. Vol.126, Iss. 2. pp. 183–235.

[7]. Springintveld J., Vaandrager F. and D'Argenio P. Testing timed automata. Theoretical Computer Science, 254 (1-2). pp. 225-257, 2001.

[8]. Kushik N., Lopez J., Cavalli A., Yevtushenko N. Improving Protocol Passive Testing through 'Gedanken' Experiments with Finite State Machines. Proceedings 2016 IEEE International Conference on Software Quality, Reliability and Security, pp. 315-322.

[9]. Hierons R., Turker U. Parallel Algorithms for Testing Finite State Machines: Generating UIO Sequences. IEEE Transactions on Software Engineering, 42(11),7429774. pp. 1077-1091, 2016.

[10]. K. El-Fakih, M. Gromov, N. Shabaldina, N. Yevtushenko. Distinguishing Experiments for Timed Non-Deterministic Finite State Machines. Acta Cybernetica. 2013. Vol. 21, № 2. pp. 205–222.

[11]. Tvardovskii A., Yevtushenko N. Minimizing timed Finite State Machines. Vestnik TGU [The Bulletin of TSU], 2014. Vol. 4 (29). pp. 77-82 (in Russian).

[12]. O. Kondratyeva, N. Yevtushenko, and A. Cavalli. Parallel composition of nondeterministic finite state machines with timeouts. Journal of Control and Computer Science. Tomsk State University, Russia. 2014. Vol. 2(27). pp. 73–81 (in Russian).

[13]. O. Kondratyeva, N. Yevtushenko, A. Cavalli. Solving parallel equations for Finite State Machines with Timeouts. Trudy ISP RAN / Proc. ISP RAS, vol. 26, issue 6, pp. 85–98 (in Russian). DOI: 10.15514/ISPRAS-2014-26(6)-8.
[14]. Shabaldina N., Gromov M. Using BALM-II for deriving parallel composition of timed finite state machines with outputs delays and timeouts: work-in-progress. System Informatics [Sistemnaya informatika], № 8, 2016, pp. 33-42.
[15]. Gromov M..L, Shabaldina N.V. Using balm-ii for deriving cascade parallel composition of timed finite state machines. Modeling and Analysis of Information Systems [Modelirovanie i analis inforamzionnych system], 23:3 (2016). pp. 699-712 (in Russian).
[16]. http://www.uppaal.com/ (date of access: 21.04.2016)
[17]. N. Shabaldina , M. Gromov. FSMTest-1.0: a manual for researches. Proceedings of IEEE East-West Design & Test Symposium (EWDTS'2015). Ukraine, Kharkov: SCITEPRESS, 2015. pp. 216-219.

Эксперименты по построению параллельной композиции временных автоматов

А.П. Сотников <sotnikhtc@gmail.com>
Н.В. Шабалдина <nataliamailbox@mail.ru>
М.Л. Громов <maxim.leo.gromov@gmail.com>
Томский государственный университет,
634050, Россия, г. Томск, пр. Ленина, д. 36

Аннотация. В данной работе мы продолжаем наши исследования параллельной композиции временных конечных автоматов. Мы рассматриваем композицию временных автоматов с таймаутами и задержками выходных символов. Для того чтобы оценить, насколько часто в параллельной композиции недетерминированных временных автоматов (с таймаутами и без таймаутов) возникают бесконечные множества задержек выходных символов, мы провели компьютерные эксперименты. Для проведения таких экспериментов мы реализовали два инструмента: первый позволяет преобразовать временной конечный автомат в полуавтомат (данный инструмент встроен в BALM-II), второй позволяет преобразовать глобальный полуавтомат композиции во временной автомат. Ориентируясь на известные работы по данной тематике, мы описываем бесконечные множества задержек выходных символов конечным образом, а именно, при помощи линейных функций, и нужно знать, как часто такое множество линейных функций возникает, чтобы оценить важность дальнейших исследований параллельной композиции временных автоматов (особенно случая каскадной композиции). Результаты экспериментов показали, что в значительном количестве случаев (около 50 %) временной автомат композиции содержит бесконечное множество задержек выходных символов. Кроме того, мы оценили размер глобального полуавтомата и автомата композиции. При проведении экспериментов мы не рассматривали глобальные полуавтоматы с большим числом состояний (более 10000).

Ключевые слова: временные конечные автоматы; параллельная композиция; BALM-II.

DOI: 10.15514/ISPRAS-2017-29(3)-13

Для цитирования: Сотников А.П., Шабалдина Н.В.,, Громов М.Л. Эксперименты по построению параллельной композиции временных автоматов. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 233-246. DOI: 10.15514/ISPRAS-2017-29(3)-13

Список литературы

[1]. Gill A. Introduction to the theory of finite state machines, New-York, McGraw-Hill, 1962.

[2]. N. Yevtushenko, T. Villa, R. Brayton, A. Petrenko, and A. Sangiovanni-Vincentelli. Solution of parallel language equations for logic synthesis. In The Proceedings of the International Conference on Computer-Aided Design. 2001. pp. 103–110.

[3]. G. Castagnetti, M. Piccolo, T. Villa, N. Yevtushenko, A. Mishchenko, Robert K. Brayton. Solving Parallel Equations with BALM-II. Technical Report No. UCB/EECS-2012-181, Electrical Engineering and Computer Sciences University of California at Berkeley. 2012. http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-181.pdf (дата доступа 21.04.2016).

[4]. Gregorio Diaz, Juan-Jos e Pardo, Mar a-Emilia Cambronero, Valent n Valero, and Fernando Cuartero. Automatic Translation of WS-CDL Choreographies to Timed Automata, volume 3670 of Lecture Notes in Computer Science, book section 17, pages 230{242. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-28701-8. doi: 10.1007/11549970 17

[5]. M. Lallali, F. Zaidi, and A. Cavalli. Timed modeling of web services composition for automatic testing. In Signal-Image. Technologies and Internet-Based System, 2007. Bibliography 102 SITIS '07. Third International IEEE Conference on, pages 417- 426, Dec 2007. DOI: 10.1109/SITIS.2007.110.

[6]. R. Alur and D. L. Dill. A theory of timed automata. Theoretical computer science. 1994. Vol.126, Iss. 2. pp. 183–235.

[7]. Springintveld J., Vaandrager F. and D'Argenio P. Testing timed automata. Theoretical Computer Science, 254 (1-2). pp. 225-257, 2001.

[8]. Kushik N., Lopez J., Cavalli A., Yevtushenko N. Improving Protocol Passive Testing through 'Gedanken' Experiments with Finite State Machines. Proceedings 2016 IEEE International Conference on Software Quality, Reliability and Security, pp. 315-322.

[9]. Hierons R., Turker U. Parallel Algorithms for Testing Finite State Machines: Generating UIO Sequences. IEEE Transactions on Software Engineering, 42(11),7429774. pp. 1077-1091, 2016.

[10]. K. El-Fakih, M. Gromov, N. Shabaldina, N. Yevtushenko. Distinguishing Experiments for Timed Non-Deterministic Finite State Machines. Acta Cybernetica. 2013. Vol. 21, № 2. pp. 205–222.

[11]. А.С. Твардовский, Н.В. Евтушенко. К минимизации автоматов с временными ограничениями. Вестн. Том. гос. ун-та. Управление, вычислительная техника и информатика, 2014. № 4 (29), стр. 77-82

[12]. О.В. Кондратьева, Н.В. Евтушенко, А.Р. Кавалли. Параллельная композиция конечных автоматов с таймаутами. Вестн. Том. гос. ун-та. Управление, вычислительная техника и информатика, 2014. № 2 (27), стр. 73–81

[13]. О.В. Кондратьева, Н.В. Евтушенко, А.Р. Кавалли. Решение автоматных уравнений для временных автоматов относительно параллельной композиции. Труды ИСП РАН, том 26, вып. 6, стр. 85–98

[14]. Shabaldina N., Gromov M. Using BALM-II for deriving parallel composition of timed finite state machines with outputs delays and timeouts: work-in-progress. Системная информатика, № 8, 2016, pp. 33-42.

[15]. Громов М. Л., Шабалдина Н. В. Построение каскадной параллельной композиции временных автоматов в balm-ii. Моделирование и анализ информационных систем, Т. 23, No 6 (2016), стр. 699-712.

[16]. http://www.uppaal.com/ (дата доступа 21.04.2016)

[17]. N. Shabaldina, M. Gromov. FSMTest-1.0: a manual for researches. Proceedings of IEEE East-West Design & Test Symposium (EWDTS'2015). Ukraine, Kharkov: SCITEPRESS, 2015, pp. 216-219.

Объектно-ориентированный каркас для программной реализации приложений теории расписаний

[1] *А.С. Аничкин <anton.anichkin@ispras.ru>*
[1,2] *В.А. Семенов <sem@ispras.ru>*
[1] *Институт системного программирования РАН,*
109004, Россия, г. Москва, ул. А. Солженицына, д. 25
[2] *Московский физико-технический институт,*
141700, Московская область, г. Долгопрудный, Институтский пер., 9

Аннотация. Статья адресована вопросам программной реализации моделей, методов и приложений теории расписаний с использованием объектно-ориентированного каркаса. Каркас представляет собой систему классов вместе с предусмотренными механизмами взаимодействия и расширения, что обеспечивает эволюционную разработку серий приложений на единой методологической, программной и инструментальной основе. В статье детально обсуждаются принципы организации и функционирования разработанного каркаса, а также его возможности для разработки приложений теории расписаний и, в частности, перспективных систем календарно-сетевого планирования и управления проектами.

Ключевые слова: теория расписаний; календарно-сетевое планирование; программная инженерия; объектно-ориентированное программирование.

DOI: 10.15514/ISPRAS-2017-29(3)-14

Для цитирования: Аничкин А.С., Семенов В.А. Объектно-ориентированный каркас для программной реализации приложений теории расписаний. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 247-296. DOI: 10.15514/ISPRAS-2017-29(3)-14

1. Введение

Теория расписаний и календарно-сетевое планирование находят широкое применение в научных и индустриальных областях, связанных с управлением производством, организацией транспортных потоков, управлением вычислительными ресурсами. Однако многообразие существующих математических моделей и вычислительных методов, а также перманентное появление новых ставит перед программистами

довольно острую проблему эволюционной разработки серий приложений на единой методологической, программной и инструментальной основе. В частности, подобная проблема возникает при создании перспективных систем календарно-сетевого планирования и управления индустриальными проектами, в которых задачи составления расписаний решаются в обобщенной постановке с учетом множества факторов, влияющих на ход выполнения проектных работ. В таких постановках учитываются не только типовые временные условия, отношения предшествования между работами, ресурсные ограничения, но и специфические требования пространственно-временной согласованности проектных работ, их финансового и логистического обеспечения. Данные требования существенны для масштабных индустриальных программ, в которых риски технологических и организационных ошибок чрезвычайно высоки, а сроки и бюджеты жестко ограничены. Примерами специфических требований могут служить условия привлечения инвестиционных средств, ограничения по поставкам материалов, правила размещения и использования оборудования, особенности монтажа элементов конструкций возводимого сооружения, условия резервирование рабочих зон при организации проектных работ.

Использование универсальных математических библиотек для решения подобных задач, как правило, оказывается невозможным из-за принципиальных отличий в их постановках или крайне неэффективным в силу зависимости вычислительной сложности составления расписания от частных условий. Например, задача проектного планирования проектов (Resource-Constrained Project Scheduling Problem; RCPSP), являющаяся NP-полной, редуцируется к частным постановкам «открытой линии», «рабочего цеха» или «потоковой линии», имеющим полиномиальную сложность при небольшом числе машин, простых моделях обслуживания и отсутствии директивных сроков [1]. Применение универсальных средств проектного планирования для подобных частных постановок было бы чрезмерно затратным.

Стратегия разработки программ составления расписаний заново для каждого нового типа приложения также является неприемлемой в силу сложности современных математических моделей и вычислительных методов. Данные методы должны учитывать большое количество факторов и использовать развитые системы эвристик для поиска приемлемых приближенных решений в тех случаях, когда задача имеет высокую размерность. Разработка и программная реализация подобных методов часто оказывается предметом интенсивных научных исследований. Адаптация унаследованных открытых кодов, изначально непредназначенных для подобных целей и не предусматривающих возможности для их развития, также малоэффективна даже для написания программ близкой функциональности.

В связи с этим актуальным представляется создание единой инструментальной среды для программной реализации моделей, методов и

приложений теории расписаний. Подобная среда должна предоставлять развитые средства для разработки новых программ на основе ранее реализованных модулей. При этом возможности развития, адаптации и конфигурации модулей должны обеспечивать построение эффективных программ составления расписаний, релевантных условиям и сложности решаемых прикладных задач.

Ранее предпринимались попытки организации подобных сред в виде расширяемых математических библиотек. В качестве одного из значимых результатов следует упомянуть библиотеку PSPLIB [2], в которой Колиш и Шпрехер реализовали несколько методов проектного планирования. Библиотека претерпела определенное развитие [3, 4, 5, 6], однако в настоящее время она преимущественно используется для тестирования других аналогичных программ и оценки производительности на специально подготовленных наборах контрольных примеров (benchmarks). Библиотека имеет принципиальные ограничения для решения индустриально значимых задач высокой размерности, а ее архитектура плохо приспособлена для реализации новых моделей и методов.

Более интересной в этом отношении является библиотека проектного планирования LibRCPS, разработанная Лемменем [7]. В ней предусмотрены интерфейсы для задания входных данных, а также имеется возможность специфицировать некоторые алгоритмические детали поиска решения, например, выбрать тип целевой функции и применяемую эвристику. К сожалению, значительная часть планируемых возможностей и функций библиотеки осталась нереализованной. Несмотря на открытые исходные коды, библиотека не получила дальнейшего развития, а сообщения об опыте ее применения крайне скудны.

Более успешными в практическом отношении оказались программные системы календарно-сетевого планирования и управления проектами. Обычно они обеспечивают автономную работу пользователя на изолированном компьютере или групповую работу в корпоративной сети. В качестве популярных программных решений следует указать Oracle Primavera, MS Project, Synchro, Spider Project, Gemini, Merlin, Zoho Projects, ManagePro [8]. Ряд систем конфигурируется в виде универсальных Интернет-сервисов и WEB-клиентов к ним. К подобным решениям относятся Smartsheet, GanttPro, Asana, Acunote, Teamweek, Bitrix24, Jira, ISETIA [8]. Несмотря на наличие программных интерфейсов доступа к данным и возможность локального или удаленного вызова функций планирования, перечисленные системы обладают существенным общим недостатком. Главным образом он связан с предопределенным характером реализованных алгоритмов и невозможностью их модификации для решения новых классов прикладных задач. Тем самым, данные системы не предоставляют инструментальных возможностей, необходимых для их дальнейшего функционального развития.

Вместе с тем, потребность в подобных инструментах велика, поскольку разрабатывается большое число специализированных систем, ориентированных на частные индустриальные приложения. Для примера приведем лишь некоторые их них, акцентируя внимание на разнообразие прикладных постановок [9].

Система PLANETS (PLanning Activities on NETworkS) [10, 11], разработанная Университетом Каталония (University of Catalonia) в Барселоне для испанской электрической компании, как инструмент календарного планирования перестройки и технического обслуживания электрической сети без нарушения обслуживания потребителей. Система ATLAS [12, 13] осуществляет планирование производства гербицидов на заводе Monsanto в Антверпене. Система MOSES [14] была разработана компанией COSYTEC для производителя питания для животных в Великобритании. Система FORWARDC [15] представляет собой систему поддержки принятия решений (СППР) и используется на нефтеперегонных заводах в Европе при планировании поставок сырой нефти, ее обработки, смешивания и доставки потребителям. Xerox использовал систему планирования различных видов работ на копировальных машинах [16]. TAP-AI [17] — активная система планирования, предназначенная для ежедневного управления деятельностью авиалиний SAS. OPTISERVICE [18] — программный пакет для назначения персонала для всех заграничных теле- и радиостанций сети RFO с учетом ограничений по времени и условиям оплаты квалифицированных журналистов и технических работников. Система MOSAR [19], разработанная компаниями Cisi и COSYTEC для министерства юстиции Франции, назначает охранников тюрем по 200 тюрьмам Франции по чередующимся сменам. Система COBRA [20] позволяет разработать диаграммы рабочих планов машинистов поездов компании North Western Trains в Великобритании. Проект DAYSY Esprit (пакет SAS-Pilot) [17] осуществляет переназначение летных экипажей по полетам. Система краткосрочного планирования (The Short Term Planning (STP)) для компании Renault [21] решает задачу транспортировки автомобилей заказчикам с учетом множества ограничений. Средства финансового планирования применяются при утверждении бюджетов районов Москвы и их последующем контроле [22]. Планирование некоторых телекоммуникационных сетей мобильной связи осуществляется с помощью системы POPULAR [23]. Примечательно, что во многих системах используются технологии логического программирования в ограничениях, которые оказываются конкурентоспособными как по гибкости задания условий прикладных задач, так и по эффективности их решения. В частности, языки и системы логического программирования в ограничениях CHIP, 2LP, ILOG, ECLiPSe послужили математической основой для реализации некоторых из перечисленных выше специализированных систем. Тем не менее, интерпретация обобщенных задач проектного планирования в терминах логического программирования невозможна и данные технологии

могут применяться лишь в качестве элемента более общих подходов к планированию.

Важный шаг к систематизации и концептуализации задач проектного планирования был сделан в связи со становлением технологий информационного моделирования процессов строительства BIM (Building Information Modeling) [24, 25]. Появившиеся информационные стандарты и, в частности, модель IFC (ISO 16739:2013 Industry Foundation Classes) [26] позволяют специфицировать некоторые типовые условия задач календарно-сетевого планирования, используя программные интерфейсы доступа к данным или альтернативные способы их представления в файлах открытых форматов. Однако данные стандарты не регламентируют математические методы планирования и поэтому не могут служить полноценной основой для реализации программных приложений соответствующей функциональности.

Настоящая статья адресована проблемам создания инструментальной среды для программной реализации моделей, методов и приложений теории расписаний в виде объектно-ориентированного каркаса или архитектурного шаблона (object-oriented framework). В дальнейшем объектно-ориентированный каркас для приложений теории расписаний и проектного планирования называется SAF-каркас (Scheduling Application Framework). Объектно-ориентированные каркасы с успехом применяются при разработке сложных программных систем с расширяемым функционалом и при создании линеек программных продуктов в смежных предметных областях. Поэтому использование данного подхода для достижения декларируемых целей представляется вполне оправданным.

Требования, предъявляемые к объектно-ориентированному каркасу, а также общие принципы его построения обсуждаются в разделе 2. Раздел 3 посвящен организации классов прикладных данных для представления условий задач проектного планирования RCPSP в расширенных постановках. Классы математических объектов и вычислительных алгоритмов для редукции задач проектного планирования к задачам условной оптимизации и их решения описываются в разделе 4. Методологические аспекты разработки программных приложений теории расписаний на основе модулей каркаса рассматриваются в разделе 5. Результаты апробации разработанного каркаса в ходе разработки информационной системы планирования и управления проектами кратко обсуждаются в разделе 6. В заключении подводятся основные итоги работы.

2. Общие принципы и организация SAF-каркаса

2.1 Понятие объектно-ориентированного каркаса

Согласно наиболее распространённому определению [27], каркас (или фреймворк — от английского «framework») представляет собой программную платформу, определяющую структуру целевой программной

системы и облегчающую разработку, сопровождение и объединение компонентов в ее составе. Целевые системы в этом случае состоят из самого каркаса, являющегося их неизменной частью, и модулей расширения, конфигурации которых могут гибко меняться для обеспечения требуемой функциональности системы и желаемых характеристик. Организация каркаса при этом должна предусматривать, так называемые, точки расширения (hot spots), благодаря которым модули с одними и теми же интерфейсами могут применяться в качестве альтернативных реализаций основных функций системы. Точки расширения во многом задают правила конфигурирования и направления возможной функциональной эволюции целевой системы.

Каркасный подход естественным образом воплощается в рамках парадигмы объектно-ориентированного программирования (ООП). Поскольку ООП предполагает систематизацию и концептуализацию предметной области, относительно просто определяются элементы и точки расширения каркаса. Ключевые понятия предметной области, допускающие специализацию, оформляются в виде абстрактных классов каркаса с предопределенным интерфейсом. Они задают потенциальные точки расширения каркаса. Непосредственная реализация методов интерфейса осуществляется в наследуемых классах, чем обеспечивается полиморфизм включения и возможность конфигурирования приложений из специализированных элементов каркаса. Примечательно, что значительная часть методов может быть имплементирована на уровне абстрактных классов, тем самым, избавляя разработчиков от необходимости повторной реализации общих и, часто нетривиальных, механизмов взаимодействия классов каркаса. В этом случае разработчики могут сосредоточиться на особенностях реализации методов для конкретных типов классов с учетом их специфических свойств и поведения. Естественно, чтобы достичь подобных преимуществ, требуется провести тщательный объектный анализ предметной области и спроектировать каркас рациональным образом, обеспечивающим его последующее многократное использование при относительно невысоких затратах на доработку целевых приложений.

Каркасный подход к построению приложений тесто связан с общими методологиями объектно-ориентированного программирования и, в частности, с теорией объектно-компонентного моделирования, предложенной и развитой Е.М. Лаврищевой [28], а также методом построения систем с расширяемым функционалом, описанным М.М. Горбуновым-Посадовым [29].

Во многом становление каркасного подхода было предопределено необходимостью разработки графических интерфейсов пользователя (GUI), которые имели тенденцию к выделению стандартной структуры приложения и типовых графических элементов. Популярные в настоящее время

графические библиотеки MFC, Qt, GNOME, KDE в полной мере служат примерами успешного применения каркасного подхода.

Принципиальным отличием каркаса от библиотеки программ является то, что он не просто предоставляет наборы отдельных, часто несвязанных между собой функций, но и во многом предопределяет архитектуру всей целевой системы. Иногда указывают и на другое отличие, состоящее в инверсии управления и вызове пользовательских функций непосредственно из модулей каркаса [30]. Однако это наблюдение является не совсем верным. Например, в библиотеках условной нелинейной оптимизации подобная инверсия применяется с целью задания математических функций в виде соответствующих императивных процедур расчета их значений и производных. Такой способ исключает необходимость интерпретации математических функций, заданных декларативным образом, и повышает эффективность вычислительного процесса.

В любом случае организация каркаса не препятствует интеграции любого числа сторонних библиотек самой разной функциональности для решения вспомогательных задач. Более того, она может предусматривать применение специализированных языков для описания прикладных задач, форматов обменных файлов для вывода и хранения результатов, языков запросов к СУБД, протоколов взаимодействия клиентских и серверных приложений и т.п. Тем самым, концепция каркаса существенно расширяет идею библиотечной организации программного обеспечения, предусматривая развитые инструментальные возможности для построения целевых приложений.

2.2 Общие требования и принципы построения SAF-каркаса

Обсуждаемый объектно-ориентированный SAF-каркас сочетает в себе функции математической библиотеки и инструментальной среды для построения программных приложений теории расписаний и проектного планирования. Сформулируем общие требования, которые предъявлялись к нему и учитывались при его разработке:

- универсальность, предполагающая наличие готовых к использованию программных модулей для математически строгой постановки и решения типовых задач теории расписаний и проектного планирования;
- эффективность, означающая в данном случае равномерно высокую производительность модулей для решения обсуждаемого класса задач и, прежде всего, для приближенного решения индустриально значимых задач проектного планирования высокой размерности;
- гибкость, подразумевающая возможность повторного использования имеющихся модулей при программной реализации новых моделей, методов и приложений теории расписаний при относительно низких затратах на доработку.

Данные отчасти противоречивые требования нуждаются в уточнениях, поскольку в значительной степени формируют общий функциональный облик всего каркаса.

Как известно, теория расписаний охватывает довольно много классов задач с разными оценками вычислительной сложности и со своими алгоритмами решения. Общепринятая нотация Грэхема $\alpha|\beta|\gamma$, в которой характеристики описывают соответствующие модели исполнения операций и машин (работ и ресурсов) и целевые функции, задает общую классификацию подобных задач. В зависимости от индивидуальных характеристик $\alpha|\beta|\gamma$ вычислительная сложность составления расписания может существенно варьироваться и поэтому для этих целей обычно применяют специальные алгоритмы, ориентированные на частные классы задач.

Важно отметить, что при разработке каркаса, как универсальной математической библиотеки, не ставилась цель предоставить средства, которые бы обеспечили решение всех задач теории расписания за оптимальное время. Вместо этого предпринята попытка эффективно решать задачи проектного планирования в постановке RCPSP, к которой редуцируются все основные задачи теории расписаний [31]. Однако и данная классическая постановка оказывается довольно частной для реализации приложений календарно-сетевого планирования и управления проектами, в которых применяются сложные многопараметрические модели работ, связей, ресурсов, календарей, счетов. Некоторые отличия в постановках задач указаны в работе [1], в которой авторы говорят о задаче RCPSP в расширенной постановке. Необходимые математические обобщения также обсуждаются и систематизируются в нашей обзорной работе [32].

В работе [31] определяется класс задач обобщенного проектного планирования Generally Constrained Project Scheduling Problem (GCPSP) и формулируются утверждения о сводимости задач RCPSP в расширенных постановках к задачам GCPSP. Существенно, что последние формулируются в математически нейтральной форме, которая определяет лишь тип целевой функции и вид алгебраических ограничений, возникающих в задачах проектного планирования. Таким образом, универсальность каркаса может обеспечиваться путем предоставления развитого набора программных модулей для задания условий задач проектного планирования в постановке GCPSP и их решения. Для математической редукции прикладных задач достаточно проинтерпретировать их условия в терминах постановки проектного планирования GCPSP, разрешить ее и представить результаты в представлении исходной задачи.

Требования эффективности также нуждаются в некоторых пояснениях. Поскольку задачи проектного планирования RCPSP и GCPSP являются NP-полными, а для индустриальной практики представляют интерес проекты с количеством работ, исчисляемых десятками и сотнями тысяч, главное внимание должно уделяться быстрым алгоритмам, обеспечивающим поиск

приближенных решений за полиномиальное время. Лучший из известных точных алгоритмов Брукера за приемлемое время может решать задачи размерности не больше 60 [33], что делает невозможным его использование в обсуждаемых индустриальных приложениях. Вместе с тем, точные алгоритмы могут применяться для валидации приближенных алгоритмов и, в частности, для выбора и настройки применяемых в них эвристических правил. Поэтому состав каркаса может предусматривать программные модули, реализующие и некоторые точные алгоритмы. Однако вопросы эффективности становятся некритичными, поскольку оценка качества найденных приближенных решений может осуществляться на тестовых задачах очень низкой размерности.

Под гибкостью каркаса как программно-инструментальной среды подразумевается возможность реализации новых математических моделей, методов и приложений теории расписаний при относительно низких затратах на доработку имеющихся программных модулей. Поскольку задачи теории расписаний редуцируются к соответствующей обобщенной постановке проектного планирования GCPSP, для задания условий и решения которой основные программные модули уже реализованы и включены в состав каркаса, доработка целевых приложений потребует небольших затрат. В тех случаях, когда необходимо решать частные классы задач за оптимальное время, потребуются дополнительные усилия на программную реализацию специальных алгоритмов. В предположении, что они основаны на уже реализованных в каркасе алгоритмах или используют общую с ними вычислительную стратегию, подобные затраты также могут быть минимизированы.

Таким образом, обсуждаемые требования универсальности, эффективности и гибкости, предъявляемые к каркасу, могут быть удовлетворены на основе изложенных выше принципов.

2.3 Организация и состав классов SAF-каркаса

Разработанный каркас представляет собой систему классов (в дальнейшем, учитывая практическую реализацию на языке Си++, будем использовать принятые термины «класс», «конкретный класс», «абстрактный класс» и «интерфейс»). В организации каркаса выделим следующие группы классов:

- классы решателей (Solvers), реализующие общие алгоритмические схемы решения задач GCPSP (Schedulers), а также эвристики для поиска приближенных решений (Heuristics);
- классы математических объектов (Mathematics), предназначенные для задания условий и представления результатов проектного планирования в обобщенной постановке GCPSP;
- классы математической редукции (Reductions), обеспечивающие сводимость прикладных задач составления расписаний к постановке

GCPSP и соответствующую интерпретацию прикладных данных;

- классы прикладных данных (Applications), используемые для представления условий и результатов решения задач проектного планирования RCPSP в расширенных постановках;
- классы средств визуализации (Visualizations), предназначенные для графического отображения результатов планирования, в том числе, с использованием текстовых отчетов, графиков, диаграмм.

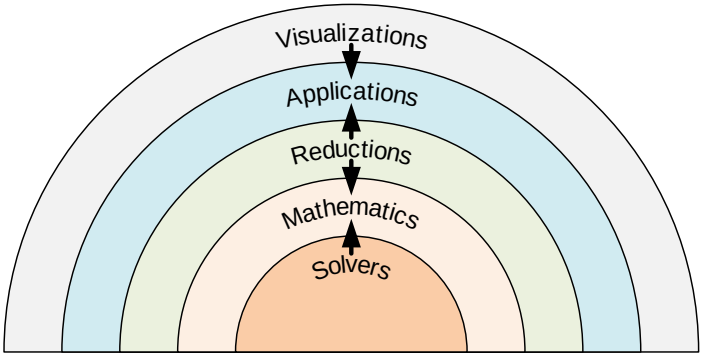


Рис. 1. Организация пакетов классов каркаса и основные отношения использования
Fig. 1. The organization of packages of classes of the framework and the basic relations of use

Классы средств визуализации будут рассмотрены в заключительных разделах, посвященных методологии разработки приложений, внедрению разработанного каркаса и созданию перспективной системы визуального планирования и управления проектами.

В следующих разделах остановимся более подробно на основных группах классов, непосредственно связанных с решением задач теории расписаний. Заметим, что часть из них реализуется как конкретные классы, допускающие непосредственное конструирование объектов. Другая часть представляет собой интерфейсы или абстрактные классы, которые по существу определяют точки расширения каркаса и позволяют предоставить их альтернативные реализации. Примечательно, что некоторые функции каркаса, в частности общие алгоритмические схемы, могут реализовываться в абстрактных классах без конкретизации типов условий решаемых задач и особенностей конкретных алгоритмов. Более того, такой способ реализации каркаса является рациональным с точки зрения повторного использования модулей и функциональной эволюции целевых приложений.

3. Организация классов прикладных данных

В данном разделе подробно описываются классы и интерфейсы каркаса, относящиеся к группе Applications и позволяющие задать условия и

результаты задач проектного планирования в расширенных постановках RCPSP.

3.1 Класс «Проект» (Project)

Класс Project агрегирует в себе все данные, необходимые для математически корректной постановки задачи проектного планирования, и предоставляет необходимые интерфейсы доступа к ним. Сам проект представляется иерархией связанных между собой работ с назначенными календарями, ресурсами и счетами. В рамках ООП перечисленные понятия реализуются соответствующими классами Project, Task, Link, Calendar, Resource, Account соответственно. Классы Task и Resource являются абстрактными, что означает невозможность создания экземпляров и необходимость предоставления конкретных реализаций методов, объявленных в интерфейсах данных классов. В следующих подразделах подробно описываются особенности подобных реализаций. В частности, поясняются способы определения простых и составных типов работ и ресурсов. Дополнительные классы TaskRate, ResourceRate, ResourceUse, Supply и Replenishment используются для ассоциирования работ, ресурсов и счетов между собой и параметризации подобных отношений.

Перечисленные выше классы являются конкретными, однако следует принять во внимание, что в их основу положены довольно общие параметрические модели, охватывающие расширенные постановки обсуждаемого класса задач RCPSP. Вместе с тем, при необходимости данные модели могут быть развиты и реализованы путем непосредственного наследования классов каркаса.

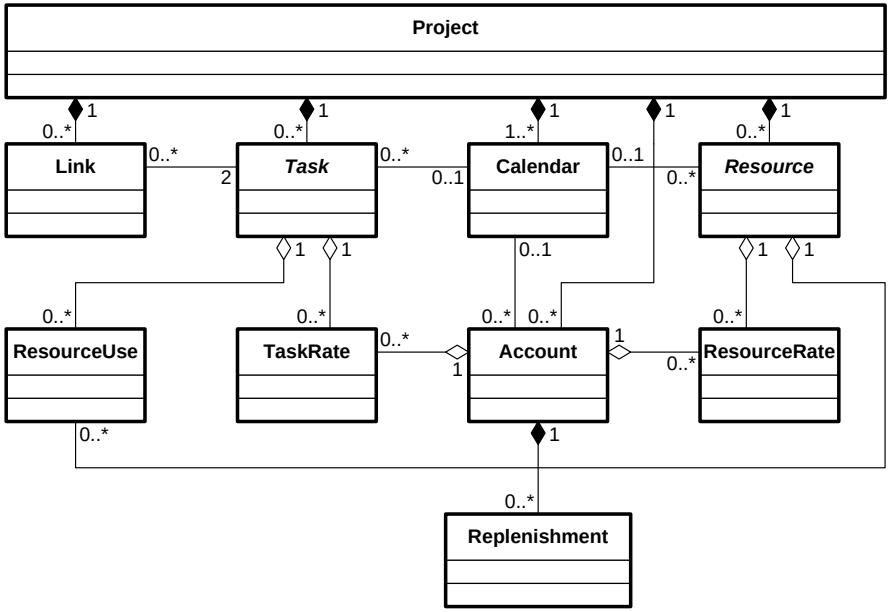


Рис. 2. UML-диаграмма основных классов прикладных данных
Fig. 2. UML diagram of main classes of application data

Кроме агрегации экземпляров указанных на диаграмме классов, класс Project определяет собственные атрибуты. Такими атрибутами являются проектный календарь, используемый в качестве основного в тех случаях, когда не определён индивидуальный календарь для работ, ресурсов и счетов, а также временные проектные ограничения, определяющие время начала и/или завершения проекта и возможную стратегию прямого и обратного планирования проектных работ (как можно раньше и как можно позже соответственно).

3.2 Календарные данные

Работа с календарной информацией занимает важное место при постановке и решении задач проектного планирования. Сами календари представляют собой объекты с довольно сложной организацией данных и нетривиальными операциями пересчета календарных дат, времен, рабочих интервалов. Поэтому в их реализации используются вспомогательные классы Time, Date, DateAndTime, TimeInterval, Duration, DayOfWeek, WorkWeek, MonthOfYear, RecurrencePattern, RecurrenceTimeInterval, которые в составе каркаса выполняют и самостоятельные функции. Рассмотрим их более подробно.

3.2.1 Класс «Время» (Time)

Класс Time предназначен для представления времени в рамках одних суток с точностью до долей секунды. Переменная времени может принимать любое значение от 00:00:00 до 24:00:00. Класс реализует методы для установки времени суток с помощью компонентов-значений часа, минуты, секунды и долей секунды, а также методы для получения соответствующих компонентов-значений времени суток и его строкового представления. Интерфейс класса предусматривает необходимые логические операции сравнения текущего времени с заданным значением, а также арифметические операции добавления заданной продолжительности к текущему времени и ее вычитания из текущего времени. Результат последних операций всегда приводится к суточному временному интервалу.

3.2.2 Класс «Дата» (Date)

Класс Date используется для представления календарной даты в виде компонентов-значений числа, месяца и года. Интерфейс класса предусматривает методы для установки календарных дат, сравнения дат, добавления к дате и вычитания из неё заданной продолжительности, вычисления продолжительности временного интервала между текущей датой и заданной.

3.2.3 Класс «Дата и время» (DateAndTime)

Класс DateAndTime предназначен для консолидированного представления календарной даты и времени суток, позволяющего оперировать абсолютными временными метками. Интерфейс класса во многом повторяет интерфейсы рассмотренных выше классов Date и Time.

3.2.4 Класс «Временной интервал» (TimeInterval)

Класс TimeInterval позволяет оперировать временными интервалами в пределах одних суток. Временной интервал задается нижней и верхней границей в предположении, что нижняя граница принадлежит интервалу, а верхняя — нет. Класс предоставляет методы для задания и получения границ временного интервала, вычисления его продолжительности, определения статуса принадлежности заданного момента времени текущему интервалу, а также вычисления теоретико-множественных операций пересечения и объединения текущего интервала с заданным интервалом.

3.2.5 Класс «Продолжительность» (Duration)

Данный класс используется для представления продолжительности проектных работ и лагов между ними. Продолжительность исчисляется с точностью до долей секунды и может быть положительной, отрицательной или нулевой. В классе реализуются арифметические операции сложения и вычитания продолжительностей, операции умножения и деления текущей продолжительности на число, а также логические операции сравнения заданных продолжительностей.

3.2.6 Тип данных «День недели» (DayOfWeek)

Тип данных DayOfWeek предназначен для представления дней недели и естественным образом реализуется как перечислимый тип с семью предопределенными значениями: Monday, Tuesday, Wednesday, Thursday, Friday, Saturday и Sunday.

3.2.7 Тип данных «Месяц года» (MonthOfYear)

Тип данных MonthOfYear предназначен для представления месяцев в году и реализуется в каркасе как перечислимый тип с двенадцатью предопределенными значениями: January, February, March, April, May, June, July, August, September, October, November и December.

3.2.8 Класс «Рабочая неделя» (WorkWeek)

Класс WorkWeek позволяет приписать логические признаки дням недели, например, с целью задания и определения их рабочего статуса. Класс реализуется как массив из семи логических значений, каждое из которых соответствует определенному дню недели в перечислении DayOfWeek. Интерфейс класса позволяет определить статус заданного дня недели и при необходимости изменить его.

3.2.9 Тип данных «Регулярное правило» (RecurrencePattern)

Обычно временные интервалы в рабочих календарях подчиняются некоторым регулярным правилам. Например, рабочие часы организации могут повторяться «ежедневно», «еженедельно», «ежемесячно», «в определённое число каждого месяца» и т. д. Для описания подобных регулярных правил в составе каркаса предусмотрен вспомогательный перечислимый тип данных RecurrencePattern со следующими предопределенными значениями: Daily, Weekly, Monthly, MonthlyByDayOfMonth, MonthlyByPosition, ByDayCount, ByWeekdayCount, YearlyByDayOfMonth, YearlyByPosition. Именованные значения имеют понятную интерпретацию.

3.2.10 Класс «Регулярный временной интервал» (RecurrenceTimeInterval)

Класс RecurrenceTimeInterval предназначен для задания регулярных временных интервалов. Отличием от рассмотренного выше класса TimeInterval является более компактный, не избыточный способ представления временных интервалов в тех случаях, когда они подчиняются регулярным правилам, специфицируемым типом RecurrencePattern. Использование в подобных случаях класса TimeInterval привело бы к необходимости конструирования и последующего анализа огромного числа интервальных объектов для каждого проектного дня. Класс RecurrenceTimeInterval обобщает модель данных класса TimeInterval путем определения дополнительных атрибутов для задания регулярного правила, начальной и конечной даты его применения и соответствующих ему параметров (например, «день недели», «число месяца», «день года» и т.п.).

3.2.11 Класс «Календарь» (Calendar)

Исполнение работ и привлечение ресурсов в рамках проектной деятельности обычно осуществляется на основе рабочих календарей. Корректная постановка задачи проектного планирования предполагает, что должен быть определен, по крайней мере, один проектный календарь, используемый по-умолчанию. Для работы с календарями в состав объектно-ориентированного каркаса включен конкретный класс Calendar.

Календари могут быть рационально организованы в виде двух множеств, одно из которых соответствует регулярным интервалам рабочего времени, а другое — их исключениям в особые календарные дни. В классе Calendar для этих целей используется две коллекции workTime и exception с элементами соответствующего типа RecurrenceTimeInterval. Первая коллекция используется для описания типовой рабочей недели (например, с понедельника по пятницу с 9:00 до 17:00). Вторая коллекция используется для описания исключений, которые могут происходить, например, в праздничные или предпраздничные дни. Фактическое рабочее расписание получается путем теоретико-множественной операции вычитания исключительных интервалов из основных рабочих интервалов. Выполнение данной операции может быть сопряжено со значительным объемом вычислений, поэтому следует избегать избыточной фрагментации исключений. Например, ежедневный обеденный перерыв с 13:00 до 14:00 нерационально описывать в виде исключения, а лучше представить прерываемыми рабочими интервалами.

Календари организуются в виде иерархической структуры с введенным отношением наследования между родительскими и дочерними элементами. Наследуемый календарь может уточнять или переопределять фактическое рабочее расписание родительского календаря. Предполагается, что собственные рабочие интервалы расширяют расписание родителя, а собственные исключения — ограничивают. Фактическое расписание наследника получается путем объединения собственных рабочих интервалов с рабочими интервалами родителя после вычитания из них исключительных интервалов родителя и последующего вычитания из полученного результата исключительных интервалов наследника.

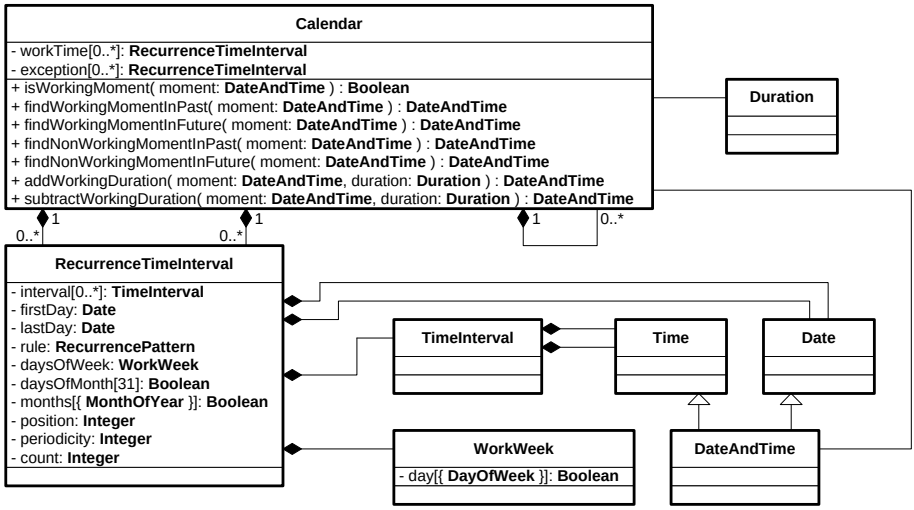


Рис. 3. UML-диаграмма классов календарных данных
Fig. 3. UML diagram of calendar data classes

Класс Calendar реализует развитый набор операций для определения статуса заданной даты или заданного временного интервала, вычисления ближайших рабочих и нерабочих дат календаря, а также для пересчета даты завершения (или начала) работы по заданной дате ее начала (или завершения) и продолжительности. Интерфейс класса предусматривает также операции unite и intersect, которые позволяют сконструировать новые календари, фактическое расписание которых является объединением или пересечением расписаний заданных календарей-операндов. Данные операции упрощают реализацию основных вычислительных процедур составления расписаний с учетом календарей, которые могут быть индивидуально приписаны проектным работам, ресурсам, ограничениям предшествования и нуждаются в комплексном анализе при пересчете фактических дат начала и завершения проектных работ.

3.3 Проектные работы

Каркас использует естественное разделение проектных работ на простые и составные. Простыми работами являются элементарные активности Activity, вехи StartMilestone, FinishMilestone и «гамаки» ShortHammock, LongHammock. Для многоуровневого представления проектного плана в виде иерархии работ используются структуры работ WBS (принятое сокращение от Work Breakdown Structure) и мультимодальные работы MultimodalTask. Все перечисленные виды работ в каркасе реализуются соответствующими конкретными классами, наследуемыми от общего абстрактного класса Task.

Мы допускаем, что некоторые атрибуты, декларируемые в нем, могут оказаться производными в конкретных реализациях и поэтому попытки их установки могут приводить к исключительным ситуациям. Тем не менее, наличие общего интерфейса Task оказывается важным фактором, предопределяющим единую дисциплину реализации классов работ.

3.3.1 Класс «Работа» (Task)

Абстрактный класс Task определяет общие методы получения и задания общих параметров работы. Прежде всего, это — планируемые и фактические даты начала и завершения работы, ее продолжительность, рабочий календарь, наложенные ограничения, используемые ресурсы, стоимость, счёт, приоритет, режим исполнения, статус и процент выполнения. В зависимости от вида работы логика реализации методов претерпевает существенные изменения и поэтому вынесена на уровень конкретных классов. Особенности полиморфной реализации далее обсуждаются на примере методов получения временных параметров работы.

Кратко уточним назначение перечисленных атрибутов работ. Рабочий календарь — календарь, в соответствии с которым осуществляется планирование и исполнение работы. В определении класса Task он реализуется опциональной ссылкой на объекты рассмотренного выше класса Calendar. При ее отсутствии применяется календарь проекта, наличие которого обязательно при конструировании объектов класса Project.

Статус определяет планируемое, стартованное, прерванное, возобновленное или финишированное состояние работы и представляется перечислимым типом TaskStatus с соответствующими значениями Planned, Started, Suspended, Resumed, Finished. Приоритет — натуральное число, характеризующее предпочтения пользователя по приоритизации работ в ходе составления расписания. Обычно необходимость в их использовании возникает, когда одновременное выполнение работ невозможно из-за ограниченности общих ресурсов и требуется принять решение о порядке их выполнения.

Явные временные ограничения определяют допустимые интервалы или полуинтервалы для дат начала или завершения работ. Данные ограничения снабжаются спецификатором обязательности или приоритетности над ограничениями предшествования в тех случаях, когда формируемая система алгебраических уравнений и неравенств перегружена и не может быть полностью разрешена. Если спецификатор предписывает обязательное выполнение временного ограничения и это препятствует каким-либо ограничениям предшествования, то расписание строится до конца, но снабжается отчетом о неразрешенных ограничениях.

Правило выравнивания устанавливает необходимость принудительного выравнивания планируемой даты на начало или конец каждой минуты, часа, суток, недели, месяца или года и представляется в каркасе перечислимым типом SnapMode с соответствующими значениями SnapToMinute,

SnapToHour, SnapToDay, SnapToWeek, SnapToMonth, SnapToYear. Правило выравнивания является опциональным атрибутом работы и может рассматриваться в качестве специфического временного ограничения, определяющего область допустимых значений для даты начала работы.

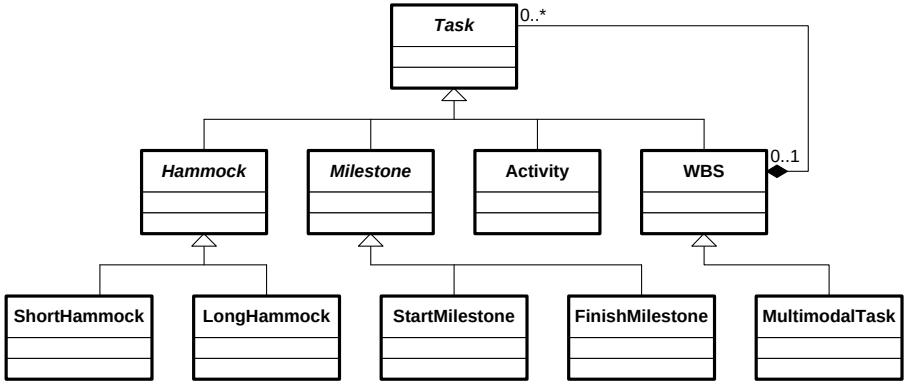


Рис. 4. UML-диаграмма иерархии классов работ
Fig. 4. UML diagram of the hierarchy of task classes

Для некоторых видов работ может быть указано максимально допустимое число прерываний, которое определяет возможные режимы их исполнения. Если данный атрибут не задан, то предполагается, что количество прерываний не ограничено. Нулевое значение интерпретируется как недопустимость прерываний.

3.3.2 Класс «Активность» (Activity)

Класс Activity реализует понятие простых операций, которые представляют собой терминальные работы в иерархическом представлении проектного плана. Конкретный класс Activity наследует и реализует интерфейс класса Task таким образом, что любой параметр работы может быть задан индивидуально, но при одновременной коррекции других связанных с ним параметров. Тем самым, обеспечивается семантическая согласованность представления данных. Например, при установке планируемых дат начала и завершения работы, пересчитывается ее продолжительность. При установке доли выполнения пересчитывается прогнозируемое время завершения работы и т.п.

3.3.3 Классы «Вехи» (Milestone)

Близкое поведение реализуют классы StartMilestone и FinishMilestone. Основным отличием вех от простых операций является нулевая продолжительность и, как следствие, совпадающие даты начала и завершения работ. При этом даты в классе StartMilestone принудительно выравниваются на момент времени, допустимый для начала работы, а в классе FinishMilestone

— на момент времени, допустимый для завершения работы. Установка даты начала вехи приводит к коррекции даты завершения и наоборот. Вызов метода установки продолжительности для вехи порождает соответствующее исключение.

3.3.4 Класс «Структура работы» (WBS)

Главной особенностью реализации класса WBS является наличие множественной композиции children на объекты типа Task, благодаря которой структуры работ могут содержать в себе дочерние работы любых типов, в том числе и работы более низких уровней. Таким образом, класс WBS позволяет структурировать проектный план в виде многоуровневой иерархии разнотипных работ. Структуры WBS не обязаны содержать дочерних работ, поскольку детализация проектного плана обычно происходит постепенно.

Явное задание временных параметров и других производных атрибутов WBS, используя методы наследуемого интерфейса Task, является некорректным и приводит к исключительным ситуациям. В самом деле, временные, ресурсные и стоимостные характеристики структуры работ определяются ее дочерними работами. Например, дата начала структуры работы определяется самым ранним стартом дочерних работ, а дата ее завершения — самым поздним финишем. Вычисление данных параметров осуществляется путем рекурсивного обхода многоуровневого представления структуры работ и уточнения минимальных и максимальных значений соответствующих дат в дочерних работах. Аналогичным образом вычисляются ресурсные и стоимостные характеристики структур работ. Последние, в частности, применяются при оценке качества найденных решений в постановках, нацеленных на минимизацию сроков и стоимости проекта в условиях жестких ресурсных ограничений.

3.3.5 Классы «Гамаки» (Hammock)

Подобно структурам работ WBS реализуются «гамаки» классов ShortHammock и LongHammock. Они не предусматривают явного задания дат работ, поскольку рассчитываются по временным параметрам предшествующих и последующих работ. Дата начала «Короткого гамака» определяется самым поздним финишем из всех предшественников, а дата его завершения — самым ранним стартом последователей. Для «Длинного гамака» дата начала совпадает с самым ранним финишем из всех предшественников, а дата завершения — с самым поздним стартом последователей. В случае отсутствия предшественников и/или последователей «гамак» вырождается в работу с продолжительностью, равной продолжительности проекта.

3.3.6 Класс «Мультимодальная работа» (MultimodalTask)

Важным требованиям к средствам планирования сложных проектов является возможность задания альтернативных режимов выполнения. С этой целью в состав каркаса включен класс MultimodalTask, определение которого во многом повторяет класс WBS из-за использования композиции дочерних

работ children. Однако логика реализации интерфейса Task принципиально отличается, поскольку выполнение структуры работ означает выполнение всех дочерних работ, а выполнение мультимодальной работы предполагает исполнение лишь одной из дочерних работ. Поскольку тип дочерних работ не конкретизируется композицией children в WBS и MultimodalTask, с их помощью удается строить сложные стратегии проектной деятельности, например, многоуровневые альтернативы структур работ. Однако следует иметь в виду, что из-за комбинаторной неопределенности наличие мультимодальных работ в представлении проекта существенно усложняет поиск расписаний, близких к оптимальным.

3.4 Класс «Связь работ» (Link)

Класс Link предназначен для задания отношений предшествования и синхронизации между работами при постановке задач проектного планирования. В классе определяются две обязательные ассоциации на объекты типа Task, одна из которых указывает на предшествующую работу (upstreamTask), а другая — на последующую (downstreamTask). Четыре опциональных атрибута типа Duration определяют минимальную и максимальную задержку синхронизации, пересчитанную в календарные даты с использованием рабочих календарей предшественника и последователя. Незаданные минимальные задержки (upstreamMinLag и downstreamMinLag) интерпретируются как нулевые, а незаданные максимальные задержки (upstreamMaxLag и downstreamMaxLag) — как бесконечно большие величины. Опциональная ссылка на календарь типа Calendar используется для пересчета дат в соответствии с собственным календарем.

Кроме этого, объекты класса Link имеют в качестве атрибута спецификатор связи, представленный перечислимым типом LinkType. Данный атрибут может принимать одно из следующих значений: SSLink, SFLink, FSLink или FFLink. Значение SSLink означает, что связь устанавливается между началом предшественника и началом последователя, SFLink — между началом предшественника и окончанием последователя, FSLink — между окончанием предшественника и началом последователя и FFLink — между окончанием предшественника и окончанием последователя. С учётом данного спецификатора минимальные и максимальные задержки приобретают более понятный смысл. Например, для связи с типом FSLink при заданной минимальной задержке данный вид связи устанавливает требование начать последующую работу не раньше, чем через установленное время после завершения предшествующей работы. При заданной максимальной задержке — не позже, чем через установленное время после завершения предшествующей. Примечательно, что задержки могут быть отрицательными и приводить к обратной последовательности выполнения предшественников и последователей.

3.5 Ресурсы

Каркас поддерживает несколько категорий ресурсов, реализуемых соответствующими классами простого ресурса SimpleResource, группового ресурса GroupResource, объединения ресурсов JointResource и семейства ресурсов FamilyResource. За исключением простого ресурса все остальные виды являются составными, что предполагает композицию дочерних или ассоциацию сторонних ресурсов.

3.5.1 Класс «Ресурс» (Resource)

Абстрактный класс Resource определяет базовый тип, от которого наследуются все перечисленные выше конкретные классы. Данный класс не предусматривает общий интерфейс для получения и задания ресурсных параметров в силу того, что простые и составные ресурсы параметризуются различными способами. Единственными общими методами, определяемыми в данном классе, являются правила исчисления стоимости ресурса.

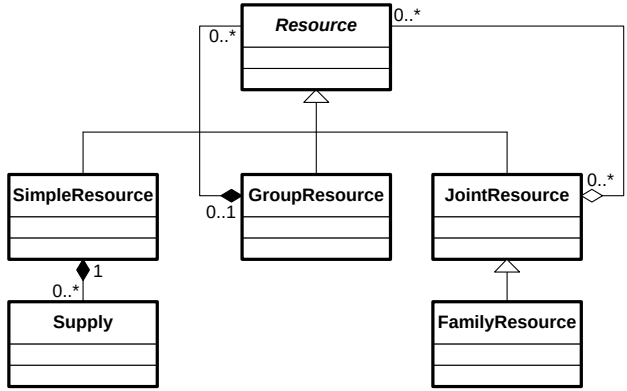


Рис. 5. UML-диаграмма иерархии классов ресурсов
Fig. 5. UML diagram of the hierarchy of resource classes

3.5.2 Класс «Простой ресурс» (SimpleResource)

Класс SimpleResource реализует понятие простого ресурса со следующим набором параметров. Это — признаки возобновимости и разделяемости ресурса, рабочие календари, временные лаги и доступное количество ресурса.

Признак возобновимости представляется перечислимым типом со значениями Renewable и Expendable. Первое значение указывает на возврат используемого количества ресурса по окончании каждой работы в общий пул. Второе устанавливает, что ресурс тратится в ходе выполнения каждой работы, где он используется, и его доступное количество в ходе выполнения проекта только уменьшается. Обычно к возобновимым ресурсам относят рабочий персонал и технику, а к невозобновимым — расходные материалы и энергетические ресурсы.

Признак разделяемости представляется перечислимым типом со значениями Discrete и Continuous, устанавливающими привлекается ли ресурс дискретным образом или непрерывным. Например, если некоторая работа выполняется за один день и ее трудоемкость составляет 1,5 человеко-дня, то данный признак позволяет решить, необходимо ли привлечь для ее выполнения двух сотрудников на целый день (Discrete) или в условиях неполной занятости они могут одновременно участвовать в других параллельных работах (Continuous). Следует отметить, что доступное количество дискретных ресурсов всегда исчисляется в целых. Оба признака являются обязательными атрибутами простого ресурса, поскольку участвуют в оценке его доступности и влияют на логику составления расписания.

Основной календарь устанавливает рабочее время, в которое ресурс доступен для проектных работ. Обычно в качестве основного календаря используется проектный календарь, в соответствии с которым выполняются все основные работы. Однако возможны ситуации, когда ресурсные календари могут иметь отличия, например, в силу технологических особенностей применяемого оборудования или индивидуальных планов сотрудников. В подобных случаях, фактический календарь выполнения работы строится путем теоретико-множественного пересечения ее собственного календаря с календарями всех используемых ресурсов. Для поддержки основного календаря используется обязательная объектная ссылка на класс Calendar.

Дополнительный календарь реализуется аналогичным образом, но является необязательной объектной ссылкой. Его основное назначение — планирование работ в сверхурочное время, обычно оплачиваемое по повышенным тарифам. Если целью планирования является скорейшее завершение проекта за счет возможного увеличения бюджета, то расписание строится с учетом основного и дополнительного календарей.

Временные лаги ресурса определяются как два опциональных атрибута класса, имеющие тип продолжительности Duration. Если атрибуты не установлены, то они интерпретируются как имеющие нулевую продолжительность. Первый атрибут определяет время доставки, установки или наладки ресурса перед его использованием в ходе выполнения работы, а второй атрибут — время остановки, демонтажа или возврата ресурса, необходимое для его освобождения и последующего использования в других работах. Временные лаги имеют спецификатор зависимости от количества используемого ресурса. При его задании итоговый временной лаг получается домножением на количество ресурса, используемого в конкретной работе. Перечисленные временные параметры также применяются в случаях прерывания и возобновления работ аналогично тому, как это происходит при их начале и завершении.

3.5.3 Класс «Поставка» (Supply)

Для получения доступного количества простого ресурса на заданный момент времени часто необходимо реконструировать историю поставок. С это целью в состав каркаса включен специальный класс Supply, а в классе SimpleResource определяется множественная композиция объектов данного класса. Множество объектов класса Supply реализует понятие «Цепочки поставок» путем определения для каждого объекта двух наборов атрибутов, определяющих планируемые и фактические значения следующих параметров: дата поставки, объем поставки или списания ресурса, стоимость организации всей поставки, стоимость за единицу ресурса, а также счет, если он отличается он единого счета ресурса. Атрибуты, связанные с планируемыми величинами, являются обязательными, а атрибуты, связанные с фактическими величинами — опциональными, поскольку актуализация поставок осуществляется уже в процессе проектной деятельности.

Объем поставки является ее ключевой характеристикой, поскольку предопределяет количество ресурса, доступное на начало проекта или на начало любой спланированной работы. В ходе выполнения проекта ресурсы могут захватываться, потребляться, освобождаться, генерироваться, в результате чего их доступное количество меняется.

При составлении расписания важно учесть, что на протяжении всего проекта доступное количество каждого ресурса не может быть отрицательным. Нарушение этого ограничения означает, что проектные работы спланированы неправильно и используют несуществующие объемы ресурса. Если проект не предусматривает пополнение или генерацию ресурса, то ни одна работа не может использовать ресурса больше, чем доступно на начало проекта. Это условие может проверяться уже на этапе постановки задачи.

3.5.4 Класс «Групповой ресурс» (GroupResource)

Класс группового ресурса GroupResource применяется для организации разнородных ресурсов в единое иерархическое многоуровневое представление в соответствии с требованиями пользователей. С этой целью в данном классе, наследуемом от абстрактного интерфейса Resource, определяется множественная композиция composedOf объектов типа Resource. Это позволяет в каждый групповой ресурс включить любое количество разнородных ресурсов, в том числе и групповые ресурсы более низких уровней.

3.5.5 Класс «Объединение ресурсов» (JointResource)

Класс объединения ресурсов JointResource во многом аналогичен классу GroupResource за исключением того, что вместо композиции composedOf определяется множественная ассоциация assembledFrom на объекты типа Resource. Класс предназначен для задания альтернативных способов группирования разнородных ресурсов, например, при формировании и

комплектовании бригад. Назначение объединенного ресурса на работу предполагает использование в работе всех его ассоциируемых ресурсов.

3.5.6 Класс «Семейство ресурсов» (FamilyResource)

Класс семейства ресурсов FamilyResource аналогичен классу JointResource и использует множественную ассоциацию associatedWith типа Resource, что позволяет в одном объекте группировать разные ресурсы. Однако принципиальным семантическим ограничением является требование, чтобы все ассоциируемые ресурсы были однородными. Например, если один из простых ресурсов является возобновимым, то и все остальные ресурсы, включенные в семейство, должны быть возобновимыми. Семейства ресурсов непосредственно назначаются на работы, однако любое такое назначение допускает использование любых комбинаций ассоциируемых родственных ресурсов. Например, если семейство ресурсов определяет группу сотрудников соответствующей специальности и квалификации, то назначение семейства на работу будет означать, что для выполнения работы может привлекаться любой свободный сотрудник или сотрудники, незанятые в период выполнения работы.

3.5.7 Класс «Использование ресурса» (ResourceUse)

Класс ResourceUse позволяет ассоциировать работу и используемый ей ресурс путем установки ссылок на соответствующие объекты и задания значений атрибутов, определяющих условия привлечения ресурса, включая количество или лимиты использования ресурса, профиль потребления или генерации ресурса на протяжении работы.

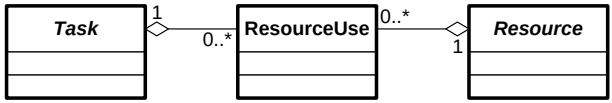


Рис. 6. UML-диаграмма класса использования ресурсов
Fig. 6. UML diagram of resource use class

Последний атрибут определяется как перечислимый тип ResourceProfile со значениями BackLoaded, BellShaped, FrontLoaded, Linear, OffsetTriangular, ThreeStep, Trapezoidal, TriangularDecrease, TriangularIncrease, DoublePeak, EarlyPeak, определяющими вид соответствующих скалярных функций одной переменной. Реальные профили потребления ресурса получаются путем масштабирования нормированных функций по оси абсцисс на период выполнения работы и по оси ординат на количество привлекаемого ресурса.

Следует иметь в виду, что установленное количество ресурса может быть отрицательным, что означает генерацию работой данного ресурса и возможность использования дополнительного количества другими работами. Примерами работ, генерирующих возобновимые и невозобновимые ресурсы,

могут служить краткосрочная аренда дополнительного оборудования и производство вспомогательных материалов в ходе проектной деятельности.

3.6 Финансовое обеспечение

Важным аспектом проектной деятельности является бюджетно-финансовое планирование и обеспечение. С точки зрения дизайна каркаса ключевыми элементами здесь являются бюджетный счет, а также различные правила исчисления стоимости работ и ресурсов.

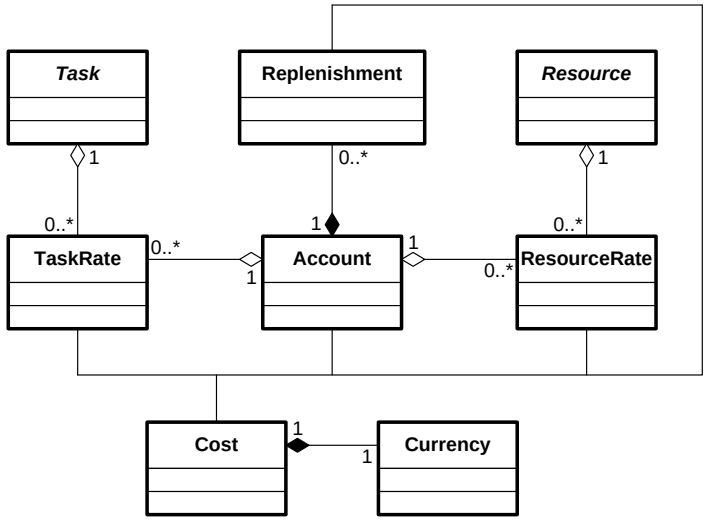


Рис. 7. UML-диаграмма классов финансового обеспечения
Fig. 7. UML diagram of financial support classes

3.6.1 Классы «Стоимость» (Cost) и тип данных «Валюта» (Currency)

Вспомогательный класс Cost реализует понятие стоимости, выраженной в денежно-валютном эквиваленте. Объекты данного класса представляются парой значений: денежным номиналом и видом валюты, в которой данный номинал представлен. Вид валюты задается перечислимым типом данных Currency. Номинал может быть положительным, отрицательным или нулевым. Для объектов класса определены арифметические операции сложения, вычитания, умножения и деления на вещественное число и логические операции сравнения. В случаях, когда валюты двух стоимостных операндов не совпадают, значения номиналов приводятся к единой валюте по предопределенному обменному курсу. Для подобных целей класс предусматривает статические методы установки кросс-курсов и применения их к заданным номиналам.

3.6.2 Тип данных «Стоимость за…» (CostType)

Тип данных CostType предназначен для спецификации выставляемой стоимости, ставки или тарифа. Другими словами, атрибуты данного типа позволяют определить, за что указана стоимость. Данный тип представлен в каркасе как перечисляемый тип со следующими предопределёнными значениями: FixedCost (стоимость фиксирована и не зависит от какой-либо продолжительности или количества), CostPerMinute, CostPerHour, CostPerDay, CostPerWeek, CostPerMonth, CostPerYear (приведенная стоимость за соответствующую единицу времени), CostPerUnitMinute, CostPerUnitHour, CostPerUnitDay, CostPerUnitWeek, CostPerUnitMonth, CostPerUnitYear (приведенная стоимость за использование одной единицы ресурса в течении единицы времени), CostPerUnit (стоимость за использование одной единицы ресурса).

3.6.3 Класс «Счёт» (Account)

Класс Account реализует понятие банковского счёта или кошелька. Основным атрибутом класса являются начальный баланс счёта initBalance типа Cost. Все счета в проекте представлены плоским списком, любой счёт может использоваться как для списания финансовых средств, так и для их пополнения. Счета могут быть ассоциированы с работами, ресурсами, календарями или проектом в целом.

3.6.4 Класс «Пополнение счёта» (Replenishment)

Для учёта доступности финансовых средств каркас предусматривает класс Replenishment, экземпляры которого описывают поступление финансовых средств на счёт. Каждый экземпляр класса Replenishment хранит объектную ссылку на счёт назначения (зачисления), сумму поступлений с указанием валюты (типы Cost и Currency), а также планируемую и актуальную даты поступления финансовых средств на счёт. При этом планируемая дата является обязательным атрибутом, а актуальная — опциональным. Используя приписанные объекты Replenishment можно реконструировать профиль доступных финансовых средств на счете. В этом смысле назначение и организация данного класса аналогична рассмотренному выше классу Supply.

3.6.5 Класс «Исчисление стоимости работы» (TaskRate)

Класс TaskRate позволяет ассоциировать работу и финансовый счет, откуда привлекаются средства для ее выполнения или куда зачисляются средства в случае ее прибыльности. Каждый экземпляр класса хранит объектные ссылки на работу и на финансовый счет, а также значения атрибутов, устанавливающих характер расходов или доходов (атрибут типа CostType), фиксированную или приведенную стоимость выполнения работы (атрибут типа Cost), стоимость прерываний работы (атрибут типа Cost) и профиль финансирования (атрибут типа CostProfile). Опциональными атрибутами класса являются актуальная стоимость и период действия применяемого

тарифа. Данные атрибуты необходимы, чтобы оценить штрафные санкции в случае задержки или опережения работ относительно планируемых дат.

Приведенная стоимость может быть отнесена к единице рабочего или календарного времени, а также к единице трудозатрат работы. Профиль финансирования представляется перечислимым типом CostProfile со значениями AtStart, AtEnd и Uniform, устанавливающими, что средства списываются со счета или зачисляются на счет в начале соответствующего временного периода, в его конце или расходуются равномерно на протяжении всего периода. Примечательно, что с одной и той же работой может быть ассоциировано несколько экземпляров данного класса. Кроме того, следует принимать во внимание, что в роли ассоциированной работы могут быть не только простые активности, но и любые другие виды работ (вехи, гамаки, структуры работ или мультимодальные работы).

3.6.6 Класс «Исчисление стоимости ресурса» (ResourceRate)

Организация класса ResourceRate «Исчисление стоимости ресурса» аналогична рассмотренному выше классу TaskRate за исключением того, что он определяет не стоимость выполнения работы, а стоимость привлечения использования ресурсов при выполнении работ. В данном классе определяются объектные ссылки на соответствующие экземпляры ресурса и счёта, опциональные даты начала и конца действия тарифа, планируемые и актуальные значения стоимости, характер стоимости и её выплаты. С одним ресурсом может быть ассоциировано несколько экземпляров данного класса. Итоговая стоимость использования ресурса в той или иной работе будет определяться как сумма затрат по каждому из тарифов с учётом временных показателей работы. Коллекция объектов ResourceRate с установленными датами действия тарифов позволяет реконструировать всю историю изменений стоимости ресурса в ходе проектной деятельности.

4. Организация классов математических объектов и решателей

Пакеты классов математических объектов (Mathematics) и решателей (Solvers) в определённой степени изолированы от классов прикладных данных (Applications), рассмотренных выше. Данные классы реализуют математические понятия и алгоритмы теории расписаний. Для сведения прикладных задач к обобщенной постановке проектного планирования, формулируемой в математически нейтральной форме, предусмотрены специальные классы редукции (Reductions), которые реализуют интерфейсы математических объектов с учетом особенностей прикладных задач и, тем самым, выполняют функции посредников между прикладными и математическими классами каркаса.

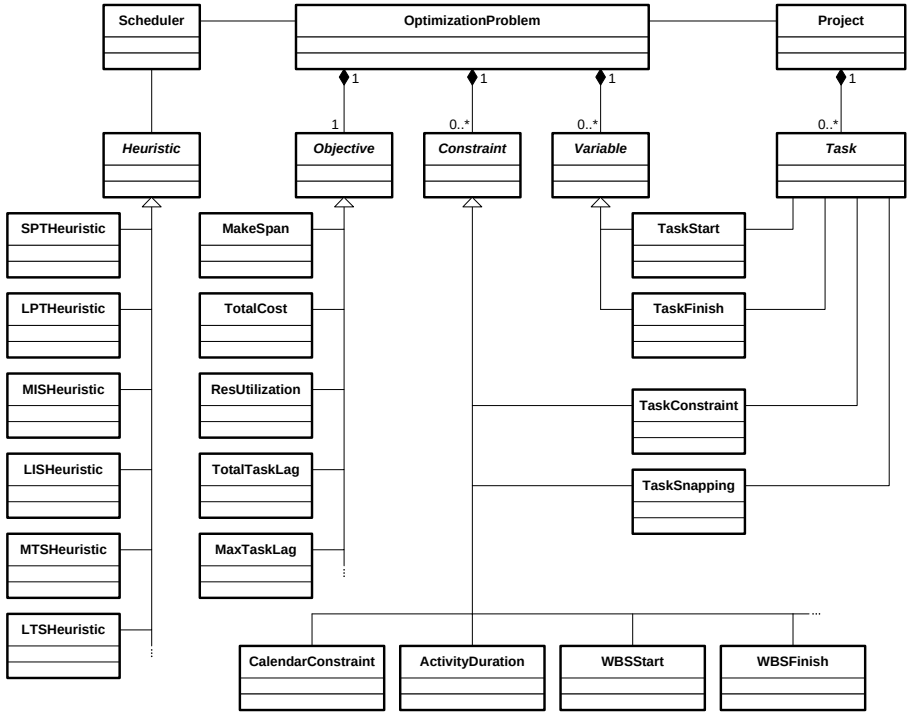


Рис. 8. UML-диаграмма классов математических объектов и решателей
Fig. 8. UML diagram of classes of mathematical objects and solvers

4.1 Класс «Оптимизационная задача» (OptimizationProblem)

Класс OptimizationProblem предназначен для постановки задачи условной нелинейной оптимизации путем задания множества переменных, целевой функции и системы нелинейных алгебраических ограничений. Класс реализуется как композиция всех математических объектов, участвующих в постановке задачи. Такими объектами являются переменные задачи класса Variable, целевая функция класса Objective и алгебраические ограничения класса Constraint.

Поскольку постановка оптимизационной задачи полностью определяется особенностями прикладной задачи проектного планирования, конструирование объектов OptimizationProblem и их наполнение условиями задачи осуществляется в классе Project с помощью вызова метода initOptimizationProblem(). Реализация данного метода предполагает обход всех экземпляров прикладных классов, ассоциируемых с проектом, и вызов соответствующих одноимённых методов для них. В результате каждый экземпляр прикладных классов, в том числе и сам проект, конструирует и инициализирует соответствующие математические объекты и включает их в

композицию класса OptimizationProblem. Заметим, что конструируемые математические объекты являются экземплярами редукционных классов, которые наследуют интерфейсы математических классов Variable, Objective, Constraint и реализуют их с учетом особенностей прикладной задачи. С этой целью они хранят ссылки на соответствующие прикладные данные. Например, для инициализации целевой функции оптимизационной задачи типа Objective создается соответствующий объект, который хранит ссылку на прикладной объект Project и через нее получает доступ ко всем проектным данным, участвующим в вычислении значений целевой функции. Аналогично задаются алгебраические ограничения. Опишем реализацию используемых вспомогательных классов более подробно.

4.2 Класс «Область допустимых значений» (ValueDomain)

Вспомогательный класс ValueDomain предназначен для задания области допустимых значений переменных в решаемой задаче. Поскольку переменные задачи могут выражать разные понятия и описываться разными типами, то наиболее рациональной представляется реализация класса ValueDomain в виде шаблона, параметризуемого типом переменной. В обсуждаемой обобщенной постановке проектного планирования все переменные задачи связаны с временными характеристиками, поэтому применяется конкретная реализация.

Область значений может быть представима отдельными точками, интервалами, бесконечными полуинтервалами, причем все элементы множества не пересекаются и упорядочены. Это обеспечивает эффективность операций проверки принадлежности заданного значения или интервала заданной области, пересечения и объединения заданных областей.

В интерфейсе данного класса для этих целей предусмотрены соответствующие методы:

- boolean isEmpty() — возвращает True, если множество допустимых значений пусто, и False в противном случае;
- boolean contains(ValueDomain&) — возвращает True, если множество допустимых значений полностью содержит заданную область, и False в противном случае;
- ValueDomain unite(ValueDomain&) — объединяет текущую область допустимых значений с другой, переданной в качестве параметра, и возвращает результат как новый экземпляр класса;
- ValueDomain intersect(ValueDomain&) — пересекает текущую область допустимых значений с другой, переданной в качестве параметра, и возвращает результат как новый экземпляр класса.

4.3 Интерфейс «Переменная» (Variable)

Интерфейс Variable определяет методы доступа к переменным задачи проектного планирования, получения, хранения и установки их значений. Используя интерфейс Variable и стандартные шаблоны коллекций, можно определить вектор переменных задачи, например, как ArrayOf<Variable*>.

Поскольку данные переменные связаны с прикладными данными, а именно с временными характеристиками проектных работ, то в составе каркаса предусмотрены конкретные классы TaskStart и TaskFinish, которые наследуя интерфейс Variable, предоставляют доступ к дате и времени начала и завершения работ типа DateAndTime. Кроме заданных значений даты и времени, каждая переменная может принимать неустановленное значение Unset и неизвестное значение Unknown. Например, неизвестное значение присваивается всем переменным перед решением задачи для указания необходимости выполнения соответствующих вычислений. Статус неустановленной переменной означает, что данная переменная была исключена из процесса решения в силу логических условий, связанных с выполнимостью соответствующих работ. Кроме методов доступа к переменным интерфейс Variable определяет метод получения ограничений задачи, в которых данная переменная участвует. Данный метод используется в ходе составления расписаний при анализе и согласованном разрешении наложенных ограничений.

4.4 Интерфейс «Целевая функция» (Objective)

Обсуждаемая обобщенная постановка задач проектного планирования допускает использование альтернативных целевых функций. Поэтому целевая функция в составе каркаса определяется как абстрактный класс с единым интерфейсом, от которого наследуются возможные конкретные реализации, а именно: MakeSpan (время выполнения всего проекта), TotalCost (суммарная стоимость всего проекта), ResUtilization (использование ресурсов), TotalTaskLag (суммарная задержка по всем работам относительно их директивных сроков), MaxTaskLag (максимальная задержка из всех работ относительно директивных сроков) и другие.

Единый интерфейс Objective определяет два основных метода:

- float getValue(ArrayOf<Variable*>&) — возвращает вещественное значение целевой функции по заданному вектору переменных задачи. Реализация метода в конкретных классах должна допускать корректную работу в тех случаях, когда не все переменные имеют предустановленные значения, но может быть вычислена, например, нижняя оценка целевой функции;
- DateAndTime getArgMin(ArrayOf<Variable*>&, int index, ValueDomain&) — возвращает значение заданной переменной, при которой достигается минимум целевой функции на заданной области при фиксированных значениях других переменных.

4.5 Интерфейс «Ограничение» (Constraint)

Принципы организации интерфейса Constraint и наследуемых от него конкретных классов прикладных ограничений во многом аналогичны рассмотренным выше. Интерфейс Constraint определяет следующие методы:

- bool isSatisfied() — возвращает True, если ограничение удовлетворено на ассоциируемом с ним множестве переменных и False в противном случае;
- getVariables(ArrayOf<Variable*>&) — возвращает множество ассоциируемых с ограничением переменных;
- DependencyType getDependency(Variable&) — возвращает одно из значений Independent, Dependent, Vague, определяющее характер зависимости переменной с указанным индексом;
- int getPriority() — возвращает целочисленный приоритет ограничения, используемый при разрешении переопределенных систем;
- ValueDomain resolveFor(ArrayOf<Variable*>&, int index) — возвращает множество значений для заданной переменной, при которых ограничение разрешается при фиксированных других переменных. Предполагается, что реализация данного метода в ограничениях регулярного алгебраического вида должна учитывать переменные с возможным неизвестным состоянием, которые в этом случае интерпретируются как отсутствующие.

Классы прикладных ограничений, наследуя интерфейс Constraint, реализуют данные методы в результате доступа к соответствующим прикладным данным и поэтому отнесены к пакету Reductions. Например, класс CalendarConstraint, ассоциируемый с классом Calendar, реализует ограничение допустимого рабочего времени начала или завершения работы. Классы TaskConstraint и TaskSnapping, ассоциируемые с классом Task, реализуют явные временные условия и правила выравнивания работ, атрибуты которых являются фактическими параметрами порождаемых алгебраических ограничений.

Аналогичным образом реализуются другие классы прикладных ограничений. Класс ActivityDuration определяет строгую алгебраическую зависимость между временами начала и завершения работы с учетом её продолжительности и применяемого календаря. Классы WBSStart и WBSFinish определяют соответствующие зависимости старта и завершения структуры работ от соответствующих параметров дочерних работ. Классы ShortHammockStart, ShortHammockFinish, LongHammockStart и LongHammockFinish определяют аналогичные зависимости «гамаков» от взаимосвязанных предшественников и последователей, классы LinkMinLag и LinkMaxLag — условия предшествования работ с учётом минимального и максимального временного лага, ResourceConstraint — условие доступности ресурса, AccountConstraint — условие наличия средств на счете.

4.6 Интерфейс «Эвристика» (Heuristic)

Для решения задач проектного планирования в обобщенной постановке разработан приближенный алгоритм, основанный на эвристиках [31]. Каркас предоставляет реализацию данного алгоритма в виде соответствующего класса Scheduler. Однако использование последнего предполагает задание эвристик в виде упорядоченного множества объектов соответствующего типа Heuristic.

Интерфейс Heuristic определяет единственный метод:

- VariablePriority compare(Variable& var1, Variable& var2) — возвращает результат сравнения приоритетов пары переменных в виде одного из значений перечислимого типа VariablePriority: FirstOverSecond (первая переменная приоритетнее), SecondOverFirst (вторая переменная приоритетнее), Equal (приоритеты переменных равны). В качестве входных переменных метод принимает ссылки на сравниваемые переменные задачи.

Данный метод реализуется в конкретных классах, наследуемых от интерфейса Heuristic. В частности, каркас предоставляет готовые к использованию реализации следующих эвристик:

- MISHeuristic (Most Immediate Successors) — выбирает переменную с наибольшим количеством непосредственных переменных-последователей;
- LISHeuristic (Least Immediate Successors) — выбирает переменную с наименьшим количеством непосредственных переменных-последователей;
- MTSHeuristic (Most Total Successors) — выбирает переменную с наибольшим количеством всех переменных-последователей;
- LTSHeuristic (Least Total Successors) — выбирает переменную с наименьшим количеством всех переменных-последователей;
- SPTHeuristic (Shortest Process Time) — выбирает переменную с наименьшей разностью значений с переменными-последователями;
- LPTHeuristic (Longest Process Time) — выбирает переменную с наибольшей разностью значений с переменными-последователями.

Более строгое описание эвристик приводится в [31]. Поскольку конкретные классы переменных имеют непосредственную связь с прикладными объектами, формирующими вектор переменных, то становится возможной реализация предметно-ориентированных эвристик с учетом особенностей прикладной задачи и более быстрых способов составления расписаний.

Поскольку применение отдельной эвристики не гарантирует вынесение окончательного вердикта о приоритете одной переменной над другими, на практике применяются иерархические стратегии, состоящие в последовательном применении нескольких эвристик. В каркасе такая

возможность обеспечивается заданием упорядоченного множества разнотипных объектов типа Heuristic.

4.7 Класс «Решатель» (Scheduler)

Scheduler является конкретным классом, реализующий алгоритмы точного и приближенного решения оптимизационной задачи построения расписания. Интерфейс класса определяет пять основных метода:

- boolean state(OptimizationProblem&) — задаёт оптимизационную задачу составления расписания. Возвращает True в случае корректно заданных условий и False в противоположном случае;
- setHeuristics(ArrayOf<Heuristic*>&) — задать упорядоченное множество эвристик для приближенного решения;
- boolean solveApproximately() — ищет решение поставленной задачи, используя описанный приближенный алгоритм с предустановленными эвристиками. Возвращает True, если расписание было успешно составлено и False в противоположном случае;
- boolean solveExactly() — ищет решение заданной оптимизационной задачи, используя точный алгоритм границ и ветвей. Возвращает True в случае успешного поиска решения и False в противоположном случае, например, при исчерпании отведенного процессорного времени;
- generateReport(Report&) — генерирует отчет о процессе работы алгоритмов и качестве найденного приближенного решения.

Рассмотрим листинг программы на языке Си++, иллюстрирующий типовую последовательность вызова методов, а также реализацию основных и вспомогательных методов класса. Основное внимание уделим методу реализации приближенного алгоритма solveApproximately().

```
void main()
    {
    …
    Project project;
    …
    OptimizationProblem problem;
    project. initOptimizationProblem( problem );
    ArrayOf<Heuristic*> heuristics;
    heuristics.push_back( new MISHeuristic() );
    heuristics.push_back( … );
    …
    Scheduler scheduler;
    if ( scheduler.state( problem ) )
        {
```

```
        scheduler. setHeuristics( heuristics );
        if ( scheduler. solveApproximately() )
            {
            ArrayOf<Variable*> &allVariables = m_problem.getVariables();
            for ( int i = 0; i < allVariables.size(); ++i )
                allVariables[ i ].submit();
            }
        }
    Report report;
    scheduler. generateReport( report );
    …
    }

bool Scheduler::state( OptimizationProblem &problem )
    {
    m_problem = &problem;
    return true;
    }

void Scheduler::setHeuristics( ArrayOf<Heuristic*> &heuristics )
    {
    m_heuristics = &heuristics;
    }

bool Scheduler::solveApproximately()
    {
    ArrayOf<Variable*> &allVariables = m_problem.getVariables();
    checkAllVariablesAsUnknown( allVariables );
    ArrayOf<int> activeVariableIndices;
    computeIndependentVariableIndices( activeVariableIndices, allVariables );
    while ( activeVariableIndices.size() > 0 )
        {
        int selectIndex = selectVariable( activeVariableIndices, allVariables );
        if ( ! computeVariable( allVariables, selectIndex ) )
            return false;
        updateActiveVariableIndices( activeVariableIndices, allVariables, selectIndex );
        }
    return true;
    }
```

Вначале метод переводит переменные задачи в неизвестное состояние Unknown вызовом вспомогательного метода setAllVariablesAsUnknown(). Далее определяется множество независимых переменных с помощью метода computeIndependentVariableIndices() и инициализируется множество активных переменных activeVariableIndices. Дальнейшая работа алгоритма предполагает циклическую обработку данного множества. На каждом шаге цикла алгоритм выбирает одну из переменных на основе предустановленных эвристик (метод selectVariable()) и пытается найти её допустимое значение, удовлетворяющее всем ассоциированным с ней ограничениям (метод computeVariable()). Если допустимое значение найти не удаётся, то работа алгоритма прерывается с вердиктом о некорректно заданных условиях поставленной задачи. В случае успешного поиска множество активных переменных обновляется путем исключения найденной переменной и добавления новых переменных, зависящих только от уже обработанных (метод updateActiveVariableIndices()). Ниже приведён листинг перечисленных вспомогательных методов.

```
void Scheduler::checkAllVariablesAsUnknown( ArrayOf<Variable*> &variables ) const
    {
    for ( int i = 0; i < variables.size(); ++i )
        variables[ i ]->checkAsUnknown();
    }

void Scheduler::computeIndependentVariableIndices( ArrayOf<int> &activeVariableIndices,
                                                   const ArrayOf<Variable*> &allVariables ) const
    {
    activeVariableIndices.clear();
    for ( int i = 0; i < allVariables.size(); ++i )
        if ( isNonDependentVariable( allVariables[ i ]->getConstraints(), allVariables[ i ] ) )
            activeVariableIndices.push_back( i );
    }

bool Scheduler::isNonDependentVariable( const ArrayOf<Constraint*> constraints,
                                        const Variable* variable ) const
    {
    for ( int i = 0; i < constraints.size(); ++i )
        if ( constraints[ i ]->getDependency( variable ) == DEPENDENT )
            return false;
    return true;
    }

int Scheduler::selectVariable( const ArrayOf<int> &activeVariableIndices,
                               const ArrayOf<Variable*> &allVariables ) const
```

```
    {
    ArrayOf<int> candidates = activeVariableIndices;
    ArrayOf<int> result;
    for ( int i = 0; i < ( *m_heuristics ).size(); ++i )
        {
        getPriorityVariableIndices( result, candidates, allVariables, ( *m_heuristics )[ i ] );
        if ( result.size() == 1 )
            return ( result[ 0 ] );
        candidatIndices = result;
        }
    return ( result[ 0 ] );
    }

void Scheduler::getPriorityVariableIndices( ArrayOf<int> &result,
                                            const ArrayOf<int> &candidates,
                                            const ArrayOf<Variable*> &allVariables,
                                            Heuristic* heuristic ) const
    {
    result.clear();
    result.push_back( candidates [ 0 ] );
    for ( int i = 1; i < candidates.size(); ++i )
        {
        VariablePriority priority = heuristic->compare( ( *( allVariables[ result[ 0 ] ] ) ),
                                                        ( *( allVariables[ candidates[ i ] ] ) ) );
        if ( priority == FIRST_OVER_SECOND )
            continue;
        if ( priority == SECOND_ OVER_FIRST )
            result.clear();
        result.push_back( candidates[ i ] );
        }
    }

bool Scheduler::computeVariable( ArrayOf<Variable*> &allVariables,
                                 const int &selectedIndex ) const
    {
    ArrayOf<Constraint*> drivingConstraints;
    getDrivingConstraints( drivingConstraints,
                           allVariables[ selectedIndex ]->getConstraints(),
                           allVariables[ selectedIndex ] );
    sortConstraintsByPriority( drivingConstraints );
    if ( drivingConstraints[ 0 ]->canSkipVariable( allVariables[ selectedIndex ] ) )
```

```
        {
        allVariables[ selectedIndex ]->unset();
        return true;
        }
    ValueDomain resultD;
    ValueDomain tempD;
    for ( int i = 0; i < drivingConstraints.size(); ++i )
        {
        tempD = drivingConstraints[ i ]->resolveFor( allVariables, selectedIndex );
        if ( i == 0 )
            {
            if ( tempD.isEmpty() )
                {
                reportError( drivingConstraints[ i ], allVariables[ selectedIndex ] );
                return false;
                }
            else
                resultD = tempD;
            }
        else
            {
            tempD = tempD.intersect( resultD );
            if ( tempD.isEmpty() )
                reportUnresolvedConstraint( drivingConstraints[ i ],
                                            allVariables[ selectedIndex ] );
            else
                resultD = tempD;
            }
        }
    allVariables[ selectedIndex ] = ( m_problem->getObjective() ).getArgMin( allVariables,
                                                                              selectedIndex,
                                                                              resultD );
    return true;
    }

void Scheduler::getDrivingConstraints( ArrayOf<Constraint*> &drivingConstraints,
                                       const ArrayOf<Constraint*> constraints,
                                       const Variable* variable ) const
    {
    drivingConstraints.clear();
    for ( int i = 0; i < constraints.size(); ++i )
```

```
        if ( constraints[ i ]->getDependency( *variable ) != INDEPENDENT )
            drivingConstraints.push_back( constraints[ i ] );
    }

void Scheduler::updateActiveVariableIndices( ArrayOf<int> &activeVariableIndices,
                                             const ArrayOf<Variable*> &allVariables,
                                             const int &selectedIndex ) const
    {
    activeVariableIndices.erase( activeVariableIndices.find( selectedIndex ) );
    const ArrayOf<Constraint*> &constraints = allVariables[ selectedIndex ]->
                                                                    getConstraints();
    for ( int i = 0; i < constraints.size(); ++i )
        if ( constraints[ i ]->getDependency( *( allVariables[ selectedIndex ] ) ) ==
                                                                    INDEPENDENT )
            {
            const ArrayOf<Variable*> &assVars = constraints[ i ]->getVariables();
            for ( int j = 0; j < assVars.size(); ++j )
                if ( constraints[ i ]->getDependency( *( assVars[ j ] ) ) == DEPENDENT )
                    if ( isActiveVariable( assVars[ j ] ) )
                        activeVariableIndices.push_back( allVariables.find( assVars[ j ] ) );
            }
    }

bool Scheduler::isActiveVariable( const Variable* variable ) const
    {
    const ArrayOf<Constraint*> &constraints = variable->getConstraints();
    for ( int i = 0; i < constraints.size(); ++i )
        if ( constraints[ i ]->getDependency( *variable ) == DEPENDENT )
            {
            const ArrayOf<Variable*> &assVars = constraints[ i ]->getVariables();
            for ( int j = 0; j < assVars.size(); ++j )
                if ( constraints[ i ]->getDependency( assVars[ j ] ) == INDEPENDENT )
                    if ( assVars[ j ]->isUnknown() )
                        return false;
            }
    return true;
    }
```

5. Методология разработки приложений теории расписаний на основе SAF-каркаса

Одним из принципиальных требований, предъявляемых к SAF-каркасу, была возможность повторного использования имеющихся модулей при

программной реализации новых моделей, методов и приложений теории расписаний при относительно низких затратах на доработку.

Во многом данное требование удается удовлетворить благодаря принятым в качестве методологической основы принципам объектно-ориентированного программирования и оригинальной многослойной архитектуре каркаса. Рассмотрим их более подробно на примере разработки программных приложений теории расписаний и, в частности, приложений календарно-сетевого планирования.

Каркасом предусматривается довольно развитый набор готовых к использованию модулей и поэтому разработка типового приложения календарно-сетевого планирования, главным образом, сводится к реализации графического интерфейса пользователя (GUI), подключению и конфигурированию имеющихся модулей. В приведенной многослойной архитектуре каркаса элементы GUI составляют самый внешний слой, имеют непосредственный доступ к прикладным данным и могут использовать средства стандартных графических библиотек. Например, популярные GUI библиотеки, такие как Qt, MFC, BCG, предоставляют средства для визуализации проектного плана в виде диаграммы Ганта, построения графиков и диаграмм общего вида, отображения календарей и других данных, характерных для календарно-сетевого планирования.

Для поддержки приложением требуемых функций управления прикладными данными достаточно воспользоваться пакетом Applications, а для решения соответствующих задач составления расписаний — пакетами Reductions, Mathematics и Solvers. Поскольку они предоставляют все необходимые классы для задания условий задач проектного планирования в расширенных постановках и их решения, то реализация данных функций сводится к использованию классов прикладных данных и решателей. Если известны математические особенности прикладной задачи, а к ее решению предъявляются повышенные требования эффективности, то можно сконфигурировать классы решателей соответствующими целевыми функциями и эвристиками, реализации которых также включены в состав каркаса. При этом следует учесть, что выбор эвристик и порядок их применения во многом диктуется целевой функцией оптимизационной задачи.

Таким образом, разработка типового приложения календарно-сетевого планирования с функциями управления данными и решения задач проектного планирования RCPSP в расширенных постановках требует относительно низких затрат, обусловленных, главным образом, следующими работами:

- разработка GUI целевого приложения на основе графических библиотек общего назначения;
- использование имеющихся классов прикладных данных для задания условий задачи планирования;
- использование и конфигурирование имеющих классов решателей соответствующими целевыми функциями и эвристиками для

эффективного приближенного решения поставленной задачи планирования.

В других, более специальных случаях целевых приложений могут потребоваться дополнительные усилия, связанные с выполнением некоторых из перечисленных ниже работ:

- развитие пакета классов Applications для редукции задач теории расписаний к постановке RCPSP;
- развитие пакета классов Applications для представления условий задач RCPSP в расширенных постановках;
- развитие пакета классов Reductions для редукции прикладной задачи к математической постановке условной оптимизации GCPSP;
- развитие пакета классов Solvers для реализации новых точных и приближенных алгоритмов, а также новых эвристик для них.

Обсудим перечисленные возможности разработки целевых приложений теории расписаний в результате развития и конфигурирования классов каркаса.

5.1 Развитие пакета Applications для редукции задач теории расписаний к постановке RCPSP

Обычно задачи теории расписаний не формулируются как задачи проектного планирования, хотя в большинстве случаев могут быть проинтерпретированы в их терминах или сведены к ним за полиномиальное время. В подобных случаях требуется разработка специальных классов прикладных данных, выражающих соответствующие понятия рассматриваемой предметной области и реализующих их, например, с помощью классов пакета Applications.

В качестве примера рассмотрим задачу составления школьного расписания. Будем считать, что в школе занятия проводятся в соответствии с единым регулярным расписанием уроков и перемен, повторяющимся каждый день с понедельника по пятницу. Исключение составляют дни каникул и праздничные дни. В терминах проектного планирования временной аспект функционирования школы описывается с помощью понятия рабочего календаря. Поскольку каркас предоставляет обобщенную реализацию рабочего календаря с регулярными и исключительными правилами, для представления школьного календаря можно воспользоваться соответствующим классом Calendar. Школьный календарь может быть реализован в целевом приложении с помощью конструирования соответствующего объекта и инициализации его атрибутов. Альтернативная реализация состоит в определении класса-наследника и в уточнении рабочих интервалов, регулярных правил и исключений непосредственно в его конструкторе.

У каждого класса учеников есть свой предопределённый образовательными стандартами план занятий на год и на каждый семестр. В терминах проектного

планирования циклы занятий представляются вложенными структурами работ, а каждое занятие или урок в отдельности — активностью с продолжительностью 45 минут. При реализации целевого приложения можно воспользоваться классами WBS и Activity, входящими в состав пакета каркаса Applications, а в целевом приложении построить годовой план школьных занятий. Для упрощенного задания циклов занятий по отдельным предметам и количествам учебных часов можно определить специальные классы-наследники Cycle и Lesson и реализовать в них необходимые вспомогательные методы.

Для проведения занятия в определенное время необходимо, чтобы были свободны ученики класса, а также были доступны учитель и учебный класс. В терминах задачи RCPSP ученики класса, учитель и учебный класс следует рассматривать как возобновимые ресурсы соответствующих типов с доступным единичным количеством. При этом учитель обычно проводит весь цикл занятий для одного класса по одному из своих предметов. А занятия по некоторым предметам требуют специально оборудованных классов. Данные условия в рамках задачи RCPSP задаются путем назначения индивидуальных ресурсов на соответствующие работы. Обычно для организации циклов занятий по отдельным предметам необходимо предусмотреть временные интервалы между ними, например, для подготовки домашних занятий или отдыха учащихся. Данные условия естественным образом интерпретируются в терминах отношений предшествования между работами с соответствующими задержками. Для этих целей может быть определен класс Rule, который наследуя класс каркаса Link, уточняет способ параметризации данных условий.

Таким образом, рассмотренный пример задачи составления школьного расписания может быть сведён к задаче проектного планирования, а разработка программного приложения — к непосредственному использованию классов каркаса или к их наследованию с доопределением вспомогательных методов инициализации прикладных данных. В первом случае может оказаться полезным использование алиасов или предкомпиляторных директив для переименования классов каркаса. Тогда разработка GUI целевого приложения может осуществляться в терминах предметной области, а не проектного планирования. Вместо классов Project, WBS, Activity, Resource, Link можно использовать более естественные для целевого приложения названия School, Cycle, Lesson, Teacher, Room, Class, Rule. Во втором случае реализуются новые предметно-ориентированные классы путем наследования от классов каркаса или в результате их использования с определением нового интерфейса, характерного для условий решаемой прикладной задачи.

5.2 Развитие пакета Applications для представления условий задач RCPSP в расширенных постановках

Хотя пакет каркаса Applications предоставляет довольно развитый набор классов прикладных данных для задания условий RCPSP задач в расширенных постановках, вполне допустимы ситуации, когда требуется еще расширить данный набор, например, для поддержки новых моделей исполнения работ, привлечения ресурсов, специфических типов ограничений и т.п. С математической точки зрения класс задач остается неизменным, но появляется дополнительная специфика, связанная с новыми типами целевых функций и наложенных ограничений. В нашей работе [31] обосновывается возможность математической постановки и решения, так называемых, задач GCPSP с целевыми функциями и наложенными алгебраическими ограничениями общего характера.

В соответствии с объектно-ориентированным подходом развитие пакета Applications может осуществляться различными способами. Например, новые классы прикладных данных можно наследовать от существующих классов с определением новых свойств объектов и уточнением способов параметризации условий прикладных задач. Можно создавать новые классы прикладных данных, определяя их свойства и устанавливая отношения ассоциации, композиции, агрегации с существующими классами каркаса. В отличие от простого переименования классов, упоминаемого в предыдущем разделе, становится возможной реализация обобщенных моделей прикладных данных для задания специальных условий RCPSP задач в расширенных постановках.

В качестве примера, иллюстрирующего необходимость развития пакета Applications, приведем приложение визуального пространственно-временного моделирования проектов. В отличие от традиционных информационных систем календарно-сетевого планирования и управления проектами, составление расписаний в подобных приложениях осуществляется также с учетом пространственных ограничений, регламентирующих условия выполнимости планируемых работ на проектной площадке. Для поддержки пространственных ограничений потребуются дополнительные классы, чтобы представить проектные данные в виде трехмерных геометрических моделей, связать их с проектными работами или порождаемыми ими событиями, а также определить характер пространственных коллизий для составления согласованных расписаний. Примечательно, что реализация классов для задания других видов ограничений, таких как временные условия, отношения предшествования, ресурсные лимиты, рабочие календари, уже предусмотрена типовой конфигурацией пакета Applications и не потребует дополнительных усилий. Поэтому развитие каркаса для обсуждаемых случаев не представляется сложным.

5.3 Развитие пакета Reductions для редукции прикладных задач к постановке GCPSP

Рассмотренные выше способы развития пакета Applications сами по себе не влияют на ход составления расписания алгоритмами, реализации которых уже включены в состав каркаса. Чтобы учесть новые условия задачи проектного планирования, необходимо реализовать соответствующие классы пакета Reductions, с помощью которых данные условия могут быть выражены и проинтерпретированы в математически нейтральных терминах постановки условной оптимизации GCPSP. Программисту, прежде всего, требуется решить, порождают ли введенные или производные прикладные данные новые переменные, меняют ли они вид целевой функции, а также приводят ли они к новым типам алгебраических ограничений. Если это имеет место, то требуется реализовать заново или унаследовать классы, реализующие интерфейсы Variable, ValueDomain, Objective, Constraint, и соответствующим образом модифицировать тело методов формирования условий математической задачи OptimizationProblem. С учетом того, что пакет Reductions содержит реализации всех классов, необходимых для математически корректной редукции задач RCPSP к постановке GCPSP, разработка нескольких дополнительных классов близкой функциональности не потребует значительных усилий.

5.4 Развитие пакета Solvers для реализации новых алгоритмов и эвристик

Пакет Solvers предоставляет готовые к использованию реализации точного и приближенного алгоритмов решения задач GCPSP. В работе [31] формулируются достаточные условия существования решения в данной постановке, а также эквивалентность обобщенного приближенного алгоритма популярному алгоритму последовательной диспетчеризации в случае классической постановки RCPSP. Однако каркас не предоставляет средств анализа существования решений в случае произвольно заданных целевых функций и систем ограничений, а также средств оценки качества найденных приближенных решений. Отчасти вторая проблема нивелируется возможностью поиска точного решения, по крайней мере, для задач низкой размерности. Однако вся ответственность целиком ложится на разработчиков приложений, которые должны обеспечить согласованность задаваемых условий прикладных задач и применяемых алгоритмов.

Одним из способов настройки обобщенного приближенного алгоритма является задание эвристик. Поскольку эвристики реализуются как объекты с общим интерфейсом Heuristic, то разработчики могут предоставить собственные реализации эвристик с учетом прикладных особенностей решаемых задач и сконфигурировать соответствующим образом решатель. В случаях, когда в приложении требуются новые алгоритмы, разработчики

могут их реализовать, основываясь на уже имеющихся в каркасе классах математических объектов и типовых алгоритмов. Примечательно, что используя интерфейсы математических объектов, разработчики, тем самым, предоставляют обобщенные реализации алгоритмов, которые могут быть использованы при решении прикладных задач с произвольными условиями, представимыми математическими объектами наследуемых классов. Однако вопросы конструктивности применения отдельных алгоритмов к конкретным задачам опять же относятся к компетенции разработчиков приложения.

6. Апробация SAF-каркаса

Описанный выше каркас и связанная с ним методология разработки программных приложений теории расписаний были успешно апробированы в ходе построения системы визуального пространственно-временного моделирования проектов Synchro [34]. Данный коммерческий продукт консолидирует в себе традиционные функции систем календарно-сетевого планирования и управления проектами, таких как Oracle Primavera, MS Project, Asta Powerproject, и функции визуального моделирования проектной деятельности.

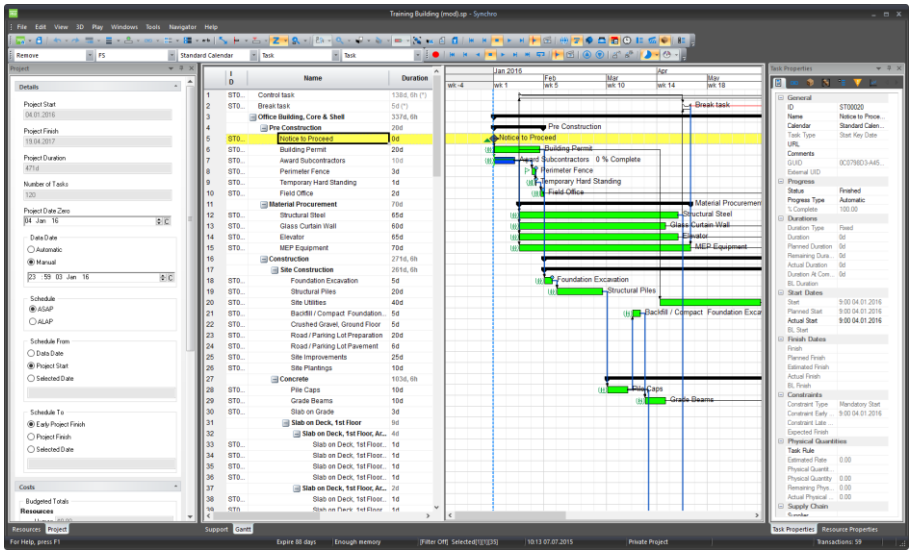


Рис. 9. Графический интерфейс пользователя системы Synchro
Fig. 9. Graphical user interface of the Synchro system

Как следствие, система Synchro должна поддерживать развитый набор пространственных, временных, ресурсных, финансовых ограничений, которые могут быть наложены на проектные работы, а также предоставлять эффективные средства составления согласованных расписаний с учетом всех

видов ограничений. Более того, по мере функциональной эволюции продукта допустим пересмотр данных требований в сторону обобщения и расширения набора ограничений.

На рисунке 9 приведен снимок экрана, иллюстрирующий основные элементы графического интерфейса пользователя системы Synchro, включая диаграмму Ганта, окна задания индивидуальных атрибутов работ, окна задания параметров проектов. Используя последний, пользователь, в частности, может задать некоторые типовые ограничения на даты начала и завершения всего проекта. Временные условия на индивидуальные работы, правила их выравнивания, а также рабочие календари могут быть заданы, используя окно атрибутов работ.

Данные требования удалось удовлетворить в результате использования SAF-каркаса в составе приложения. Каркас не только предоставил готовые к использованию программные средства постановки и решения задач RCPSP, но и обеспечил возможность добавления и поддержки новых видов ограничений в рамках описанной выше методологии.

Проведенные вычислительные эксперименты с большими проектными планами показали высокую эффективность средств каркаса, несмотря на виртуализацию основных вычислительных операций и обобщенную многопараметрическую реализацию моделей прикладных данных. В частности, сравнение с упомянутыми выше популярными приложениями показало, что разработанная система Synchro не уступает им по производительности, однако предоставляет более широкие функциональные возможности, в том числе, из-за поддержки развиваемого набора ограничений.

7. Заключение

Таким образом, обсуждены основные вопросы программной реализации моделей, методов и приложений теории расписаний с использованием SAF-каркаса. Детально рассмотрены принципы организации и функционирования разработанного каркаса, а также его возможности для разработки приложений теории расписаний и, в частности, перспективных систем календарно-сетевого планирования и управления проектами. Успешная апробация каркаса в ходе разработки коммерческого продукта показала правильность принятых проектных решений, а также перспективность использования каркаса для эволюционной разработки серий приложений теории расписаний на единой методологической, программной и инструментальной основе.

Список литературы

[1]. Лазарев А. А., Гафаров Е. Р. Теория расписаний. Задачи и алгоритмы. МГУ им. М. В. Ломоносова, Москва, 2011 г., 222 с.

[2]. Kolisch R., Sprecher A. PSPLIB — A project scheduling library. European Journal of Operational Research, том 96, выпуск 1, 1997 г., с. 205-216.

[3]. Kolisch R., Schwindt C., Sprecher A. Benchmark instances for project scheduling problems. Глава из книги «Handbook on recent advances in project scheduling», под ред. Weglarz J., 1999 г., стр. 197-212.
[4]. Kolisch R., Hartmann S. Heuristic algorithms for solving the resource-constrained project scheduling problem — Classification and computational analysis. Глава из книги «Handbook on recent advances in project scheduling», под ред. Weglarz J., 1999 г., стр. 147-178.
[5]. Hartmann S., Kolisch R. Experimental evaluation of state-of-the-art heuristics for resource constrained project scheduling. European Journal for Operational Research, том 127, выпуск 2, 2000 г., стр. 394-407.
[6]. Kolisch R., Hartmann S. Experimental Investigation of Heuristics for Resource-Constrained Project Scheduling: An Update European Journal of Operational Research, том 174, 2006 г., стр. 23-37.
[7]. Lemmen R. Modeling Resource Alternatives in Project Scheduling. Munich University of Applied Sciences. 29 марта 2005 г.
[8]. Интернет-ресурс: «43 полезных сервиса для управления проектами. Без эпитетов». Статья-обзор от 9 февраля 2016, https:habrahabr.ru/post/276873/, дата обращения 25.06.2017
[9]. Щербина О.А. Удовлетворение ограничений и программирование в ограничениях. Интеллектуальные системы. Теория и приложения, том 15, выпуск 1-4, 2011 г., стр. 53-170.
[10]. Creemers T. (et al.) Constraint-based Maintenance Scheduling on an Electric Power Distribution Network Proc. of the 3rd International Conference and Exhibition on Practical Applications of Prolog. Париж, Alinmead Software Ltd., апрель 1995 г.. стр. 135-144.
[11]. Интернет-ресурс: «PLanning Activities on NETworkS» Интернет-сайт разработчика, http://www.iri.upc.edu/research/webprojects/planets/, дата обращения 25.06.2017
[12]. Simonis H., Cornelissens T. Modelling producer/consumer constraints Proceedings 1st Int. Conference on Principles and Practice of Constraint Programming (CP95). Springer-Verlag, LNCS 976, 1995 г., стр. 449-462.
[13]. Интернет-ресурс: «Atlas Venture» Интерней-сайт разработчика, https://atlasventure.com, дата обращения 25.06.2017
[14]. Aggoun A., Gloner Y., Simonis H. Global constraints for scheduling in CHIP. Invited Industrial Presentation. JFPLC 99, 1999 г.
[15]. Glaisner F., Richard L.-M. FORWARD-C: A refinery scheduling system Proc. conf. on Practical Applications of Constraint Technology (PACT97). 1997 г.
[16]. Fromherz M., Gupta V., Saraswat V. Model-based computing: constructing constraint-based software for electro-mechanical systems. Proc. conf. on Practical Applications of Constraint Technology (PACT95). 1995 г., стр. 63-66.
[17]. Baues G., Kay P., Charlier P. Constraint based resource allocation for airline crew management. Proc. ATTIS'94. 1994 г.
[18]. Collignon C. Gestion optimisee de ressources humaines pour l'audiovisuel. Proc. CHIP users' club. 1996 г.
[19]. Интернет-ресурс: «COSYTEC» Интернет-сайт разработчика, http://www.cosytec.com/constraint_programming/cases_studies/administration.htm, дата обращения 25.06.2017

[20]. Simonis H., Charlier P. Cobra — a system for train crew scheduling. Proc. DIMACS workshop on constraint programming and large scale combinatorial optimization. 1998 г.
[21]. Chew T., David J.-M. A constraint-based spreadsheet for cooperative production planning. Proc. AAAI SIGMAN workshop on knowledge-based production planning, scheduling and control. 1992 г.
[22]. Shvetsov I., Kornienko V., Preis S. Interval spreadsheet for problems of financial planning. Proc. PACT97. 1997 г., стр. 373-385.
[23]. Fruhwirth T., Brisset P. Optimal planning of digital cordless telecommunication systems. Proc. PACT97. 1997 г.
[24]. Shih-Ming Chena, F.H. (Bud) Griffisb, Po-Han Chenc, Luh-Maan Chang. A framework for an automated and integrated project scheduling and management system. Automation in Construction, том 35, 2013 г., стр. 89-110.
[25]. Jan Tulke, Mohamed Nour, Karl Beucke. A Dynamic Framework for Construction Scheduling based on BIM using IFC. IABSE Congress Report, 17th Congress of IABSE. 2008 г., стр. 158-159.
[26]. Интернет-ресурс: «ISO 16739:2013» Интернет-страница описания стандарта, http://www.iso.org/iso/catalogue_detail.htm?csnumber=51622, дата обращения 25.06.2017
[27]. Интернет-ресурс: «Фреймворк» Интернет-страница электронной энциклопедии, https://ru.wikipedia.org/wiki/Фреймворк, дата обращения 25.06.2017
[28]. Лаврищева Е.М. Software Engineering компьютерных систем. Парадигмы, технологии и CASE-средства программирования. Киев, Наукова думка, 2013 г., 283 с.
[29]. Горбунов-Посадов М.М. Расширяемые программы. Москва, Полиптих, 1999 г., 336 с.
[30]. Интернет-ресурс: Martin Fowler: InversionOfControl. Статья-исследование, https://martinfowler.com/bliki/InversionOfControl.html, дата обращения 25.06.2017
[31]. Аничкин А.С., Семенов В.А. Математическая формализация задач проектного планирования в расширенной постановке. Труды ИСП РАН, том 29, выпуск 2, 2017 г., стр. 231-256. DOI: 10.15514/ISPRAS-2017-29(2)-9
[32]. Аничкин А.С., Семенов В.А. Современные модели и методы теории расписаний и календарно-сетевого планирования. Труды ИСП РАН, том 26, выпуск 3, 2014, стр. 212-262. ISSN 2220-6426
[33]. Brucker P., Knust S. Complex scheduling. Springer-Verlag, Berlin, Heidelberg, Germany, 2006 г., 292 с.
[34]. Интернет-ресурс: «Synchro Software» Официальный Интернет-сайт продукта Synchro, http://synchroltd.com, дата обращения 25.06.2017

Object-oriented framework for software development of scheduling applications

[1] A.S. Anichkin <anton.anichkin@ispras.ru>
[1,2] V.A. Semenov <sem@ispras.ru>
[1] Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia
[2] Moscow Institute of Physics and Technology (State University), 9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia

Abstract. Theory of scheduling and project planning is widely applied in diverse scientific and industrial areas. To effectively solve application-specific problems, it is necessary to state right objectives as well as to take into account a lot of factors, such as task execution models, precedence relationship between tasks, resource limitations, directive deadlines, working calendars, conditions for financial and logistics support of project tasks, specific spatio-temporal requirements, et al. Therefore, the development of scheduling applications becomes more and more complicated purposes, risky and costly ones. In this paper, we present an innovative object-oriented Scheduling Application Framework (SAF) designed to simplify and accelerate the software development processes. The presented SAF framework is a system of C++ classes that implement basic abstractions of the scheduling theory as well as provide ready-to-use components to build target applications of typical scheduling functionality. As a general-purpose mathematical library, the framework enables to set and solve so-called RCPSP problems (Resource-Constrained Project Scheduling Problem) in extended statements peculiar to popular project management systems. Branch and bound and linear dispatching algorithms have been implemented and included as a part of the framework. A dozen of heuristics has been implemented and has been provided by the framework too to solve large-scale problems more effectively. Thereby target application developers can adjust the application solver properly taking into account application-specific issues and making the search of suboptimum schedules more effective. As a software toolkit the framework enables developers to implement own components and to configure target applications in unified and flexible manner. Due to object-oriented paradigm, multi-layer architecture and class package organization, the application development takes relatively small efforts. The SAF framework has been successively validated during development of a software application intended for visual modeling and planning of projects under diverse spatial-temporal, resource and finance constraints. Due to achieved advantages, the framework looks promising for development of both sophisticated multi-disciplinary systems and effective domain-specific scheduling applications.

Keywords: scheduling theory; project planning and scheduling; software application framework.

DOI: 10.15514/ISPRAS-2017-29(3)-14

For citation: Anichkin A.S., Semenov V.A. Object-oriented framework for software development of scheduling applications. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 3, 2017. pp. 247-296 (in Russian). DOI: 10.15514/ISPRAS-2017-29(3)-14

References

[1]. Lazarev A. A., Gafarov E. R. Scheduling theory. Tasks and algorithms. Lomonosov Moscow State University, Moscow, 2011, 222 p (in Russian).

[2]. Kolisch R., Sprecher A. PSPLIB — A project scheduling library. European Journal of Operational Research, vol. 96, issue 1, 1997, pp. 205-216.

[3]. Kolisch R., Schwindt C., Sprecher A. Benchmark instances for project scheduling problems. Chapter in the book «Handbook on recent advances in project scheduling», ed. Weglarz J., 1999, pp. 197-212.

[4]. Kolisch R., Hartmann S. Heuristic algorithms for solving the resource-constrained project scheduling problem — Classification and computational analysis. Chapter in the book «Handbook on recent advances in project scheduling», ed. Weglarz J., 1999, pp. 147-178.
[5]. Hartmann S., Kolisch R. Experimental evaluation of state-of-the-art heuristics for resource constrained project scheduling. European Journal for Operational Research, vol. 127, issue 2, 2000, pp. 394-407.
[6]. Kolisch R., Hartmann S. Experimental Investigation of Heuristics for Resource-Constrained Project Scheduling: An Update European Journal of Operational Research, vol. 174, 2006, pp. 23-37.
[7]. Lemmen R. Modeling Resource Alternatives in Project Scheduling. Munich University of Applied Sciences. March 29, 2005.
[8]. Internet: [«43 useful services for project management. Without epithets».] Article-review, February 9, 2016, https://habrahabr.ru/post/276873/, accessed 25.06.2017
[9]. Shcherbina O.A. Satisfaction of constraints and programming in constraints. Intellektualnyie sistemyi. Teoriya i prilozheniya. [Intellectual systems. Theory and applications] Vol. 15, issue 1-4, 2011, pp. 53-170 (in Russian).
[10]. Creemers T. (et al.) Constraint-based Maintenance Scheduling on an Electric Power Distribution Network Proc. of the 3rd International Conference and Exhibition on Practical Applications of Prolog. Paris, Alinmead Software Ltd., April 1995, pp. 135-144.
[11]. Internet: «PLanning Activities on NETworkS» Website of developer, http://www.iri.upc.edu/research/webprojects/planets/, accessed 25.06.2017
[12]. Simonis H., Cornelissens T. Modelling producer/consumer constraints Proceedings 1st Int. Conference on Principles and Practice of Constraint Programming (CP95). Springer-Verlag, LNCS 976, 1995, pp. 449-462.
[13]. Internet: «Atlas Venture» Website of developer, https://atlasventure.com, accessed 25.06.2017
[14]. Aggoun A., Gloner Y., Simonis H. Global constraints for scheduling in CHIP. Invited Industrial Presentation, JFPLC 99. 1999.
[15]. Glaisner F., Richard L.-M. FORWARD-C: A refinery scheduling system Proc. conf. on Practical Applications of Constraint Technology (PACT97). 1997.
[16]. Fromherz M., Gupta V., Saraswat V. Model-based computing: constructing constraint-based software for electro-mechanical systems. Proc. conf. on Practical Applications of Constraint Technology (PACT95). 1995, pp. 63-66.
[17]. Baues G., Kay P., Charlier P. Constraint based resource allocation for airline crew management. Proc. ATTIS'94. 1994.
[18]. Collignon C. Gestion optimisee de ressources humaines pour l'audiovisuel. Proc. CHIP users' club. 1996 г.
[19]. Internet: «COSYTEC» Website of developer, http://www.cosytec.com/constraint_programming/cases_studies/administration.htm, accessed 25.06.2017
[20]. Simonis H., Charlier P. Cobra — a system for train crew scheduling. Proc. DIMACS workshop on constraint programming and large scale combinatorial optimization. 1998.
[21]. Chew T., David J.-M. A constraint-based spreadsheet for cooperative production planning. Proc. AAAI SIGMAN workshop on knowledge-based production planning, scheduling and control. 1992.
[22]. Shvetsov I., Kornienko V., Preis S. Interval spreadsheet for problems of financial planning. Proc. PACT97. 1997, pp. 373-385.

[23]. Fruhwirth T., Brisset P. Optimal planning of digital cordless telecommunication systems. Proc. PACT97. 1997.
[24]. Shih-Ming Chena, F.H. (Bud) Griffisb, Po-Han Chenc, Luh-Maan Chang. A framework for an automated and integrated project scheduling and management system. Automation in Construction, vol. 35, 2013, pp. 89-110.
[25]. Jan Tulke, Mohamed Nour, Karl Beucke. A Dynamic Framework for Construction Scheduling based on BIM using IFC. IABSE Congress Report, 17th Congress of IABSE. 2008, pp. 158-159.
[26]. Internet: «ISO 16739:2013» Web-page of description of the standard, http://www.iso.org/iso/catalogue_detail.htm?csnumber=51622, accessed 25.06.2017
[27]. Internet: «Software framework» Web-page of the electronic encyclopedia, https://en.wikipedia.org/wiki/Software_framework, accessed 25.06.2017
[28]. Lavrischeva E.M Software Engineering of computer systems. Paradigms, technologies and CASE-programming tools. Kiev, Naukova dumka, 2013, 283 p. (in Russian).
[29]. Gorbunov-Possadov M.M. Extensible programs. Moscow, Poliptih, 1999, 336 p. (in Russian).
[30]. Internet: Martin Fowler. InversionOfControl. Research article, https://martinfowler.com/bliki/InversionOfControl.html, accessed 25.06.2017
[31]. Anichkin A.S., Semenov V.A. Mathematical formalization of project scheduling problems. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 2, 2017, pp. 231-256 (in Russian). DOI: 10.15514/ISPRAS-2017-29(2)-9
[32]. Anichkin A.S., Semenov V.A. A survey of emerging models and methods of scheduling. Trudy ISP RAN/Proc. ISP RAS, vol. 26, issue 3, 2014, pp. 5-50 (in Russian). DOI: 10.15514/ISPRAS-2014-26(3)-1
[33]. Brucker P., Knust S. Complex scheduling. Springer-Verlag, Berlin, Heidelberg, Germany, 2006, 292 p.
[34]. Internet: «Synchro Software» Official website of the product Synchro, http://synchroltd.com, accessed 25.06.2017