Министерство образования и науки Российской Федерации

САНКТ-ПЕТЕРБУРГСКИЙ
ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО

Российская академия наук

ФЕДЕРАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЦЕНТР
«ИНФОРМАТИКА И УПРАВЛЕНИЕ» РАН

Exactpro
London Stock Exchange Group

ИНСТРУМЕНТЫ И МЕТОДЫ АНАЛИЗА ПРОГРАММ – 2015
TOOLS & METHODS OF PROGRAM ANALYSIS (TMPA – 2015)

Материалы
Международной научно-практической конференции
12–14 ноября 2015 года

Санкт-Петербург
Издательство Политехнического университета
2015

УДК 004.4
И72

Инструменты и методы анализа программ – 2015. Tools & Methods of Program Analysis (TMPA – 2015) : материалы Междунар. науч.-практ. конф. 12–14 ноября 2015 года. – СПб.: Изд-во Политехн. ун-та, 2015. – 220 с.

Редакционная коллегия: председатель В.М. Ицыксон;
члены редколлегии: В.Н. Захаров, И.Л. Иткин, А.-М. А. Лукина,
Н.А. Тамм, А.С. Эдеш, К.В. Загоруйко, Е.С. Аввакумова, А.В. Зверев,
П.Н. Рудовский, М.П. Рудовский

В сборнике публикуются доклады участников Международной научно-практической конференции «Инструменты и методы анализа программ – 2015» / «Tools & Methods of Program Analysis (TMPA – 2015)», состоявшейся 12–14 ноября 2015 года в Санкт-Петербургском политехническом университете Петра Великого.

Для широкого круга читателей.

ISBN 978-5-7422-5159-0

© Санкт-Петербургский политехнический университет Петра Великого, 2015

Conference
EXTENT

Trading Technology Trends &
Quality Assurance

www.extentconf.com

О конференции

МЕЖДУНАРОДНАЯ НАУЧНО-ПРАКТИЧЕСКАЯ КОНФЕРЕНЦИЯ: TOOLS & METHODS OF PROGRAM ANALYSIS, TMPA-2015 (ИНСТРУМЕНТЫ И МЕТОДЫ АНАЛИЗА ПРОГРАММ) 12-14 НОЯБРЯ, САНКТ-ПЕТЕРБУРГ, 2015

Конференция посвящена одному из наиболее актуальных и важных направлений программной инженерии – анализу качества программного обеспечения.

Вопросы эффективности и корректности функционирования программного обеспечения являются ключевыми для большинства наукоемких отраслей современной экономики, включая ИТ-индустрию, финансовый сектор, транспорт, медицину, высокотехнологическую промышленность и многие другие. Разработка новых и усовершенствование существующих инструментов и методов анализа программ – одно из необходимых условий внедрения инноваций в этих отраслях.

Конференция нацелена на развитие индустрии разработки программного обеспечения и внедрение новейших разработок в области тестирования, анализа и верификации.

В рамках конференции были проведены пленарные доклады и лекционные мини-курсы экспертов; доклады участников, отобранные программным комитетом из числа поступивших заявок; презентации открытых проектов, короткие сообщения, представляющие новые идеи, незавершенные исследования или новые инструменты.

Темы, рассматриваемые на конференции, включают (но не ограничиваются):

- автоматизация тестирования программного обеспечения;
- статический анализ программ;
- верификация;
- динамические методы анализа программ;
- тестирование и анализ параллельных и распределенных систем;
- тестирование и анализ высоконагруженных систем и систем высокой доступности;
- анализ и верификация программно-аппаратных систем;
- методы создания качественного программного обеспечения;
- инструментальные средства анализа, тестирования и верификации;

Программный комитет TMPA-2015

- Захаров Виктор Николаевич, *д.т.н., федеральный исследовательский центр «Информатика и управление» РАН*
- Ицыксон Владимир Михайлович, *к.т.н., доцент кафедры компьютерных систем и программных технологий СПбПУ, сопредседатель*
- Петренко Александр Константинович, *д.ф.-м.н., зав. отделом Технологий программирования Института системного программирования РАН, профессор кафедры Системного программирования ВМиК МГУ им. М.В.Ломоносова*
- Ахин Марат Халимович, *научный сотрудник СПбПУ*
- Глухих Михаил Игоревич, *к.т.н., доцент, JetBrains, СПбПУ*
- Николя Гюльфи, *PhD, факультет наук, технологии и коммуникации, Университет Люксембурга*
- Иосиф Иткин, *директор Exactpro, London Stock Exchange Group*
- Камкин Александр Сергеевич, *к.ф.-м.н., в.н.с. ИСП РАН*
- Кириленко Яков Александрович, *старший преподаватель, МатМех СПбГУ*
- Коннов Игорь, *PhD, факультет информатики, Венский технический университет*
- Кулямин Виктор Вячеславович, *к.ф.-м.н., в.н.с. ИСП РАН, доцент ВМК МГУ*
- Моисеев Михаил Юрьевич, *к.т.н., старший программист, Интел*
- Пакулин Николай Витальевич, *к.ф.-м.н., Ученый секретарь ИСП РАН*
- Павлова Елена Анатольевна, *к.т.н., координатор программ, Microsoft*
- Рудовский Максим, *Exactpro, London Stock Exchange Group*
- Цителов Дмитрий Игоревич, *руководитель Evolution Lab, Devexperts, LLC*
- Дженнифер Трелевич, *Risk CTO, Deutsche Bank, IEEE Technology and Engineering Management Society*
- Йозеф Виддер, *PhD, факультет информатики, Венский технический университет*
- Захаров Владимир Анатольевич, *д.ф.-м.н., доцент кафедры МК, зав. лабораторией МПКБ*

Содержание

СТАТИЧЕСКИЙ И ДИНАМИЧЕСКИЙ АНАЛИЗ ПРОГРАММ

Динамический анализ исполняемого кода в формате ELF на основе статической бинарной инструментации 14
Михаил Ермаков

Применение статического анализа для оптимизации динамического поиска гонок 21
Яков Роскошный, Дмитрий Цителов, Виталий Трифанов, Роман Елизаров

Лексический анализ динамически формируемых строковых выражений 27
Марина Полубелова, Семён Григорьев

Применение технологии иерархических параметризируемых шаблонов для автоматизированного исправления ошибок в программном коде 38
Артем Алексюк, Владимир Ицыксон

Автоматическое тестирование линеаризуемости реализаций многопоточных структур данных 48
Антон Евдокимов, Дмитрий Цителов, Роман Елизаров, Виталий Трифанов

ТРЕЙДИНГОВЫЕ СИСТЕМЫ

Спонсорский доступ с малой задержкой посредством FPGA 58
Валерий Флоров, Павел Гарин, Павел Смирнов, Максим Метельков

Reference test harness for algorithmic trading platforms 65
Виктория Леончик, Алексей Сухов, Евгений Ушаков, Иосиф Иткин, Анна-Мария Лукина

Инструмент для автоматизированного тестирования систем проведения расчетов и клиринга ClearTH 72
Анна Торопова, Екатерина Димова, Иосиф Иткин

Автоматизированное создание сценариев тестирования финансовых протоколов и подключений 82
Анна Торопова, Сергей Павлов, Андрей Соловьев, Александр Бормотин, Иосиф Иткин

КОРОТКОЕ СООБЩЕНИЕ
Dynamic verification of input and output data streams for market data aggregation and quote dissemination systems (Ticker Plant) 95
Алёна Булда, Мария Орлова

ВЕРИФИКАЦИЯ

Implementing the metaVCG approach in the C-light system 101
Алексей Промский, Дмитрий Кондратьев

Расширение метагенерации условий корректности концепцией семантической разметки 107
Дмитрий Кондратьев

A need to specify and verify standard functions 119
Николай Шилов

Формальные методы в робототехнике 123
Дмитрий Мордвинов, Юрий Литвинов

КОРОТКИЕ СООБЩЕНИЯ
Верификация функциональных программ методом построения диаграмм состояний 135
Андрей Миронов

Towards a usable defect prediction tool: crossbreeding machine learning and heuristics 146
Владимир Коваленко, Галина Алперович

Задача верификации выполнения QoS требований с учетом случайного поведения вычислительных систем 150
Сергей Френкель, Виктор Захаров

ИНСТРУМЕНТЫ ТЕСТИРОВАНИЯ И ОТЛАДКИ

Мультиплатформенный метод обратной отладки виртуальных машин 161
Павел Довгалюк, Мария Климушенкова, Денис Дмитриев, Владимир Макаров

Трассировка многомодульного приложения в среде операционной системы z/OS 169
Ростислав Ефремов

КОРОТКИЕ СООБЩЕНИЯ

Тестирование среды программирования роботов 176
Дмитрий Мордвинов, Юрий Литвинов

Generation of test scenarios for non deterministic and concurrent telecommunication applications 186
Павел Дробинцев, Всеволод Котляров, Никита Воинов

Система информационной поддержки разработки макропрограмм автономного управления космическим аппаратом 195
Андрей Тюгашев, Антон Насекин

Статический и динамический анализ программ

Секционные доклады

Динамический анализ исполняемого кода в формате ELF на основе статической бинарной инструментации

Михаил Ермаков
Институт Системного Программирования РАН
Москва, Россия
mermakov@ispras.ru

***Аннотация*—Статья посвящена возможностям применения статической инструментации исполняемых файлов и файлов динамических библиотек в формате ELF для проведения динамического анализа программ. В статье приводится обзор существующих решений в данной и смежных областях и предлагается подход к обработке бинарных файлов ARM ELF. Описываются основные особенности предлагаемого подхода: разметка целевых файлов и генерация инструментационного кода на основе пользовательских спецификаций, использование промежуточного языка описания инструкций ассемблера ARM, компоновка итоговых инструментированных файлов исполняемого кода. Приводятся результаты применения реализации данного подхода для создания модулей трассировки помеченных данных, используемых в рамках инструмента автоматического поиска дефектов Avalanche.**

Ключевые слова—динамический анализ, статическая бинарная инструментация, ELF, платформа ARM

I. Введение

В настоящее время существует большое количество систем и инструментов, производящих автоматическую и полуавтоматическую обработку программного обеспечения для достижения одной или нескольких целей, таких как:

- проведение анализа целевой программы на наличие критических дефектов и несоответствия заявленной функциональности;
- сбор статистических данных о выполнении программы с целью выявления блоков, осуществляющих неэффективное использование системных и временных ресурсов, для проведения последующей оптимизации данных блоков;
- создание блоков защиты от взлома и раскрытия используемых алгоритмов и обратная данной задача — извлечение трасс выполнения, распознавание внутренней организации программы с целью идентификации используемых алгоритмов.

Приведённые примеры взаимодействия с программным обеспечением относятся в первую очередь к аспектам времени выполнения целевой программы. На практике реализация подобной функциональности подразумевает произведение дополнительных действий во время работы целевой программы — либо на уровне внешнего системного инструмента, имеющего возможность отслеживать и влиять на поведение программы, либо на уровне модифицированного исполняемого кода программы или исполняемого кода, внедрённого в программу. Для программного обеспечения, выполняющегося с помощью промежуточных средств (виртуальные машины, интерпретаторы, эмуляторы), добавляется возможность взаимодействия с целевой программой на уровне данных промежуточных средств. Большинство системных инструментов осуществляют внешний контроль за выполнением целевой программы (отслеживание состояния процесса программы, использования выделенных ресурсов, трассировка входных и выходных данных). Данный подход обеспечивает минимизацию влияния на исполнение целевой программы, однако предоставляет ограниченные возможности извлечения внутренних данных программы для проведения анализа. Это в свою очередь повышает сложность самого анализа, если необходимо учитывать конкретные аспекты исполняемого кода, такие как параметры структурных компонент (инструкций, управляющих блоков, функций и т.д.).

Средства инструментации приводят к прямой модификации исполняемого кода целевой программы — дополнительные и изменённые блоки инструкций выполняются непосредственно в контексте её процесса. Инструментация влияет на различные аспекты исследуемой программы — размер областей виртуальной памяти, выделенной под исполняемый код и данные, время работы отдельных функций, взаимодействие с динамической памятью и другими ресурсами. Необходимость гарантировать отсутствие значительного влияния на аспекты, оценка которых является основной целью инструмента анализа, — один из наиболее важных ограничивающих факторов подхода инструментации. Тем не менее, повышенный уровень внедрения в целевую программу позволяет организовывать значительно более точный анализ по сравнению с системными средствами.

A. Средства инструментации

Среди средств инструментации можно выделить три основные группы:

1) Статическая инструментация исходного кода

Статическая инструментация исходного кода — автоматическая модификация или расширение исходного кода по заданным спецификациям. Данный подход в первую очередь требует наличия исходного кода и возможности встраиваться в процесс компиляции и сборки программы. Проведение подобной инструментации требует мощного статического анализа для построения модели исходного кода; дополнительно при изменении и добавлении кода необходимо избегать проблем внешних и внутренних зависимостей, конструкций условной компиляции (если таковые поддерживаются) и особенностей компилятора и сборщика. Шаг оптимизации может также нарушить логику инструментационного кода. Тем не менее, инструментация исходного кода позволяет изменять целевые программы для их запуска на разных платформах без необходимости явной поддержки данных платформ (за исключением особенностей компилятора и сборщика).

2) Динамическая инструментация исполняемого кода

Динамическая инструментация исполняемого кода — внедрение в процесс выполнения программы с целью перехвата блоков инструкций и их модификации перед выполнением на процессоре. Модификация блока инструкций требует проведения декодирования инструкций и обратного кодирования во время работы программы под контролем системы инструментации, что вносит дополнительные накладные расходы. Тем не менее, при использовании систем динамической инструментации достаточно лишь контролировать целостность блоков инструкций, а не всего образа исполняемого файла или библиотеки. Размер блоков может быть выбран достаточно малым, чтобы затраты на подобный контроль были минимальными. При этом сохраняется возможность осуществлять доступ к элементам исходного образа исполняемого кода и данных, загруженного в виртуальную память, так как он не изменяется системой инструментации.

3) Статическая инструментация исполняемого кода

Статическая инструментация исполняемого кода — модификация и пересборка исполняемого файла или файла библиотеки для получения целостной версии данного файла с дополнительной функциональностью. В отличие от систем динамической инструментации, статические системы работают не напрямую с блоками инструкций, а с цельными образами, возможно содержащими управляющие структуры, исполняемый код и данные программы. Это приводит к увеличению сложности декодирования блоков целевых файлов и к значительному ограничению возможностей модификации — изменение инструкций внутри блока может приводить к нарушению работы других инструкций блока и корректности информации в управляющих структурах. Обработка защищённых (например, с помощью упаковки) и обфусцированных файлов исполняемого кода с помощью систем статической инструментации крайне затруднена, а программы, осуществляющие генерацию и изменение собственного кода во время выполнения, в общем случае не могут быть обработаны в полном объёме. Тем не менее, этапы декодирования и кодирования производятся один раз, что даёт значительное снижение накладных расходов при множественных запусках целевой программы. Дополнительно, при проведении инструментации существует возможность рассматривать файлы целевой программы для выявления параметров глобальных зависимостей и оптимизации инструментационного кода для связанных блоков инструкций.

B. Структура статьи

В рамках данной статьи будет рассматриваться подход к проведению статической инструментации исполняемого кода и особенности текущей реализации данного подхода для файлов в формате ARM ELF. В секции 2 будут рассмотрены существующие аналоги и другие разработки в данной области. Секция 3 содержит описание этапов инструментации, проводимых в представляемом подходе. В секции 4 приведено краткое описание архитектуры реализации системы инструментации и результаты применения инструментов, разработанных в рамках системы для задачи автоматического поиска дефектов. В секции 5 приводится краткое заключение по представленному материалу и рассматриваются перспективные направления дальнейших исследований.

II. Обзор систем инструментации

Для большинства существующих систем инструментации центральным принципом является предоставление возможностей по проведению настраиваемой обработки целевых программ — блоки исполняемого кода, в которых производится модификация или внедрение дополнительного кода, а также функциональная логика необходимых изменений задаются разработчиком модуля инструментации. Основными факторами, ограничивающими область применимости систем инструментации, являются следующие:

- Поддерживаемые машинные архитектуры и специфические наборы инструкций для данных архитектур;
- Поддерживаемые форматы организации файлов исполняемого кода;
- Предоставляемая разметка точек и блоков инструментации;
- Тип организации инструментационного кода;
- Ограничения, накладываемые на целевые файлы исполняемого кода.

A. Ранние системы инструментации

Ранние системы инструментации создавались для специфических архитектур и технологий и в основном не применимы для использования на данный момент из-за отсутствия поддержки со стороны авторов или устаревания целевых платформ. Тем не менее, данные проекты представляют значительный интерес за счёт развития области и формулирования основных подходов.

Система ATOM [1] была разработана для архитектуры DEC в связке с системой управления и организации исполняемого кода OM [2]. Данная система одной из первых предоставляла пользователю возможности описания логики инструментации, в то время как

непосредственно процесс инструментации выполнялся внутренним модулем ATOM. Дополнительно пользователю необходимо было осуществить разметку файлов целевой программы для обозначения точек вставки дополнительного кода.

Авторы системы EEL [3] сфокусировали внимание на возможности применения структурных элементов исполняемого кода целевой программы (инструкций, блоков инструкций, функций) в качестве точек инструментации. Это позволило значительно упростить процесс разметки целевого кода и спецификации логики модуля инструментации. Дополнительно, подход, используемый в системе EEL, обеспечивал гибкую структуру для поддержки различных архитектур — кодирование и декодирование целевого кода и генерацию инструментационного кода осуществлял специфический модуль, реализующий некоторый общий интерфейс.

Среди систем, разработанных под целевые архитектуры, широко используемые в данный момент, можно выделить BIRD [4] (Windows/x86) и BitRaker Anvil [5] (Linux/ARM).

Подход к инструментации в системе BIRD заключался во вставке инструкций вызова внешних функций в исполняемый код целевой программы и сборка инструментационного кода в виде динамической библиотеки, предоставляющей реализацию данных внешних функций.

Система BitRaker Anvil была основана на использовании кросс-платформенного анализа — в исполняемый код целевой программы для платформы ARM вставлялись инструкции вызова внешней функции. При работе целевой программы в среде эмуляции, функционирующей в рамках устройства x86, вызовы инструментационного кода осуществлялись в процессе среды эмуляции, что значительно повышало скорость проведения анализа.

B. Современные системы инструментации

Более современные системы инструментации активно используются для разработки отдельных модулей или полноценных инструментов анализа. Существует несколько направлений, разработки в которых интегрируются в системы инструментации для повышения их точности, эффективности и гибкости:

- Развитие интерфейса инструментации, доступного пользователю для описания логики необходимого модуля инструментации.
- Поддержка большого количества целевых платформ и разработка специфических оптимизаций для повышения эффективности работы инструментов на данных платформах.
- Повышение точности алгоритмов декодирования фрагментов данных, управляющих структур и кода целевых программ и разработка алгоритмов извлечения дополнительных данных о связях и особенностях объектов целевых программ для использования инструментом анализа.
- Оптимизация генерируемого инструментационного кода.

Среди наиболее значимых систем статической инструментации можно выделить MAQAO [6, 7], PEBIL [8] и Dyninst [9], предоставляющих возможности работы с файлами исполняемого кода для платформ Linux/x86 (PEBIL, Dyninst), Linux/x86-64 (MAQAO, PEBIL, Dyninst) и Linux/ppc, Linux/ppc64 (Dyninst).

Система MAQAO изначально была разработана для разбора исполняемого кода и внедрения модулей регистрации событий для профилирования и оценки эффективности работы компонентов программы. В дальнейшем был разработан язык спецификации точек инструментации MIL [7] и модуль генерации инструментационного кода для поддержки возможности разработки дополнительных инструментов анализа. Важной особенностью системы MAQAO является высокая точность и корректность работы с программами, активно использующими параллельные вычисления — именно для анализа подобных программ было разработано ядро системы.

Система PEBIL производит разбор исполняемых файлов целевой программы, модификацию секций кода и управляющих структур с целью подготовки к инструментации и добавляет инструментационный код в виде дополнительной секции. Блоки инструкций в точках инструментации заменяются на инструкции перехода в добавленные секции. Система PEBIL поддерживает генерацию инструментационного кода в виде внешней динамической библиотеки или коротких блоков инструкций, не требующих подключения дополнительных библиотек. В системе используется двухпроходный декодер инструкций и модуль оптимизации, осуществляющий минимизацию количества инструкций, не относящихся к логике инструмента анализа, но необходимых для корректной передачи управления на инструментационный код.

Система Dyninst предоставляет широкий интерфейс описания инструментационного кода и спецификаций точек инструментации. Внутренние модули обеспечивают произведение необходимых модификаций, причём возможно проведение как и статической инструментации на основе полномасштабного разбора файлов целевой программы, так и динамической инструментации на основе стандартного подхода перехвата и модификации блоков инструкций.

III. Инструментация ARM ELF

Разработка подхода, описанного в рамках данной статьи, была обоснована отсутствием систем статической инструментации, предоставляющих поддержку архитектуры ARM на базе операционной системы Linux (перспективные целевые платформы — Android и Tizen). Востребованность подхода статической инструментации продиктована возможностями разработки модулей динамического анализа, осуществляющегося на множестве входных данных и путей выполнения для обеспечения достаточного покрытия целевой программы, — при проведении подобного анализа накладные расходы

статической инструментации меньше, чем накладные расходы динамической инструментации [8].

Среди основных принципов разрабатываемого подхода, определяемых целевыми задачами динамического анализа, можно выделить следующие:

- Фокус на инструментации структурных единиц исполняемого кода (инструкций) для сбора подробной трассы, включающей типы и аргументы совершаемых операций.
- Поддержка создания инструментационного кода в виде исходного кода на языке C/C++ с возможностью использования прямых ассемблерных вставок.
- Поддержка возможности задания областей исполняемого кода целевой программы на основе фильтрации по функциям и модулям.

A. Этапы инструментации

1) Разбор исполняемого кода и генерация инструментационного кода

Первым этапом процесса инструментации является обработка пользовательских спецификаций и проход декодера файлов исполняемого кода целевой программы. Язык задания спецификаций позволяет описывать целевые типы точек инструментации, набор фильтров для каждой типа и непосредственно инструментационный код, выполнение которого необходимо в точках инструментации. В числе поддерживаемых типов точек инструментации выступают следующие:

- Структурные точки инструментации, задающие все инструкции определённой функциональной группы (арифметические и логические инструкции, инструкции доступа к памяти, вызова процедур и т.д.).
- Модульные точки инструментации, задающие позицию в некотором логическом блоке (входные и выходные инструкции функции, входные инструкции базовых блоков, блоки условного выполнения).

Задаваемые фильтры обеспечивают возможность осуществлять инструментацию точек определённого типа внутри отдельных функций или групп функций. Язык описания фильтров поддерживает групповые символы для возможности использования префиксов (например, для работы с функциями классов C++).

Задание инструментационного кода осуществляется путём написания коротких блоков исходного кода на языке C/C++, которые автоматически компилируются в исполняемый код, который впоследствии внедряется в файлы целевой программы. Существует возможность использования ассемблерных вставок, глобальных переменных (с помощью задания заголовка, общего для всех блоков) и внешних зависимостей (для которых необходимо задать имя подключаемой библиотеки).

Блоки инструментационного кода привязаны к инструкциям, находящимся в точках инструментации. При написании инструментационного кода существует возможность использования специального интерфейса, предоставляющие операции над инструкциями двух типов:

- Операции доступа к статическим элементами инструкций, таким как номера регистров операндов, значения константных операндов, флаги, специфичные для отдельных групп инструкций.
- Операции доступа к динамическим элементами инструкций, таким как значения регистров и содержимого ячеек памяти, значения регистра флагов для работы с блоками условного выполнения и т.д.

Реализация операций первого типа настроена таким образом, чтобы приводить к генерации явных константных значений в исходном инструментационном коде. Работа блока оптимизации при компиляции инструментационного кода позволяет понизить накладные расходы по осуществлению доступа к статическим элементам. Для реализации операций второго типа производится генерация блока инструкций, выполнение которого приведёт к вычислению необходимых значений при выполнении инструментационного кода.

2) Подготовка файлов целевой программы к инструментации

Вторым этапом инструментации является изменение структуры файлов целевой программы в рамках формата ARM ELF для получения корректных образов. В список операций, проводимых на данном этапе, входят следующие:

- Копирование исходной структуры файлов и добавление дополнительных секций инструментационного кода и данных.
- Перемещение и расширение ряда секций, содержащих управляющую информацию и исполняемый код, для добавления внешних зависимостей, указанных в инструментационном коде.
- Добавление исполняемого кода и заполнение управляющих структур в расширенных областях секций согласно стандарту ARM ELF и протоколам генерации кода, используемых компиляторами и сборщиками, для корректной работы загрузчика при выполнении инструментированных исполняемых файлов и библиотек.

3) Инструментация подготовленных образов

Непосредственная инструментация заключается в замене одной инструкции или блока инструкций в указанных точках на инструкцию перехода в секцию инструментационного кода, добавленную на предыдущем этапе. Последующие действия связаны с обеспечением корректности модифицированного кода путём вставки дополнительных инструкций сохранения состояния программы и восстановления заменённых инструкций.

Сгенерированный инструментационный код состоит из набора обособленных сегментов, каждый из которых

соответствует точке инструментации. При этом сегменты имеют блочную структуру, разделённую на 4 области: начальный «пустой» блок, состоящий из инструкций, не имеющих никаких побочных эффектов, непосредственно блок кода, реализующий необходимую для анализа функциональность, дополнительный «пустой» блок и блок данных, используемый инструментационным кодом. Начальный и дополнительный «пустые» блоки имеют размер, определяемый особенностями инструкций в точке инструментации, и используются для вставки служебных инструкций.

В процессе инструментации в первый «пустой» блок осуществляется вставка инструкций сохранения регистров, изменяемых целевым блоком, во второй «пустой» блок осуществляется вставка инструкций восстановления регистров и вставка инструкций кода, которые были заменены в точке инструментации на инструкцию перехода. Последней во второй «пустой» блок вставляется инструкция возвратного перехода в основную секцию кода на позицию после точки инструментации. При наличии двух смежных точек инструментации с целью снижения накладных расходов вместо возвратного перехода может осуществляется переход на следующий блок инструментационного кода. Пример структуры кода представлен на рис. 1, показывающем три блока инструкций: исходный блок инструкций, содержащий точку инструментации, блок инструкций после вставки перехода на инструментационный код и соответствующий блок инструментационного кода.

Во время переноса исходных инструкций во второй «пустой» блок может возникнуть необходимость модифицировать параметры инструкций или заменить инструкцию на блок, производящий идентичный эффект. В первую очередь подобные сложности вызывают инструкции, которые напрямую или косвенно используют значение счётчика команд: инструкции перехода по относительному смещению, инструкции доступа к памяти по адресу, вычисляющемуся по смещению от текущего и т.д.

4) Коррекция смещений в инструментационном коде

Последним этапом инструментации является восстановление смещений, используемых инструкциями доступа к глобальными переменным и внешним зависимостям в инструментационном коде. Стандартный процесс сборки исполняемого кода из нескольких объектных файлов включает в себя выставление корректных смещений в сегментах исполняемого кода по общей секции работы с глобальными данными и зависимостями. В процессе инструментации сгенерированный инструментационный код подключается к существующему исполняемому файлу или библиотеки сторонними от системы сборки средствами (использование стандартных сборщиков напрямую невозможно из-за отсутствия дополнительных таблиц в исходных файлах целевых файлов) и, соответственно, требует проведения подобного шага.

IV. Особенности практической реализации подхода

Система инструментации интегрирована с набором инструментов binutils[10]. Базовые модули binutils включают в себя библиотеку для разбора и управления файлами в формате ELF, причём текущая версия комплекса уже включает в себя средства удаления, добавления и перемещения секций, представления управляющих структур в необходимом формате. Наконец, в составе комплекса binutils существует библиотека libopcodes, предоставляющая гибкий интерфейс для реализации модулей декодирования набора инструкций определённой архитектуры.

Инструменты readelf и objdump в составе комплекса binutils используются для извлечения структуры целевых файлов ARM ELF, разметки точек инструментации и автоматической генерации исходного инструментационного кода. Инструмент objdump

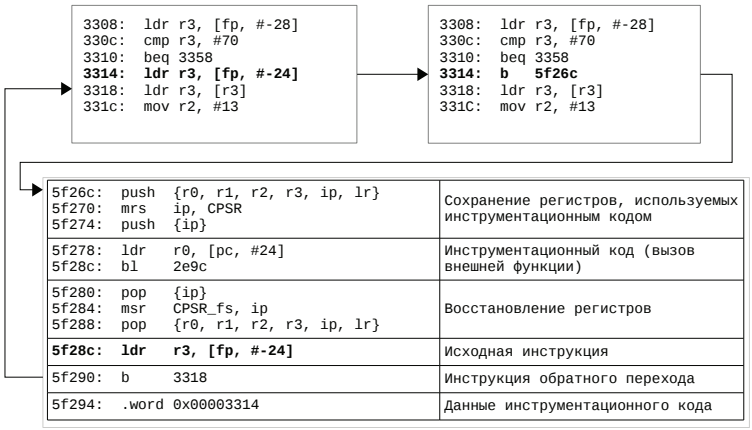


Рис. 1: Структура кода при инструментации

реализует линейное дизассемблирование; данный подход в отличие от метода рекурсивного разбора накладывает дополнительные ограничения на входные файлы (например, наличие таблицы символов) для обеспечения точности результата. Трансформация инструментационного кода в объектные файлы осуществляется с помощью компиляторной системы gcc, конкретная версия которой может быть предоставлена пользователем. Перераспределение исходных секций и добавление новых секций осуществляется с помощью инструмента objcopy. Для коррекции смещений и сегментов, осуществляющих работу с внешними зависимостями, на основе внутренней библиотеки работы с форматом ELF binutils был разработан инструмент addsymbol. Наконец, модуль расширения для библиотеки libopcodes, используемый для разметки точек инструментации, интегрирован с новым инструментов rewriter, проводящим непосредственно инструментацию.

A. Внутреннее представление инструкций

При работе инструмента objdump происходит разбор инструкций файлов исполняемого кода целевой программы и перевод инструкций во внутреннее представление, описывающее их основные элементы и параметры. Среди наиболее значимых элементов можно выделить следующие:

- Тип аргументов инструкции (регистры, константные значения).
- Значения аргументов инструкции (для констант).
- Код операции.
- Дополнительные параметры, специфичные для групп операций (режимы индексирования, триггеры обновления регистра флагов).

Основная функциональная мощность по обработке данных параметров лежит на инструментационном коде, однако прямой доступ к значениям параметров значительно упрощает структуру инструментационного кода и снижает накладные расходы при проведении анализа за счёт единовременного извлечения статических данных на этапе инструментации.

B. Практическое применение системы инструментации

В рамках практических экспериментов по использованию системы статической инструментации были разработаны модули инструментации для проведения динамического анализа, нацеленного на автоматический поиск дефектов. Данные модули разрабатывались для существующей системы анализа Avalanche [11], стандартно использующей динамическую инструментацию на основе комплекса Valgrind [12] для сбора необходимых данных.

Система Avalanche осуществляет обход возможных путей выполнения целевой программы на основе отслеживания потока помеченных данных, составления систем булевых формул для описания путей выполнения и автоматической генерации новых входных данных с помощью решателя булевых формул. Система осуществляет итеративный анализ, причём на каждой итерации целевая программа выполняется под контролем двух модулей, извлекающих дополнительные данные, — tracegrind (модуль инструментации с целью получения полной трассы инструкций, оперирующих помеченными данными) и covgrind (модуль инструментации с целью вычисления метрики покрытия на основе количества выполненных базовых блоков). Для работы с удалёнными и внешними устройствами, обладающими ограниченными вычислительными мощностями, в системе Avalanche предусмотрен специальным режим, при использовании которого на целевом устройстве осуществляется только запуск программы, а полная обработка трасс и работа решателя булевых формул выполняется на хост-устройстве.

Реализации модулей covgrind и tracegrind на основе статической инструментации были использованы в рамках указанного режима работы и позволили значительно ускорить анализ набора целевых программ. Различия в обработке инструкций систем статической и динамической инструментации привели к малому изменению порядка обхода ветвей, однако повышение скорости обхода ветвей были зафиксированы на всех проектах. Обнаруженные дефекты соответствуют дефектам, найденным системой Avalanche ранее на рассмотренных проектах для архитектуры x86_64. В таблице 1 приведён краткий обзор результатов работы системы с использованием двух типов инструментации.

Таблица I. Результаты работы системы Avalanche

Программа	**Итерации анализа**		**Количество дефектов**		**Время обнаружения дефекта, с**	
	Стат.	***Дин.***	***Стат.***	***Дин.***	***Стат.***	***Дин.***
swfdump	575	105	4	1	4	31
mpeg2dec	302	55	1	1	3	25
cjpeg	1331	236	1	1	20	600
qtdump	1027	189	1	1	33	1269
mpeg3dump	125	54	2	1	40	112

Время анализа для всех запусков было ограничено двумя часами (стандартная практика для инструмента Avalanche), в качестве начальных входных данных использовались файлы, соответствующие предыдущим результатам применения Avalanche. В список проанализированных программ входят:

- утилита swfdump (swftools-0.9.0) для извлечения информации о содержимом файлов формата SWF;
- утилита mpeg2dec (libmpeg2-0.5.1) для декодирования файлов формата MPEG-2;
- утилита cjpeg (libjpeg-7) для кодирования файлов в формат JPG;
- утилита qtdump (libquicktime-1.1.3) для декодирования файлов нескольких форматов (например, AVI)
- утилита mpeg3dump (libmpeg3-1.8) для декодирования файлов в формате MPEG-3.

Для 4 целевых программ из 5 при применении статического инструментации было достигнуто 5-кратное увеличение количества итераций анализа (количество итераций пропорционально количеству обойдённых ветвей и проверенных наборов формул). Известные по предыдущим результатам работы Avalanche дефекты были достигнуты ранее и для двух проектов было найдено больше различных дефектов. Для программы mpeg3dump было зафиксировано менее значительное ускорение, что связано с большим количеством ветвей выполнения, приводящих к зацикливанию программы. Зацикливание приводит к задержке по ожиданию срабатывания таймера и к снижению полезного времени анализа.

V. Заключение

Рассмотренный подход позволяет создавать точные и эффективные инструменты анализа, осуществляющие трассировку данных и инструкций. Наибольший выигрыш по производительности по сравнению с существующими динамическими аналогами достигается при выполнении многочисленных запусков модифицированных целевых программ (с целью проверки различных путей выполнения и сценариев работы). На данный момент основной фокус реализации подхода стоит на работе с платформой ARM, однако интеграция с системой binutils обеспечивает возможность поддержки других систем (в рамках формата ELF) без необходимости модификации базовых модулей системы инструментации.

В настоящее время рассматриваются следующие направления будущей деятельности:

- Расширение множества поддерживаемых типов точек инструментации и интерфейса, предоставляемого пользователю для описания инструментационного кода.
- Исследования возможностей оптимизации инструментации за счёт уменьшения количества инструкций, обеспечивающих корректность перехвата управления, и реорганизации переходов между исполняемым кодом целевой программы и блоками инструментационного кода.
- Повышение точности работы декодера исполняемого кода при работе с файлами ELF, не содержащими дополнительные секции управляющей информации.
- Изменение структуры внутреннего представления инструкций для повышения гибкости интерфейса, предоставляемого для написания инструментационного кода, и адаптации к поддержке различных наборов инструкций. Одним из возможных решений в данном направлении является полномасштабная трансформация блоков инструкций в операции промежуточного языка (например, REIL [14]).

Ссылки

[1] Amitabh Srivastava, Alan Eustace. ATOM: A System for Building Customized Program Analysis Tools, WRL Research Report 94/2, Western Research Laboratory, Palo Alto, CA, USA.

[2] Amitabh Srivastava and David W. Wall. A Pratical System for Intermodule Code Optimization at Link-Time. Journal of Programming Language, l(l), pp 1-18, March 1993.

[3] James R. Larus, Eric Scnharr. EEL: Machine-Independent Executable Editing. PLDI '95 Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation, 1995. pp. 291-300.

[4] Susanta Nanda, Wei Li, Lap-Chung Lam, Tzi-cker Chiueh. BIRD: Binary Interpretation using Runtime Disassembly. International Symposium on Code Generation and Optimization, 2006 .

[5] B. Calder, T. Austin, D. Yang, T. Sherwood, S. Sair, D. Newquist, and T. Cusac. BitRaker Anvil: Binary Instrumentation for Rapid Creation of Simulation and Workload Analysis Tools. In Global Signal Processing (GSPx) Conference, 2004 .

[6] Lamia Djoudi, Denis Barthou, Patrick Carribault, Christophe Lemuet, Jean-Thomas Acquaviva, and William Jalby. MAQAO: Modular Assembler Quality Analyzer and Optimizer for Itanium 2. In Workshop on Explicitly Parallel Instruction Computing Techniques, Santa Jose, California, March 2005.

[7] Andres S. Charif-Rubial, Denis Barthou, Cedric Valensi, Sameer S Shende, Allen D. Malony and William Jalby. MIL: A language to build program analysis tools through static binary instrumentation. In 20th Annual International Conference on High Performance Computing, HiPC'13, Hyderabad, India, December 2013.

[8] Michael A. Laurenzano, Mustafa M. Tikir, Laura Carrington, Allan Snavely. PEBIL: Efficient static binary instrumentation for Linux. 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS), 2010 .

[9] Barton P. Miller and Andrew R. Bernat, Anywhere, Any Time Binary Instrumentation, ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE), Szeged, Hungary, 2011 .

[10] GNU binutils, http://www.gnu.org/software/binutils/. Дата обращения: 10.08.2015.

[11] Исаев И.К., Сидоров Д.В. Применение динамического анализа для генерации входных данных, демонстрирующих критические ошибки и уязвимости в программах. Программирование, №4, 2010.

[12] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), San Diego, California, USA, 2007 .

[13] Thomas Dullien and Sebastian Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. In CanSecWest, 2009.

Применение статического анализа для оптимизации динамического поиска гонок

Роскошный Яков
Университет ИТМО
watson9494@gmail.com

Цителов Дмитрий
ООО "Эксперт-Система"
tsitelov@acm.org

Трифанов Виталий
ООО "Эксперт-Система"
trifanov@devexperts.com

Елизаров Роман
ООО "Эксперт-Система"
elizarov@devexperts.com

***Аннотация*—Обнаружение гонок – актуальная задача верификации многопоточных программ. Существующие методы подразделяются на динамические, статические и смешанные. Одной из основных проблем динамических методов являются высокие накладные расходы, в частности, необходимость контролировать все обращения к разделяемым переменным, по которым возможна гонка. Для обеспечения корректной синхронизации при обращении к потенциально разделяемых переменным часто применяются устойчивые шаблонные решения. В статье рассматриваются некоторые наиболее типичные шаблоны обеспечения синхронизации доступа к разделяемым переменным и перспективы доказательства их корректности средствами статического анализа. Предложен алгоритм определения переменных защищённых во всех возможных ветвях исполнения и реализован соответствующий инструмент для языка Java. Инструмент апробирован на различных синтетических тестах и реальных программах, интегрирован с ранее разработанным авторами динамическим детектором гонок jDRD, который был представлен на TMPA-2013 [1].**

Ключевые слова—статический анализ; поиск гонок; многопоточность; Java;

I. Введение

Многие современные языки программирования, в том числе Java, используют многопоточную модель с разделяемой памятью. Для публикации изменений, сделанных потоком, и для получения изменений, сделанных другими потоками предусмотрены операции синхронизации. Контракты модели памяти описывают формальные гарантии, которые предоставляют различные операции синхронизации. Использование различных операций синхронизации обеспечивает частичную упорядоченность между операциями в различных потоках. Если между обращениями к разделяемой переменной нет упорядочивания по времени, и хотя бы одна операция является записью, то наступает *состояние гонки*. Обнаружить гонку на этапе тестирования очень сложно, поскольку исполнение многопоточных программ в общем случае недетерминировано и зависит от большого количества неконтролируемых факторов. Эффект гонки может проявляться только при очень специфическом порядке выполнения программы.. Если гонка произошла, то испорченные данные могут долго распространяться по программе, и несоответствие может обнаружиться совсем в другом месте. Повторный запуск программы на тех же данных может не привести к возникновению гонки. Существующие методы поиска гонок подразделяются на *динамические*, *статические* и *смешанные*. Задача нахождения гонок статическим анализом является NP-трудной[2, 3], поэтому при статическом подходе обычно снижается точность и полнота результата. Одной из главных проблем динамического подхода является снижение производительности анализируемой программы, поскольку в основе лежит регистрация обращений ко всем потенциально разделяемым переменным.

Для синхронизации разделяемых переменных часто применяются устойчивые шаблонные решения, которые гарантируют отсутствие гонок при доступе к этим переменным. В данной работе рассматриваются шаблоны обеспечения синхронизации доступа к данным, а также возможности доказательства их корректности средствами статического анализа.

Представлен алгоритм и инструмент для нахождения переменных, защищенных во всех ветвях исполнения. Инструмент интегрирован с динамическим детектором гонок для Java-программ jDRD.

II. Шаблоны обеспечения синхронизации

Для обеспечения синхронизации используются различные конструкции. Некоторые конструкции удовлетворяют повторяющимся шаблонам, для которых можно доказать корректность синхронизации объекта.

Для выделения шаблонов рассматривались поля, по которым были зарегистрированы обращения и не было найдено гонок динамическим детектором. Анализировались поля, обращения к которым встречались наиболее часто. Были выделены следующие шаблоны :

- *Безопасная публикация объекта без последующих изменений (Safe published read-only).* Поля класса инициализируются на начальном этапе конфигурации, а затем только читаются (речь не идёт о так называемых *неизменяемых (immutable)* объектах с исключительно final полями). Типичный представитель – конфигурация приложения, доступная через статический контекст. Гонки по таким полям могут возникнуть только при отсутствии синхронизации между начальной инициализацией объектов данного класса и их дальнейшим использованием. Для доказательства отсутствия гонок при обращении к таким полям потребуется проанализировать последовательность операций инициализации и передачи объектов между потоками, при этом, в общем случае, отсутствие гонок может быть гарантировано только для некоторых объектов класса. Доказательство теории об отсутствии гонок по полю будет при этом *локальным*, то есть относиться к некоторой ограниченной области кода или зависеть от точки порождения объекта.
- Объект, *принадлежащий потоку (Thread confined).* Поля класса, экземпляры объектов которого не покидают контекст одного потока. Это свойство также в общем случае зависит от контекста использования или точки порождения объекта.
- *Безопасная публикация объекта (Safe publishing).* Поля класса, экземпляры которого передаются потоку, и затем не покидают его пределов. Типичный случай – конструирование потомка класса Thread или Runnable для инициализации нового потока. Это свойство также в общем случае зависит от экземпляра класса. Для верификации таких полей понадобится различать экземпляры объектов
- *Внутренне синхронизированное поле (Internally synchronized).* Поле корректно защищено внутренней блокировкой. То есть, существует такая блокировка, что любая операция с полем проводится с этой блокировкой. Данное свойство является *глобальным*, то есть не зависит от экземпляра объекта и контекста использования.
- *Неизменяемые (Immutable)* объекты – все их поля final и ссылка на объект (this) не «утекает» при конструировании. Данное свойство глобально и тривиально доказывается.[1]

Если теория об отсутствии гонок по полю глобальна, то можно исключить из динамического анализа все обращения к этому полю. Проверка локальных теорий в случае динамического детектора потребует встраивания дополнительных проверок на соответствие контексту при инструментировании байт-кода или генерации альтернативных реализаций класса, как предложено в [11]. Оба подхода потребовали бы существенно более сложных технических решений, поэтому было принято решение начать с исследования оптимизаций на основе глобальных свойств. В данной работе рассматривается поиск внутренне синхронизированных полей.

[1] Объекты такого типа не попали в область часто используемых полей, поскольку jDRD исключает их из анализа автоматически.

III. Алгоритм для поиска корректно-синхронизированных полей.

Будем называть поле корректно-синхронизированным, если существует такая блокировка, что любая операция с полем проводится под этой блокировкой.

Формально, поле *f* является корректно-синхронизированным, если существует такой синхронизационный объект *l*, что любая операция доступа к полю *access(f)* в потоке T_i выполняется между взятием и освобождением объекта *lock* в том же потоке при любом сценарии исполнения ($a \rightarrow b$ означает *a happens-before b*):

T_i: *acquire(l)* → *access(f)* → *release(l)*

Поскольку по определению синхронизационного объекта для произвольных обращений либо T_i:*release(l)* → T_j:*acquire(l)* либо T_j:*release(l)* → T_i:*acquire(l)*, то для любой пары операций T_i:*access(f)* и T_j:*access(f)* найдётся такая пара обращений (T_i:*release(l)* → T_j:*acquire(l)*) или (T_j:*release(l)* → T_i:*acquire(l)*), что либо будет верно T_i:*access(f)* → T_i:*release(l)* → T_j:*acquire(l)* → T_j:*access(f)*, либо T_j:*access(f)* → T_j:*release(l)* → T_i:*acquire(l)* → T_i:*access(f)* соответственно. Таким образом, между любыми двумя обращениями к полю *f* есть отношение *happens-before* и гонка при обращении к этому полю невозможна.

Алгоритм ищет для полей блокировки, удовлетворяющие условиям. Алгоритм можно разделить на 3 части.

- Получение промежуточного представления и графа потока исполнения.
- Получение множества возможных блокировок.
- Обход графа потока исполнения и выделение корректно-синхронизированных полей.

A. *Получение промежуточного представления и графа потока исполнения.*

Граф потока исполнения (англ. control flow graph, CFG) — это все возможные пути исполнения части программы, представленные в виде графа. Вершинами данного графа являются последовательности операций, не содержащие в себе ни операций передачи управления, ни точек, на которые управление передается из других частей программы. Ребра показывают возможные переходы между операциями [4]. Построение данных графов зависит от выбранной формы представления программы. Для Java-программ изначально доступно представление в виде байт-кода. Если доступен исходный код, то можно проводить анализ самой Java-программы. Данные представления неудобны для последующего анализа. Байт-код имеет большое количество инструкций. Java-код имеет большое

количество синтаксических конструкций, которые трудно проанализировать.

Промежуточное представление (англ. Intermediate representation, IR) — язык абстрактной машины, упрощающий проведение анализа.

Для данной работы было выбрано промежуточное представление приведенное в *SSA*-форму [5]. SSA-форма технически упрощает получение множества блокировок. В промежуточном представлении, используемом в данной работе, вся работа со стеком заменена на локальные переменные. Как и в байт-коде имеется две примитивные операции синхронизации: *monitorEnter* и *monitorExit*. Эти операции отвечают за взятие и освобождение монитора объекта. Каждая переменная имеет единственное место присваивания, так как представление удовлетворяет *SSA*-форме. Для присваивания переменной может использоваться *φ-функция* Если локальная переменная может принимать несколько значений, то данная переменная принимает значение φ-функции от всех возможных ее значений. В данной работе промежуточное представление и CFG получены с помощью библиотеки Soot [6]. Присваивания в данном представлении удовлетворяют следующей грамматике (local — локальная переменная, field — поле класса, constant – константа).

Таблица I. Грамматика используемого IR.

Продукции грамматики	
imm	local constant
expr	imm_1 binop imm_2 (type) imm imm instanceof type invokeExpr new refType newarray (type) [imm] neg imm
assignStmt	local = φ(imm_1, imm_2, ...) local = local.field | field | imm | expr field = imm local.field = imm

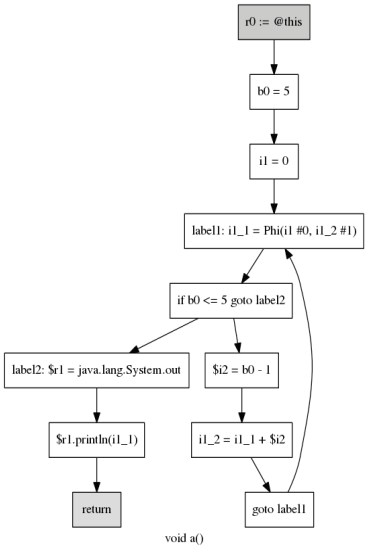
Листинг I. Пример Java-программы

```
public void a() {
        int x = 5;
        int y = 0;
        while (x > 5) {
                y += x − 1;
        }
        System.out.println(y) ;
}
```

Листинг II. Описанное промежуточное представление

```
public void a() {
        r0 := @this ;
        b0 = 5;
        i1 = 0;
        label1 :
        i1_1 = Phi(i1 #0, i1_2 #1);
        if b0 <= 5 goto label2 ;
        $i2 = b0 − 1;
        i1_2 = i1_1 + $i2 ;
        goto label1 ;
        label2 :
        $r1 = java.lang.System.out;
        $r1.println(i1_1) ;
        return ;
}
```

Рис 1. Схема CFG метода



B. Получение множества возможных блокировок.

Алгоритм должен быть точным, то есть сообщать только о тех полях, которые действительно корректно-синхронизированы. В полученном представлении операция взятия блокировки осуществляется с локальной переменной. Под блокировкой подразумевается ссылка на объект, по которому осуществляется синхронизация(носитель монитора при встроенной синхронизации или объект блокировки для java.util.concurrent). При обходе графа, который будет рассмотрен далее, нужно поддерживать текущее множество взятых блокировок. Если нельзя статически доказать, что локальная переменная всегда ссылается на одно поле, или переменная ссылается на поле, которое может измениться, то операции взятия блокировки по

Листинг III. Блокировка с несколькими возможными значениями

```
Class A {
        final Object lock1 = new Object() ;
        final Object lock2 = new Object() ;
        void a() {
                Object lock ;
                if (System.currentTimeMilis() % 10000 == 0) {
                        lock = lock1 ;
                } else {
                        lock = lock2 ;
                }
                syncronized(lock) {
                        // some code
                }
        }
}
```

Листинг IV. БЛОКИРОВКА ПО ПОЛЮ ДРУГОГО КЛАССА.

```
Class A {
        final B b = new B() ;
        public void a() {
                synchronized (b. lock) {
                        //some code
                }
        }
}
```

таким переменным не должны добавлять информацию относительно взятых блокировок. Так как полученное представление является SSA, то каждая переменная имеет единственное место инициализации. Если переменная инициализируется как *φ*-функция то эта переменная может иметь не единственное значение. Также если локальная переменная *l1* принимает значение другой локальной переменной *l2* или поля локальной переменной *l2.field*, и *l2* может иметь не единственное значение, то *l1* тоже может иметь не единственное значение и не может являться блокировкой.

Сначала выделим множество переменных, которые инициализируются φ-функциями. Затем построим замыкание данного множества относительно операций присваивания. Полученное множество, очевидно, будет искомым.

Поле, которое может являться потенциальной блокировкой должно быть неизменяемым, то есть должно иметь модификатор final. В общем случае каждое поле в пути блокировки должно иметь модификатор final. В случае со static полями достаточно проверить, что поле имеет модификатор final[1].

C. *Обход графа потока исполнения.*

Обход является рекурсивным, напоминает обход в глубину, но с некоторыми отличиями. При обходе поддерживается множество текущих взятых блокировок *curLocks*. Также для каждой вершины *CFG* хранится множество блокировок, с которыми обход уже посещал данную вершину *v.locks*. При входе в вершину v нужно сравнить *curLocks* и *v.locks*. Если $v.locks \subseteq curLocks$, то можно не продолжать обход вершины v. Иначе, в *v.locks* и *curLocks* запишем $v.locks \cap curLocks$ и продолжим обход. Записывать нужно пересечение, так как если существует ветка обхода, в которой вершина *v* посещена без блокировки *l*, то нельзя гарантировать, что операция вершины *v* защищена блокировкой *l*. Изменять *curLocks* нужно при операциях взятия и освобождения блокировки. Для каждого поля сохраним множество блокировок, с которыми обращались к данному полю *f.locks*. При обращении к полю *f* нужно пересечь *f.locks* и *curLocks*.

Рассмотрим подробно обработку операций синхронизации и обращений к полям.

1) *Операции синхронизации.* Если при обходе встретилась операция синхронизации, то нужно изменить *curLocks*. Но сначала нужно проверить, что переменная, над которой осуществляется операция синхронизации, может являться блокировкой. Данная проверка описана в предыдущем разделе. Далее, если текущая операция — операция взятия блокировки, то добавляем блокировку в *curLocks*; а если операция освобождения, то удаляем блокировку из *curLocks*. Помимо стандартных операций *monitorEnter* и *monitorExit* в данной работе рассмотрены блокировки пакета java.util.concurrent и их парные операции *lock*() и *unlock*().

2) *Обращения к полям.* При обращении к полю может возникнуть состояние гонки. Но, если существует блокировка *l*, такая что любая операция чтения и записи с полем *v* производится со взятой *l*, то поле *v* корректно-синхронизировано. Таким образом, для каждого поля *f* надо поддерживать множество блокировок *f.locks*, с которыми гарантировано обращались к данному полю. А при обращении к полю *f* сужать *f.locks* до пересечения *f.locks* и *curLocks*.

D. *Обработка методов, защищенных блокировкой.*

Пока в алгоритме рассматривался обход *CFG* каждого метода независимо. Но существуют методы, вызовы которых всегда защищены определенной блокировкой. Соответственно, любая операция в данном методе защищена этой блокировкой. Анализ методов может добавить информации о множестве текущих блокировок, что приведет к увеличению найденных корректно-синхронизированных полей. Предварительно выделив для метода множество блокировок, с которыми он гарантировано вызывается, можно улучшить анализ.

Для поиска блокировок, которыми защищен метод, можно использовать алгоритм аналогичный базовому алгоритму поиска корректно-синхронизированных полей. Для каждого метода *m* надо поддерживать множество блокировок *m.locks*, с которыми гарантировано вызывался данный метод. При операции вызова метода m записывать

Листинг V. ПСЕВДОКОД ОБХОДА ГРАФА

```
function visit( CFGVertex v, Set<Lock> currentLocks)
        if v.locks ⊆ currentLocks then
                break
        else
                v.locks ← v.locks ∩ currentLocks
                currentLocks ← v.locks ∩ currentLocks
        op ← v.getOperation()
        if op.isMonitorEnterOperation() then
                currentLocks.add(v.getOperations.getLock())
        if op.isMonitorExitOperation() then
                currentLocks.remove(v.getOperations.getLock())
        if op.isFieldAssignmentOperation() then
                field ← op.getField()
                field.locks ← field.locks ∩ currentLocks
        for all CFGVertex c : v.childs
                do visit(c, currentLocks)
end function
```

[1] Строго говоря, поле хранящее ссылку на объект блокировки может и не быть final, если гарантируется его неизменность и инициализация предшествует любому использованию. Однако это является плохим стилем и на практике редко встречается.

в *m.locks* пересечение *curLocks* и *m.locks*.

После одного обхода для каждого метода *m* будет сформировано множество *m.locks*. Далее можно повторить обход графа с появившимися новыми блокировками. Второй и последующие обходы нужны, так как после каждого обхода множество блокировок, которые защищают метод может увеличиться. Если после очередного обхода в множество блокировок, защищающих метод *m*, добавлена блокировка *l*, то все операции метода *m* защищены *l*. Это означает, что при следующем обходе, любой метод *k*, вызываемый из m может стать защищенным блокировкой *l*. Оценим количество таких обходов. Если после очередного обхода для каждого метода *m* не изменилось *m.locks*, то можно завершать алгоритм. Теоретически может понадобиться M×L обходов (M — количество анализируемых методов, L — количество блокировок, которыми защищен хотя бы один метод). На практике нескольких (трех или четырех) таких обходов достаточно, так как операции синхронизации редко используются для того, чтобы синхронизировать операции через пять вызовов метода.

Также при обработке методов необходимо корректно обрабатывать виртуальные методы. Класс, у которого будет вызван метод, можно определить только во время выполнения. При статическом анализе можно утверждать, что при вызове метода *m* у объекта типа *t* будет выполнен метод одного из наследников класса *t*. Таким образом, все внутренние вызовы метода *m* класса *t* гарантировано защищены блокировкой *l*, если все вызовы метода *m* предков класса *t* защищены *l*.

IV. Практические результаты

Реализован инструмент для поиска корректно-синхронизированных полей, использующий алгоритм из предыдущего раздела. Инструмент написан на языке программирования Java и интегрирован с динамическим детектором гонок jDRD.

jDRD получает список корректно-синхронизированных полей через конфигурационный файл, генерируемый разработанным инструментом. В jDRD был также включен сбор статистики, который отслеживает количество обращений к корректно-синхронизированным полям, выделенным статическим анализом.

Предусмотрено два режима работы: локальный и глобальный. *Локальный режим* подразумевает, что анализируемый код может использоваться сторонними приложениями, не входящими в область анализа. Таким образом, если поле корректно-синхронизировано в рамках анализируемой программы, но оно доступно для изменения по правилам видимости, то в локальном режиме оно не считается корректно-синхронизированным. Аналогичные рассуждения касаются методов. Такой режим нужен для анализа различных библиотек. *Глобальный режим* подразумевает, что в приложении используется только код из области анализа. Если поле корректно-синхронизировано в рамках анализируемой программы, то даже если оно доступно для изменения, глобальный режим отметит его как корректно-синхронизированное. Данный режим предназначен для тестирования законченных приложений. Множество переменных, выделенное в глобальном режиме включает в себя множество, выделенное при работе в локальном режиме.

В качестве операций синхронизации поддерживаются *synchronized* методы и блоки, а также парные операции захвата и освобождения блокировки java.util.concurrent.locks.ReentrantLock.

Было проведено тестирование разработанной программы на различных тестах. Создан набор синтетических тестов, покрывающий большинство конструкций Java-программ. В качестве тестов были использованы программы использующие различные операции синхронизации, обработку исключений(exceptions), статические и нестатические блокировки и поля, внутренние классы и т.д. Также были проведены запуски на реальных библиотеках и приложениях с последующей проверкой результатов.

Инструмент был запущен на различных приложениях, которые активно используют конкурентный доступ к данным. Далее представлено краткое описание тестируемых программ.

- *MARS*(Monitoring and reporting system) — система мониторинга реального времени. Используется для отображения различных данных приложения.
- *dxFeed* — система, отвечающая за быструю доставку больших объёмов данных(котировок).
- *Tomcat* — контейнер сервлетов. Позволяет запускать веб-приложения.
- *jtt* — система менеджмента времени, использующая в качестве сервера JIRA

Таблица II. Полученные результаты

Приложение	Общее количество полей	Корректно-синхронизированных полей	Процент
dxFeed	5212	982	19
MARS	2311	477	21
Tomcat	6793	1546	23
Jtt	1165	176	15

Таблица III. Полученные результаты, без учета final полей

Приложение	Общее количество полей	Корректно-синхронизированных полей	Процент
dxFeed	1555	89	6
MARS	545	20	4
Tomcat	2924	132	5
Jtt	405	28	7

V. Аналогичные исследования

В настоящее время статический анализ рассмотрен довольно широко. Существует большое количество инструментов, которые проводят статический анализ Java-программ для различных целей.

Существует несколько реализаций статического подхода для обнаружения гонок[7, 8]. Также существуют инструменты, такие как FindBugs [9] и ThreadSafe [10], которые проводят анализ конкурентного доступа к данным. Но все эти реализации ориентированы на поиск ошибок в программах и для оптимизации динамического детектора не подходят.

В работе [11] приведен способ для нахождения объектов, которые не покидают контекст потока или метода. Результаты данной работы используются для выделения объектов на стеке вместо кучи, а также для ликвидации ненужных операций синхронизации. Целью работы являлась оптимизация компилятора, однако метод анализа, рассмотренный в данной работе теоретически можно использовать для оптимизации динамического детектора за счет исключения из области динамического анализа отдельных экземпляров объектов. Однако применение результатов такого анализа может потребовать нетривиальных изменений самого детектора и его характеристик. Это направление является возможной областью развития данной работы.

VI. Заключение

Поиск гонок – актуальная задача верификации многопоточных программ и динамические методы являются одним из основных подходов в этой области. Одной из главных проблем динамических методов является снижение производительности анализируемой программы. В данной работе был рассмотрен вопрос оптимизации динамических детекторов, путем проведения предварительного статического анализа.

Рассмотрены некоторые наиболее типичные шаблоны обеспечения синхронизации доступа к разделяемым переменным и перспективы доказательства их корректности средствами статического анализа.

В рамках работы был разработан алгоритм для поиска корректно-синхронизированных полей. Данный алгоритм позволяет обнаруживать поля, при обращении к которым гарантированно не может возникнуть гонки. Применение результатов предложенного анализа позволяет сократить время выполнения и объем потребления памяти динамического детектора без внесения существенных изменений. Полученный алгоритм позволяет отслеживать различные операции синхронизации, используемые для защиты обращений к разделяемым переменным. Алгоритм основан на обходе графов потока исполнения методов

Разработан инструмент, реализующий предложенный алгоритм. Корректность работы программы анализа была протестирована на наборе синтетических тестов, покрывающем различные варианты условий, предусмотренных алгоритмом. Полученный инструмент предоставляет интерфейс для интеграции с динамическими детекторами. В рамках данной работы он был интегрирован с существующим динамическим детектором гонок для Java-программ.

Также результаты анализа могут быть использованы для поиска некоторых ошибок синхронизации, поскольку в результате работы инструмента выдаётся информация о критических обращениях, препятствующих признанию поля корректно синхронизированным.

Данная работа имеет хорошие перспективы для развития. Используя результаты работы и методы *escape*-анализа, планируется дальнейшая оптимизация динамического детектора за счет исключения из области динамического анализа отдельных экземпляров объектов.

[1] Трифанов В.Ю, Цителов Д.И. Динамический поиск гонок в Java-программах на основе синхронизационных контрактов. Материалы конференции «Инструменты и методы анализа программ (ТМРА-2013”), Кострома, 2013. сс. 273-285

[2] Netzer R. H. B. Race Condition Detection for Debugging Shared-memory Parallel Programs. Ph.D. thesis. Madison, WI, USA: University of Wisconsin at Madison, 1991. UMI Order No. GAX91-34338.

[3] Netzer R. H. B., Miller B. P. What Are Race Conditions?: Some Issues and Formalizations // ACM Lett. Program. Lang. Syst. New York, NY, USA, 1992. Vol. 1, no. 1. P. 74–88. URL: http://doi.acm.org/10.1145/130616.130623.

[4] Soot - a Java Bytecode Optimization Framework / R. Vall´ee-Rai, P. Co, E. Gagnon et al. 2000. URL: http://dl.acm.org/citation.cfm?id=781995.782008.

[5] Bilardi G., Pingali K. Algorithms for Computing the Static Single Assignment Form // J. ACM. New York, NY, USA, 2003. Vol. 50, no. 3. P. 375–425. URL: http://doi.acm.org/10.1145/765568.765573.

[6] Soot. A framework for analyzing and transforming Java and Android Applications. URL: http://sable.github.io/soot/.

[7] Naik M., Aiken A., Whaley J. Effective Static Race Detection for Java // SIGPLAN Not. New York, NY, USA, 2006. Vol. 41, no. 6. P. 308–319. URL: http://doi.acm.org/10.1145/1133255.1134018.

[8] Chord: A Program Analysis Platform for Java. URL: http://pag.gatech.edu/chord.

[9] FindBugs. URL: http://www.ibm.com/developerworks/java/library/j-findbug1/.

[10] ThreadSafe. URL: http://www.contemplateltd.com/threadsafe.

[11] Whaley J., Rinard M. Compositional Pointer and Escape Analysis for Java Programs // SIGPLAN Not. New York, NY, USA, 1999. Vol. 34, no. 10. P. 187–206. URL: http://doi.acm.org/10.1145/320385.320400.

Лексический анализ динамически формируемых строковых выражений

Полубелова Марина Игоревна
Санкт-Петербургский Государственный Университет
Санкт-Петербург, Россия
polubelovam@gmail.com

Григорьев Семён Вячеславович
Санкт-Петербургский Государственный Университет
Санкт-Петербург, Россия
Semen.Grigorev@jetbrains.com

***Аннотация*—Строковые выражения могут использоваться для формирования и последующего исполнения кода во время выполнения основной программы. Такой подход обладает высокой выразительностью, однако затрудняет разработку, отладку и сопровождение, а также является источником таких уязвимостей, как внедрение SQL и межсайтовый скриптинг. Статический анализ строковых выражений предназначен для борьбы с недостатками подхода посредством проверки, что все формируемые выражения удовлетворяют некоторым свойствам, без запуска программы. Лексический анализ или токенизация формируемого кода является важным шагом такого статического анализа. В данной статье описан автоматизированный подход к созданию лексических анализаторов динамически формируемого кода, который позволит упростить создание инструментов, предназначенных для статического анализа такого кода.**

***Ключевые слова*—анализ строковых выражений, генератор лексических анализаторов, лексический анализ, встроенные языки, string analysis, lexing, string-embedded language**

I. Введение

Многие языки программирования позволяют работать со строковыми выражениями. Они могут формироваться динамически с использованием строковых операций и языковых конструкций, например, условных операторов и циклов. Такие выражения широко используются в программных интерфейсах ODBC, ADO.NET, JDBC, предназначенных для формирования запросов к базе данных в языках программирования C++, C#, Java, соответственно, а также в web-программировании. Некоторые примеры использования динамически формируемых строковых выражений представлены в листингах 1, 2.

```
private void Go(bool cond)
{
  string tableName = cond ? "Sold " : "OnSale ";
  string query =
      "SELECT ProductID, UnitPrice, ProductName"
          + " FROM dbo.products_" + tableName
          + " WHERE UnitPrice > 1000 "
          + " ORDER BY UnitPrice DESC;";
  Program.ExecuteImmediate(query);
}
```

Листинг 1: Пример встроенного SQL в C#

```
<?php
//Embedded SQL
$query = 'SELECT * FROM '. $my_table;
$res = mysql_query($query);
//HTML markup generation
echo "<table>\n";
while($line=mysql_fetch_array($res, MYSQL_ASSOC)){
echo "\t<tr>\n";
foreach ($line as $col_value){
echo "\t\t<td>$col_value</td>\n";}
echo "\t</tr>\n";}
echo "</table>\n";
?>
```

Листинг 2: Использование нескольких встроенных в PHP языков (MySQL, HTML)

Динамически формируемые строковые выражения воспринимаются компилятором как обычные строки, что усложняет разработку и сопровождение системы. Во-первых, о наличии ошибок, таких как лексических или синтаксических, в сформированном выражении становится известно только в момент выполнения программы, когда оно начинает выполняться в своем программном окружении. Во-вторых, при ненадлежащей обработке пользовательского ввода система становится уязвимой, например, для SQL инъекций или межсайтового скриптинга. Указанные проблемы можно решить, включив обработку строковых выражений в статический анализ программы.

Классическим подходом к статическому анализу является проведение лексического анализа и синтаксического разбора исходного кода. Синтаксический анализ строит структурное представление, которое используется в дальнейшем, например, для семантического анализа или трансформаций, проводимых в контексте реинжиниринга. В рамках данного подхода широко распространен автоматизированный способ создания

лексических и синтаксических анализаторов такими генераторами как Lex, Yacc и их потомками. Использование подобных инструментов сокращает затраты на создание программных продуктов, требующих построения структурного представления кода. Однако существующие генераторы лексических и синтаксических анализаторов не применимы для создания инструментов обработки динамически формируемого кода из-за того, что такой код, как правило, не представим в виде линейного потока, принимаемого на вход классическими анализаторами.

Таким образом, есть необходимость в создании генератора лексических анализаторов для динамически формируемого кода, который предоставляет функциональность, аналогичную классическому. Это позволит упростить создание инструментов, которые предназначены для решения задач, возникающих при обработке динамически формируемого кода. Примерами таких задач являются трансформация запросов с одного языка на другой, возникающая в контексте реинжиниринга информационных систем, подсветка синтаксиса и ошибок в интегрированных средах разработки, а также подсчет различных метрик.

В данной статье описан автоматизированный подход создания лексического анализатора для динамически формируемого кода. В рамках работы были разработаны алгоритм лексического анализа и генератор лексических анализаторов, за основу которого была взята библиотека FsLex [1]. Указанные компоненты были реализованы как часть проекта YaccConstructor [2], который является модульным инструментом для проведения лексического анализа и синтаксического разбора, а также платформой для поддержки встроенных языков.

II. Обзор

В данном обзоре рассматриваются существующие инструменты, предназначенные для работы с динамически формируемыми выражениями, генератор лексических анализаторов FsLex и проект YaccConstrucor, в котором ведется разработка автоматизированного подхода создания лексических анализаторов для динамически формируемого кода.

A. Обзор существующих инструментов

Для работы с динамически формируемыми строковыми выражениями существует ряд инструментов. Почти все они предназначены для решения какой-то одной конкретной задачи: либо проверка выражения на соответствие описанию некоторой эталонной грамматики, либо статический анализ программы на уязвимость. Так как генерация всех значений динамически формируемого выражения значительно снижает скорость проведения анализа и возможны ситуации, когда количество принимаемых выражением значений может быть бесконечным, то целесообразно иметь конечное представление множества значений данного выражения и уже над ним проводить анализ.

Конечное представление множества значений строкового выражения впервые было использовано в инструменте Java String Analyzer [3]. Этот инструмент предназначен для анализа строк и строковых операций в Java-программе. Результатом этого приближения стал конечный автомат, который используется для проверки включения языков: проверяется включение языка, порождаемого программой, в язык, описанный пользователем. Затем инструмент PHP String Analyzer [4], использовав идею предыдущего инструмента, уточнил проводимую аппроксимацию, результатом которой стала контекстно-свободная грамматика. Этот инструмент применяется для статической валидации HTML-страниц, генерируемых в PHP-программе.

В инструменте Pixy [5], предназначенном для поиска SQL инъекций и межсайтового скриптинга в PHP-программах, применяется техника path pruning, позволяющая проводить анализ только тех значений строкового выражения, которые оно может принять в момент выполнения программы Инструмент Stranger [6] расширяет указанный подход, используя в качестве структурного представления динамически формируемого кода конечный автомат над алфавитом символов обрабатываемого языка. При этом рассматривается общий случай: когда аргументами строковых операций являются конечные автоматы. В рамках инструмента Stranger разработан алгоритм, который вычисляет результат этих операций и возвращает его в виде конечного автомата, что позволяет достичь более высокой точности анализа по сравнению с аналогами.

Разработчики следующего инструмента расширили круг решаемых задач, сформулировав вопросы безопасности, корректности и производительности сформированных запросов с использованием таких программных интерфейсов, как ADO.NET и JDBC. Описание этого инструмента дается в статье [7], в которой также была указана разработанная функциональность: поиск SQL инъекций, извлечение множества всех значений для строкового выражения, удаление неиспользуемых переменных в формируемом запросе, а также проверка на соответствие типов возвращаемого запросом значения с ожидаемым в программе.

Инструмент SAFELI [8], также предназначенный для поиска уязвимостей в web-приложениях, отличается от рассмотренных тем, что структурным представлением динамически формируемого кода является синтаксическое дерево разбора. Результат получается посредством сопоставления полученного дерева с синтаксическим деревом шаблона уязвимости, параметризованного реальными данными. Однако SAFELI не поддерживает обработку строковых выражений, которые могут быть получены при участии строковых операций.

Для проведения лексического анализа и синтаксического разбора множества значений строкового выра-

жения был разработан инструмент Alvor [9], который является расширением для среды разработки Eclipse, предназначенным для статической валидации SQL-выражений, встроенных в программы на Java. Одной из возможностей инструмента является поиск лексических и синтаксических ошибок, однако поддержка нового языка генерируемого кода является нетривиальной задачей из-за отсутствия генераторов лексических и синтаксических анализаторов. Кроме того, Alvor не поддерживает обработку выражений, полученных с помощью строковых операций (кроме конкатенации) и циклов.

B. Инструмент YaccConstructor

YaccConstructor [2] является модульным инструментом с открытым исходном кодом [10], предназначенным для исследований в области лексического анализа и синтаксического разбора, а также платформой для поддержки встроенных языков [11]. Данный инструмент реализован на платформе Microsoft .NET, основной язык разработки — F#.

Разработанный ранее механизм анализа встроенных языков, представленный в работах [12], [11], имел ограничения на структуру динамически формируемого выражения: лексический и синтаксический анализаторы могли обрабатывать только аппроксимацию, представленную в виде ориентированного ациклического графа (DAG). Это не позволяло корректно обрабатывать выражения, полученные с помощью циклов.

В данной работе такое ограничение снимается: лексический анализатор работает с произвольным конечным автоматом над алфавитом символов обрабатываемого языка. Разработанный модуль для лексического анализа, который состоит из генератора лексических анализаторов и интерпретатора, соответствующего предложенному алгоритму лексического анализа, внедрен в инструмент YaccConstructor.

C. Генератор лексических анализаторов FsLex

При проведении лексического анализа часто используются генераторы лексических анализаторов, которые по спецификации обрабатываемого языка строят описание конечного преобразователя, на основе которого входной поток символов преобразуется в поток токенов. Для реализации инструмента для проведения лексического анализа динамически формируемого кода был выбран генератор лексических анализаторов FsLex [1]. Этот выбор обусловлен тем, что реализованный механизм является компонентом инструмента YaccConstructor, основным языком разработки которого является язык программирования F#.

Генератор лексических анализаторов на вход принимает файл с расширением .fsl, в котором описана лексическая спецификация языка, формат определения которой представлен в листинге 3 .

```
{
module Lexer
(* header: any valid F# code can appear here *)
open Parser (* in this module type of tokens
is described *)
}
(* regex macros  *)
let ident = regexp ...
(* rules *)
rule entrypoint = parse
| regexp { action }
| ...
and entrypoint = parse
...
```

Листинг 3: Формат определения спецификации для языка

В описании спецификации выделяется три блока: заголовок (`header`), блок именованных регулярных выражений и блок точек вхождения (`entrypoint`).

Обычно заголовок содержит директивы `open`, необходимые для выполнения действий (`action`), а также вспомогательные функции. Между заголовком и точками вхождений можно определять имена для регулярных выражений. Для каждой точки вхождения генерируется функция на языке F# с таким же именем. Аргументом для таких функций является буфер лексических анализаторов, а возвращают они вычисленное действие. При этом возвращаемое значение должно иметь тип `Parser.token`, который описывает токены.

Буфер лексических анализаторов — это абстрактный тип данных, реализованный в модуле стандартной библиотеки `Lexing` (Microsoft.FSharp.Text.Lexing). Методы `FromString`, `FromBytes`, `FromChars` создают лексические буферы, принимающие данные из строки символов, массива байтов и символов, соответственно. Данный тип позволяет сохранять для лексемы координаты отступов начала и конца в исходном коде и накопленную строку.

Лексический анализатор читает символы из буфера и сравнивает их с регулярным выражением, которое соответствует правилу, до тех пор, пока префикс не совпадает с одним из регулярных выражений. Если префикс совпадает с несколькими регулярными выражениями, то из них выбирается то, которое первым определено в спецификации. Например, если у нас есть два правила, которые на выражения “*” и “**” должны возвращать токены `MULT` и `POW`, соответственно, то в определении спецификации необходимо сперва указать правило для токена `POW`, а только потом для токена `MULT`. После выбора регулярного выражения, к накопленной строке применяется действие, соответствующее этому регулярному выражению.

Пример спецификации для языка арифметических выражений представлен в листинге 4.

```
{
module Calc.Lexer
open Calc.Parse
open Microsoft.FSharp.Text.Lexing

let lexeme lb = LexBuffer<_>.LexemeString lb
}

let digit = ['0'-'9']
let whitespace = [' ' '\t' '\r' '\n']

rule token = parse
| whitespace { token lb }
| ['-']? digit+ ('.'digit+)? (['e' 'E'] digit+)?
      { NUMBER(lexeme lb) }
| '-'  { MINUS(lexeme lb) }
| '/'  { DIV(lexeme lb) }
| '+'  { PLUS(lexeme lb) }
| "**" { POW(lexeme lb) }
| '*'  { MULT(lexeme lb) }
```

Листинг 4: Лексическая спецификация языка арифметических выражений для генератора лексических анализаторов FsLex

Результатом работы генератора является файл с расширением .fs с кодом F# для лексического анализатора. Этот файл содержит код, указанный в заголовке спецификации, конечный преобразователь и функции, которые были созданы на каждую точку вхождения, а также вызов функции интерпретатора построенного конечного преобразователя. Для того чтобы использовать такой лексический анализатор, полученный файл вместе с описанием типов токенов, которые автоматически строятся по грамматике эталонного языка, необходимо добавить в модуль, предназначенный для лексического анализа.

III. Лексический анализ

Основной задачей лексического анализа является преобразование входного потока символов в поток токенов, соответствующих спецификации обрабатываемого языка, и сохранение привязки лексических единиц к исходному коду. В классическом лексическом анализе входной поток является линейным. Для проведения лексического анализа динамически формируемого строкового выражения необходима структура, которая является конечным представлением множества значений этого выражения. Для построения аппроксимации используется алгоритм, предложенный в статье [13], поэтому такой структурой является конечный автомат над алфавитом символов обрабатываемого языка.

Результатом работы лексического анализа динамически формируемого строкового выражения является конечный автомат над алфавитом токенов эталонной грамматики языка. В классическом лексическом анализе токен можно представить в виде структуры, содержащей идентификатор токена и последовательность символов, выделенных из входного потока. В контексте данной статьи токен представляет структуру, содержащую идентификатор токена и *конечный автомат*, описывающий все возможные последовательности символов для данного токена в данной позиции. При этом для каждого символа хранится информация: из какой строки получен этот символ и координаты его позиций внутри этой строки. Это необходимо для того, чтобы сохранить информацию о происхождении токена, так как он мог быть сформирован из различных строковых переменных в исходном коде.

Таким образом, **основная задача лексического анализа динамически формируемого строкового выражения** заключается в переводе конечного автомата над алфавитом символов обрабатываемого языка в конечный автомат над алфавитом токенов эталонной грамматики языка с сохранением привязки лексических единиц к исходному коду.

A. Конечные преобразователи и конечные автоматы

В данной статье для конечных автоматов и конечных преобразователей используются определения, представленные ниже.

Конечным автоматом называется кортеж $A = \langle Q, \Sigma, \Delta, q_0, F\rangle$, где Q — конечное множество состояний, Σ — входной алфавит, при этом Δ задает на множестве Q структуру ориентированного графа, дуги которого помечены символами (x), где $x \in \Sigma \cup \{\varepsilon\}$. q_0 — начальное состояние, $F \subseteq Q$ — множество конечных состояний.

Конечным преобразователем называется кортеж $M = \langle Q, \Sigma, \Sigma', \Delta, q_0, F\rangle$, где Q — конечное множество состояний, Σ, Σ' — входной и выходной алфавиты соответственно, при этом Δ задает на множестве Q структуру ориентированного графа, дуги которого помечены парами $(x : y)$, где $x \in \Sigma$, $y \in \Sigma' \cup \{\varepsilon\}$. q_0 — начальное состояние, $F \subseteq Q$ — множество конечных состояний.

В алгоритме лексического анализа используется операция **композиции** над двумя конечными преобразователями. Композиция конечных преобразователей [14] — это два последовательно взаимодействующих конечных преобразователя: выход первого конечного преобразователя является входом для второго конечного преобразователя. Пусть даны $M_1 = \langle Q_1, \Sigma_1, \Sigma_1', \Delta_1, q_{01}, F_1\rangle$ и $M_2 = \langle Q_2, \Sigma_2, \Sigma_2', \Delta_2, q_{02}, F_2\rangle$, то результатом композицией M_1 и M_2 будет конечный преобразователь $\langle Q_1 \times Q_2, \Sigma_1, \Sigma_2', \Delta, q_{01} \times q_{02}, F_1 \times F_2\rangle$, при этом Δ задает на множестве $Q_1 \times Q_2$ структуру ориентированного графа, дуги которого помечены парами $(x : y)$, где $x \in \Sigma_1$, $y \in \Sigma_2'$, если $\exists z \in \Sigma_1' \cap \Sigma_2$: в M_1 есть дуга, помеченная $(x : z)$, и в M_2 есть дуга, помеченная $(z : y)$.

B. Генератор лексических анализаторов

Для проведения лексического анализа динамически формируемого выражения применяется операция композиции к двум конечным преобразователям, которая использует явное представление этих преобразователей, что порождает ограничения на формат определения лексической спецификации для языка.

Генератор лексических анализаторов FsLex строит конечный преобразователь, в котором входным алфавитом являются символы, имеющие кодировку ASCII или Unicode, выходным алфавитом — функции, тип возвращаемых значений которых соответствует типу `Token`. Эти функции соответствуют действиям (`action`), которые определены в спецификации для языка. Однако бывают ситуации, когда нужно исключить некоторые выражения, например, пробелы и комментарии, из результата. Обычно это происходит на этапе проведения лексического анализа: в соответствующем действии не происходит возвращение токена, однако возвращаемое значение должно иметь тип `Token`. В классическом случае происходит вызов функции, соответствующей точке вхождения, от измененного состояния буфера лексического анализатора. Такой способ не применим для лексического анализа динамически формируемого кода из-за ограничения на выходной алфавит: так как входная и выходная структуры данных не являются потоками, то исключение токена не может быть реализовано простым удалением символов из буфера анализатора. В данной работе предлагается использование типа `Option`: `Some` обозначает возвращение токена, `None` — его отсутствие.

Использование нескольких точек вхождений в определении спецификации означает рекурсивное определение функций, которые создаются на каждую точку вхождения: в соответствующем действии происходит вызов одной из этих функций. Обычно такой подход используют для обработки вложенных конструкций, например, комментариев. В терминах конечного преобразователя это означает, что Δ задает на множестве Q структуру ориентированного графа, дуги которого помечены парами $(x : y)$, где $x \in \Sigma$, $y \in \Sigma' \cup \{\varepsilon\} \cup Q$. То есть конечный преобразователь должен уметь выдавать не только символ из выходного алфавита, но и состояние другого конечного преобразователя, в которое должен перейти анализатор. Обработав этот конечный преобразователь, анализатор должен вернуться в состояние исходного конечного преобразователя, которое было получено при переходе. Определение операции композиции над такими типами конечных преобразователей на данный момент не обнаружено, что создает ограничение на количество используемых точек вхождений: можно использовать только одну точку вхождения.

С введенными выше ограничениями спецификация для языка арифметических выражений будет иметь вид, представленный в листинге 5. Основное отличие от аналогичного описания, представленного в листинге 4 заключается в коде функций `action`: для конструирования “значимых” токенов используется `Some` , а пробелы нужно опускать, для чего используется `None`.

```
{
module Calc.Lexer
open Calc.Parser
}

let digit = ['0'-'9']
let whitespace = [' ' '\t' '\r' '\n']

rule token = parse
| whitespace { None }
| ['-']? digit+ ('.'digit+)? (['e' 'E'] digit+)?
      { Some(NUMBER(gr)) }
| '-'  { Some(MINUS(gr)) }
| '/'  { Some(DIV(gr)) }
| '+'  { Some(PLUS(gr)) }
| "**" { Some(POW(gr)) }
| '*'  { Some(MULT(gr)) }
```

Листинг 5: Лексическая спецификация языка арифметических выражений для генератора лексических анализаторов динамически формируемых выражений

В результате своей работы генератор создает файл с расширением .fs с кодом на языке F# для лексического анализатора. В этом файле содержится код, указанный в заголовке спецификации, конечный преобразователь, массив действий и функция `tokenize`. Функция `tokenize` осуществляет лексический разбор. Массив действий состоит из функций, задающих отображение из некоторого конечного автомата в тип `Option<'token>`. Для спецификации языка, представленного в листинге 5, массив действий будет иметь следующий вид:

```
[|
  (fun (gr:FSA<_>) -> None );
  (fun (gr:FSA<_>) -> Some(NUMBER(gr)));
  (fun (gr:FSA<_>) -> Some(MINUS(gr)));
  (fun (gr:FSA<_>) -> Some(DIV(gr)));
  (fun (gr:FSA<_>) -> Some(PLUS(gr)));
  (fun (gr:FSA<_>) -> Some(POW(gr)));
  (fun (gr:FSA<_>) -> Some(MULT(gr)));
|]
```

Листинг 6: Массив действий для спецификации языка, указанной в листинге 5

C. Алгоритм лексического анализа

На этапе построения аппроксимации множества значений динамически формируемого строкового выраже-

ния происходит сохранение привязки: с каждым символом сохраняются координаты позиций этого символа в исходной строке, а также привязка этой строки к исходному коду. Результатом этого этапа является конечный автомат A, в котором входной алфавит состоит из элементов вида `('символ', привязка)`.

Над конечным автоматом A запускается процедура построения детерминированного конечного автомата. При этом конечный автомат детерминирован так, чтобы на дугах из одной вершины не было двух одинаковых символов с одинаковой привязкой. Из полученного конечного автомата $A' = \langle Q, \Sigma, \Delta', q_0, F\rangle$ строится конечный преобразователь $M = \langle Q, \Sigma, \Sigma', \Delta, q_0, F\rangle$ для лексического анализатора, в котором выходной алфавит $\Sigma' = \{s \mid (s, _) \in \Sigma\}$. То есть дуги графа, являющегося представлением конечного преобразователя M, помечены парами `(('символ', привязка) : 'символ')`. Отображение Δ отличается от отображения Δ' только тем, что появился выходной алфавит. Такое преобразование необходимо для выполнения операции композиции.

В конечный преобразователь M необходимо добавить переход по символу `'eof'` от всех конечных состояний в новое состояние, которое теперь будет являться конечным. Этот символ означает окончание строки и необходим для корректной работы лексического анализатора, так как вычисление действия (`action`) к накопленной строке происходит при чтении следующего символа.

На вход лексический анализатор принимает два конечных преобразователя, один из которых получен в результате построения аппроксимации, а второй — из описания, построенного генератором лексических анализаторов. Предлагаемый алгоритм для проведения лексического анализа динамически формируемого выражения состоит из двух этапов.

Этап 1. Выполнение операции композиции над двумя входными конечными преобразователями. Результатом этой операции является либо набор лексических ошибок, либо конечный преобразователь $M_1 = \langle Q_1, \Sigma_1, \Sigma'_1, \Delta_1, q_{01}, F_1\rangle$. Наличие лексических ошибок возможно в двух случаях: либо конечный преобразователь M содержит символы, которых нет в лексической спецификации, либо конечный преобразователь M порождает такой язык, который не принимает на вход лексический анализатор. Возможна также ситуация, когда конечный преобразователь M порождает язык, в котором есть слова, не принимаемые на вход лексическим анализатором. В этой ситуации набор лексических ошибок получается следующим образом: применяется операция разности к двум конечным автоматам, полученных из конечных преобразователей M и M_1 с помощью входной проекции соответствующих конечных преобразователей. После чего происходит лексический анализ подмножества языка, принимаемого лексическим анализатором.

Этап 2. Если конечный преобразователь M_1 получен, то происходит этап его интерпретации, результатом которой будет являться конечный автомат A_{token} над алфавитом токенов. В конечном преобразователе M_1 выделяются *action-вершины* — это вершины, из которых выходит хотя бы одна дуга, содержащая функцию, возвращающую Some(token). После action-вершины всегда возвращается один тип токена, но может выходить дуга, помеченная символом $(_, \varepsilon)$, что означает, что токен продолжает накапливаться. Пример такого конечного преобразователя представлен на рис. 1. Из одной action-вершины могут быть достижимы несколько других action-вершин. Количество достижимых вершин соответствует количеству токенов, которые нужно выделить в конечном преобразователе M_1 для данной action-вершины.

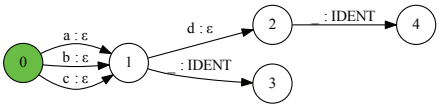


Рис. 1: Вершины 1 и 2 являются *action*-вершинами, но из вершины 1 токен продолжает накапливаться

Этот этап состоит из шагов, представленных ниже. В них используется структура `GraphAction` (см. листинг 7). Эта структура предназначена для выделения токенов в конечном преобразователе M_1 и содержит поля для стартовой вершины, набора конечных вершин и конечного автомата. Стартовой вершиной может быть только action-вершина или начальное состояние, конечной вершиной — action-вершина или конечное состояние.

```
struct GraphAction =
    int startV
    array<int> endV
    FSA fsa  //finite state automaton
```

Листинг 7: Структура GraphAction

В описании алгоритма для дуги конечного преобразователя M_1 используется нотация $e = (u, (a, b), v)$, где u — начальная вершина дуги, v — конечная вершина, (a, b) — метка дуги, $a \in \Sigma_1$, $y \in \Sigma'_1 \cup \{\varepsilon\}$. Добавление дуги в конечный автомат, который содержится в структуре `GraphAction`, осуществляется с помощью функции `graphAction.fsa.AddEdge(e)`, которая добавляет дугу из вершины u в вершину v, помеченную символом a в соответствующий конечный автомат. В алгоритме используются очередь Q для того, чтобы контролировать порядок обхода конечного преобразователя и структура `GraphAction` *grAct* для выделения конечного автомата, который содержит токен.

Шаг 1. Для конечного преобразователя M_1 стро-

ится набор вершин $actV$, которые являются action-вершинами.

Шаг 2. Для начального состояния конечного преобразователя M_1 и вершин из набора $actV$ запускается обход, который накапливает для токена конечный автомат. Порядок обхода представлен в алгоритме 1. Результатом обхода для всех вершин является коллекция $tokenAct$, состоящая из элементов типа `GraphAction`.

Алгоритм 1 Порядок обхода конечного преобразователя для сохранения привязки лексических единиц к исходному коду

```
function BFS(vStart, M1)
    grAct.startV ← vStart
    Q.Enqueue(vStart)
    while Q is not empty do
        v ← Q.Dequeue()
        if v is not visited then
            VISIT(v)
            for all e = (v, _, u) in M1 do
                if e = (v, (_, ε), u) then
                    c ← v is init state of M1
                    if v ≠ vStart or c then
                        grAct.fsa.AddEdge(e)
                        if u = vStart then
                            grAct.endV.Add(u)
                        Q.Enqueue(u)
                else
                    if v = vStart then
                        grAct.fsa.AddEdge(e)
                        if u is final state in M1 then
                            grAct.endV.Add(u)
                        else
                            Q.Enqueue(u)
                    else
                        grAct.endV.Add(v)
```

Шаг 3. Для точного определения конечного автомата, сохраняющего связь между токеном и исходным кодом, необходимо пройти конечный преобразователь M_1 в обратном направлении. Такая необходимость возникает в случае, если конечный преобразователь M_1 содержит циклы. На этом шаге также запускается обход для вершин из набора $actV$ и начального состояния конечного преобразователя M_2, который является инвертированным конечным преобразователем M_1. Порядок обхода представлен в алгоритме 2. Результатом обхода для всех вершин является коллекция $tokenActInv$, состоящая из элементов типа `GraphAction`.

Шаг 4. Ищутся пересечения конечных автоматов, полученных на двух предыдущих шагах, при условии, что $tokenAct.endV.current = tokenActInv.startV$. Токен определяется функцией, лежащей на дуге, исходящей из вершины $tokenAct.endV.current$, в него записывается результат пересечения. При этом в конечный автомат A_{token} добавляется переход по этому токену из вершины $tokenAct.startV$ в вершину $tokenAct.endV.current$.

Алгоритм 2 Порядок обхода инвертированного конечного преобразователя для сохранения привязки лексических единиц к исходному коду

```
function BFSInv(vStart, M2, M1)
    grAct.startV ← vStart
    Q.Enqueue(vStart)
    while Q is not empty do
        v ← Q.Dequeue()
        if v is not visited then
            VISIT(v)
            for all e = (v, _, u) in M2 do
                grAct.fsa.AddEdge(e)
                c ← u is action-vertex or init state of M2
                if c then
                    if ∃e = (u, (_, ε), _) in M1 then
                        Q.Enqueue(u)
                else
                    Q.Enqueue(u)
```

Так как символ `'eof'` является вспомогательным символом для лексического анализа, то для всех вершин, из которых выходит дуга, помеченная `'eof'`, добавляется переход в новое состояние по токену `EOF` в конечный автомат A_{token}, которое теперь является конечным состоянием A_{token}. Над полученным конечным автоматом запускается процедура построения детерминированного конечного автомата. При этом автомат детерминирован так, чтобы на дугах из одной вершины не было двух токенов с одинаковым идентификатором и конечным автоматом.

D. Пример

Рассмотрим работу алгоритма для лексического анализа на примере. Пусть результатом аппроксимации является конечный автомат A, представленный на рис. 2. Для этого примера сохранение привязки для символов опускаем.

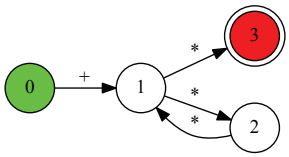


Рис. 2: Результат аппроксимации

Преобразуем этот конечный автомат во входную структуру для алгоритма. Для этого сперва строится детерминированный конечный автомат A' (см. рис. 3), затем — конечный преобразователь M (см. рис. 4).

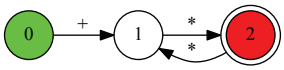


Рис. 3: Детерминированный конечный автомат A'

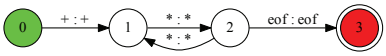


Рис. 4: Конечный преобразователь M для лексического анализа

Для обработки конечного преобразователя M используется спецификация для языка арифметических выражений (см. листинг 5). Результат композиции конечного преобразователя M с конечным преобразователем, построенным генератором лексических анализаторов, показан на рис. 5.

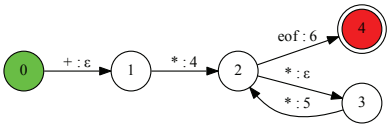


Рис. 5: Результат композиции

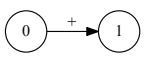
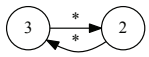
На рис. 5 цифры соответствуют индексу в массиве действий, который представлен в листинге 6 (нумерация элементов в массиве начинается с 0). Индекс 4 соответствует функции, которая возвращает токен `Some(PLUS)`, 5 — токен `Some(POW)`, 6 — токен `Some(MULT)`. Так как лексических ошибок получено не было, то происходит этап интерпретации полученного конечного преобразователя M_1, который состоит из 4 шагов. Ниже представлены результаты выполнения каждого шага.

Шаг 1. В набор $actV$ добавляются action-вершины 1, 2, 3.

Шаг 2. Запускается обход, представленный в алгоритме 1, для вершин из набора $actV$ и вершины 0, которая является начальным состоянием конечного преобразователя M_1. Результатом обхода для всех вершин является коллекция $tokenAct$, которая представлена в виде таблицы I.

Шаг 3. Запускается обход, представленный в алгоритме 2, для вершин из набора $actV$ и вершины 49, которая является начальным состоянием конечного преобразователя M_2. Результатом обхода для всех вершин является коллекция $tokenActInv$, которая представлена в виде таблицы II.

Шаг 4. Выполняются операции пересечения конечных автоматов, результаты которых представлен в таблице III.

startV	0	1
FSA		
endV	1	2,3

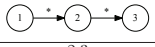
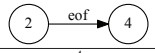
startV	2	3
FSA		
endV	2,3	4

Таблица I: Коллекция $tokenAct$ для Шага 2

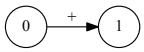
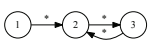
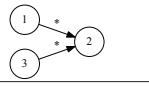
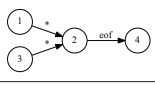
startV	1	3
FSA		
startV	2	4
FSA		

Таблица II: Коллекция $tokenActInv$ для Шага 3

Результатом лексического анализа будет конечный автомат, представленный на рис. 6. Каждая дуга графа содержит токен, который хранит в себе конечный автомат. Например, дуга от вершины 1 к вершине 3 содержит токен `POW`, у которого хранится конечный автомат с переходами от состояния 1 к состоянию 2 по символу '*', от 2 к 3 — по символу '*', полученный в результате пересечения конечных автоматов.

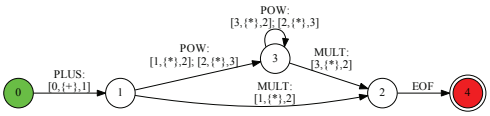


Рис. 6: Результат лексического анализа

IV. Архитектура модуля для лексического анализа

Архитектура инструмента, реализующая рассмотренный механизм, представлена на рис. 7.

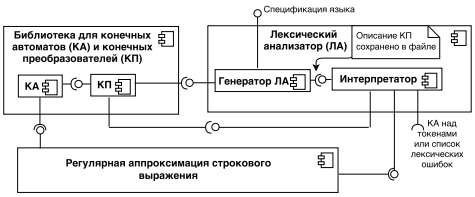


Рис. 7: Архитектура модуля для лексического анализа

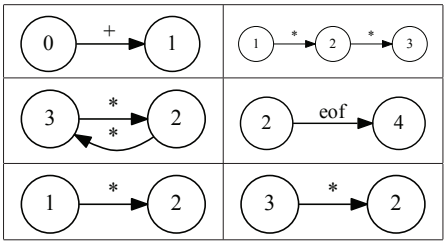


Таблица III: Результат пересечения конечных автоматов из таблиц I и II

Компонента **Лексический анализатор** состоит из **Генератора лексических анализаторов** и **Интерпретатора**. Генератор лексических анализаторов строит конечный преобразователь, описание которого взято из **Библиотеки для конечных автоматов и конечных преобразователей**, и сохраняет результат в отдельный файл. Такой подход позволяет многократно использовать данный конечный преобразователь для обработки кода, написанного на одном языке. **Интерпретатор** принимает на вход два конечных преобразователя, один из которых из которых получен в результате построения аппроксимации (за это отвечает компонент **Регулярная аппроксимация строкового выражения**), а второй построен генератором лексических анализаторов. Результатом работы **Интерпретатора** является либо конечный автомат над алфавитом токенов эталонной грамматики языка, либо список обнаруженных лексических ошибок.

Библиотека для конечных автоматов и конечных преобразователей предоставляет ряд операций, которые необходимы для построения аппроксимации и проведения лексического анализа. Для конечных автоматов используются операции: конкатенация, replace, дополнение, пересечение, а для конечных преобразователей — композиция.

V. Апробация

В данном разделе описан механизм использования реализованного инструмента, а также рассмотрены примеры, демонстрирующие разработанную функциональность.

Для осуществления лексического разбора необходимо выполнить следующие шаги.

Шаг 1. Запустить генератор, указав путь к файлу с расширением .fsl, в котором написана спецификация. В результате создается файл с расширением .fs, содержащий конечный преобразователь и вспомогательные функции.

Шаг 2. Необходимо в отдельном файле с расширением .fs указать описание типов токенов, которые автоматически строятся по грамматике эталонного языка. Полученные файлы подключить в модуль, предназначенный для получения результата лексического разбора.

Шаг 3. Получить конечный автомат, аппроксимирующий множество значений строкового выражения и удовлетворяющий описанию используемой библиотеки для конечных автоматов и конечных преобразователей. Преобразовать этот конечный автомат в конечный преобразователь (см. Алгоритм лексического анализа).

Шаг 4. Вызвать функцию `tokenize` из сгенерированного файла. Результатом этой функции будет либо конечный автомат над алфавитом токенов эталонной грамматики языка, либо список лексических ошибок.

Рассмотрим примеры, которые показывают преимущества реализованного решения, а именно, возможность сохранения конечного автомата внутри структуры токена и обработки циклов во входном конечном автомате.

Пример 1. Рассмотрим следующий пример кода:

```
private void Go(int number){
   String query =
        "SELECT nameX FROM tableY WHERE x < ";
   while(query.Length < number){
      query += "+ 1 ";
   }
   Program.ExecuteImmediate(query);
}
```

Листинг 8: Пример формирования выражения в цикле

Результатом аппроксимации выражения `query` является конечный автомат, представленный на рис. 8. Результат лексического анализа представлен на рис. 9. В результирующем конечном автомате учитываются две ситуации: цикл не выполняется (путь $0 \to \ldots \to 7 \to 9$) и выполняется (путь $0 \to \ldots \to 7 \to 8 \to 10 \to 8 \to \ldots \to 10 \to 9$), что позволяет уточнить проводимый анализ в целом.

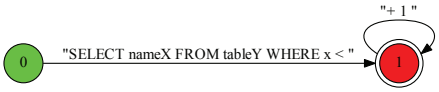


Рис. 8: Результат аппроксимации

Рис. 9: Результат лексического анализа

Пример 2. Рассмотрим следующий пример кода, в котором конечный автомат токена содержит цикл:

```
String query = "SELECT name";
for(int i = 0; i < 10; i++){
   query  += "X";
}
query += " FROM tableY";
Program.ExecuteImmediate(query);
```

Листинг 9: Пример кода, в котором конечный автомат токена содержит цикл

Результатом лексического анализа будет конечный автомат, представленный на рис. 10. Конечный автомат первого токена IDENT, содержащий цикл, представлен на рис. 11. На этом же рисунке показана сохраненная привязка символов к исходному коду. Таким образом, если цикл содержится только в конечном автомате токена, то в результирующем конечном автомате циклы отсутствуют, что позволяет упростить структуру входных данных для дальнейшего анализа.

Рис. 10: Результат лексического анализа

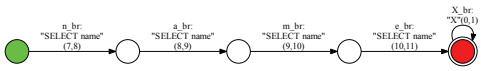


Рис. 11: Конечный автомат первого токена IDENT, содержащий цикл

На практике основным сценарием для динамически формируемого выражения является следующая ситуация, где строки x1, x2, . . . , xn возвращают одинаковый идентификатор токена:

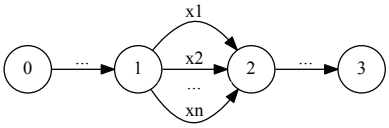


Рис. 12: Конечный автомат

Разработанный механизм лексического анализа вернет один токен при переходе от состояния 1 к состоянию 2, что значительно упрощает входные данные для синтаксического разбора.

VI. Заключение

В данной работе описан алгоритм лексического анализа и основанный на нем генератор лексических анализаторов для динамически формируемого кода. Данный алгоритм был реализован в проекте YaccConstructor на языке программирования F#. Разработанный алгоритм позволяет при проведении лексического анализа для токена сохранять конечный автомат, что значительно упрощает входные данные для синтаксического разбора. Также алгоритм позволяет хранить привязку лексических единиц к исходному коду, которая необходима для позиционирования места ошибок или навигации по коду.

Разработанный генератор лексических анализаторов позволяет получать по спецификации языка соответствующий анализатор, использующий описанный алгоритм. Таким образом, был разработан автоматизированный подход создания лексических анализаторов для динамически формируемого кода.

В дальнейшем планируется снять ограничения на формат написания спецификации, по которому генератор лексических анализаторов строит соответствующий анализатор, а также оптимизировать полученный инструмент за счет подбора структур данных и алгоритмов для работы с конечными автоматами [15].

Список литературы

[1] Сайт проекта fslex. [Online]. Available: http://fsprojects.github.io/FsLexYacc/

[2] I. Kirilenko, S. Grigorev, and D. Avdiukhin, “Syntax analyzers development in automated reengineering of informational system,” *St. Petersburg State Polytechnical University Journal. Computer Science. Telecommunications and Control Systems*, no. 174, pp. 94 – 98, 2013.

[3] A. S. Christensen, A. Møller, and M. I.Schwartzbach, “Precise analysis of string expressions,” in *Proceedings of the 10th International Conference on Static Analysis*. Berlin, Heidelberg: Springer-Verlag, Jun. 2003, pp. 1–18.

[4] Y. Minamide, “Static approximation of dynamically generated web pages,” in *Proceedings of the 14th International Conference on World Wide Web*. New York, NY, USA: ACM, 2005, pp. 432–441.

[5] N. Jovanovic, C. Kruegel, and E. Kirda, “Pixy: A static analysis tool for detecting web application vulnerabilities,” in *Security and Privacy, 2006 IEEE Symposium on*. Berkeley/Oakland, CA: IEEE, 2006, pp. 263–269.

[6] F. Yu, M. Alkhalaf, and T. Bultan, “Stranger: An automata-based string analysis tool for php,” in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2010, pp. 154–157.

[7] A. Dasgupta, V. Narasayya, and M. Syamala, “A static analysis framework for database applications,” in *Computer Software and Applications Conference*. IEEE, 2007, pp. 87–96.

[8] X. Fu and K. Qian, “Safeli: Sql injection scanner using symbolic execution,” in *Proceedings of the 2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications*. New York, NY, USA: ACM, 2008, pp. 34–39.

[9] A. Annamaa, A. Breslav, J. Kabanov, and V. Vene, “An interactive tool for analyzing embedded sql queries,” in *Programming Languages and Systems*. Springer Berlin Heidelberg, 2010, pp. 131–138.

[10] Репозиторий проекта yaccconstructor. [Online]. Available: https://github.com/YaccConstructor/YaccConstructor

[11] S. Grigorev, E. Verbitskaia, A. Ivanov, M. Polubelova, and E. Mavchun, “String-embedded language support in integrated development environment,” in *Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia*. ACM, 2014, pp. 21:1–21:11.

[12] Григорьев, Вербицкая, Полубелова, Иванов, and Мавчун, “Инструментальная поддержка встроенных языков в интегрированных средах разработки,” *Моделирование и*

анализ информационных систем, vol. 21, no. 6, pp. 131–143, 2014.

[13] F. Yu, M. Alkhalaf, T. Bultan, and O. H. Ibarra, "Automata-based symbolic string analysis for vulnerability detection," in *Formal Methods in System Design*. Springer US, 2014, pp. 44–70.

[14] T. Hanneforth. Finite-state machines: Theory and applications unweighted finite-state automata. Lecture Notes. [Online]. Available: http://tagh.de/tom/wp-content/uploads/fsm_unweigtedautomata.pdf

[15] P. Hooimeijer and M. Veanes, "An evaluation of automata algorithms for string analysis," in *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer-Verlag, 2011, pp. 248–262.

Применение технологии иерархических параметризируемых шаблонов для автоматизированного исправления ошибок в программном коде

Алексюк А.О.
Санкт-Петербургский
Политехнический Университет
Петра Великого
Email: artyom.h31@gmail.com

Ицыксон В.М.
Санкт-Петербургский
Политехнический Университет
Петра Великого
Email: vlad@icc.spbstu.ru

Аннотация—Статья посвящена автоматизированному исправлению ошибок в программном коде с использованием результатов статического анализа. Рассмотрены существующие подходы, применяемые в данной области, основные задачи и сложности, стоящие при создании инструментов автоматического исправления ошибок. Предложенный в статье способ описания исправлений на основе иерархических параметризируемых шаблонов позволяет находить и заменять участки кода, учитывая в процессе поиска то, что некоторые элементы программы, такие как идентификаторы, могут иметь произвольное значение в рамках заданных условий. Разработана архитектура системы, в которой применён описанный способ описания исправлений. Архитектура воплощена в прототипе программной системы, и подготовлен набор шаблонов, исправляющих сигнатурные ошибки, найденные статическим анализатором. Проведенное тестирование разработанного инструмента показало работоспособность подхода.

I. Введение

В настоящее время происходит постоянное увеличение сфер применения программного обеспечения. Такие важные области жизни человека, как медицина, авиация, финансовые структуры давно используют программы и вычислительные системы для решения своих задач. Нельзя не отметить тенденцию к широкому распространению «умных» потребительских устройств в рамках концепции «Internet of Things», в результате чего многие окружающие нас устройства получают вычислительные модули со сложным набором ПО внутри. С учётом стремления к снижению затрат на разработку программных систем для повышения конкурентноспособности и одновременным усложнением и повышением ответственности решаемых задач, становится всё актуальнее необходимость в поддержании и контроле качества ПО.

Важную роль в этом играет статический анализ программного кода. С помощью этого подхода можно обнаружить широкий спектр нефункциональных дефектов в ПО и тем самым повысить такие характеристики качества, как надёжность, эффективность, сопровождаемость и мобильность.

Одним из основных достоинств статического анализа является, при определённых условиях, возможность полностью автоматической работы, поэтому основной способ использования инструментов этого типа - регулярный запуск, например, каждую ночь, или по событиям, например, после выгрузки фиксации ("коммита") в систему контроля версий. Результаты анализа сохраняются и просматриваются человеком впоследствии. Кроме того, процедуру статического анализа можно задействовать и однократно, например, для проверки унаследованного кода или кода библиотек программного проекта.

Достаточно большое число дефектов, выявляемых статическими анализаторами, являются типовыми и могут быть исправлены автоматически. Как правило, это ошибки, связанные с не всегда очевидными особенностями языка, опечатками.

Целью данной работы является разработка системы автоматической модификации программного кода, использующей результаты работы одного из средств статического анализа.

Статья состоит из четырех разделов. Первый раздел содержит анализ подходов и инструментов для автоматической модификации программного кода, их основные особенности. Во втором разделе производится анализ ограничений, присущих автоматической модификации программ. В третьем разделе описываются архитектурные решения, принятые в процессе разработки системы. В четвертом разделе производится тестирование разработанной системы.

II. Обзор аналогов

В данном разделе рассматриваются существующие инструменты автоматической модификации кода. Про-

ведена классификация, рассмотрены особенности каждого класса инструментов.

Инструменты различаются по сфере применения - универсальные системы и узкоспециализированные, созданные для исправления одного типа ошибок. В качестве примера узкоспециализированных инструментов можно привести системы для устранения ошибок, связанных с многопоточностью. В ходе их работы в программу вводится дополнительная синхронизация с помощью соответствующих примитивов. Процедура исправления осложняется тем, что при неправильном изменении кода могут возникнуть новые ошибки, такие как взаимные блокировки (deadlocks).

В качестве примера подобных инструментов можно привести Grail[1], AFix[2] и Axis[3]. Далее в статье рассматриваются только универсальные системы.

Проведём классификацию по характеру используемой информации - статические инструменты, использующие только исходный код, и динамические, которые дополнительно используют информацию о ходе выполнения программы.

Как правило, инструменты первого типа являются модулями для интегрированных сред разработки (IDE), такой подход позволяет переиспользовать существующие системы модификации кода. Подобные системы имеются в таких IDE, как Eclipse и IntelliJ IDEA.

В интегрированной среде разработки IntelliJ IDEA имеется встроенный статический анализатор[4], который непрерывно анализирует код и при наличии предупреждений сообщает об этом пользователю (подсвечивая соответствующий участок кода жёлтым цветом). При этом пользователь может вызвать меню «Quick Fix», которое содержит список предлагаемых исправлений. На рисунке 1 изображён пример использования этих возможностей IDEA.

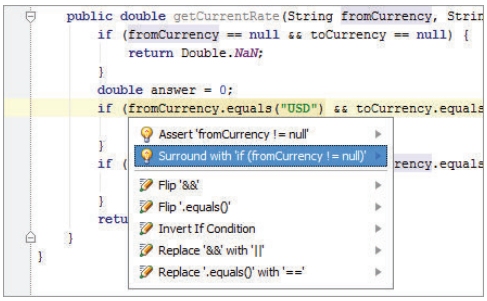


Рис. 1. Предупреждение от статического анализатора в IntelliJ IDEA и всплывающее меню Quick Fix

Довольно большое количество исправлений изначально содержится в IDEA, однако имеется возможность добавить свои шаблоны с помощью функции Structural Search and Replace[5]. Данная функция во многом похожа на обычный диалог замены текста (Search and Replace), однако учитывает синтаксис и семантику кода. Например, при поиске вхождения не учитывается разница в форматировании между шаблоном поиска и исходным кодом, также не учитывается порядок объявления членов при поиске класса. Если нет необходимости искать точное совпадение, в шаблоне можно использовать переменные.

Из-за ограниченного количества используемых источников информации, инструменты данного класса позволяют легко добавлять новые исправления с помощью механизма шаблонов и проще в использовании, однако для них высока вероятность внесения некорректного исправления в код.

Примером инструментов того же класса, но без использования шаблонов, является проект GenProg[6], [7], использующий генетические алгоритмы. В случае GenProg функцией приспособленности является количество пройденных тестов. Генотипом является набор изменений для абстрактного синтаксического дерева программы, содержащей ошибку. Имеются следующие варианты мутаций: удаление поддерева из АСД, обмен местами нескольких поддеревьев, добавление нового поддерева, аналогичного одному из уже присутствующих в программе. Кроме мутаций, возможно скрещивание нескольких генотипов, в таком случае комбинируется несколько наборов изменений.

У данного подхода есть несколько недостатков:

- Необходимость большого тестового покрытия, как положительными, так и отрицательными тестами. Авторы статьи отмечают, что при недостаточном количестве положительных тестов имеет место удаление части функциональности создаваемой заплаткой («патчем»). Кроме того, очевидно, что при недостатке отрицательных тестов GenProg становится сложнее понять, какие именно изменения приближают процесс исправления к конечной цели. В итоге направленный поиск (с учетом функции приспособленности) превращается в обычный перебор.
- Так как все добавляемые в АСД поддеревья берутся из другой части программы, не все ошибки могут быть исправлены. Например, если для исправления дефекта необходимо добавить проверку в определённое место программы, в другой части кода уже должна быть подобная проверка.
- Для исправления ошибок требуется достаточно много времени. В статье указано, что на исправление ошибок в наборе приложений среднего размера (lighttpd, flex и т.п.) потребовалось около 6 минут, а при попытке запуска на наборе из больших приложений (Wireshark, интерпретатор PHP) за 12 часов была исправлена только половина дефектов.

В качестве примера противоположного подхода, при котором вносятся изменения в работающую програм-

му, можно назвать такие инструменты, как Juzi[8] и AutoFix-E[9]. Первый относится к инструментам, корректирующим структуры данных, а второй - к инструментам, вносящим исправления в логику работы программы. Принцип действия у них схож - по мере работы программы вычисляется разница между теми состояниями данных, которые приводят к сбою, и корректными состояниями. При обнаружении состояния, которое приводит к сбою, оно либо непосредственно исправляется в соответствии с вычисленной разницей между состояниями (Juzi), либо в программу добавляются вызовы существующего кода так, чтобы состояние было исправлено (AutoFix-E).

Недостатком рассмотренного выше подхода является тот факт, что он сложен в реализации и требует, чтобы ошибки преобразовывались в сбои, например, с помощью контрактов.

III. Проблемы автоматической модификации кода

В этом разделе рассмотрены основные условия и задачи, возникающие при проектировании системы автоматической модификации кода.

Одним из важнейших условий, влияющих на формирование требований и выбор решений при проектировании систем автоматической модификации кода, является строго ограниченная степень допустимого изменения поведения программы. Другими словами, так как предполагается, что инструмент будет запускаться регулярно и полностью автоматически, разработчик может ожидать, что исправная логика работы кода не будет искажена инструментом.

Для начала стоит сказать, что инструменты статического анализа кода имеют ненулевую вероятность ошибок первого и второго рода, то есть могут пометить абсолютно корректный код как ошибочный или наоборот, пропустить дефект в коде. Для этого есть множество причин: отсутствие формальной спецификации программного проекта на входе, упрощения и допущения в логике анализа, различного рода оптимизации, направленные на уменьшение потребления памяти анализатором и ускорение его работы.

Наиболее опасны именно ошибки первого рода, так как они могут привести к внесению бесполезных или даже нарушающих работу программы изменений в процессе исправления несуществующего дефекта. Для борьбы с такого рода ошибками в статических анализаторах имеется возможность запретить выдачу предупреждений для некоторого участка кода. Поэтому, так как функция фильтрации ошибок уже реализована в анализаторе, система модификации кода может считать, что все сообщения от анализатора являются корректными, а все найденные ошибки должны быть исправлены.

Рассмотрим следующую проблему: некорректное исправление реально существующей ошибки. Несмотря на то, что исправление ошибок не является рефакторингом, в данном аспекте у этих двух операций возникают во многом схожие трудности. Нужно отметить, что даже ручное исправление дефекта разработчиком не гарантирует того, что ошибка действительно устранена, логика работы кода не исказилась и изменения не затронули другие части проекта. Например, разработчик может иметь недостаточно знаний о проекте. Для преодоления таких проблем можно использовать просмотр кода (code review) и/или тестирование.

Таким образом, система автоматической модификации не может и не должна гарантировать полную корректность преобразований, а для контроля его работы необходимо использовать дополнительные практики и инструменты. Однако система должна быть спроектирована так, чтобы частота появления подобных проблем была минимальной.

IV. Разработанное средство модификации ПО

Структурно разрабатываемую систему можно разделить на три части:

1) Взаимодействие со статическим анализатором, получение списка предупреждений
2) Модуль для модификации кода
3) Набор исправлений

В качестве статического анализатора был выбран FindBugs. Так как предполагается полностью автоматическое использование разрабатываемой системы, модуль взаимодействия с анализатором не должен быть интерактивным и всю информацию должен получать через аргументы командной строки или из файла настроек.

Модуль модификации кода должен удовлетворять следующим требованиям:

- Универсальность. Применяемый в модуле подход должен позволять вносить максимальное количество видов исправлений. Минимальное требование: подход в том или ином виде должен поддерживать переменные в шаблонах, т.е. не зависеть от названия классов, методов, переменных программы, расположения искомого участка в файле с исходным кодом и так далее.
- Корректность преобразований. Модуль не должен каким-либо образом изменять логику обрабатываемого кода, если этого не требует шаблон модификации. Так как провести формальное доказательство эквивалентности исходной и исправленной программы для всех возможных замен на практике весьма сложно, можно считать, что инструмент осуществляет корректное преобразование, если он протестирован на нескольких известных, находящихся в промышленной эксплуатации проектах.
- Сохранение форматирования кода. Разрабатываемый инструмент должен сохранять отступы и комментарии в модифицируемом коде.

- Полнота поддержки языка программирования. Модуль должен использовать наиболее полную грамматику языка и уметь работать с максимальным количеством конструкций. Инструменту следует поддерживать последние версии языка программирования.
- Расширяемость. Используемый подход должен обеспечивать возможность простого добавления новых модификаций, без существенного изменения уже написанного кода инструмента. Наиболее подходящий вариант, если для добавления новых модификаций не требуется изменять или перекомпилировать код вообще.

В качестве основы для модуля модификации кода был выбрана технология модификации программного кода, основанная на параметризуемых шаблонах[10]. Структура проекта изображена на рисунке 2.

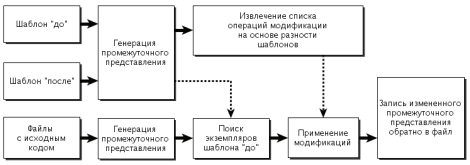


Рис. 2. Структура проекта TemplateRefactorings

Основное достоинство проекта TemplateRefactorings - высокая расширяемость. Модификации к коду описываются в виде двух шаблонов: шаблон «до» и шаблон «после». Для описания шаблонов используется незначительно модифицированный язык Java, нет необходимости изучать какие-либо другие языки или библиотеки, что значительно снижает порог вхождения для пользователей. Для добавления нового шаблона не нужно каким-либо образом изменять код инструмента. Кроме того, за счёт использования параметров («селекторов») в шаблонах достигается высокая универсальность.

В рамках разработки системы автоматической модификации кода необходимо было выполнить следующие шаги:

1) Организовать взаимодействие с FindBugs
2) Улучшить алгоритм нахождения шаблонов в исходном коде
3) Обновить библиотеку ANTLR до версии 4
   a) Адаптировать грамматику паттернов для новой версии
   b) Использовать другой способ формирования изменённого исходного кода
   c) Модифицировать код запуска парсера
4) Составить набор шаблонов-исправлений

Исходные коды разработанной системы доступны по адресу [11].

A. Организация взаимодействия с FindBugs

В результате своей работы утилита FindBugs формирует два файла: HTML, предназначенный для просмотра пользователем, и XML для передачи информации другим приложениям. XML-файл содержит следующие сведения о результатах анализа:

- Тип ошибки
- Подробное описание ошибки (для отображения пользователю)
- Путь к файлу с исходным кодом
- Номер строки, в которой была обнаружена ошибка

Таким образом, разрабатываемая система может получить все необходимые для своей работы параметры из отчёта FindBugs. Возможен автоматический запуск системы сразу после завершения работы анализатора, без участия человека.

Нужно отметить, что в одном файле с исходным кодом могут содержаться несколько ошибок одного типа. Для этого в отчёте указывается номер строки, в которой обнаружена ошибка. Однако TemplateRefactorings в исходном варианте заменяет все вхождения шаблона, поэтому было необходимо добавить фильтрацию по строкам.

Будем считать, что ошибка исправлена, если было найдено такое вхождение шаблона в исходном коде, которое затрагивает указанную строку. При этом может быть несколько вхождений, соответствующих этому критерию, но только одно из них вносит необходимое изменение, а не изменяет какой-либо другой код, который не относится к ошибке, но подходит под тот же шаблон.

Для простоты будем считать, что первое вхождение соответствует ошибочному участку кода, а последующие - изменяют не относящийся к ошибке участок кода. Очевидно, данное допущение не всегда верно. Чтобы точнее определять, какое именно изменение нужно применять, необходимо иметь больше информации, а именно:

1) Позицию в строке (столбец), где была найдена ошибка. FindBugs не сообщает подобную информацию.
2) Семантическую информацию, например, имена переменных, используемые в ошибочном выражении. Эти сведения зависят от типа ошибки, поэтому использование таких сведений уменьшает универсальность решения.

B. Улучшение алгоритма нахождения шаблонов в исходном коде

Рассмотрим процедуру нахождения вхождений дерева-шаблона. Эта задача является классической и обычно именуется как Pattern Matching in Trees. Существует достаточно большое количество алгоритмов для решения этого вопроса, некоторые из них рассмотрены в [12].

При разработке системы автоматической модификации кода решается несколько модифицированный вариант этой задачи, где некоторые узлы дерева могут быть селекторами. Задача усложняется тем, что селектору могут соответствовать произвольное число узлов исходного дерева, в том числе и нулевое (здесь и далее под деревом понимается АСД в случае оригинального проекта TemplateRefactorings и дерево разбора в случае системы автоматической модификации кода).

Рассмотрим способ решения, используемый в проекте TemplateRefactorings:

1) В дереве модифицируемого файла найти такой узел, который бы имел такой же тип и значение, как и корень шаблона. Далее будет рассматриваться поддерево, у которого корень - найденный узел.
2) Попытаться сопоставить узлы из найденного поддерева программы и узлы дерева шаблона.
3) Провести проверку того, что узлы сопоставлены верно: каждый узел дерева шаблона, не являющийся селектором, должен иметь один и только один соответствующий ему узел из найденного поддерева.
4) Если проверка пройдена успешно, поиск закончен, иначе - перейти к п. 1 и искать другие поддеревья.

Сопоставление происходит пошагово, за один шаг рассматриваются узлы только на одном уровне. Подробнее алгоритм рассмотрен в [10].

Однако не во всех ситуациях данный подход работает корректно. В ходе тестирования был найден пример шаблона, при котором данному способу не удавалось найти сопоставление, хотя оно существовало. На рисунке 3 показаны деревья, для которых приведённый подход работает некорректно. Слева изображено дерево исходного файла, а справа - дерево шаблона, серым цветом обозначены узлы-селекторы.

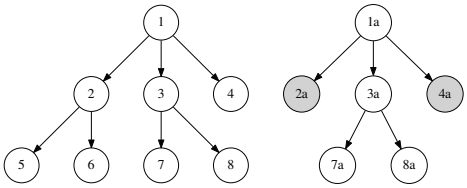


Рис. 3. Пример деревьев, для которых приведённый подход работает некорректно

Будем считать, что тип и значение узлов 1 и 1a совпадает, поэтому переходим сразу к пункту 2 из вышеприведённого плана. Очевидно, узлы 2a, 3a и 4a можно сопоставить узлам 2, 3 и 4 соответственно. Однако возможен и другой вариант - узел 2a не имеет соответствующих ему узлов, узел 3a сопоставляется с узлом 2, а к узлу 4a ставятся в соответствие узлы 3 и 4.

Однозначно сделать выбор нельзя, не рассмотрев тип и значения узлов 5, 6, 7, 8, 7a и 8a. Этому мешает тот факт, что подход предусматривает рассмотрение только одного уровня дерева на каждом шаге. При этом если сделан неправильный выбор, то не удаётся правильно создать соответствие для узлов 7a и 8a, в результате чего проверка на следующем этапе не проходит, и начинается поиск следующего поддерева. В данном примере подходящих поддеревьев нет, поэтому будет сделан вывод о том, что вхождений шаблона не существует для данного файла.

Одно из решений данной проблемы - интегрировать проверку в процесс сопоставления. Таким образом, если возникает несколько вариантов назначения соответствия, для каждого варианта запускается процедура сопоставления для дочерних узлов, производится проверка, и выбирается тот вариант, который прошёл проверку.

C. Обновление ANTLR, модификация грамматики языка Java

Для генерации лексического и статического анализатора в проекте используется утилита ANTLR[13][14]. Мажорная третья версия ANTLR была выпущена в 2007 году и на момент написания TemplateRefactorings была самой новой версией ANTLR. Версия 4 была выпущена в 2013 году как продолжение развития ANTLRv3. Новая версия несовместима со старой, как с точки зрения API, так и с точки зрения формата грамматики. Основное отличие между ними в том, что третья версия на выходе даёт АСД, а четвертая - CST.

Грамматика паттернов в TemplateRefactoring написана для ANTLRv3 и, в силу несовместимости, не может быть использована в ANTLRv4. Кроме того, исходная грамматика поддерживает только устаревшую пятую версию языка Java (JRE и JDK с поддержкой этой версии языка были выпущены в 2004 году).

Так как оригинальная грамматика была лишь незначительно изменена, было решено взять за основу грамматику для новой версии языка Java и внести в неё аналогичные модификации.

За основу была взята грамматика для Java 7 из официального репозитория грамматик ANTLRv4 [15]. На данный момент выпущена и более новая версия языка, 8, однако её грамматика значительно усложнена (1457 строк без комментариев по сравнению с 842 строками для Java 7). Для анализа того, имеет ли смысл поддерживать более новую версию языка, воспользуемся тем фактом, что для запуска программ на новом языке нужна версия JRE не ниже восьмой. Статистика используемых версий JRE[16] за 2015 год показывает, что только на 20,84% опрошенных машин используется JRE 8. Таким образом, широко используемые проекты

ещё не имеют возможности перейти на новую версию языка.

В процессе модификации важно сохранить возможность использовать одну и ту же грамматику, как для исходного кода, так и для шаблонов.

Начальное правило в исходной грамматике - compilationUnit, описывающее весь файл с Java-кодом. Это не очень удобно для описания исправлений, так как почти всегда шаблон замены описывает только один блок или оператор. Поэтому для удобства описания шаблонов добавим новое начальное правило javaStart:

```
1  javaStart
2  :    compilationUnit EOF
3  |    block EOF
4  |    blockStatement EOF
5  |    memberDeclaration EOF
6  |    expression EOF
7  ;
```

Таким образом, для файлов с модифицируемым исходным кодом разбор оставлен вариант с compilationUnit, а для файлов шаблонов будут использоваться остальные варианты.

Добавим правило для селекторов. Как и в оригинальной грамматике проекта TemplateRefactorings, селекторы начинаются с символа #:

```
1  selector
2  :    '#' Identifier
3  ;
```

В правило expression (выражение языка программирования) добавим вариант с селектором:

```
1  expression
2  :    primary
3  |    expression '.' Identifier
4  |    expression '.' 'this'
5  |    expression '.' 'new' nonWildcardTypeArguments
       ?
6   innerCreator
7  |    expression '.' 'super' superSuffix
8  |    expression '.' explicitGenericInvocation
9  |    expression '[' expression ']'
10 |    expression '(' expressionList? ')'
11 |    'new' creator
12 |    '(' type ')' expression
13 |    expression ('++' | '--')
14 |    ('+'|'-'|'++'|'--') expression
15 |    ('~'|'!') expression
16 |    expression ('*'|'/'|'%') expression
17 |    expression ('+'|'-') expression
18 |    expression ('<' '<' | '>' '>' '>' | '>' '>')
19  expression
20 |    expression ('<=' | '>=' | '>' | '<')
       expression
21 |    expression 'instanceof' type
22 |    expression ('==' | '!=') expression
23 |    expression '&' expression
24 |    expression '^' expression
25 |    expression '|' expression
26 |    expression '&&' expression
27 |    expression '||' expression
28 |    expression '?' expression ':' expression
29 |    selector
30 |   <assoc=right> expression
31 (    '='
32 |    '+='
33 |    '-='
34 |    '*='
35 |    '/='
36 |    '&='
37 |    '|='
38 |    '^='
39 |    '>>='
40 |    '>>>='
41 |    '<<='
42 |    '%='
43 )
44 expression
45 ;
```

В правило для блока добавлена поддержку селекторов:

```
1  block
2  :    '{' (selectorStatement | blockStatement)* '}'
3  ;
4
5  selectorStatement
6  :    selector ';'
7  ;
```

Добавим возможность использования селекторов в объявлениях переменных:

```
1  localVariableDeclaration
2  :    (selector|variableModifier* type)
       variableDeclarators
3
4  variableDeclarator
5  :    (selector | variableDeclaratorId) ('='
       variableInitializer)?
6  ;
```

Изменим правило для комментариев и пробелов. В исходной грамматике они отбрасывались ещё на этапе лексического анализа. Однако для того, чтобы в сформированном исходном коде сохранить форматирование и комментарии, необходимо каким-либо образом запомнить соответствующие лексемы. Если назначить токенам метку скрытого канала, то они не будут обрабатываться парсером, но останутся в потоке токенов:

```
1  WS
2  :    [ \t]+ -> channel(HIDDEN)
3  ;
4
5  BR
6  :    [\r\n\u000C]+ -> channel(HIDDEN)
7  ;
8
9  COMMENT
10 :    '/*' .*? '*/' -> channel(HIDDEN)
11 ;
12
13 LINE_COMMENT
14 :    '//' ~[\r\n]* -> channel(HIDDEN)
15 ;
```

В связи с переходом от AST к CST необходимо модифицировать способ превращения дерева обратно в код. В исходном варианте TemplateRefactorings АСД преобразуется в DOM (Document Object Model), при этом для каждого типа элементов DOM был написан метод для превращения этого элемента в строку, после чего данный метод вызывался для корня.

В случае же CST формирование кода значительно проще, так как в дереве сохраняются все терминалы. Если обойти дерево вглубь и подряд записать в строку все терминалы, то в результате подобной процедуры можно получить исходный код программы. Можно отметить, что изменение дерева в процессе исправления

ошибки не влияет на работоспособность этого подхода. Кроме того, при этом автоматически переносятся комментарии и оформление. Недостаток данного подхода - сложность расстановки пробелов в участках кода, где была произведена модификация кода, из-за чего в этих местах форматирование может быть некорректным.

D. Составление набора шаблонов-исправлений

Проект должен содержать исправления для следующих типов ошибок:

1) Отсутствие явного указания кодировки по умолчанию при чтении текстовых файлов
2) Сравнение строк с помощью оператора ==
3) Отсутствие проверки на null в методе equal()
4) Отсутствие проверки типа аргумента в методе equal()
5) Хранение в коде абсолютного пути к файлу
6) Использование конструкторов для классов-оболочек
7) Использование приближенного значение математических констант
8) Использование метода removeAll() для очистки коллекции
9) Завершение всей JVM с помощью вызова System.exit() при обработке нештатных ситуаций
10) Метод toString() возвращает null
11) Метод clone() возвращает null
12) Отсутствие проверки значения, возвращаемого методом read()
13) Сравнение массивов с помощью метода equal()
14) 32-разрядная арифметика для вычисления epoch time
15) Сравнение возвращаемого методом compareTo() значения на равенство с константой
16) Нулевой индекс при обращении к SQL PreparedStatement
17) Получение объекта Thread только для вызова статического метода isInterrupted()
18) Вызов метода toString() для массива
19) Смена регистра без указания языка
20) Вызов метода toString() для строки

Каждое исправление хранится в отдельной поддиректории. Пример структуры каталога показан на рисунке 4. Для каждого исправления должно быть создано как минимум три файла:

- before.java - шаблон "до"
- after.java - шаблон "после"
- desc.txt - текстовый файл, содержащий все типы ошибок, которые могут быть исправлены данной парой шаблонов, каждый тип - на отдельной строке

Перечисленный ранее набор исправлений был составлен из следующих соображений. Во-первых, для того, чтобы исправить ошибку, необходимо, чтобы анализатор смог найти её, поэтому набор исправляемых

```
fixes
├── equal_sign
│   ├── after.java
│   ├── before.java
│   └── desc.txt
└── hardcoded_path
    ├── after.java
    ├── before.java
    └── desc.txt
```

Рис. 4. Структура каталога с исправлениями

ошибок ограничен множеством ошибок, которые умеет обнаруживать анализатор. Список поддерживаемых статическим анализатором FindBugs ошибок опубликован на официальном сайте проекта[17].

Во-вторых, из этого набора были исключены те шаблоны замены, которые в процессе тестирования вызывали некорректные исправления. Как было обозначено в разделах 3 и 4, это не гарантирует, что оставшиеся шаблоны всегда будут корректно исправлять ошибки. Если в обрабатываемом проекте какие-либо шаблоны непозволительно часто вызывают проблемы, их можно удалить из набора или отключить поиск соответствующих ошибок в анализаторе.

Можно отметить, что дефекты и шаблоны имеют связь «многие-ко-многим», то есть один шаблон может исправлять произвольное количество дефектов, а один дефект может соответствовать нескольким шаблонам.

В качестве примера будет рассмотрена одна из самых известных ошибок [18, p. 445], которую допускают при написании кода на языке Java: сравнение строк с помощью оператора == (ES_COMPARING_PARAMETER_STRING_WITH_EQ).

```
1 String foo1 = "foo";
2 String foo2 = String.valueOf( new char [] { 'f',
      'o', 'o' } );
3 foo1 == foo2 // false!
4 foo1.equals( foo2 ) // true
```

Шаблон «до» в таком случае выглядит следующим образом:

```
1 #a == #b;
```

Шаблон «после»:

```
1 #b.equals(#a);
```

Аргументы были сознательно переставлены местами, чтобы не возникало исключения NullPointerException при b == null.

Третий файл содержит тип исправляемой ошибки - ES_COMPARING_PARAMETER_STRING_WITH_EQ.

Рассмотрим более сложный пример - предупреждение DMI_HARDCODED_ABSOLUTE_FILENAME. Оно возникает, когда в коде хранится абсолютный

путь к какому-либо файлу. Формально это не является ошибкой и никак не мешает нормальному функционированию программного проекта, однако при запуске такого кода на другой машине скорее всего возникнут проблемы. Таким образом, ухудшаются такие характеристики качества ПО как сопровождаемость и мобильность.

Наиболее корректный способ исправления этого дефекта - либо создание файла настроек, который бы содержал все используемые пути, либо замена абсолютного пути на относительный. Первый вариант требует модификации структуры проекта, а для второго варианта необходимо знать базовую директории, относительно которой можно было бы рассчитать относительный путь. Таким образом, ни один из этих способов не может быть легко автоматизирован, поэтому в качестве исправления вынесем путь в строковую константу.

Пример кода, который создаёт такую проблему - `File file = new File("/path/to/file");`. В таком случае исправление будет выглядеть следующим образом:

```
1 final String hardcoded_filename_0 = "/path/to/
      file";
2 File file = new File(hardcoded_filename_0);
```

Сложность примера в том, шаблон «до» содержит один операнд, а шаблон «после» - два. При синтаксическом разборе файлов шаблонов создаётся дерево, а у дерева не может быть два корня (соответствующие двум операторам). Операторы можно объединить в блок, но область видимости переменной будет ограничена созданным блоком.

По этой причине формируемые операторы необходимо добавить в тот же блок, в котором находился исходный оператор. Сначала рассмотрим случай, когда переменная уже объявлена. Ниже приведён шаблон «до» для такого случая:

```
1 {
2 #expr1;
3 #type #file = new File(#expr);
4 #expr2;
5 }
```

Шаблон «после»:

```
1 {
2 #expr1;
3 final String hardcoded_filename_0 = #expr;
4 #type #file = new File(hardcoded_filename_0);
5 #expr2;
6 }
```

Аналогично для случая, когда переменная уже объявлена.

Также можно добавлять новые операторы в метод. Рассмотрим ошибку, когда метод equal() не проверяет, что сравниваемый объект не является null. Шаблон «до»:

```
1 boolean equals(Object obj) {
2 #expr;
3 }
```

Шаблон «после»:

```
1 boolean equals(Object obj) {
2   if (obj == null) {
3     return false;
4   }
5   #expr;
6 }
```

V. Применение средства на реальных проектах

Для проверки того, что разработанная система корректно функционирует, попробуем исправить ошибки в одном из известных, находящихся в промышленной эксплуатации проектов.

В качестве такого проекта была выбрана библиотека JGraphT[19]. Основные мотивы её выбора следующие:

- Проект активно развивается, последний коммит был создан 29 дней назад (на момент написания этого раздела)
- Использует Java 7, поэтому можно протестировать, как модуль модификации кода работает с новыми возможностями языка
- Имеет достаточно большое количество тестов - 439 штук
- Имеет минимальное число зависимостей - только JUnit и XMLUnit для тестирования, что упрощает сборку
- Проект среднего размера - 27 тысяч строк

При тестировании использовалась ревизия ea16483 из Git-репозитория проекта. В результате статического анализа с помощью FindBugs было найдено 46 потенциальных ошибок, из них 14 могут быть исправлены автоматически с помощью разработанных шаблонов. 8 ошибок не могут быть исправлены из-за того, что FindBugs сообщает для них неправильный номер строк (анализатор работает на уровне байт-кода, поэтому не всегда корректно делает отображение в исходный код). Остальные ошибки пока что не могут быть исправлены, так как для них ещё не написаны шаблоны замены. Отчёт анализа был сохранен в XML-файл, после чего был запущен разработанный инструмент автоматической модификации кода.

При таком сценарии проверялись следующие части проекта и их функции:

- Модуль взаимодействия со статическим анализатором FindBugs - корректный разбор XML, получение из него путей к файлам, номеров строк и типов ошибок
- Модуль модификации кода - корректность грамматики языка, алгоритмов нахождения разницы между деревьями, поиск вхождений дерева разбора шаблона в дерево разбора исходного файла, превращение CST обратно в форму текста
- Набор исправлений

После завершения работы системы необходимо было проконтролировать, что модификация прошла успешно и корректно. Для этого:

- Было проверено, что изменения были внесены только в нужные файлы. Так как проект находился под управлением системы контроля версий git, для этого была использована команда git status.
- Было прослежено, что изменения внесены в соответствии с шаблонами. Аналогично, для этого с помощью команды git diff были вручную просмотрены изменения в файлах.
- Для контроля того, что изменения синтаксически корректны, проект был заново собран.
- Для проверки того, что исправления не затронули логику проекта, были запущены все имеющиеся в проекте тесты. Ни один из них не провалился.
- FindBugs был запущен повторно и сообщил о 32 дефектах, что соответствует числу неисправленных ошибок.

Приведем примеры исправлений. Первый случай - неэффективное создание объекта-обертки:

```
1  buckets.get(degree[nb]).remove(new Integer(nb));
```

Ниже приведена замена:

```
1  buckets.get(degree[nb]).remove( Integer.valueOf(
       nb));
```

Второй пример - отсутствие проверки на null и тип аргумента в методе equal. Исходный вариант:

```
1  @Override public boolean equals(Object obj)
2  {
3      LabelsEdge otherEdge = (LabelsEdge) obj;
4      if ((this.source == otherEdge.source)
5          && (this.target == otherEdge.target))
6      {
7          return true;
8      } else {
9          return false;
10     }
11 }
```

Замена:

```
1  @Override public boolean equals(Object obj)
2  {
3  if (obj == null) {
4      return false;
5  }
6  if (!obj.getClass().isInstance(this)) {
7      return false;
8  }
9      LabelsEdge otherEdge = (LabelsEdge) obj;
10     if ((this.source == otherEdge.source)
11         && (this.target == otherEdge.target))
12     {
13         return true;
14     } else {
15         return false;
16     }
17 }
```

VI. Заключение

Существующие системы модификации кода являются автоматизированными и поэтому требуют ручного вмешательства, что ограничивает область применения подобных инструментов. Кроме того, для добавления новых типов исправлений зачастую необходимо модифицировать код инструмента.

По результатам анализа была спроектирована и разработана полностью автоматическая система, имеющая высокую степень расширяемости: шаблоны замены хранятся отдельно от кода инструмента, для их описания используется лишь немного модифицированный язык Java, что снижает порог вхождения для пользователей инструмента и упрощает добавление новых исправлений. Язык шаблонов был дополнен новыми возможностями и теперь базируется на последних версиях языка Java. В качестве статического анализатора был выбран FindBugs, а в систему был добавлен модуль взаимодействия с ним. Из всего списка обнаруживаемых ошибок были выбраны те, которые с минимальной вероятностью искажают логику модифицируемого ПО, и для них был разработан набор шаблонов-исправлений.

Основным применением разработанного инструмента является регулярный запуск в рамках практики непрерывной интеграции для поддержания качества разрабатываемого ПО. Кроме того, система автоматической модификации кода может использоваться при модернизации унаследованных программ.

В качестве направлений дальнейшего развития системы можно назвать добавление поддержки других статических анализаторов для расширения спектра находимых ошибок. Расширение грамматики языка шаблонов позволит точнее указывать место, которое необходимо изменить. Помимо исправления ошибок, система может быть легко адаптирована и под смежные задачи: увеличение быстродействия кода, повышение безопасности.

Список литературы

[1] P. Liu, O. Tripp, and C. Zhang, "Grail: Context-aware fixing of concurrency bugs," in Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 318–329. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635881

[2] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," SIGPLAN Not., vol. 46, no. 6, pp. 389–400, Jun. 2011. [Online]. Available: http://doi.acm.org/10.1145/1993316.1993544

[3] P. Liu and C. Zhang, "Axis: Automatically fixing atomicity violations through solving control constraints," in Software Engineering (ICSE), 2012 34th International Conference on, June 2012, pp. 299–309.

[4] Intellij idea how-to: Static code analysis. JetBrains. [Online]. Available: https://www.jetbrains.com/idea/documentation/static_code_analysis.html

[5] Structural search and replace: What, why, and how-to. JetBrains. [Online]. Available: https://www.jetbrains.com/idea/documentation/ssr.html

[6] C. L. Goues, S. Forrest, and W. Weimer, "Current challenges in automatic software repair," Software Quality Journal, vol. 21, pp. 421–443, 2013. [Online]. Available: http://www.cs.cmu.edu/~clegoues/docs/legoues-sqjo13.pdf

[7] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," Software Engineering, IEEE Transactions on, vol. 38, no. 1, pp. 54–72, Jan 2012.

[8] B. Elkarablieh and S. Khurshid, "Juzi," in Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on, May 2008, pp. 855–858.

[9] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in Proceedings of the 19th International Symposium on Software Testing and Analysis, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 61–72. [Online]. Available: http://doi.acm.org/10.1145/1831708.1831716
[10] Д.А.Тимофеев, "Разработка технологии модификации программного кода, основанной на параметризуемых шаблонах," Master's thesis, Санкт-Петербургский государственный политехнический университет, Санкт-Петербург, 2010.
[11] Исходные коды разработанного проекта. Bitbucket, code hosting. [Online]. Available: https://bitbucket.org/h31/fixmycode
[12] F. Ma, "On the study of tree pattern matching algorithms and applications," Ph.D. dissertation, The University Of British Columbia, 2006.
[13] Веб-сайт проекта antlr. ANTLR. [Online]. Available: http://www.antlr.org/
[14] T. Parr, The Definitive ANTLR 4 Reference, 2nd ed. Pragmatic Bookshelf, 2013.
[15] Grammars written for antlr v4. ANTLR. [Online]. Available: https://github.com/antlr/grammars-v4
[16] Java version statistics: 2015 edition. Plumbr. [Online]. Available: https://plumbr.eu/blog/java/java-version-statistics-2015-edition
[17] Findbugs bug descriptions. FindBugs. [Online]. Available: http://findbugs.sourceforge.net/bugDescriptions.html
[18] Нимейер, Программирование на Java. Эксмо, 2014. [Online]. Available: https://books.google.ru/books?id=zVpUBQAAQBAJ
[19] Jgrapht. JGraphT - a free Java graph library. [Online]. Available: http://jgrapht.org/

Автоматическое тестирование линеаризуемости реализаций многопоточных структур данных

Евдокимов А.А.
Университет ИТМО
evdokimov@rain.ifmo.ru

Цителов Д.И.
ООО "Эксперт-Система"
tsitelov@acm.org

Елизаров Р.А.
ООО "Эксперт-Система"
elizarov@devexperts.com

Трифанов В.Ю.
ООО "Эксперт-Система"
trifanov@devexperts.com

***Аннотация*—В разработке алгоритма помимо теоретической корректности важна правильная реализация. Для однопоточного случая задача проверки реализации решается, например, модульным и нагрузочным тестированием. Однако, в многопоточных системах применение этих средств затруднено, поскольку выполнение программы зависит от чередования операций в потоках. Существует потребность в инструментах, которые проверяют корректность реализации многопоточного алгоритма. Аналогом понятия «корректности» для многопоточных программ служит понятие линеаризуемости — свойство алгоритма, при котором результат любого параллельного выполнения эквивалентен некоторому последовательному. В статье предлагается инструмент, проверяющий структуру данных на линеаризуемость при определенных допущениях (структура данных должна быть неблокирующейся и не зависящей от внешней среды). Инструмент работает «по определению» — ищет набор операций над структурой данных, параллельное исполнение которых не может быть объяснено последовательным. С помощью инструмента было показано, что несколько структур данных из общедоступных библиотек ведут себя некорректно на определенных наборах операций.**

Ключевые слова*—*линеаризуемость; Java; многопоточные алгоритмы

I. Введение

При разработке алгоритма важна не только теоретически доказанная корректность, но и правильная программная реализация. В однопоточной среде для проверки реализации применяются такие подходы к тестированию, как модульное и нагрузочное тестирование. Однако, в связи с развитием многопроцессорных систем, увеличивается число многопоточных алгоритмов. Упомянутые подходы к тестированию в целом оказываются мало эффективны в многопоточном окружении, так как программа может работать без ошибок почти всегда, кроме редких ситуаций, когда ошибка проявит себя, вследствие особой конфигурации потоков и чередования операций в них. Существует потребность в инструменте, который проверяет корректность реализации многопоточного алгоритма.

Общепринятым критерием «корректности» для многопоточных программ является понятие *линеаризуемости* [1] – свойство программы, при котором результат любого параллельного выполнения операций эквивалентен некоторому последовательному. В статье описан инструмент, который проверяет структуру данных на линеаризуемость при некоторых допущениях. Инструмент пытается найти набор операций, на котором линеаризуемость будет нарушена. Если такой набор будет найден, это значит, что структура данных нелинеаризуема. В противном случае никакого определенного ответа дать нельзя.

Для его реализации потребовалось решить задачи генерации тестовых наборов, генерации многопоточных исполнений для этих наборов и проверки полученных многопоточных исполнений.

С помощью созданного инструмента были найдены наборы операций, приводящие к некорректному поведению некоторых структур данных из общедоступных библиотек.

II. Корректность многопоточных структур данных

В качестве критерия корректности реализации многопоточной структуры данных можно использовать понятие *линеаризуемости*.

Если говорить неформально, структура данных *линеаризуема*, если можно считать, что каждая ее операция производит действие над структурой мгновенно, в один момент времени, лежащий между вызовом операции и ее возвратом. Такая операция называется *атомарной*. Место, где операция производит действие, называется *точкой линеаризации* [1].

Пример, иллюстрирующий точки линеаризации, показан на рисунке 1. Показано многопоточное обращение к структуре данных «счетчик»: по две операции incAndGet(), вызываемые из двух потоков A и B. Несмотря на то, что операции перекрываются, каждая операция возымела эффект в точке, помеченной крестом. И, соответственно, каждая операция корректно отработала, вернув свое, уникальное, значение.

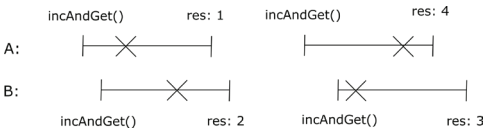


Рис. 1. Пример точек линеаризуемости

Теперь определим понятие линеаризуемости более формально, следуя классической монографии The Art of Multiprocessor Programming [1].

Исполнение многопоточной системы представлено *историей* – конечной последовательностью *событий вызова* и *возврата*, где *событие вызова* – момент вызова операции над структурой данных, а *событие возврата* – момент возврата из нее. Подыстория истории H – подпоследовательность событий истории H. Мы будем записывать событие вызова, как $\langle o\ m(args)\ A\rangle$, где o – объект, m – его метод с аргументами $args$, а A – идентификатор потока. Событие возврата будем записывать, как $\langle o\ r(ret)\ A\rangle$, где o – объект,

r – это либо Ok, либо код ошибки, ret – результирующее значение, а A – идентификатор потока.

Событие возврата *соответствует* событию вызова, если они принадлежат одному объекту и потоку. В случае, если метод был вызван несколько раз из одного потока, образуется несколько событий возврата и все они соответствуют методу вызова. Введем понятие *вызова метода*. *Вызовом метода* будем называть пару из события вызова и следующего соответствующего ему события возврата. Событие вызова называется *незавершенным*, если дальше в H нет ни одного соответствующего ему события возврата.

Расширение истории H – история, полученная из H добавлением *событий возврата* к некоторым (нулю или более) незавершенным событиям вызова. Будем обозначать подысторию H, состоящую из всех соответствующих друг другу событий вызова и возврата, как $complete(H)$.

История H *последовательна*, если она состоит из чередующихся событий вызова и соответствующих им событий возврата и начинается с события вызова.

Подыстория потока, или $H|A$ – подпоследовательность всех событий истории H для потока A. Две истории H и H' *эквивалентны*, если для любого потока A $H|A$ совпадает с $H'|A$. История H *правильно сформирована*, если для каждого потока A $H|A$ – последовательна.

Последовательная история H *корректна*, если подыстория для каждого объекта корректна, то есть соответствует спецификации поведения объекта.

Вызов метода m_0 *предшествует* вызову m_1 в истории H, если событие возврата для m_0 произошло до события вызова m_1.

Итак, история H *линеаризуема*, если есть такое ее расширение H' и такая корректная последовательная история S, что:

- $complete(H')$ эквивалентна S;
- Если вызов метода m_0 предшествовал вызову m_1 в H, то это должно соблюдаться и в S.

Будем называть S *линеаризацией* H.

Таким образом, было введено формальное определение линеаризуемости. И если найдется способ проверять его программным путем, то появится возможность автоматически проверять структуру данных на линеаризуемость.

III. Похожие разработки

Есть несколько других работ, в которых описаны подобные инструменты.

В одной из работ описан инструмент COLT [2]. Этот инструмент проверяет атомарность композиции заведомо атомарных операций структуры данных, пользуясь определением линеаризуемости. Для этого в произвольных местах проверяемой композиции вставляются другие операции над структурой данных, как будто выполняемые параллельно. Это возможно вследствие атомарности операций. Далее полученные наборы операций исполняются и полученные результаты проверяются. Однако такой подход не подходит для тестирования непосредственно структуры данных, но в работе есть полезные замечания о взаимовлиянии операций, которые учтены в разработанном инструменте.

В другой работе описан инструмент Line-Up [3], который проверяет линеаризуемость структуры данных. В нем используется иной подход генерации тестовых наборов: там не учитываются взаимовлияние операций и согласованность аргументов. Также там используется иной подход к генерации многопоточных исполнений. Для генерации используется программа проверки моделей (model checker) CHESS [4], что вносит свою специфику, так как в ней используется управление чередованием операций за счет вмешательства в механизм диспетчеризации исполнения нативного кода и такой метод плохо подходит для обнаружения ошибок, которые могут возникнуть при JIT-оптимизации и связанных с моделью памяти аппаратного уровня. В нашем подходе используется генерация многопоточных исполнений конечного, скомпилированного и оптимизированного кода.

Также существует инструмент нагрузочного тестирования jcstress [5]. Он позволяет тестировать многопоточное исполнение кода, но не в атоматическом режиме. Требуется непосредственно реализовать функции, которые будут исполняться параллельно, и все ожидаемые результаты. В нашем же подходе различные многопоточные сценарии и возможные корректные результаты генерируются в автоматическом режиме. В некотором роде созданный инструмент можно считать развитием jcstress.

IV. Метод проверки корректности

Целью исследования является разработка и реализация метода, позволяющего проверять линеаризуемость структуры данных в автоматическом режиме «по определению», и при этом работающего достаточно эффективно и требующего минимального количества настроек со стороны программиста.

Формальное доказательство линеаризуемости является сложной задачей. Наш метод будет стремиться найти нарушения линеаризуемости, а именно искать наборы операций, на которых многопоточная структура данных ведет себя некорректно.

В качестве первого шага на пути к решению сузим решаемую задачу и не будем рассматривать структуры данных, которые при вызове какой-то операции могут заблокироваться, ожидая выполнения какого-то условия. Это значит, что любая операция должна завершаться вне зависимости от текущего состояния структуры. Если же операция не завершается, значит внутри произошел dead-lock, зацикливание или проблема иного рода и это сразу же сигнализирует об ошибке. После этого допущения во всех опредениях не требуется рассматривать операции, которые являются *незавершенными*. Таким образом, рассмотрение блокирующихся структур данных выходит за рамки нашего исследования.

После этого определение линеаризуемости принимает следующий вид:

История H *линеаризуема*, если существует такая корректная последовательная история S, что:

- H эквивалентна S;
- Если вызов метода m_0 предшествовал вызову m_1 в H, то это должно соблюдаться и в S.

В качестве второго шага вводится следующее предположение: для проверки структуры данных на линеаризуемость нужно перебрать достаточно большое количество многопоточных историй, проанализировать результаты выполнения и попытаться их объяснить последовательной перестановкой. Если каждая многопоточная история будет объяснена, то можно с достаточно хорошей вероятностью утверждать, что структура данных линеаризуема.

Стоит отметить, что такой подход не позволяет непосредственно найти ошибку в реализации, но он может сиг-

нализировать о ее наличии, предоставляя тест, на котором поведение структуры данных в многопоточной среде будет отличаться от ожидаемого.

Введенные предположения позволяют предложить следующий метод.

1) Описание метода: Выбирается небольшое количество потоков, в которых будет тестироваться структура данных. Далее создается тестовый набор операций. Он представляет собой небольшой фиксированный набор операций для каждого потока, которые будут вызваны над объектом. Для этого набора генерируются все возможные последовательные исполнения для объекта и фиксируются результаты выполнения операций каждого такого исполнения. Последовательное исполнение состоит из набора операций, являющихся перестановкой исходного набора операций. Перестановка, суженная на конкретный поток, должна давать в точности те операции и в том же порядке, что предназначены для этого потока. Набор последовательных исполнений конечен, так как рассматриваются только перестановки исходного набора операций.

Мы получаем эталонные исполнения этих операций над объектом и, если в многопоточном режиме получается какое-то новое исполнение, это однозначно сигнализирует об ошибке.

Причем желательно, чтобы этот набор операций был не абсолютно случайным. Некоторые операции над структурой данных принимают аргументы. Можно ввести понятие классов эквивалентности входных параметров. Входные параметры из одного класса эквивалентности вызывают одно и тоже внутреннее поведение структуры.

Например, скорее всего неважно, какое значение добавлять в очередь. Поведение структуры никак не изменится от этого значения. Таким образом, все входные параметры для операции добавления в очередь образуют один класс эквивалентности.

Но если значение – ключ для хеш-таблицы, то операции, вызванные с разными ключами в разных потоках параллельно могут никак не пересечься по логике структуры и будут выполнены изолированно друг от друга. Поэтому, каждое значения ключа для хеш-таблицы образует свой класс эквивалентности (если не учитывать коллизии). Если ключи будут принадлежать одному классу эквивалентности, то есть будут одинаковыми, то могут возникать различные эффекты, которые и могут помочь выявить ошибку. Например вставка в хеш-таблицу разных значений с одинаковым ключом.

Также стоит обратить внимание на операции, никак не изменяющие структуру данных, а лишь обращающиеся к данным и определенным образом их обрабатывающие. Подобные операции, выполненные параллельно, скорее всего будут работать ровно также как и в однопоточном режиме. Поэтому при генерации тестовых наборов стоит это учесть и не ставить в одновременное параллельное выполнение такие операции.

Следующим шагом после генерации последовательных исполнений является генерация многопоточных исполнений набора операций. Для каждого многопоточного исполнения фиксируется результат выполнения каждой операции. Далее этот набор результатов нужно проверить. Для любого многопоточного исполнения должно существовать последовательное исполнение, приводящее к тому же результату. Поэтому перебираются многопоточные исполнения, и ищется соответствие результатов среди последовательных. Если найдено последовательное исполнение, совпадающее с многопоточным, значит многопоточное исполнение корректно, и мы переходим к следующему. В противном случае обнаружилось многопоточное исполнение, которое нелинеаризуемо, а значит, структура данных не удовлетворяет этому свойству. Дополнительно получен тестовый набор операций, на котором произошла ошибка и его, далее, можно использовать для анализа и поиска ошибки в реализации.

Метод можно представить в виде блок-схемы, изображенной на рисунке 2.

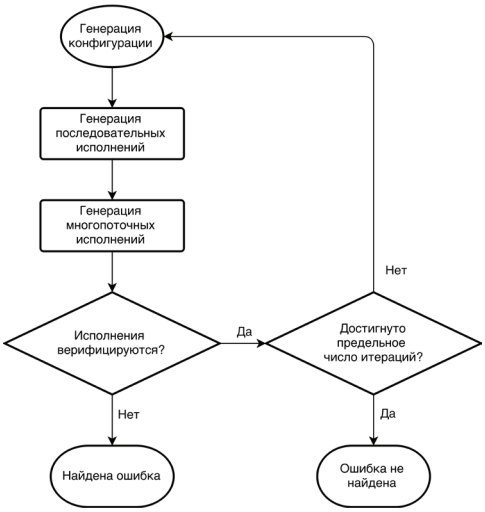


Рис. 2. Блок-схема алгоритма

2) Границы применимости метода: Структура данных не должна зависеть от внешних факторов – например, внутри нее не должно быть работы с сетью. При повторных выполнениях одной и той же последовательности операций должны получаться одни и те же результаты. В противном случае нет возможности создать эталонные последовательные исполнения и, таким образом, проверить многопоточные.

Как было сказано, данный метод находит не непосредственно ошибки, а набор операций, на которых многопоточная структура данных ведет себя некорректно. Некорректное поведение может быть вызвано такими ошибками, как незащищенный доступ к разделяемой памяти, неправильная синхронизация исполнения или логические ошибки в частях алгоритма, призванных обеспечить корректность алгоритма при многопоточном исполнении – иными словами, любые ошибки, которые приводят к иному поведению в многопоточной среде по сравнению с однопоточной.

В случае нахождения набора операций, на котором структура данных ведет себя некорректно, можно утверждать, что структура данных нелинеаризуема. В противном случае никакого определенного ответа дать нельзя. Будем

считать, что ожидается корректность структуры данных. То есть нулевая гипотеза – структура данных линеаризуема. Тогда инструмент может давать следующие варианты ответов: «линеаризуема и это неправильно», «линеаризуема и это правильно» и «нелинеаризуема и это правильно». Таким образом, ошибки первого рода отсутствуют (ложных срабатываний быть не может), второго рода – присутствуют (могут не найтись наборы операций для некорректных структур).

Итого, границы применимости метода можно определить так: метод точен, но не полон и его можно применять на неблокирующихся структурах данных, не зависящих от внешней среды и заявляющих себя, как безопасные при исполнении в многопоточной среде (то есть удовлетворяющие свойству линеаризуемости).

V. Особенности реализации метода

A. *Генерация тестовых наборов*

Требуется каким-то образом предоставить входные данные анализатору. Просто передать ссылку на соответствующий класс недостаточно – непонятно, какие методы вызывать, с какими аргументами их вызывать и с какими параметрами тестировать данный класс.

Для решения этой проблемы был реализован механизм аннотаций. Идея была взята из библиотеки jcstress [5]. Чтобы протестировать какую-то структуру данных, требуется создать класс и разметить его специальным образом с помощью аннотаций.

```
1   @CTest(iter = 300, actorsPerThread = {"1:3", "1:3"})
2   @CTest(iter = 300, actorsPerThread = {"1:3", "1:3", "1:3"})
3   public class QueueTest {
4       public Queue<Integer> q;
5
6       @Reset
7       public void reload() {
8           q = new GenericMPMCQueue(16);
9       }
10
11
12      @Actor(args = {"1:10"})
13      public void offer(Result res, Object[] args) throws Exception {
14          Integer value = (Integer) args[0];
15          res.setValue(q.offer(value));
16      }
17
18      @Actor(args = {})
19      public void poll(Result res, Object[] args) throws Exception {
20          res.setValue(q.poll());
21      }
22
23      @Immutable
24      @Actor(args = {})
25      public void peek(Result res, Object[] args) throws Exception {
26          res.setValue(q.peek());
27      }
28  }
```

Листинг 1. Пример класса для тестирования очереди

Рассмотрим пример такого класса (листинг 1). Для настройки анализатора применяется аннотация @CTest, которая имеет следующие параметры: iter – количество сгенерированных тестовых наборов, actorsPerThread задает количество потоков и диапазон количества сгенерированных операций для каждого потока. Значение для actorsPerThread – набор строк, количество строк соответствует количеству потоков, а отдельная строка задает открытый справа диапазон количества операций для соответствующего потока.

Аннотация @Reset помечает метод, который будет использован для инициализации структуры (требуется переинициализировать структуру после каждого исполнения операций на ней).

Для того чтобы указать методы, требующие тестирования, и аргументы, для них нужно создать специальные методы и пометить их аннотацией @Actor. В качестве параметра args задается диапазон сгенерированных значений для каждого аргумента метода. Внутри созданного метода требуется вызвать нужный метод структуры данных с предоставленными значениями и сообщить результат исполнения операции в служебную структуру Result с помощью соответствующих процедур. Если вызываемый метод над структурой данных завершается исключением (это может быть частью контракта структуры данных – например, извлечение из пустой очереди), то это будет корректно обработано и записано соответствующим образом в структуру Result.

Достаточно сложно оценить качество тестового набора. Но о некоторых наборах можно сказать, что они вряд ли помогут выявлению ошибок. К таким наборам относятся те, которые никак не изменяют структуру данных, или те, в которых результаты всех операций неотличимы друг от друга (ничего не возвращают).

Существует опциональная возможность пометить метод как неизменяющий структуру данных (если есть такое априорное знание) для улучшения качества сгенерированного набора с помощью аннотации @Immutable. В таком случае в тестовом наборе среди параллельных операций обязательно найдется операция, изменяющая структуру данных.

Представленный класс определяет следующую конфигурацию: два теста, в каждом генерируется по 300 тестовых наборов. В первом тесте 2 потока, во втором 3. Для каждого потока генерируется от 1 до 2 (включительно) операций. Возможные операции: offer (с аргументом от 1 до 10), poll и peek.

Пример тестового набора представлен в листинге 2.

```
1   A: q.peek(); q.peek();
2   B: q.offer(3); q.offer(4);
```

Листинг 2. Пример тестового набора для очереди

B. *Генерация многопоточных исполнений*

После того, как тестовый набор операций был создан, необходимо сгенерировать многопоточные исполнения для него.

Для этого созданный набор исполняется большое количество раз в многопоточном режиме и результаты исполнения сохраняются.

1) Качество генератора: Важным условием, влияющим на работоспособность создаваемого анализатора, является качество генератора многопоточных исполнений. Если при запусках будет преобладать какой-то конкретный порядок исполнения команд, то значимых результатов не получится. Нужно стремиться к тому, чтобы операции над

структурой данных исполнялись максимально разнообразно, покрывая все возможные состояния структуры.

Также требуется, чтобы параллельные операции исполнялись действительно параллельно, иначе многопоточное исполнение вырождается в последовательное, и нарушений линеаризуемости найдено не будет. Пример такого вырождения – когда вследствие особенностей диспетчеризации потоков операции для первого потока исполняются раньше, чем запускается второй поток, и генерируется один и тот же вариант исполнения.

Качество генератора оценивается по скорости работы, так как более быстрый генератор в единицу времени сгенерирует большее количество многопоточных исполнений, что должно привести к лучшему покрытию всевозможных исполнений. Также оценивается распределение возможных исходов – если последовательный анализ показал какое-то количество определенных исходов, а генератор покрывает только некоторые из них, либо доминирует какой-то конкретный исход, скорее всего мы не получим достоверного покрытия вероятных исходов, что ухудшит способность генератора к нахождению ошибок реализации.

Для того, чтобы наборы операций стартовали одновременно и исполнялись без лишних задержек, были проделаны следующие шаги.

Для общего снижения издержек на создание и инициализацию различных служебных структур данных создается пул потоков, в котором и исполняются наборы операций. При моделировании многопоточных исполнений никаких служебных структур не создается – они создаются заранее и передаются в многопоточную модель.

Для того, чтобы наборы операций начинали исполняться одновременно, потоки, исполняющие их, синхронизируются с помощью барьера. При достижении барьера поток блокируется. После того, как все потоки достигнут барьера, барьер снимается, и все потоки одновременно продолжают исполнение.

2) Вызов методов: Следующий вопрос, требующий решения – каким образом вызывать методы у структуры данных. Java предоставляет специальную библиотеку, Reflection API [6], позволяющую получать методы у класса и вызывать их. Однако на практике она показала себя не очень хорошо при использовании в нашем инструменте. Во-первых, вызов методов с ее помощью происходит с большими накладными расходами по времени, а с учетом того, что нам требуется делать большое количество вызовов методов у структуры, это становится критичным. Во-вторых, ее использование вносит различные, неконтролируемые задержки между вызовами операций (стек служебных вызовов, приводящий в итоге к вызову нужной нам операции), что снижает качество генератора и снижает возможности по контролируемому исполнению кода. В-третьих, библиотека может вносить неявную синхронизацию при вызове методов, что негативно влияет на качество генерации многопоточных исполнений. Поэтому было решено отказаться от Reflection API в пользу генерации Java байткода.

Идея состоит в генерации кода с минимальным добавлением служебных операций, который легко бы поддавался оптимизации JIT-компилятором. Для реализации этого подхода используется библиотека ASM [7]. Был создан базовый класс, в котором определен нужный метод. Далее генерируется по одному наследнику для каждого потока от этого класса и у них переопределяется этот метод так, чтобы он содержал вызовы нужных операций. После этого при генерации многопоточных исполнений достаточно вызывать метод у нужного наследника.

3) Кеш-линии: На выполнение многопоточного кода может повлиять так называемая проблема False Sharing [8]. Процессор оперирует памятью не побайтово, а так называемыми кэш-линиями, которые представляют собой блоки фиксированного размера (часто используется размер 64 байта, но могут быть и другие размеры). И если два разных объекта разделяют общую кэш-линию, то изменение одного объекта может повлечь за собой загрузку данных из памяти для второго объекта заново, хотя данные и не менялись, что может повлиять на исполнение многопоточного кода. Один из подходов для решения этой проблемы – добавление фиктивных полей в структуру, чтобы ее размер был кратен размеру кэш-линии и, таким образом, она не разделяла какую-то кэш-линию с другим объектом. Для облегчения выравнивания в Java 8 была введена аннотация @Contended и проаннотированный таким образом класс выравнивается автоматически. Аннотация была применена к структуре Result, в которую и происходит запись данных из разных потоков. Выравнивание объектов Result никак не влияет на структуру памяти исследуемого объекта и все связанные с этим особенности его реализации остаются неизменными.

4) Управление выполнением: После того как наборы операций исполняются эффективно, можно попробовать внести контролируемые изменения в порядок исполнения. Для этого нужно наблюдать за получающимися результатами и каким-то образом влиять на исполнение для обеспечения максимального покрытия. Это может быть сделано добавлением небольших, динамически изменяющихся задержек перед каждой операцией. Однако на данный момент в инструменте используется упрощенный подход. Фаза генерации многопоточных исполнений была разбита на две части. В первой фазе все операции исполняются без задержек. Во второй фазе перед каждой операцией вносится случайная задержка из небольшого диапазона. Такой подход привел к немного лучшему распределению результатов многопоточных исполнений (многопоточным исполнениям соответствуют однопоточные и хороший генератор будет производить различные многопоточные исполнения, приводящие к разным результатам, для плохого же генератора будут доминировать одни и те же многопоточные исполнения) и повысил процент тестовых наборов, засчитанных как приводящие к ошибке.

Нужно отметить, что на первых итерациях многопоточного запуска может наблюдаться поведение, необычное для последующих запусков. Это так называемая фаза прогрева, когда большую роль играют особенности диспетчеризации потоков и JIT-оптимизация. Однако это не имеет негативного влияния на разработанный метод, поскольку он не имеет ложных срабатываний. Потребовалось учесть фазу прогрева так, чтобы итерации этой фазы не составляли большую часть общего количества итераций.

Количество многопоточных запусков ограничено фиксированным числом итераций и является разумным компромиссом между временем работы инструмента и качеством нахождения ошибок. Для этого было взято достаточно большое количество запусков, при котором ошибки находились. Далее было определено реальное число запусков, которое обеспечивало нахождение ошибок (к примеру, делалось

100000 запусков, но ошибки находились уже на первых 50000 итераций).

C. Проверка многопоточных исполнений

Для проверки многопоточного исполнения требуется найти последовательное исполнение, ему соответствующее. Соответственно, необходим способ построения последовательных исполнений.

Для построения последовательных исполнений требуется сгенерировать все перестановки исходного набора операций такие, что перестановка, суженная на конкретный поток, дает в точности те операции и в том же порядке, что для него предназначены. Можно сказать, что планировщик исполнения случайным образом выбирает поток, из которого он будет выполнять следующую операцию.

Пример последовательных перестановок для примера на листинге 2 представлен в листинге 3.

```
1  [q.peek(); q.peek(); q.offer(3); q.offer(4); ]
2  [q.peek(); q.offer(3); q.peek(); q.offer(4); ]
3  [q.peek(); q.offer(3); q.offer(4); q.peek(); ]
4  [q.offer(3); q.peek(); q.peek(); q.offer(4); ]
5  [q.offer(3); q.peek(); q.offer(4); q.peek(); ]
6  [q.offer(3); q.offer(4); q.peek(); q.peek(); ]
```

Листинг 3. Пример последовательных перестановок

После того, как все такие перестановки были сгенерированы, для каждой перестановки последовательно выполняются ее операции над структурой данных и сохраняется результат.

VI. Практические результаты

A. Тестирование на синтетических примерах

Для первичного тестирования инструмента было реализовано несколько структур данных и в них были допущены синтетические ошибки.

Рассмотрим структуру данных **счетчик** и ее ошибочную реализацию и покажем, что было найдено анализатором. Также это поможет проиллюстрировать его работу. Примером использования **счетчика** является подсчет количества появлений каких-либо событий в многопоточной среде.

В листинге 4 представлена корректная реализация.

```
1  public class Counter {
2      private volatile int c = 0;
3
4      public int get() {
5          return c;
6      }
7
8      public synchronized int getAndIncrement() {
9          return c++;
10     }
11 }
```

Листинг 4. Структура данных "счетчик"

Ключевое условие, обеспечивающее корректность реализации – защищенный с помощью ключевого слова synchronized метод incrementAndGet. Это значит, что внутри метода единовременно может находиться только один поток.

В случае, если по каким-то причинам ключевое слово будет потеряно, возникает классическая проблема – *состояние гонки*. Это происходит, потому что операция c++ не атомарна. При совершении этой операции сначала значение переменной c читается, потом увеличивается, потом записывается обратно. Поэтому, если два потока считали одно и тоже значение, потом увеличили его и записали обратно, счетчик увеличится на значение 1. А ожидалось, что на 2.

Или, в худшем случае, один поток считал значение, а другой в это время увеличил счетчик несколько раз и записал результат. Первый же поток увеличивает свое значение и записывает его обратно, что приводит к откату счетчика.

Анализатор находит данную ошибку. В листинге 5 отображена найденная конфигурация и недопустимое поведение для многопоточного счетчика. В фигурных скобках показан результат выполнения операции. Обе операции incrementAndGet вернули значение 1, что не является допустимым поведением.

```
1  [0_incrementAndGet() {1}]
2  [1_incrementAndGet() {1}]
```

Листинг 5. Пример для ошибочного счетчика

Также были реализованы многопоточная очередь и примитив банковского приложения (функциональность, которую должна обеспечивать структура – получение количества средств на счете клиента, установка значения средств на счете клиента, и перевод средств между разными клиентами) и в них были допущены ошибки синхронизации при доступе к разделяемым переменным. Для всех реализаций инструмент представил пример, на котором линеаризуемость нарушается.

В качестве примеров были выбраны несложные структуры данных, корректность работы которых обеспечивалась с помощью блокировок, так как эти ошибки в таких структурах достаточно наглядны, и это те примеры, на которых анализатор точно должен находить ошибки. Однако потенциально, спектр обнаруживаемых ошибок не ограничивается потерями блокировок, ведь анализируется не корректность синхронизации, а общее поведение структуры данных, которое в многопоточном режиме должно также быть корректным и соответствовать спецификации поведения структуры данных. Исходя из сути метода, могут быть обнаружены только те ошибки, которые приводят к неправильному поведению именно в многопоточной среде.

B. Тестирование на доступных библиотеках

Было проведено тестирование некоторых доступных библиотек созданным инструментом.

Сначала была протестирована стандартная библиотека java.util.concurrent [9] и тестирование не выявило ошибок. Далее была протестирована библиотека Google Guava [10] и ошибок также не нашлось.

Эти библиотеки используются в большом количестве проектов и скорее всего все возможные ошибки уже были выявлены и исправлены.

Далее были протестированы экспериментальные библиотеки, содержащие реализации многопоточных очередей и хэш-таблиц - jctools [11], zchannel [12] и hish_scale_lib [13]. Тестирование показало, что некоторые структуры оттуда реализованы некорректно и, действительно, возможны

Таблица I. Статистика для найденных ошибок

Структура	Тестовый набор	Многопоточное исполн.	Время (мс)
Counter	1 – 1 (1)	2 – 1695 (530)	4 – 110 (33)
Queue	1 – 28 (7)	6 – 997 (307,6)	5 – 38786 (8635)
Accounts	1 – 39 (14,9)	19 – 5547 (1199,4)	3 – 50606 (18697)
NonBlockingSetInt	1 – 7 (3,2)	1846 – 54522 (29736,4)	52 – 10414 (4291)
NonBlockingHashSet	1 – 33 (8,5)	1 – 58749 (24661,7)	5 – 66975 (16111)
MpmcArrayQueue(2)	2 – 25 (11,1)	335 – 67590 (34806,5)	2352 – 45628 (19974)
MPMCQueue(2)	1 – 18 (10,6)	9034 – 96977 (52290)	1814 – 33958 (19714)
MPMCQueue(16)	2 – 20 (5,8)	454 – 23292 (14409,8)	1877 – 33858 (8651)
LockFreeQueue	6 – 193 (76,375)	2588 – 66147 (32718)	8825 – 342199 (135202)

такие наборы операций, что структура ведет себя неправильно.

А именно, были найдены ошибки в

- NonBlockingSetInt, NonBlockingHashMap, NonBlockingHashSet из high_scale_lib.
- MpmcArrayQueue из jctools
- GenericMPMCQueue(2), GenericMPMCQueue(16) из zchannel
- lockfreequeue, найденная на GitHub [14]

Инструмент показал, что некоторые структуры данных из доступных библиотек, предоставляющих многопоточные реализации структур данных, реализованы некорректно, и в некоторых случаях их поведение может отличаться от ожидаемого.

Может быть, эти ошибки и нечасто встречаются в реальной жизни, но сам факт того, что они могут встретиться, важен. И структуры, которые заявлены как безопасные к использованию в многопоточной среде, по существу, таковыми не являются.

В таблице 1 показана статистика параметров работы инструмента для найденных ошибок. В ней отражен номер тестового набора, приводящего к ошибке, количество многопоточных запусков, потребовавшихся для выявления некорректного поведения, а также общее время поиска. В таблице указаны диапазоны полученных значений и усредненное значение для 10 запусков.

VII. Заключение

В работе было предложено решение, позволяющее проверять линеаризуемость структуры данных в автоматическом режиме. Был создан инструмент, реализующий это решение. Он не способен доказать выполнение линеаризуемости, однако способен предъявить тест, на котором удалось обнаружить нарушение линеаризуемости. Далее этот тест может быть использован для анализа реализации и нахождения ошибки.

Инструмент состоит из трех частей – генератора тестовых наборов для структуры, генератора последовательных исполнений этих тестовых наборов и генератора многопоточных исполнений тестовых наборов.

Созданный инструмент находит наборы операций, вызывающие неправильное поведение реализаций с допущенными ошибками, а также с помощью него было показано, что некоторые структуры данных из доступных библиотек, таких как jctools, zchannel и high_scale_lib, на некоторых найденных тестовых наборах ведут себя некорректно.

Ключевыми моментами, влияющими на качество анализатора, являются генератор тестовых наборов и генератор многопоточных исполнений.

В качестве возможного направления развития инструмента можно выделить развитие генератора наборов, используя априорные сведения о взаимовлиянии операций и исключая из анализа наборы, которые не могут привести к некорректному многопоточному исполнению.

Другим возможным развитием работы может являться улучшенный генератор многопоточных исполнений, который использует взамен множественного запуска тестовых наборов контролируемое исполнение этих наборов и обеспечивает более эффективное покрытие возможных путей исполнения. Применение данного подхода приведет к своим компромиссам в виде выбора подмножества всех возможных вариантов исполнения и искажения низкоуровневых эффектов реализации. Потребуется отдельное сравнительное исследование эффективности и скорее всего имеет смысл поддерживать оба варианта. Также в случае использования такого подхода будет получен не только тест, приводящий к ошибке, но и непосредственно образец выполнения кода, что поможет в локализации проблемы.

Также возможна интеграция инструмента с анализатором DRD (Data Race Detector) [15], что расширит диапазон обнаруживаемых ошибок.

На данном этапе реализованный инструмент показал свою практическую применимость и может быть использован при разработке многопоточных структур данных.

Список литературы

[1] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*, 1st ed. Morgan Kaufmann, 2008.

[2] O. Shacham, N. Bronson, A. Aiken, M. Sagiv, M. Vechev, and E. Yahav, “Testing atomicity of composed concurrent operations” *ACM SIGPLAN Notices - OOPSLA '11*, vol. 46, no. 10, pp. 51--64, Oct. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2048073

[3] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan, “Line-up: a complete and automatic linearizability checker” *ACM SIGPLAN Notices - PLDI '10*, vol. 45, no. 6, pp. 330--340, Jun. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1806634

[4] “Chess”. [Online]. Available: http://research.microsoft.com/en-us/projects/chess/

[5] “jcstress”. [Online]. Available: http://openjdk.java.net/projects/code-tools/jcstress/

[6] “Reflection API”. [Online]. Available: https://docs.oracle.com/javase/tutorial/reflect/

[7] “ASM library”. [Online]. Available: http://asm.ow2.org/

[8] “False Sharing”. [Online]. Available: http://robsjava.blogspot.ru/2014/03/what-is-false-sharing.html

[9] “java.util.concurrent”. [Online]. Available:

http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html
[10] “Google Guava”. [Online]. Available: https://code.google.com/p/guava-libraries/
[11] “jctools”. [Online]. Available: https://github.com/JCTools/JCTools
[12] “zchannel”. [Online]. Available: http://landz.github.io/
[13] “High Scale Lib”. [Online]. Available: https://github.com/stephenc/high-scale-lib
[14] “lockfreequeue”. [Online]. Available: https://github.com/yaitskov/lock-free-queue
[15] “Data Race Detector”. [Online]. Available: https://code.devexperts.com/display/DRD

Трейдинговые системы

Секционные доклады

Спонсорский доступ с малой задержкой посредством FPGA

Валерий Флоров, Павел Гарин, Павел Смирнов, Максим Метельков
EXACTPRO
{valery.florov, pavel.garin, pavel.smirnov, maxim.metelkov}@exactprosystems.com

***Abstract* — В статье рассматривается реализация спонсорского доступа к бирже с помощью FPGA платы с несколькими сетевыми разъемами. Рассмотрен способ снижения задержки прохождения пакетов через такое устройство за счет ранней передачи пакетов, еще не прошедших полную обработку (прием). Описываются использованные инструменты для тестирования разработанной системы, процедура проверки корректности ее функционирования и механизм измерения задержек.**

Keywords—sposored access; trading; fpga; low latency

I. Насущность проблемы

В мире биржевой торговли существует понятие “высокоскоростной трейдинг” (High-frequency trading, HFT). Стратегии, применяемые в HFT, используют преимущество в скорости при доступе к данным торгов (market data, MD) и при формировании заявок на покупку/продажу. Разница в ценах может быть незначительна, но поскольку сделок совершается очень большое количество (сотни сделок в секунду на одного клиента и больше), то в целом прибыль оказывается значительной. В целом позиции открываются на очень короткое время, среднее время удержания 22 секунды, и не переносятся на следующий день (внутридневная торговля). Согласно исследованиям Aite Group рынок HFT составлял 25% от общего объема торгов фьючерсами с ожиданием роста до 40% в 2015 году [2].

Актуальность самой задачи рассмотрена уже неоднократно, например [3] и [4]. Заметим, что наибольшее влияние на результативность сделок при высокоскоростной торговле оказывает значение задержки от момента публикации данных с биржевой площадки, до момента, когда сформированный ордер будет передан в ПО биржи (matching engine, ME). Естественно, большое значение играет и качество самого алгоритма, но оно не является предметом рассмотрения данной статьи.

II. Известные пути решения

Рассмотрим из чего состоят задержки при распространении ордера от клиента до биржи. Цепочка в общем случае может выглядеть так:

- ПО клиента (торговый алгоритм)
- Сетевая карта клиента, (Network interface card, NIC)
- Среда передачи, сюда может входить активное сетевое оборудование, а также физические линии передачи (провода или оптические кабели)
- ПО брокера
- Среда передачи
- Входной шлюз биржи

Кроме длины цепочки, на задержку влияют еще следующие факторы:

- скорость передачи данных. Используются скорости 10-40G
- размер сообщения (ордеров и ответов биржи)
- протокол передачи (обычно TCP или UDP, но могут быть и проприетарные протоколы)

Оценка сетевых задержек на стороне клиента достаточно подробно рассмотрена в [5].

Некоторые пункты в этих двух списках могут быть использованы клиентом для улучшения качества торговли, например: использование более скоростных подключений, написание алгоритма на языках/платформах, которые лучше управляют временем жизни объектов и, как следствие, уменьшают накладные расходы по управлению памятью, делают их более детерминированными и позволяют, таким образом, ускорить выполнение этих алгоритмов по сравнению, например, с языками с автоматической сборкой мусора; использование более производительных сетевых карт с прямым доступом к буферам пакетов, минуя ядро операционной системы (вариации таких сетевых карт можно найти на ресурсе NetFPGA [6] также множество готовых решений предлагает например компания Myricom [7] и другие поставщики); сокращение физической дистанции до брокера, в том числе сокращение количества “прыжков” по активному оборудованию; подключение к тому брокеру, кто вносит меньшие задержки.

Остальные пункты определяются биржей и брокером и не могут быть изменены так просто – протокол передачи, формат сообщения и т.п. Но есть еще одна возможность для существенного сокращения издержек. Некоторые биржи предоставляют непосредственное подключение к своим шлюзам, минуя ПО брокера (sponsored access, SA). Но при этом брокер все равно отвечает за своего клиента. Таким образом, есть некоторое противоречие – с одной стороны брокер и биржа заинтересованы в более активной торговле своих клиентов, но, с другой стороны, брокер сам может понести потери из-за их некорректного поведения.

Таким образом, стоит задача – с одной стороны снизить задержку до минимума, с другой стороны – позволить брокеру своевременно вмешаться и защитить себя (а возможно и клиента) от потерь при неправильном поведении торгового алгоритма клиента.

III. Экстремальный путь решения.

Сейчас можно найти много решений для HFT реализованных на программируемых логических матрицах (Field-Programmable Gate Array, FPGA), т.е. данный подход в настоящее время рассматривается как перспективный, но в основном решения фокусируются на стороне клиента, на таких аспектах, как анализ MD, ускорение принятия решений, ускорение обработки сетевого трафика. Хорошим примером такого обзора является [8].

Также на рынке уже присутствует несколько решений, нацеленных на решение задачи спонсорского доступа с микросекундными задержками: HFT Risk Gateway FPGA based solution [9] – 5 мкс; Ullink FPGA Risk Solution [10] – 2 мкс, но в [1] упоминается (но не рассматривается) подход при котором задержки можно существенно уменьшить. Автор предлагает начинать обрабатывать сообщения, еще до того, как они были приняты полностью.

Обычно в таких устройствах используются те или иные реализации аппаратного стека TCP/IP, например nxTCP for Xilinx [11]. Но это неминуемо вводит задержку на прием и передачу ввиду буферизации пакетов.

Эта задача также может быть решена с применением технологии FPGA. Ускорение достигается заменой последовательной логики, реализуемой процессором общего назначения (central processing unit, CPU) на схемную реализацию – логику, в виде параллельно работающей схемы везде, где это возможно. Такая схема обычно описывается на языке описания схем (hardware description language, HDL) VHDL, Verilog, SystemVerilog или подобном. Компилируется в специальный файл прошивки, который загружается в целевую микросхему FPGA и организует в ней связи для реализации задуманной схемы. Несмотря на то, что тактовая частота для такой схемы обычно меньше чем тактовая частота в 3-4 GHz для топовых моделей процессоров, выигрыш в производительности достигается за счет высокой параллельности (многостадийные конвейеры, при необходимости - множество экземпляров, работающих параллельно) и заточенности под конкретную задачу каждого из блоков. Последним шагом в ускорении таких алгоритмов может быть только реализация их в виде заказных микросхем (application specific integrated circuit, ASIC). Как правило, они имеют более высокую предельную частоту, по сравнению с той же схемой, реализованной в FPGA (за счет исключения всех накладных расходов на организацию таких схем, а также за счет лучшей низкоуровневой оптимизации). Оборотной стороной является высокая стоимость такого решения при малых производимых партиях, а также меньшая гибкость, по сравнению с FPGA.

IV. Цели проекта.

Разработать решение для предоставления спонсорского доступа клиенту со временем задержки менее 200 нс.

Доступ предоставить для Ethernet на скорости 10G по протоколу Native LSE [13], данные MD использовать MITCH LSE [27] Блокировать любое действие клиента, которое может нарушить условия брокера. Рассматривать нарушение следующих условий: Нулевой объем сделки; цена, выходящая за статический диапазон; цена, выходящая за динамический диапазон от цены последней сделки; превышение лимитов торговли как по отдельным инструментам, так и по сумме портфеля в целом. Предоставить возможность брокеру задавать границы проверок для статических и динамических цен, задавать пределы по торговле для каждого клиента,

V. Анализ целей.

Попробуем оценить минимально возможное время задержки. Минимальное время задержки может быть реализовано как быстродействующий “предохранитель”, встроенный непосредственно в линию, связывающую клиента и порт сетевого подключения на площадке биржи. Опираться будем на следующие предпосылки:

1) Клиент и биржа находятся на одной площадке, фактически клиент подключает свой сетевой порт непосредственно к порту входного коммутатора биржи.

2) Используется скоростной протокол нижнего уровня Ethernet 10G, оптика или медь.

3) Используется бинарный протокол Native LSE, а значит устанавливается tcp-сессия.

Оценим время передачи одного сообщения от клиента к бирже, нас интересует главным образом сообщение "New Order". Именно это сообщение дает распоряжение купить или продать определенный актив по указанной цене, основной тип сообщений при HFT. Оценку будем давать для указанной скорости передачи. Размер сообщения — 106 байт. Размер всех заголовков (TCP/IP, Ethernet frame) — 55 байт. Общий размер 161 байт. Итого, при идеальной среде передачи без помех, не учитывая задержку на распространение сигнала в медном/оптическом кабеле, время передачи такого пакета составит ~130 нс (10G) Таким образом, если устройство встраивается в разрыв между клиентом и биржей и использует буферизацию на прием всего пакета, то задержка увеличивается как минимум на двойное время передачи пакета, которое складывается из задержек PHY-MAC, MAC-TCPoffload,

что можно наблюдать как декларируемая задержка для решений [1]. Увеличение размера пакета при передаче сообщений большей длины соответственно увеличивает и задержку.

Возможно ли снизить это время и если возможно, то до каких величин? Да, возможно, если отказаться от буферизации, хотя бы частично. Можно начинать транзитную передачу пакета от клиента, не дожидаясь полного приема. Можно лишь незначительно задержать информацию внутри (на несколько тактов, необходимых на принятие решения — не нарушает ли пересылаемое сообщение некоторых правил торговли). Оценим необходимую задержку в тактах опорной частоты, пересчитав затем их в единицы времени.

Опорную частоту разумно выбрать 156 МГц, как необходимую для тактирования модулей приемо-передатчиков (Small Form-factor Pluggable, SFP). Период одного такта — 6.4 нс. Глубины конвейера в 30 тактов будет достаточно для анализа текущего пакета и принятия решения об разрешении или запрете. Таким образом можно уложиться в поставленный предел по задержке в 200 нс.

Для подключения через Ethernet на скорости 10G можно использовать плату расширения на несколько SFP+ слотов, с которой можно использовать несколько приемо-передатчиков для нужной скорости / среды передачи.

Чтобы не дать возможность клиенту послать некорректный ордер на биржу, нужно иметь возможность его анализировать в реальном времени в процессе передачи, для этого уметь извлекать из ордера такие поля как цена, объем, тип ордера, сторона (покупка/продажа), инструмент. Для бинарного протокола это не представляется сложным.

Основная идея состоит в том, что клиент не должен устанавливать соединение с устройством, оно должно быть абсолютно прозрачно для клиента до тех пор, пока он не нарушает определенных правил торговли. Отсутствие буферизации, связанная с контролем целостности протокола TCP, убирает двойную задержку на прием/отправку каждого транзитного пакета 400 нс + 400 нс [1]. Устройство лишь наблюдает за транзитным потоком, выстраивая одновременно анализ этого потока и принимая решения “на лету”. В случае, если анализ говорит, что передача такого пакета может нарушить правила торговли, а пакет уже начал передаваться на сторону биржи, этот пакет намеренно портится (искажается его контрольная сумма, которая передается последними байтами в ethernet frame, точнее — инвертируется один бит контрольной суммы). Такой пакет будет отвергнут входным коммутатором биржи и не будет анализироваться МЕ, как не существовавший. Со стороны клиента ситуация будет выглядеть, как внезапный обрыв кабеля. Естественно, необходимо будет предпринять некоторые меры, для минимизации потерь клиента. Для этого должен быть послан определенный сигнал брокеру, отвечающему за этого клиента. Поведение брокера в таком случае может быть оговорено предварительно. Разумным поведением видится отмена всех текущих открытых позиций данного клиента и попытка связаться с клиентом для разъяснения ситуации.

VI. Детали реализации.

Рассмотрим диаграмму, представленную на рис 1.

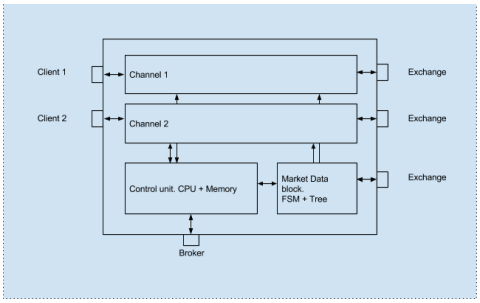


Рис 1. Устройство в целом.

На рисунке обозначены:

1. Блок управления, инициализации и мониторинга.

Представляет собой фирменную реализацию CPU от Xilinx — MicroBlaze [14]. Это 32-разрядный процессор с максимальной тактовой частотой около 200 МГц (для выбранной платы) и гибкой системой команд на основе IP core. Состоит из собственно процессора, блока памяти DDR3, коммутаторов, регистров. Процессор исполняет монолитную программу, которая обслуживает один Ethernet порт, через который представитель брокера может связываться с устройством для настройки и мониторинга. Блок управления реализован как процессор с программной логикой, потому что скорость его работы не критична (это не основной путь прохождения торговых приказов), а логика работы достаточно сложна, чтобы полностью ее реализовывать аппаратно. Он обеспечивает связь брокера с отдельными каналами клиентов, их настройку, а также выдачу сигналов об исключительных ситуациях (нарушениях правил торговли). Один блок управления может обслуживать несколько каналов клиентов, ограничение существует скорее только по производительности процессора. В том числе в задачу этого блока входит ведение журнала событий в системе с записью их на карту памяти (Secure Digital Memory Card, SD).

Блок управления связан со всеми каналами шиной управления, адреса и данных. Каждый канал может выставлять свой собственный сигнал прерывания. Блок управления обрабатывает эти сигналы прерывания и читает регистры соответствующего канала для детализации информации о прерывании.

2. Блок получения актуальных данных о ценах (market data MD).

Также имеет один порт Ethernet, но этот порт подключен уже к бирже для получения актуальных

котировок. Реализован как конечный автомат (finite state mashine, FSM), содержит также хранилище котировок по выбранным инструментам в виде красно-черного дерева, имеет свою собственную память, а также очередь запросов от каналов клиентов на обновление котировок. Посылку сообщений для подписки на прием MD осуществляется с помощью программного блока, поскольку сам процесс подписки не критичен по времени.

3. Несколько пользовательских каналов.

Каждый канал имеет два скоростных порта Ethernet в виде SFP+ модуля, к одному порту подключается клиент, второй порт подключается к коммутатору биржи. Каждый блок имеет связь с блоком управления через шину управления / обмена, а также связь с блоком MD. Подход к разбору сообщений MD рассмотрен в [15] и не представляет особого интереса в данном проекте.

Наиболее интересным является канал, именно он отвечает за обслуживание одного клиента см. рис 2.

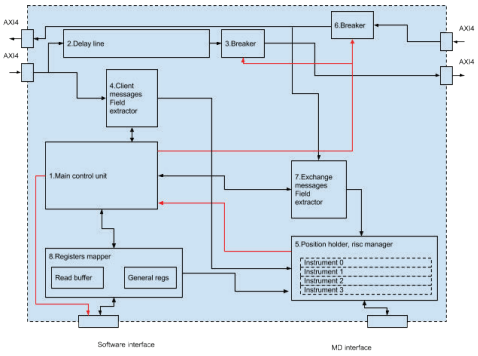


Рис 2. Один клиентский канал

Основные блоки:

1. Основной блок управления канала. (Main control unit) Реализован в виде конечного автомата. Выдает управляющие сигналы на все блоки, принимает решение о возбуждении прерываний, а также о прерывании сессии клиента.

2. Линия задержки (Delay line) 64xN, где N — подбираемый параметр. От него напрямую зависит задержка при передаче ордера. Как видно из диаграммы, задержка вносится только в канал передачи от клиента к бирже, обратный канал такой задержки не имеет. Это связано с тем, что прервать передачу мы можем только в ответ на неправильное поведение клиента. Поведение биржи по определению считается корректным.

3. Прерыватель (Breaker) именно этот модуль прекращает передачу по сигналу от центрального автомата.

4. Экстрактор полей клиентских сообщений (Client messages field extractor). Этот блок отвечает за разбор стека протоколов, выделение “чистого” потока данных. Он же определяет, что передача данных идет только в одной TCP сессии. Определяет, что данное сообщение должно быть разобрано и обработано дальнейшими блоками. Это легко, поскольку тип сообщений в протоколе Native LSE кодируется одним байтом в заголовке. В случае, если в конце приема данного пакета MAC выдаст сигнал, что пакет был принят с ошибкой контрольной суммы, пакет игнорируется, пропускается на сторону биржи, чтобы он также был отвергнут. Поскольку передается "сырой" поток и устройство не вмешивается в него, пока не нарушаются правила.

5. Держатель позиций (Position holder) По поступившей информации от экстрактора полей клиентских сообщений и от экстрактора полей биржевых сообщений подсчитывается размер доступных средств по каждому из настроенных инструментов. Этот блок также запрашивает цену последней сделки у блока обработки MD. Цены кэшируются, поскольку ответ может запаздывать. Если в установленное время блок MD не ответил, используется предыдущая (закэшированная цена по этому инструменту, если она не 0) Также держатель позиций подсчитывает общий доступный объем по всему портфелю и проверяет, что все условия соблюдаются и не выходят за указанные границы. При нарушении — выдает сигнал, по которому блок управления начинает разрыв соединения.

6. Прерыватель обратного потока (Breaker). Срабатывает одновременно с клиентским прерывателем.

7. Экстрактор полей сообщений биржи (Exchange messages field extractor). Необходим для извлечения ответных сообщений со стороны биржи.

8. Блок чтения-записи регистров (Register mapper). Позволяет записывать информацию с шины данных в адресуемые регистры, а также читать при необходимости состояние устройства.

Словари для сообщений от клиента и от биржи отличаются, поскольку в сторону биржи и в сторону клиента курсируют разные наборы сообщений.

На диаграмме обозначены связи между блоками — черным цветом — передача данных и сигналов управления, красным — сигнал прерывания (аппаратного на блоки прерывателей и программного — на интерфейс с программным блоком).

Поскольку канал не может инициировать передачу любых пакетов, как в сторону клиента, так и в сторону биржи, то есть необходимость решить проблему с потерянными пакетами и пакетами, пришедшими в неправильном порядке в пределах одной сессии TCP.

Рассмотрим места, где пакеты могли бы теряться (см. рис. 3).

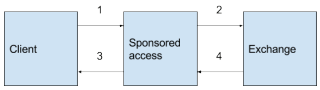


Рис 3. Варианты потерь пакетов при передаче.

1. С пакетом, который потерялся в этом месте все совершенно нормально. Плата этого пакета “не видела”, шлюз — тоже. Потеря будет обнаружена позже, стек TCP/IP со стороны биржи запросит повтор пропущенного пакета. Тогда его “увидит” и это устройство, а после передачи этого пакета дальше — также и биржа.
2. Пакет, потерянный в этом месте уже был обработан устройством, но после запроса со стороны биржи он будет повторен и пройдет через плату дважды. Для этого мы должны обнаруживать повтор такой передачи и игнорировать повторно передаваемый пакет.
3. Здесь ситуация симметрична 1, разве что инициатором передачи становится биржа. Обработка не требуется. Стек TCP/IP клиента должно обнаружить пропуск пакета и запросить повторную передачу, тогда устройство и “увидит” этот пакет.
4. Данная ситуация аналогична 2, действуют те же самые соображения.

В данной реализации устройства сделано одно упрощение — оно может работать только с корректной TCP сессией. Принадлежность Ethernet фрейма именно к текущей сессии определяется по комбинации всех значений: адреса-источника, адреса-приемника, порт-источника, порт-приемника. Эта четверка параметров захватывается при начале TCP сессии клиентом и в дальнейшем, от клиента требуется корректная поддержка установки и прерывания сессии. Фреймы, содержащие пакеты, относящиеся к другим протоколам UDP, ICMP и т.д. пропускаются беспрепятственно. Решение данной проблемы вполне возможно, но потребует расхода аппаратуры на дублирование некоторых блоков, которые обрабатывают входной / выходной поток.

Следующей проблемой, которую пришлось решать при опережающей обработке сырого потока данных это то, что сообщение, из которого извлекаются поля для обработки может оказаться в “битом” пакете. Это означает, что когда приходит сигнал о том, что текущий пакет принят полностью (а одновременно это означает, что его начало уже передается в сторону биржи) все извлеченные данные из этого пакета могут быть неверны. Значит, все поля, которые были извлечены конкретно из этого пакета должны быть сброшены и ожидать, когда данный пакет будет передан повторно.

Когда сообщение полностью разобрано и пришел сигнал подтверждения, что пакет был исправен, извлеченные поля передаются в держатель позиций. Он направляет внутри себя на один из равноправных блоков построителей книжек. Количество таких блоков параметризуется внутри проекта. Каждый из блоков представляет собой конечный автомат, диаграмма его состояний соответствует схеме обработки сообщений [13] (см. рис 4)

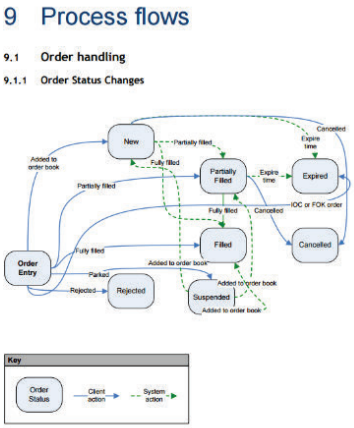


Рис 4. Схема обработки сообщений.

Все сигналы об всех событиях собираются со всех блоков в блок управления. Он, в зависимости от состояния регистра маскирования прерываний, решает — нужно ли запросить прерывание процессора для обслуживания, например для записи в журнал событий. Если же приходит сигнал от блока проверки правил торговли о нарушении какого-то правила, то блок управления вырабатывает управляющий сигнал на прерыватели потока, а также возбуждает прерывание.

Программный блок управления обслуживает сигналы прерываний от всех каналов, собирает по значения всех регистров на момент прерывания и в зависимости от типа прерывания либо просто записывает эту информацию на SD-карту, либо отправляет по резервному каналу сообщение, которое обрабатывается на стороне брокера. Для этого разработана библиотека на языке Java, которая предоставляет API для программной настройки устройства, мониторинга и сигнализации о событиях.

VII. Достигнутые результаты и их анализ.

Проект был реализован на языке SystemVerilog [16] для отладочной платы Xilinx KC-705 [12]. Для подключения четырех дополнительных SFP+ модулей была установлена плата QUAD SFP/SFP+ FMC105 [17], а также в целях отладки установлена Debug Mezzanine Card XM105 [18], на

которую выводились некоторые сигналы для внутрисхемной отладки и замеров задержки прохождения пакетов во время прогона тестов.

Использовались SFP+ модули FTLC8571D3BCL [19] с оптическими патч-кордами и связью с сетевой платой Intel X520-DA2 [20], также с оптическими SFP+ модулями.

Для подключения канала market-data использовался SFP+ слот, уже присутствующий на отладочной плате. Подключение со стороны брокера производилось через 1G ethernet разъем, также присутствующем на плате изначально.

Проект занял примерно 53% ресурсов при реализации каналов для двух клиентов. Утилизация DSP менее 3%. Была задействована внешняя память DDR3 для нужд программного ядра MicroBlaze размером 128 MB.

Для измерения получившейся задержки было выведено два сигнала “Last” от каждого из MAC — входного и выходного. Поскольку длина сообщения не изменяется, как и содержимое сообщения, то разница по времени между этими сигналами и будет искомая величина.

Задержка составила 194 нс с нулевым разбросом, как ожидалось по результатам предварительного моделирования.

Задержка не меняется для разных типов сообщений, поскольку она определяется только глубиной сдвигового регистра плюс небольшие задержки в модулях PHY & MAC. Мы получили стабильную работу при глубине сдвигового регистра N=30.

Окончательное тестирование производилось с помощь специального тестового ПО, разработанного в компании EXACTPRO — Sailfish [21]. Тестовые сценарии работали против тестового окружения площадки MIT [22] Это позволило протестировать корректную работу платы как в “прозрачном” режиме, когда клиент не нарушал контролируемых правил, а также в режиме срабатывания “предохранителя”, когда блок проверки правил обнаруживал такие нарушения.

Поскольку сервер с тестовым окружением недоступен физически, а также активно используется в других проектах, включение сетевых карт на него с оптическими коннекторами не представлялось возможным. Эта проблема была решена таким образом: компьютер, на котором было запущен Sailfish и к сетевой плате которого были подключены и вход и выход устройства SA посредством оптических кабелей, был настроен в режиме "бриджа". Пакеты с одного сетевого порта (к которому был подключен выход канала клиента SA) перенаправлялись на другой сетевой порт, уже подключенный в локальную сеть, соединенную с сервером тестового окружения. Sailfish подключался через тот сетевой порт к которому был подключен вход SA. В результате все пакеты проходили через устройство SA и в прямом и в обратном направлении.

Тестирование устройства производилось как "серого ящика". Т.е. тестовые данные подготавливались с использованием знаний о работе внутренних механизмов устройства. Это позволяет протестировать работу устройства с лучшим покрытием тестами, чем при тестировании его как "черного ящика". Подготавливая тесты для "серого ящика" можно заставить сработать все цепи во всех возможных режимах, обеспечивая максимальное покрытие тестами. Особое внимание уделяется данным, которые разбиты по соседним пакетам, а также обработке сбойных ситуаций (потери пакетов, нарушения очередности доставки, несколько TCP сессий для торговли, нарушений правил RM). Рассматривались ситуации, когда один клиент нарушает правила торговли и это не должно повредить другому клиенту, например. Отдельно рассматривается корректная работа после восстановления обрыва. Т.е. когда RM сработал и прервал соединение клиента, выдал оповещение на стороне брокера, то, после устранения причин нарушений и разрешения брокером соединения этого клиента, соединение может быть установлено снова.

В настоящее время тестирование продолжается. В планах - произвести тестирование устройства под нагрузкой, но для этого необходимо, чтобы соединение осуществлялось непосредственно с сервером, на котором запущена тестовая торговая платформа. Пока с этим есть некоторые проблемы.

VIII. Выводы.

Примененный подход позволяет действительно достигать очень низких значений задержек, значительно ниже, чем при использовании TCP/IP offload, выводя HFT на новый уровень. И, несмотря на то, что примененная стратегия управления рисками (Risk management, RM) выглядит слишком простой, этот проект позволил проверить применимость подхода и показал его расширяемость, в том числе в сторону усложнения RM. Ресурсы современных чипов FPGA позволяют это реализовать полностью. Особенно, если брать семейства, в которых уже включены ядра ARM-совместимых процессоров с повышенной (относительно MicroBlaze) производительностью, например Zinq-7000 [23]

Существенным подспорьем для решения таких задач является появление, особенно в последнее время, библиотек готовых компонентов в виде IP ядер для решения широкого круга задач — от реализаций стеков TCP и/или TCP&UDP до готовых блоков кодирования/декодирования сообщений распространенных протоколов — FIX, FastFIX, ITCH, Native etc. [24][11][25][26]. Кроме проприетарных решений в последнее время появляются и решения с открытым исходным кодом, что позволяет входить на этот рынок не только компаниям, но и отдельным разработчикам.

References

[1] John W. Lockwood, Adwait Gupte, Nishit Mehta, “A Low-Latency Library in FPGA Hardware for High-Frequency Trading (HFT)”, IEEE 20th Annual Symposium on High-Performance Interconnects, 2012

[2] A. Group, “High Frequency Trading in the Futures Markets”, 2010.

[3] Christian Leber, Benjamin Geib, “High Frequency Trading Acceleration using FPGAs”, 21st International Conference on Field Programmable Logic and Applications (FPL 2011), 2011.

[4] J.A. Brogaard, “High Frequency Trading and its Impact on Market Quality,” 5th Annual Conference on Empirical Legal Studies, 2010.

[5] S. Larsen and P. Sarangam, “Architectural Breakdown of End-to-End Latency in a TCP/IP Network,” International Journal of Parallel Programming, Springer, 2009.

[6] NetFPGA project http://netfpga.org

[7] Myricon products, https://www.myricom.com/products/network-adapters.html

[8] H. Subramoni, F. Petrini, V. Agarwal, and D. Pasetto, “Streaming, lowlatency communication in on-line trading systems,” International Symposium on Parallel & Distributed Processing, Workshops (IPDPSW), 2010.

[9] HFT Risk Gateway FPGA based solution, http://www.gatelab.com/products/hft.htm

[10] Ullink, http://www.ullink.com/

[11] nxTCP for Xilinx, http://www.enyx.com/nxtcp-xilinx/

[12] Xilinx Kintex-7 FPGA KC705 Evaluation Kit, http://www.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html

[13] Native Trading Gateway 10.1 Protocol Specification, London Stock Exchange (LSE), http://www.londonstockexchange.com/products-and-services/millennium-exchange/millennium-exchange-migration/mit203v101.pdf

[14] MicroBlaze Processor Reference Guide Embedded Development Kit EDK 11.4, http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/mb_ref_guide.pdf

[15] G.W. Morris, D.B. Thomas, and W. Luk, “FPGA Accelerated LowLatency Market Data Feed Processing,” 2009 17th IEEE Symposium on High Performance Interconnects, Aug. 2009.

[16] IEEE Standard 1800™-2012 SystemVerilog LRM, http://standards.ieee.org/getieee/1800/download/1800-2012.pdf

[17] FMC QUAD SFP/SFP+ for GbE/10GbE FMC105, http://www.vadatech.com/media/FMC105_FMC105-Data%20Sheet.pdf

[18] FMC Debug Mezzanine Card XM105, http://www.xilinx.com/support/documentation/boards_and_kits/ug537.pdf

[19] RoHS-6 Compliant 10Gb/s 850nm Multimode Datacom SFP+ Transceiver, https://www.finisar.com/sites/default/files/downloads/ftlx8571d3bcl_10gbs_850nm_multimode_datacom_sfp-plus_transceiver_product_specification_revd.pdf

[20] Intel® Ethernet Converged Network Adapter X520-DA2, http://ark.intel.com/ru/products/39776/Intel-Ethernet-Converged-Network-Adapter-X520-DA2

[21] EXACTPRO SailFish testing tool, http://www.exactprosystems.com/testtools/sf

[22] MillenniumIT, http://www.millenniumit.com/

[23] Zynq-7000 All Programmable SoC, http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[24] Algo-Logic Systems, http://algo-logic.com/lowlatency

[25] R. Mueller, J. Teubner and G. Alonso, “Streams on wires: a query compiler for FPGAs”, Proc. VLDB Endowment, vol. 2, no. 1, 2009

[26] Accelize, http://www.accelize.com/products/fpga-platform-components/finance-ip/market-data-decoders.html

[27] MIT 303 Level 2 — MITCH Specification, London Stock Exchange (LSE), http://www.londonstockexchange.com/products-and-services/millennium-exchange/millennium-exchange-migration/mit303.pdf

Reference test harness for algorithmic trading platforms

Victoria Leonchik
Exactpro Systems
victoria.leonchik@exactprosystems.com

Alexey Sukhov
Exactpro Systems
alexey.sukhov@exactprosystems.com

Eugene Ushakov
Exactpro Systems
eugene.ushakov@exactprosystems.com

Iosif Itkin
Exactpro, LSEG
iosif.itkin@exactpro.com

Anna-Maria Lukina
Exactpro Systems
anmay@exactpro.com

***Abstract*—The safety and stability of algorithmic trading software is an ongoing concern for exchanges, market participants and the society in general. Financial regulators worldwide are trying to create effective rules to prevent self-enforced market volatility and technology crashes caused by computer-aided trading. Specifying relevant requirements for dynamic software verification of algorithmic trading platforms remains is an on-going task. Yet, there has been little progress to date in locating efficient and commonly accepted approaches. This paper introduces a reference test harness implementation for algo trading platforms created by the authors.**

***Keywords*— trading, algorithm, strategy, testing.**

I. Introduction

The days of open outcry trading and trading pits are almost gone replaced by a new type of trading - algorithmic or electronic trading. Algorithmic trading is the process of using computers which execute a defined set of instructions to place orders to generate profits with speed and frequency impossible for a human to achieve [1].

Algorithmic trading became very popular during the last decade - about 40% of all financial operations are based on algorithms. Algo-trading by its nature is black box trading and there are still lots of concerns and question marks around this topic.

Algorithmic trading has received substantial attention from the society following high profile events, such as Flash Crash in 2010 when American indices (S&P 500, Dow Jones and Nasdaq 100) collapsed and recovered very rapidly [2,3], and the Knight Capital runaway algo disaster in 2012 when the firm lost $450 million in 45 minutes [4,5,6].

Regulators have started to recognize the value of algorithmic trading in the market place, but there is still concern about its safety and market risks caused by rogue algorithms. Regulators are trying to create rules and obligations regarding algorithm building, testing and deployment which could help prevent financial disasters and ensure that trading algorithms are safe and reliable.

The next section of this paper describes the role of algorithmic trading and associated legal framework. The third section introduces a reference test harness for algotrading platforms created by the authors. Section four drills into auxiliary test algo types targeted at modelling realistic market microstructure. The last section describes metrics and characteristics measured during algo trading platforms testing.

II. The role of Algorithmic Trading for Financial Market Rules and Regulations

The SEC (The United States Securities and Exchange Commission) started to formulate new rules back in 2013 and more recently FINRA published a list of suggested effective practices for firms engaging in algorithmic strategies [7]. An essential component of effective policies and procedures is testing of algorithmic strategies prior to launching them in Production.

Furthermore, the Hong Kong regulator (SFC - Securities & Futures Commission) has also worked on creating efficient rules which could be applied during the algorithm certification process: “A licensed or registered person should ensure that the algorithmic trading system and trading algorithms it uses or provides to clients for use are adequately tested to ensure that they operate as designed” [8].

ESMA has published a discussion paper for MiFID II / MiFIR which will be effective from January 2017. It includes a detailed description of the algorithmic testing procedure: “An investment firm that engages in algorithmic trading shall have in place effective systems and risk controls suitable for

the business it operates to ensure that its trading systems are resilient and have sufficient capacity" [9].

The main requirements of the SEC and ESMA can be found in the table below. (Fig.1)

SEC	ESMA
• Conducting testing: confirmation that core code components operate as intended and do not produce unintended consequences	• Clearly delineated development and testing methodologies
• Quality Assurance process should be separated from any development work	• Testing methodologies should include performance simulations / back-testing and non-live testing within a trading venue testing environment
• Periodically evaluating test controls	• Ensure that tests are commensurate with the risks that this strategy may pose to itself and to the fair and orderly functioning of the markets operated by the trading venue
• Data integrity, accuracy and workflow validation	• Periodically evaluating test controls
• Recording of all testing protocols and results • Conducting all testing in a development environment that is segregated from production	• Investment firms should ensure that the production and testing environments are kept segregated at all times

Fig.*1* SEC and ESMA requirements

One of the essential parts of algorithm testing, based on regulator requirements, is specific testing in a non-live trading environment. This would allow those involved in the certification process to make a correct assessment of algorithm risk, profitability and efficiency.

III. Reference Test Harness

We would like to introduce our view on testing solutions which could be used for testing of different algorithms and could be easily adapted to any trading platform required. The reference test harness consists of the following components:

- Algorithmic trading platform under test;
- One or several matching engines acting as execution venue simulators;
- Competing test algorithms to simulate market impact;
- Passive testing tools to gether quality, performance and efficiency stats for the algo systems and strategies under test;
- Customized order entry and market data gateways;
- Market surveillance system;
- Auxiliary proxies to control test execution.

At the core of our solution we are using an exchange matching engine as a market simulator. The authors have tried several exchange systems, including those developed by LSEG Technology services divison companies - MillenniumIT and GATElab. Let consider a full multi asset class matching platform developed by GATElab – Exchangepath – 100µs.

It is a matching engine with proven efficiency and can be used as a full replacement for a live trading system engine. The major advantages of this matching engine are:

- 100,000+ transactions per second;
- 50,000+ market data notifications per second;
- Low start up and running costs;
- Low latency – less than 100µs at the end-user gateway [10].

These conditions give us an opportunity to place a trading algorithm under test in a Production-like environment and get results which are closer aligned to what we would expect in a Production environment. Trading algorithm under test can submit orders into the matching engine and receive back execution reports and related market data. It is possible to deploy several matching engines to simulate multiple markets.

To decrease test harness hardware footprint, the authors have introduced a replay of historical data known as backtesting. Backtesting is useful as it could demonstrate the efficiency of an algorithm from a historical point of view. On the other hand, quite often backtesting tells you very little about future profitability. Because of this, backtesting is both a blessing and a curse. Many portfolio managers use backtesting to prove that a strategy is viable, but they fail to evaluate a number of issues that might be missed during backtesting as general market sentiment cannot be predicted.

It is not possible to achieve 100% accuracy in the trading day replace when dealing with highload distributed systems [11]. As soon as a client algorithm is connected to the system it will have an impact on historical data and change it, which may lead to incorrect results in the evaluation of algorithm risk and productivity. That is why a trading firm should be concerned about returning historical data to its original state without proper compensation of strategy impact. Backtesting can produce positive results which can be too far from results achievable in real conditions. In other words, during algorithm testing we should take into account the impact of the algorithm on historical market data replay. Thus for better testing this impact should be suppressed - market data should be restored by using counter flow models.

A testing tool should create counter-flows in response to user-generated, non-historic submissions. The goal of any counter-flow model is to replicate sufficiently the reaction of the market on a user strategy through the generation of additional flow of events which would be united with modified historical and user-generated events. A counter-flow model has very complex logic and consumes a lot of hardware resources that leads to increased costs and testing time.

As an alternative to using the counter-flow model and overall testing process improvement we suggest to add four points in to the system which would be configurable for any system and could help compensate testing strategy impact (fig.2):

- Customized trading GWs;
- Customized Market Data GW;
- Latency and counterflow proxies;
- Simulated traders (Active Testing Tool):
  - *Arbitraging Minirobot*
  - *Minirobots emulating 'Slicing' algorithms*
  - *Minirobots emulating 'Synthetic' algorithms*
  - *Exchange simulated orders*
  - *Aggressive buyer/seller (Market panic scenarios)*
  - *"Bandit"-algos*

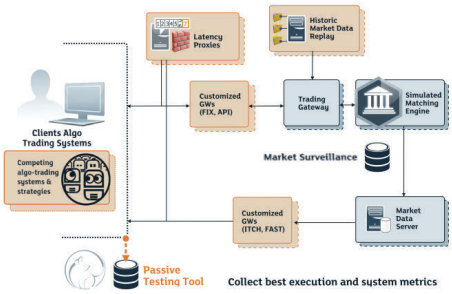


Fig.2 Test Harness for Algo trading system

Customized Trading Gateways and Market Data Gateway

In order to have an ability to put a client's algorithm in a disadvantaged position compared to another one which would be "faster", we can use customized Trading gateways. This would allow us to test the algorithm under different co-location conditions as we are able to control the latency of all messages sent or received by the client's algorithm.

Testing will answer the following question: how profitable an algorithm could be if outgoing/incoming messages have an acceptable time delay or what would happen to algorithm efficiency in case of increased latency?

The same customization could be applied to Market data gateways such as ITCH and FAST. It would give us an opportunity to evaluate the impact of different latencies in receiving market data messages on the performance of the trading algorithm.

Customized Trading gateways also allow us to change tags in sent/received client's messages. In this case we will be able to adjust our system to different clients who are using different sets of tags.

Surveillance system

In addition to testing efficiency of the trading algorithm we should take into account the involvement of control functions (such as Suveillance, Legal, Compliance, Controllers, Operations, etc.) as well as business objectives. These control measures must prevent disruptions to the fair and orderly functioning of the financial markets. Disruptions and incidents may cause damage for trader's activity, but potentially also for others traders too.

Despite the fact that an algorithm can show quite impressive results and make good profit, it does not mean that it can be used carelessly. Thus it makes sence to test the legality of algos. It may be worth to run it through Surveillance systems to make sure that trading firm will not have any issues with regulators in terms of basic market rules and the trading algorithm will not be considered abusive or disruptive [12].

Passive Testing Tool

While using highly complex algorithms it is important to store all the inbound and outbound data about all executed financial transactions and verify this data with the data of the client and the data in post trade. [13]

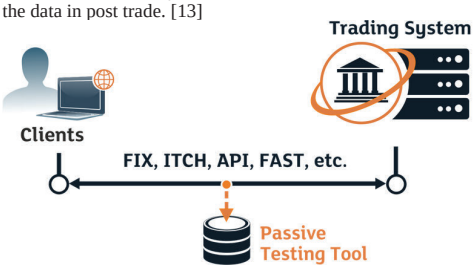


Fig.3 Passive Testing Tools in Trading Systems

Passive testing tools (Fig.3) are used for automated log collection, data structuring, monitoring, system behavior analysis and user certification [14]. Test tools allow analyzing high volume of data promptly, reacting to deviations in the system's behavior from requirements, and troubleshooting. (Fig. 4)

Item	Description
Testing Type	Passive Real-Time/Batch
Target SUT	Trading Platforms, Market Data Delivery and Post-Trade Systems
SUT Interface	Back-end (typically connected to message gateways / APIs, and DBs); GUI Testing Capabilities not supported
SUT Interaction Method	Inputs and outputs monitored by means of message capture and log parsing to analyze client activity and forecast system response; DB queries for data verification; files transfer, upload, export and comparison. Captured messages can be viewed and analyzed in real-time or post-factum

Protocols	Extant plug-ins for Industry-standard (FIX and dialects, FAST, SWIFT, ITCH, HTTP, SOAP, etc.) and Proprietary (MIT, SAIL, HSVF, RTF, RV, Reuters, Fidessa OA, Quant House, etc.) protocols. New plug-ins for additional protocols developed by request (codecs are shared between Sailfish and Shsha)
Test Scripts	Certification tests and data reconciliation may be performed by using ordinary SQL queries. Test message traffic generated in real-time or replayed from log files by other tool (e.g., Sailfish)
Test Management, Execution and Reporting	Integrated (Desktop front-end), allows for multiple simultaneous heterogeneous connections, consecutive execution of multiple planned scripts, test results summary and detailed test reports. Optional Big Button framework supported
Platform requirements	Low footprint cross-platform application, MySQL

Fig.4 Passive testing tool specification

Minirobots

Returning market data replay to its initial state can be achieved by introducing an arbitraging script formalized in one of the Minirobots tool which is to be used together with a tested algorithm in the same framework.

The Minirobots tool [10B], developed with the idea of simulating real traders' behavior in mind, is able to make decisions under specific market conditions in a common fashion, but at the same time has a certain degree of autonomy. Depending on what the testing needs are, each of the robots can act independently or jointly executing a particular trading strategy or simply replaying a stored list of orders. (Fig. 5)

Item	Description
Capacity & Precision	Hundreds – thousands of messages depending on the algorithm complexity. Millisecond precision
Testing Type	Active Multi-Participants (applicable for testing at the confluence of functional and non-functional testing)
Target SUT	Trading Platforms and Market Data Delivery Systems
SUT Interface	Back-end (typically connected to message gateways / APIs); GUI Testing Capabilities not supported
SUT Interaction Method	Message injection and capture to emulate multiple participants' activity in electronic markets (essential when there is a need to reproduce complex scenarios that can be created by trading algorithms)
Protocols	Extant plug-ins for Industry-standard (FIX and dialects, etc.) and proprietary protocols. New plug-ins for additional protocols developed by request
Test Scripts	Multi-threaded Java code specifying different liquidity profiles
Test Management, Execution and Reporting	Integrated (Web front-end), allows for multiple simultaneous heterogeneous connections, concurrent emulation of multiple participants, detailed test reports. Optional Big Button framework supported
Platform requirements	Written in Java

Fig.5 Minirobots specification

IV. Algo Test Agents Used to Simulate Liquidity and Market Impact

a. Arbitraging and Market Making Minirobots

Once arbitraging Minirobots has been implemented in the test environment, it starts sending contra-orders to add liquidity on a particular price level as soon as the order sent from the tested algorithm hits the 'market' bid or an offer. That is how replayed market data is restored to the realistic state, thereby reducing the potential impact of the tested algorithm on the event sequence of the analysed trading day. Arbitrage Minirobots are helpful when testing fragmented markets. They can move liquidity across simulated venues.

b. Minirobots emulating 'Slicing' algorithms

Another productive option is to deploy Minirobots tool emulating the behavior of market operator who is using 'Slicing' trading algorithms being widely spread at present among a variety of institutional investment funds. According to U.S. Commodity Futures Trading Commission [15] estimates, more than 90% of institutional traders use trading algorithms or other automated strategies to seek best execution for their clients. This is due to the fact that 'slicing' allows hugely sized trading orders to be executed by dividing them into small portions, which are then traded separately at different time during a trading day. This minimises the risk of orders being discovered and 'front-run' by competitive or 'dirty' traders.

The 'slicing' idea is most oftenly materialised in the usage of the so called 'Time Sliced' trading strategy. The Time Sliced algo splits an order into equal sized slices (an element of randomisation can be added too). The number of slices is determined by the start and end time and the duration between slices. Each slice is released to the market at intervals dictated by the duration between slices.

'VWAP' represents a more sophisticated implementation of a 'slicing' strategy. This algorithm is intended to manage the execution of an order in such way that it achieves the Volume Weighted Average Price (VWAP) for the order's instrument in the time period the trader has selected.

At a high level, the algorithm achieves this by releasing slices of the order at varying rates according to the analysis of historical market data. It uses additional execution logic to determine the pricing and expiry of each 'slice' created by the model.

The fact that Time Sliced/VWAP based order execution models is being popular in modern trading determines a necessity to use Minirobots tool simulating Time Sliced/VWAP trader together with a client's algorithm to understand its efficiency and competitive performance when trading is driven by someone else's automatic solution. [16,17]

c. Minirobots emulating 'Synthetic' algorithms

When it comes to better execution one can either use an automated solutions allowing the orders to be placed at the top of the market's queue. For instance, this is often a matter of immediate interest when there is a need to execute a trade at the very start of a trading day. Such algos, usually called 'Hammer', are attempting to send the order just before market open in order to reach the Exchange at the exact open time rather than wait to receive the market open signal. This is achieved by sending the order a few seconds before market open (where it will be rejected) and by continuously re-sending it until it is accepted by the Exchange. 'Hammer' algorithms behavior can also be mimicked by the Minirobots tool to see how the automated strategy tested interacts with it at the peak hours of a trading day.

If trading is circumstantial and the trader needs to place an order as a result of a particular price movement in an correlated instrument (or a group of correlated ones), another algo is used to release orders once the conditions pre-defined by a trader's logic are reached. This behavior simulated by the Minirobot tool would give a singular advantage where one wants to test 'pairs trading' or arbitraging algorithms.

d. Exchange simulated orders

Where a market data replay is recorded from the exchange that due to external reasons does not support natively some of the order types, there is always a possibility that stop, trailing stop, iceberg, ghost or market-if-touched orders are being simulated by someone else's automated solution. This in fact can affect the test results of a chosen algorithm and thus the Minirobot tool sending 'exchange simulated' orders might be another example of a more sophisticated testing approach.

e. Aggressive buyer/seller (Market panic scenarios)

Moreover, Minirobots who act like an aggressive buyer or seller can be deployed to the testing environment. With such a script implemented, it will be possible to re-create conditions of the so called 'panic' buying or selling in a particular derivative contract, underlying instrument, or a market composite index. Beyond all doubt market 'panic' is a once-off experience which may stimulate irrational price movements frequently spurring algo-traders to react irrationally and wildly. Such events a the well-known 'Flash Crash' are subject of anxiety due to a potential devastating impact on the financial stability of the stock or even the economy. Emulating this by using the Minirobots tool provides a benefit to assure that tested algorithm would not fail under stress market events. [18,19]

f. 'Bandit'-algos

With algo trading increasing in popularity in the financial markets there is a worrisome trend for anti-HFT or so called "Predatory algorithms" to be used by traders whose aim is to manipulate stock prices, forcing others to react to their benefit. Despite the MiFID I and Dodd-Frank legislative acts going into effect as long as 5 years ago, these illegal trading practices are still popular due to the fact that they are highly profitable and not always easy to detect, unfortunately. All these require us to introduce a 'bandit script' - having a 'predatory' logic - into the environment to see how it will affect the tested algorithm in terms of its performance characteristics.

Based on 'Proposed Guidance on Certain Manipulative and Deceptive Trading Practices' issued by Investment Industry Regulatory Organization of Canada (IIROC) [20,21] we might use the Minirobot tool to re-create the following abusive market behavior -

1) 'Layering'

A strategy which initiates a series of orders and trades (sometimes along with spreading false rumours in the marketplace) in an attempt to ignite a rapid price movement either up or down and induce others to trade at artificially high or low prices. An example is a "layering" strategy whereby a market participant places a bonafide order on one side of the market and simultaneously "layers" the book with non-bona fide orders on the other side of the market to bait other market participants into reacting to the non-bona fide orders and trade with the bonafide order.

2) 'Quote-stuffing'

The practice of placing an unusual number of buy or sell orders on a particular security and then immediately cancelling them to "flood" the trading systems with excessive market data messages. An objective may be to increase data latencies for marketplaces or other market participants in order to create "information arbitrage" opportunities.

3) 'Spoofing'

A practice when limit orders that are not intended to be executed are used to manipulate prices. Some strategies are related to the open or the close of regular market hours that involve distorting disseminated market imbalance indicators through the entry of non-bonafide orders, checking for the presence of an "iceberg" order, affecting a calculated opening price and/or an aggressive trading activity near the open or close for an improper purpose.

4) 'Abusive liquidity detection'

Large orders (disclosed or iceberg) are entered during the pre-open or employ "pinging" orders to detect the existence of a large buyer or seller with the intent to trade ahead of, rather than with, the large buyer or seller. After a profitable price movement, the trades are reversed, or if the price moves contrary to the position taken, the trading interest of the large buyer or seller may be viewed as a free option to trade against.

All in all, the Minirobots tool allows us to emulate any existing market participant behavior that can be used in client-to-algo or client-vs-algos mode within the same testing environment. This in fact offers a unique possibility to re-create Production-like conditions, and hence to assure that a tested algorithm is operating adequately with no real risk of losing money.

V. Algorithm efficiency and riskless criteria

The main purpose of trading algorithm testing is proving its efficiency and reliability. No one will use algorithms which only generate financial losses. There are many criteria which could be used for the evaluation of algo productivity.

Thus we decided to divide all criteria into two sections Technical and Business efficiency:

Technical Efficiency Criteria

This section is based on ISTQB classification [22].

Functional criteria

The ability of an algo to produce correct outputs for the inputs it receives according to specification. The less number of existing errors an algorithm contains, the lower the expectations of potential losses and fines are. The test harness developed by the authors parses log files to search for error and warning messages. It also analyses the consistency of the data collected by passive testing tools. Simulated matching engines and gateways reject incorrectly formatted messages and orders that fail to pass the risk controls.

Non-Functional criteria

1) Performance

How does a trading algorithm perform in terms of responsiveness and stability under a particular workloads (market data streams)? Answering this question helps Quality Assurance in understanding how an algo can cope with the number of data feeds in processes, the number of exchanges it trades on, and the types of securities it can trade. Data collected from the network capture is used to estimate internal latencies within the trading algo, including the time from market data updates to issuing orders into the market.

2) Scalability

This is the capability of an algorithm to continue functioning well under a growing amount of work, or its potential when the algo is provided with more resources (hardware mostly) in order to accommodate that growth or to meet a user need. The presence of the scalable matching engines as simulated markets allows running scalability tests for algorithmic trading platforms.

3) Reliability

The ease of an algo to perform its required functions well for a specified period of time under different specific test conditions or for a specified number of operations. Test harness automation enables us to repeat the tests many times to check the systems reliability.

4) Efficiency

The capability of an algorithm to provide appropriate performance under stated conditions, relative to the amount of resources used. Hardware metrics are collected from the trading platform to test its technical efficiency.

5) Maintainability

This is the ease with which an algorithm can be modified to correct defects, modified to meet new requirements (e.g. market conditions, regulatory acts changes), or modified to make future maintenance easier. The Knigh Capital case shows that the maintability and the ability to monitor the system is what prevents a problem from turning into a disaster.

6) Recoverability

This is the capability of an algorithm to re-establish a specified level of performance and recover the data directly affected in case of failure. Failover and recovery tests should be included into the systems testing scope.

Business Efficiency Criteria

Achieving the best possible trading price does not guarantee that the algorithm will make a profit. Best execution is a wider definition [23]. Depending on the underlying trade or investment idea on which a trading algorithm has been built, best execution includes taking all appropriate steps to achieve the best possible outcome along several dimensions. These dimensions are:

- Price (execution, price improvement, spread capture);
- Cost (explicit, market impact, adverse selection);
- Probability of execution;
- Liquidity and volatility.

There are a number of common execution performance criteria that the trading algorithm is characterized by. The most general and relevant criteria used to measure execution performance is Implementation Shortfall (IS). This approach has become an industry standard as it captures the difference between the price that an algo decided to trade and the final execution price (including commissions, taxes, etc.) for a trade. This is also known as “slippage”.

At the level of algorithmic strategy, best execution is achieved by balancing multiple conflicting goals such as best trade price, minimal market impact, optimal time and liquidity allocation, and highest possible completion rate [24].

Passive test tools implemented as part of the test harness allow capturing business efficiency parameters for every test execution and storing them into the database for regression analysis. It is necessary to repeat every test many times as real markets are not deterministic and good simulated markets are not deterministic either. Every particular test run will lead to a slightly different result. Queries executed against captured data give us the necessary analytics on the systems’ behavior and enable comparison between various versions of the systems under test.

VI. Conclusion

In this paper we have introduced our view on the development of a trading platform simulator which could be used for testing of trading algorithms and strategies. The main advantages of our approach are flexibility and simple tuning of simulator configuration depending on testing purposes.

We are planning to expand our system and develop an automated solution which could restart and change the simulator's parameters and, furthermore, collect and aggregate statistics and logs after each testing cycle. It would make testing more efficient, adjustable and understandable for clients.

We believe that the financial regulators will pay much more attention to algorithmic trades and the importance of testing trading algorithms in the future. Therefore, trading simulator solutions will become more and more popular and will be in higher demand as time goes on.

References

[1] Gov.UK, *Future of computer trading in financial markets: an international perspective,2012,* https://www.gov.uk/government/publications/future-of-computer-trading-in-financial-markets-an-international-perspective

[2] U.S. Commodity Futures Trading Commission, U.S. Securities & Exchange Commission, *Findings Regarding the Market Events of May 6, 2010*, Report of the Staffs of the CFTC and SEC to the Joint Advisory Committee on Emerging Regulatory Issues, https://www.sec.gov/news/studies/2010/marketevents-report.pdf

[3] E. Wes Bethel, D. Leinweber, O. Rübel, K. Wu, *Federal market information technology in the post flash crash era: roles for supercomputing*, WHPCF '11: Proceedings of the fourth workshop on High performance computational finance, 2011.

[4] SEC Release No. 70694, *In the Matter of Knight Capital Americas LLC*, 2013, https://www.sec.gov/litigation/admin/2013/34-70694.pdf

[5] A. Kriger, A. Pochukalina, V. Isaev, *Reconciliation Testing Aspects of Trading Systems Software Failures*. Preliminary Proceedings of the 8th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2014), ISBN 978-5-91474-020-4: 125 p. , 2014, http://syrcose.ispras.ru/2014/files/SYRCoSE2014_Proceedings.pdf

[6] G.Baxter, J. Cartlidge, *Flying by the seat of their pants: what can high frequency trading learn from aviation?* ATACCS '13: Proceedings of the 3rd International Conference on Application and Theory of Automation in Command and Control Systems, 2013.

[7] Regulatory Notice 15-09, *Equity Trading Initiatives:Supervision and Control Practices for Algorithmic Trading Strategies,* https://www.finra.org/sites/default/files/notice_doc_file_ref/Notice_Regulatory_15-09.pdf

[8] Securities and futures commission, *Code of conduct persons licensed by or registered with the securities and futures commission,2013,* http://en-rules.sfc.hk/net_file_store/new_rulebooks/h/k/HKSFC3527_1868_VER50.pdf 18.10

[9] The European Securities and Markets Authority, *MiFID/MIFIR Discussion Paper*, http://www.esma.europa.eu/system/files/2014-548_discussion_paper_mifid-mifir.pdf

[10] GATElab, *The Dive Deep into Liquidity Pools*, http://www.gatelab.com/products/matching.htm

[11]P. Protsenko, A. Khristenok, A. Lukina, A. Alexeenko, T. Pavlyuk, I. Itkin, *Trading Day Logs Replay Limitationas and Test Tools Applicability*, TMPA'2014: Proceedings of Annual International Workshop-Conference Tools & Methods of Program Analysis (TMPA), 2014

[12] U.S. Securities and Exchange Commission, *Regulatory Actions*, https://www.sec.gov/rules.shtml

[13] A.A. Averina, N.A. Antonov, I.L. Itkin, *Special features of testing tools applicable for use in trading systems production*, TMPA'13: Proceedings of Annual IEEE/ACM International Workshop-Conference Tools & Methods of Program Analysis (TMPA), 2013

[14]A. Alexeenko, A. Matveeva, D. Sharov, P. Protsenko, I. Itkin, *Compatibility Testing of Clients Protocol Connectivity to Exchange and Broker Systems*, TMPA'13: Proceedings of Annual IEEE/ACM International Workshop-Conference Tools & Methods of Program Analysis (TMPA), 2013

[15] U.S. Commodity futures trading commission,http://www.cftc.gov/index.htm

[16] A. Madhavan, *VWAP Strategies, 2002*, http://itg.com/news_events/papers/TP_Spring_2002_Madhavan.pdf

[17] J. Bialkowski, S. Darolles, G. Fol, *Improving VWAP strategies: A dynamical volume approach, 2006*, http://www.ir.canterbury.ac.nz/bitstream/10092/4534/1/12624569_VWAP_2310_2006_for_JBF.pdf

[18] O. Steinki, *Business School, Algorithmoc Trading*, http://evolutiq.com/wp-content/uploads/2014/03/Algo-Trading-Intro-2013-Steinki-Session-8.pdf

[19] Fidessa, *Advanced trading tools*, http://www.fidessa.com/products/sell-side-solutions/advanced-trading-tools

[20] Compliance: Theory and Practice in the Financial Services Industry,*Market Conduct Rules*,http://www.inhouselegal.com.au/Compliance_Course/lecture_4.htm

[21] IIROC Notice 12-0221, *Rules Notice, 2012,* http://www.iiroc.ca/Documents/2012/f62c746a-b5c9-448a-b57f-f1c04c88de14_en.pdf

[22] ISTQB Glossary, http://astqb.org/glossary/

[23] Trader Planet, Trading 101: The M&Ms for Successful Trading, http://www.traderplanet.com/tutorials/view/161657-trading-101-the-m-amp-ms-for-successful-trading/

[24] Trader Planet, *What Traders Need to Know: Best Execution*, http://www.traderplanet.com/articles/view/162821-what-traders-need-to-know-best-execution/

Инструмент для автоматизированного тестирования систем проведения расчетов и клиринга ClearTH

Анна Торопова
Exactpro Systems
anna.toropova@exactprosystems.com

Екатерина Димова
Exactpro Systems
ekaterina.dimova@exactprosystems.com

Иосиф Иткин
Exactpro Systems
iosif.itkin@exactpro.com

***Аннотация*—Системы, обеспечивающие проведение расчетов и клиринга, называемые также post trade, составляют существенную часть инфраструктуры финансовых рынков. Эти системы ответственны за большую часть доходов и расходов участников электронной торговли. Подобно торговым платформам, отвечающим за сведение транзакций в реальном режиме времени, большинство платформ для проведения расчетов и клиринга представляют собой распределенный диверсифицированный высоконагруженный набор взаимодействующих подсистем. Однако, системы для проведения расчетов и клиринга также характеризуются дополнительным набором свойств. Структура данных в таких системах часто приводит к невозможности проверки корректности их функционирования с помощью индивидуальных независимых сценариев тестирования. В статье рассматриваются особенности систем для проведения расчетов и клиринга и анализируется применимость существующих инструментов по автоматизации тестирования для их динамической верификации. Авторы описывают разработанный ими инструмент ClearTH, позволяющий структурировать библиотеку тестов в соответствии с расписанием операционного дня/цикла систем для проведения расчетов и клиринга. Описанный подход может быть применен и к другим типам систем, обладающим аналогичными свойствами.**

***Ключевые слова*— инструменты тестирования, автоматизация тестирования, ClearTH, клиринг, расчеты, расчетно-клиринговые системы, обеспечение качества**

I. Введение

В современной финансовой индустрии торговые и расчетно-клиринговые системы являются двумя зависящими друг от друга компонентами, которые беспрерывно взаимодействуют между собой. Торговые системы отвечают за заключение сделки, в то время как расчетно-клиринговые системы отвечают за успешный перевод денег и ценных бумаг, а также обеспечивают анонимность сделок, своевременное проведение корпоративных действий и многое другое[1,2,3,4]. Проблема обеспечения надёжности и качества программного обеспечения является одной из ключевых при разработке расчетно-клиринговых платформ, и решения, обеспечивающие качество систем, становятся все более востребованными [5]. Любая ошибка в программном обеспечении является дорогостоящей, а любая задержка с обнаружением дефекта вносит дополнительные затраты, как времени, так и финансов.

Заказчики программного обеспечения хотят быть уверенными, что новые изменения, внесенные в разрабатываемую платформу, не будут негативно сказываться на качестве, а наоборот, расширят функциональность продукта, что поспособствует увеличению в будущем количества клиентов, использующих данную платформу. При тестировании ставится цель максимально точно имитировать реальную активность системы и пользователей в тестовой среде. Использование тестовых инструментов позволяет быстро и точно обнаружить проблемный участок программного кода, обеспечив полное покрытие всей ключевой функциональности тестовыми сценариями.

Во второй части статьи описывается усредненная расчетно-клиринговая система, выделяются её особенности и отличия от торговых платформ. В третьем разделе проводится анализ распространенных тестовых подходов и описание тестового подхода, который наиболее целесообразно использовать при тестировании систем проведения расчетов и клиринга. Четвертая часть содержит статистический анализ использования тестовых инструментов на основе ресурсов, посвященных тестированию и разработке программного обеспечения, и статистике поисковых запросов, а также сравнение с наиболее известными тестовыми инструментами. Созданный инструмент и его особенности описаны в пятой части статьи.

II. Особенности систем проведения расчетов и клиринга

Системы проведения расчетов и клиринга являются важной частью современной инфраструктуры финансовых

рынков, но их архитектура, как правило, не рассматривается в научных публикациях. На Рис. 1 демонстрируется абстрактная внутренняя структура расчетно-клиринговых систем, релевантная для европейских и американских рынков[6,7,8], а также отображаются связи с внешними системами [9], с которыми, как правило, происходит взаимодействие в процессе клиринга или проведения расчетов.

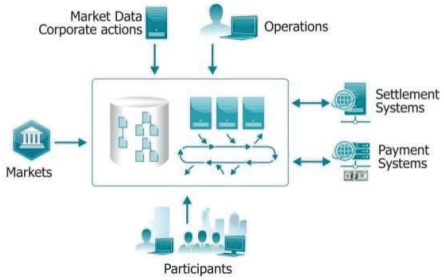


Рис. 1. Схематичное представление расчетно-клиринговой системы

Количество внешних систем, взаимодействующих с системами расчета и клиринга, может варьироваться в зависимости от особенностей конфигурации окружения[10,11], от структуры самой организации, занимающейся расчетами и клирингом, а также от законодательства страны, в которой происходят данные процессы. Однако, некая усредненная система для проведения расчетов и клиринга, схема которой представлена на Рис. 1, чаще всего взаимодействует с рынками (Markets), от которых она получает информацию о совершенных сделках (Trades, deals), с пользователями системы (Participants), с операторами (Operations), которые обеспечивают поддержку системы на том или ином уровне, с внешними расчетными (Settlement Systems) и/или платежными системами (Payment Systems) и получает рыночные данные о внешних событиях (Market Data, Corporate Actions). Связь может осуществляться через различные протоколы: MQ[12] (менеджер очередей сообщений), файловые[13] (FTP, SFTP), бинарные и многие другие. За счет этого система постоянно должна оперировать большим объемом данных[14], накопленных за разные операционные дни.

Сама же система, как правило, основывается на двух элементах: справочных данных (Reference Data/Static Data) и системном расписании (Schedule).

Справочные данные настраиваются под нужды определенных пользователей и впоследствии оказывают большое влияние на все процессы в системе. Как правило, из-за большого количества процессов в системах проведения расчета и клиринга, объем справочных данных в несколько раз превышает их количество в торговых платформах. Объем статичных данных пропорционален количеству пользователей и инструментов в системе, но они не меняются так активно, как индивидуальные транзакции, количество которых может быть не ограничено. Преимущественно в системе хранятся такие данные, как параметры рынков, информация о инструментах и пользователях, различные календари и справочные таблицы, обеспечивающие жизнедеятельность системы.

Системное расписание определяет жизненный цикл объектов в системе (как правило, он может меняться от системы к системе, но обычно длится от нескольких дней до месяцев), а также порядок процессов в течение операционного дня.

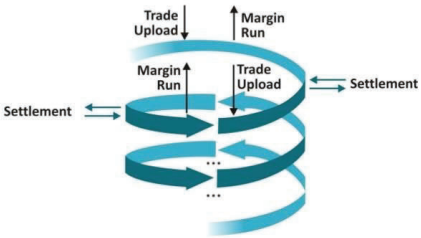


Рис. 2. Жизненный цикл системы расчета и клиринга

Как видно на Рис. 2, каждый день в расчетно-клиринговой системе происходит множество разнообразных процессов: загрузка сделок (trade upload), подсчет рисков (margin computations, margin runs), расчетные сессии (settlement sessions, settlement runs), загрузка залогового обеспечения (collateral uploads) и многие другие.

Также необходимо отметить, что очень часто изменения в законодательстве и международной политике приводят к кардинальной перестройке систем[1,2,15].

Всё это является спецификой систем для обеспечения расчетов и клиринга и находит отражение при верификации программного обеспечения на том или ином уровне.

III. Подходы к тестированию

Перед тем как определять инструменты тестирования или приступать к их проектированию, следует определить задачи, возникающие при тестировании, и подходы к тестированию, которые будут наиболее рациональны и эффективны при тестировании определенной системы.

A. Задачи, возникающие при тестировании сложных систем

При тестировании сложных систем с большим количеством данных и подключений, как правило, возникают следующие задачи:

- Сложный алгоритм подсчета проектных рисков;
- Необходимость тестировать сложные, многоступенчатые и многодневные сценарии;
- Необходимость обеспечить быстрое исполнение регрессивных тестов;

- Тестирование различных способов взаимодействия с системой: API, обмен файлами, графический интерфейс (GUI);
- Симуляция поведения внешних систем;
- Настройка справочных данных и их миграция из других систем.

B. Существующие подходы к тестированию

При тестировании, как правило, используется несколько методов, среди которых выделяются:

- Автоматическое (automation) и ручное (manual) тестирование;
- Функциональное (functional) тестирование и тестирование нагрузки/производительности (load/performance testing);
- Компонентное (unit), интеграционное (integration) тестирование и тестирование полного жизненного цикла системы (end-to-end testing).

Также существует несколько подходов к организации инструментов для тестирования:

- Data-driven testing[16] и Keyword-driven testing[17];
- Параллельный и последовательный запуск тестов.

C. Наиболее удобные подходы для тестирования расчетно-клиринговых систем

Как было сказано выше, системы проведения расчетов и клиринга представляют собой сложные структуры, которые требуют всеобъемлющего тестирования в достаточно сжатые сроки, поэтому самым эффективным решением данной задачи является автоматизация всевозможных сценариев.

При разработке тестов функциональное тестирование всегда имеет большой приоритет и практически никогда не исключается из набора тестовых сценариев. Тестирование нагрузки и/или производительности является важной частью тестирования, особенно в случаях, когда предполагается большая нагрузка на компоненты и когда система разрабатывается для большого количества пользователей.

Такие подходы как unit-, integration- и end-to-end тестирование являются основой тестирования любой функциональности и считаются неотъемлемой частью обеспечения качества программ.

Также, при выборе тестовых подходов, следует определиться с особенностями работы инструмента. На данный момент выделяются два популярных метода организации: data-driven testing и keyword-driven testing.

При data-driven testing только тестовые данные отделяются от исполняемого скрипта, тогда как при keyword-driven testing отделяются и тестовые данные, и сам скрипт[16,17]. Если для небольших программ с однотипными действиями достаточно просто отделить тестовые данные, то для расчетно-клиринговых систем, содержащих различные действия (как правило более пятидесяти [18,19]), удобнее использовать keyword-driven testing.

Системы проведения расчетов и клиринга обычно имеют определенный порядок событий и периодов в течение операционного дня[20]. При тестировании данных систем необходимо учитывать все их особенности, а также максимально точно имитировать бизнес-сценарии в тестовой среде. На рис. 3 изображено несколько сценариев, которые проверяются при тестировании систем проведения расчетов и клиринга. Они охватывают сценарии, релевантные для бизнеса, такие как:

- Дефицит средств на залоговом счете (Collateral Deficit);
- Проведение расчетов при условии, что участник торгов не выполнил своих обязательств (Delivery Default);
- Проведение корпоративных действий (Corporate Action).

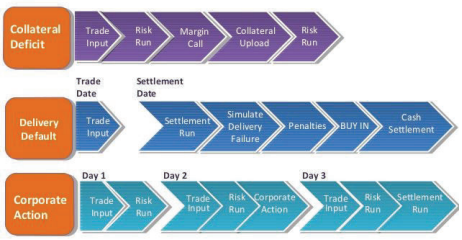


Рис. 3. Бизнес-сценарии, характерные для систем расчета и клиринга

Существующие подходы к тестированию не всегда могут быть эффективно использованы для обеспечения качества систем. Поэтому был разработан метод “concurrent test execution and scheduling”, представляющий собой гибрид популярных методов тестирования с поправкой на особенности систем проведения расчетов и клиринга.

Метод заключается в организации объекта scheduler, который представляет из себя набор тестовых keyword-driven скриптов (matrix) и расписания (sheduler config).

На Рис 4. отображена структура тестового скрипта (Matrix) для “concurrent test execution and scheduling”. Наименьшей единицей скрипта является действие (Action), которое может являться отдельным тестом или его частью. Одно действие выполнется целиком и в один момент времени. Группа действий объединяется в шаги (Global Step). Шаг, в свою очередь, является привязкой к расписанию (sheduler config), на уровне которого регулируется порядок и время их исполнения. Тестом в данном случае будет набор действий из разных шагов. Шаги объединяются в тестовый скрипт, который может содержать один или несколько тестов.

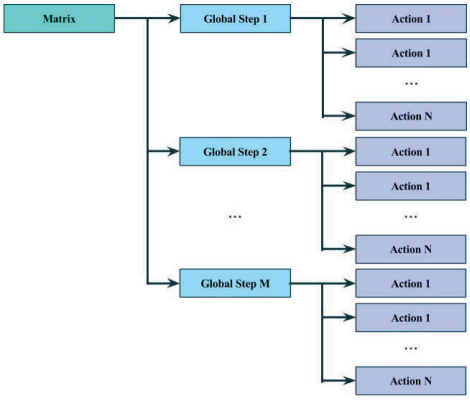


Рис. 4. Схематичное представление структуры тестового скрипта

Итак, в широком смысле подход, названный авторами “concurrent test execution and scheduling”, можно разбить на следующие субподходы:

1) *Запуск тестов по расписанию*

в данном пункте имеется в виду, что инструмент самостоятельно, по заданным параметрам или времени инициирует запуск необходимых тестовых скриптов без непосредственного вмешательства тестировщика;

2) *Одновременное выполнение тестов в одном тестовом окружении*

под одновременным исполнением здесь понимается не фактическое параллельное исполнение, которое возможно только в распределённых вычислительных системах, а скорее многозадачность, то есть в один промежуток времени исполняется часть теста (одно действие), симулируя этим параллельное выполнение тестов;

3) *Привязка к системному расписанию*

тут подразумевается привязка выполнения того или иного действия к определенному периоду или событию в системе.

IV. Существующие инструменты для обеспечения качества программного обеспечения

Обеспечение качества программного обеспечения является неотъемлемой частью жизненного цикла разработки, и с усложнением тестируемых программ появилась необходимость автоматизировать и контролировать процесс тестирования. Многие крупные компании, занимающиеся разработкой программного обеспечения имеют собственные наработки в данной области. Ярким примером является компания Hewlett-Packard Company (HP) имеющая большой набор решений для тестирования программного обеспечения[21]. С выделением процесса обеспечения качества как отдельной части разработки программного обеспечения вырос и спрос на различные инструменты для тестирования[22,23]

На текущее время создано большое количество тестовых инструментов: ресурсы, посвященные тестированию и/или разработке программного обеспечения, содержат описания до нескольких сотен различных инструментов[24,25], применимых в той или иной области.

Естественно, что при таком количестве различных решений встает вопрос о рациональном выборе инструментов для тестирования, но в случае тестирования систем, имеющих ряд специфических особенностей, выбор следует делать учитывая все особенности архитектуры системы, а также входящих в нее подсистем.

Ранее в статье мы выявили особенности систем проведения расчетов и клиринга, выбрали подходы, использование которых наиболее рационально для обеспечения качества систем, а также описали подход к тестированию, учитывающий специфику тестируемой области.

Список инструментов тестирования инструментов, реализующих распространенные тестовые подходы, достаточно обширен. Определяя решения, рациональные для описанного подхода, было решено провести анализ наиболее часто используемых тестовых инструментов.

Исследование инструментов для функционального тестирования, проведенное на различных ресурсах[26,27,28], а также статистика поисковых запросов (см. Рис 5 и Рис. 6) по заданной теме позволила выделить десять инструментов (пять коммерческих и пять с отрытым исходным кодом). Ниже приведен перечень инструментов, по которым была собрана статистика:

1) *Коммерческие инструменты:*
   a) *HP Quick Test Professional (HP QTP);*
   b) *SilkTest;*
   c) *Telerik TestStudio;*
   d) *TestComplete;*
   e) *Ranorex.*
2) *Инструменты с открытым исходным кодом:*
   a) *Selenium;*
   b) *AutoIt;*
   c) *SoapUI;*
   d) *Sahi;*
   e) *Watir ;*

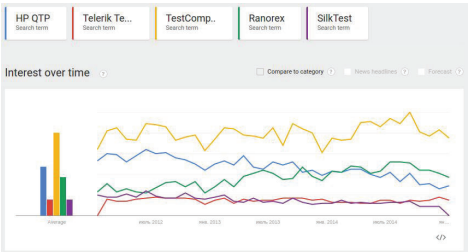


Рис. 5. Статистика поисковых запросов по коммерческим инструментам тестирования, сформированная с помощью сервиса Google Trends

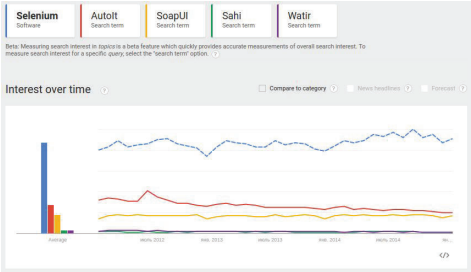


Рис. 6. Статистика поисковых запросов для инструментов тестирования с открытым исходным кодом, сформированная с помощью сервиса Google Trends

На основе имеющихся данных было решено выбрать пять из них, а именно:

- Selenium[29];
- HP Quick Test Professional (HP QTP)[30];
- Telerik TestStudio[31];
- TestComplete[32];
- Ranorex[33].

Авторами было решено провести сравнительный анализ по суб-подходам “concurrent test execution and scheduling” в связи с тем, что для определения подхода, аналогичного “concurrent test execution and scheduling”, в других инструментах могла быть выбрана другая терминология

A. Запуск тестов по расписанию и с привязкой к системным событиям

Практика использования планировщиков для организации расписания выполнения тех или иных приложений не нова и находит отражение во многих программных продуктах. Все анализируемые инструменты имеют возможность работы по расписанию. Например, Telerik TestStudio, TestComplete и HP QTP позволяют выполнять тесты в запланированное время без дополнительных плагинов или надстроек. А расписание в Selenium и Ranorex может быть реализовано при помощи серверов непрерывной интеграции (CI-серверов), например, Jenkins[34]. Привязка к системным событиям может быть осуществлена с помощью внешних плагинов и/или утилит. Например, с помощью Skybot Scheduler существует возможность организовать запуск тестов без жесткой привязки ко времени, а по какому-либо событию[35].

Несмотря на большое количество способов реализации привязки выполнения тестов ко времени или системным событиям, все способы имеют ряд недостатков, таких как сложная конфигурируемость дополнительных утилит и относительно небольшой функционал встроенных инструментов планирования.

B. Одновременное выполнение тестов

Все рассмотренные инструменты позволяют выполнять тестовые сценарии параллельно в нескольких тестовых окружениях, что позволяет распределять нагрузку при тестировании, а также сокращать время прогона независимых тестов.

Например, для выбранных коммерческих инструментов данная функция включена в поставку по умолчанию. В случае Selenium необходима интеграция с внешними модулями. Так, данное решение в сочетании с библиотекой TestNG позволяет выполнять несколько тестов параллельно, но с некоторыми ограничениями:

- Требуется сложная сетевая архитектура типа клиент-сервер;
- Тесты фактически будут выполняться в разных окружениях.

Однако с одновременным выполнением тестовых скриптов все немного сложнее, так как для всех инструментов, тест является наименьшей неделимой единицей автоматизации. Конечно, можно организовать тестовый скрипт как псевдопараллельный, но это существенно усложняет работу тестировщиков, а в случае с тестированием графического интерфейса становится практически невозможным.

Также следует проанализировать эти инструменты относительно общих требований:

- Возможность взаимодействия с системой через графический интерфейс;
- Возможность взаимодействия с системой через API;
- Тестирование Back-End;
- Гибкость и универсальность тестовых скриптов;
- Простота использования.

В Test Complete тесты могут создаваться в обоих форматах при помощи встроенных редакторов. Этот инструмент поддерживает такие языки, как: VBScript, JScript, DelphiScript и другие. Quick Test Professional позволяет записывать действия пользователя в тестируемом приложении. Эти действия хранятся в виде скриптов, которые могут быть выполнены в дальнейшем и могут быть представлены в двух форматах: как в виде кода на VBScript (expert view), так и в качестве набора последовательных шагов с действиями (keyword view).

Selenium IDE также предоставляет возможность записывать действия пользователя, в данном случае, в виде пар Command - Target, которые могут быть экспортированы в нескольких форматах: HTML, Java, Groovy, C#, Python и другие. В Ranorex и Test Studio используется Keyword-driven подход.

Несмотря на то, что все рассмотренные инструменты являются мощными и комплексными решениями, они разработаны с уклоном на тестирование широко используемых программ и приложений. Данная направленность, безусловно, важна на рынке программного обеспечения, но это значительно усложняет их конфигурирование для более специфичных систем проведения расчетов и клиринга, тем самым уменьшая эффективность процесса тестирования и повышая затраты на обеспечения качества программного обеспечения

(время, обучение персонала, настройка окружений и инструментов).

V. Инструмент для автоматизированного тестирования систем проведения расчетов и клиринга ClearTH

ClearTH - это специальный инструмент, предназначенный для автоматизации тестирования, написанный на языке Java, первичной целью которого является тестирование различного рода систем проведения расчетов и клиринга. В Таблице 1 описаны основные характеристики ClearTH.

Таблица I. Основные характеристики ClearTH

	Описание
Тип тестирования	Active Batch
Целевые системы	Расчетно-клиринговые платформы и Middle Office
Интерфейс тестируемой системы	Back-end (обычно подключается к шлюзу сообщений/APIs, и DBs); Тестирование графического интерфейса возможно через плагины или подключение в другим тестовым инструментам
Метод взаимодействия с системой	Прогон нескольких процессов по расписанию (зачастую захватывающий несколько бизнес-дней). Внутренний планировщик синхронизирован с планировщиков тестируемой системы. Запросы в базу данных для проверки данных
Протоколы	Содержит плагины для популярных протоколов (FIX и диалекты, FAST, SWIFT, ITCH, HTTP, SOAP, FTP и так далее) и частных протоколов. Новые плагины для других протоколов могут быть добавлены по требованию
Тестовый скрипт	Удобные для понимания CSV файлы. Скрипты создаются вручную аналитиками или создаются автоматически на основе системных данных.
Контроль тестирования и построение отчетов.	Интегрированный (Web фронт-энд), разрешающий несколько одновременных разнородных соединений, одновременное выполнение множества запланированных скриптов, связанных с основными глобальными шагами, краткое резюмирование результатов тестирования и детальные отчеты о прогоне. Базируется на принципе прогона всех скриптов одним нажатием - принцип "Большой кнопки"
Требования к платформе	Low footprint cross-platform application, MySQL или другие RDBMS

В ClearTH реализован подход "concurrent test execution and scheduling". Основным объектом инструмента является Sheduler, который связывает между собой тестовые скрипты (matrices) и расписанием (scheduler config).

Принцип по которому происходит выполнение тестов приведен на Рис 7.

Как видно из схемы, после инициации запуска выполнения скрипта ClearTH читает имя первого Global Step, далее приступает к поиску во всех тестовых скриптах невыполненного Action, привязанного к данному Global Step, и если Action найден, то он выполняется, если нет - ClearTH по такому же принципу обрабатывает следующий Global Step. Как только все Global Steps выполнены, ClearTH формирует отчет и завершает работу.

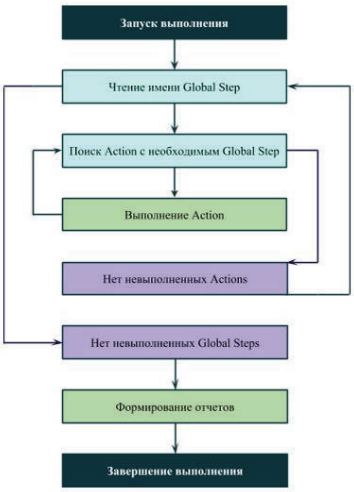


Рис. 7. Схематичное изображение работы ClearTH при выполнении теста

ClearTH имеет простой графический интерфейс, конфигурируемый для каждой системы, но базовая комплектация всегда включает в себя:

- Automation - вкладка на которой настраиваются тестовые скрипты и расписание;
- Tools - вкладка с дополнительными тестовыми инструментами;
- Connectivity - вкладка, содержащая информацию о подключениях.

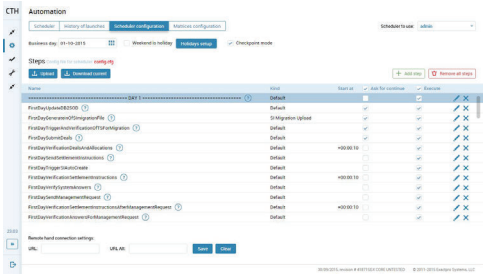


Рис. 8. Графическое представление Sheduler Config в ClearTH

Раздел Automation является базовым: планирование, запуск и выполнение тестовых сценариев происходит именно здесь. Он содержит в себе несколько Sheduler, которые в свою очередь состоят из Sheduler, History of launches, Scheduler configuration и Matrices Configuration.

Sheduler Configuration отображает Sheduler config, загруженный из файла или созданный вручную, и возможность сохранить конфигурацию в файл. Также есть возможность редактировать все поля Sheduler config через графический интерфейс, а именно:

- Имя глобального шага;
- Тип глобального шага;
- Время запуска (относительное или абсолютное);
- Запрос для продолжения (планировщик приостановит исполнение матриц после нужного шага и будет запрашивать разрешение на продолжение или остановку прогона);
- Исполнение (указывает, следует ли выполнять этот глобальный шаг или нет).

Вкладка Matrices configuration содержит список тестовых скриптов (matrices), загруженных пользователем в данный Sheduler. На этой вкладке можно загрузить, удалить или проверить matrix на предмет синтаксических ошибок, а также задать исполнение матрицы при прогоне. Matrix - это текстовый файл в формате CSV(comma separeted values). Файл в таком формате легко читается, его легко редактировать, а программ, поддерживающих данный формат, огромное количество.

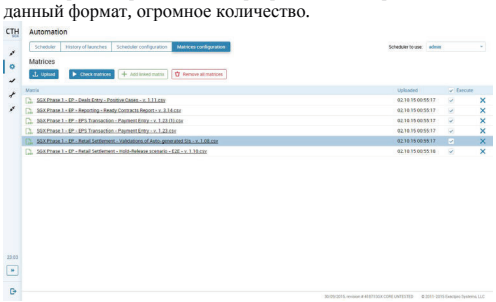


Рис. 9. Вкладка Matrices Configuration

Для каждого Action или группы задается свой заголовок, в котором указывается имя параметра, но есть и группа параметров, которая является служебной:

- #id - идетификатор Action;
- #GlobalStep - имя глобального шага;
- #Action - имя действия;
- #Enable - указывает на то, будет ли выполнятся данное действие или нет;
- #Timeout - задержка перед выполнением действия;
- #TestCase - идентификатор теста привязанный к внешнему сценарию тестирования.

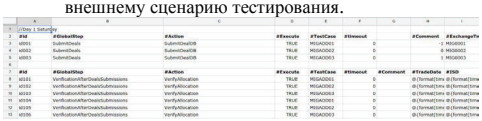


Рис. 10. Тестовый скрипт (matrix) открытый через табличный редактор

В разделе Sheduler производится запуск тестов, а также, во время прогона запланированных тестовых сценариев, ClearTH показывает все данные о прогоне:

- Текущий исполняемый глобальный шаг;
- Количество удачно завершенных и всех выполенных действий;
- Время начала и завершения каждого шага.

Затем несколько матриц, выполнение которых будет происходить синхронно, загружаются в инструмент. В конфигурационном файле прописывается порядок, в котором будут выполняться все глобальные шаги и действия, описанные в них (например, 1,3,5,2 и т.д.). Все матрицы будут проверяться одновременно, в зависимости от наличия в матрице того или иного глобального шага.

Проверка может быть остановлена в любой момент, после чего сразу же будут составлены отчеты о прогоне. Чтобы сделать перерыв в проверке, необходимо нажать на кнопку Паузы или же задать время “запроса для продолжения” после определенных глобальных шагов в файле конфигурации. В последнем случае результатом будет автоматическая остановка прогона после соответствующего глобального шага.

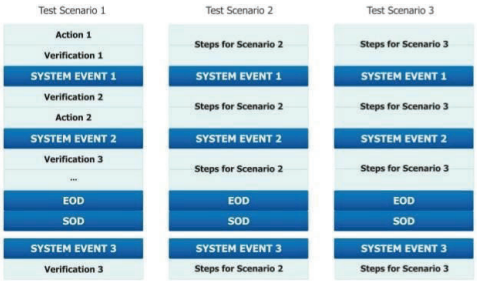
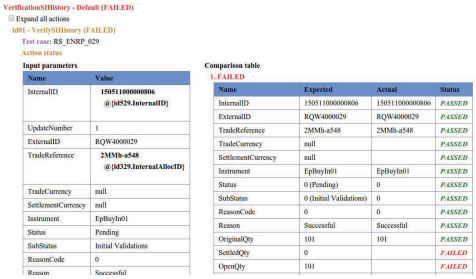


Рис. 11. Схематичное представление выполнение тестовых сценариев, привязанное к системным событиям

History of launches содержит ссылки на все отчеты, созданные в данном Sheduler, а также общую статистику по количеству удачных и неудачных тестов для каждого прогона.

Отчет содержит информацию о результатах выполнения всех Actions. Для действий он содержит информацию о входящих параметрах, а для проверок еще и таблицу сравнения ожидаемого и фактического результатов.

Если по каким либо причинам невозможно выполнить тот или иной Action типа “действие” (посылка файла, сообщения, изменения базы данных и так далее), то Action присваивается статус FAILED (неудача) и указывается причина, в обратном случае - PASSED (успех). Для присвоения статуса FAILED Action типа “проверка” есть несколько причин, например: объект не найден, фактический результат не совпадает с ожидаемым, технические проблемы.



VerificationSIHistory - Default (FAILED)

Expand all actions

idit - VerifySIHistory (FAILED)

Test case: RS_ENRP_029

Action status

Input parameters

Name	Value
InternalID	150511000000806 @{id329.InternalID}
UpdateNumber	1
ExternalID	RQW4000029
TradeReference	2MMb-a548 @{id329.InternalAllocID}
TradeCurrency	null
SettlementCurrency	null
Instrument	EpBuyIn01
Status	Pending
SubStatus	Initial Validations
ReasonCode	0
Reason	Successful

Comparison table

1. FAILED

Name	Expected	Actual	Status
InternalID	150511000000806	150511000000806	PASSED
ExternalID	RQW4000029	RQW4000029	PASSED
TradeReference	2MMb-a548	2MMb-a548	PASSED
TradeCurrency	null		PASSED
SettlementCurrency	null		PASSED
Instrument	EpBuyIn01	EpBuyIn01	PASSED
Status	0 (Pending)	0	PASSED
SubStatus	0 (Initial Validations)	0	PASSED
ReasonCode	0	0	PASSED
Reason	Successful	Successful	PASSED
OriginalQty	101	101	PASSED
SettledQty	0		FAILED
OpenQty	101		FAILED

Рис. 12. Пример сформированного отчета

Вкладка Tools в ClearTH содержит набор инструментов, который реализует дополнительный функционал, упрощающий работу тестировщика:

- Matrix from Report - генерирует на основе отчета матрицу, после выполнения которой он был получен;
- Config maker - позволяет создать sheduler config для конкретной матрицы;
- Message parser - обрабатывает сообщение, предоставляя его пользователю;
- Message to script - позволяет преобразовать сообщение в матрицу;

При необходимости могут быть добавлены специальные функции для каждого отдельного проекта или системы.

Кроме возможностей, доступных из графического интерфейса, есть набор “пассивных” функций:

- Имитация компонентов, взаимодействующих с системой;
- Подключение к базе данных;
- Анализ синтаксиса матрицы; в случае, если будут найдены ошибки, пользователь получит уведомление с возможностью просмотреть их список;
- Работа с динамическими данными с помощью mvel-выражений, позволяющих использовать java-синтаксис для автоматического заполнения полей в матрице;
- Возможность установки подключений к нескольким базам данных;
- Сохранение состояния выполнения тестовых сценариев с возможностью продолжить тестирование в дальнейшем.

Вкладка Connectivity содержит информацию о подключениях, использующихся для симуляции пользователей или внешних систем.

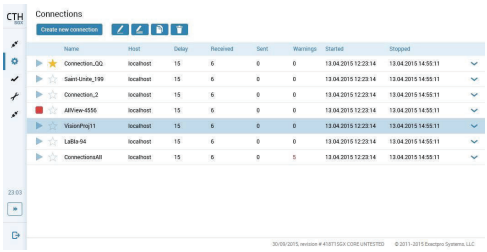


Рис. 13. Вкладка Connectivity

Кроме графического интерфейса, взаимодействие с инструментом ClearTH может быть осуществлено через API, что позволяет написать скрипт, который, например, запускает выполнение тестов, ждет его завершения, собирает отчеты в необходимую папку. Все вызовы API - это HTTP-запросы, которые отправляются по адресу, заданному при конфигурировании инструмента. В базовой комплектации поддерживаются запросы:

1) *Отображение статуса планировщика (scheduler);*

2) *Выполнение того или иного действия для заданного планировщика:*

   a) *Старт;*

   b) *Остановка;*

   c) *Удаление заданного тестового скрипта (matix);*

   d) *Удаление всех тестовых скриптов;*

   e) *Удаление всех шагов из конфигурации планировщика (scheduler cofig);*

   f) *Отображение статуса прогона;*

   g) *Получение отчета о прогоне;*

   h) *Установка нужной даты;*

3) *Отображение списка тестовых сценариев;*

4) *Отображение всех шагов (Global Steps) в планировщике;*

5) *Загрузка тестового скрипта;*

6) *Загрузка шагов.*

Все запросы возвращают результат в виде XML + HTTP-код статуса. Для отправки HTTP-запросов из скриптов можно использовать встроеные Linux-программы, например программу cURL[36].

Рис. 14. Пример API сообщений и матрицы к ним

Что касается взаимодействия с системой через API, то в ClearTH есть возможность преобразовать информацию, посылаемую в систему, в сообщение, которое примет система, и наоборот, возможность преобразовывать сообщения системы в понятную для пользователя информацию и сравнивать ее с ожидаемым результатом.

В связи с тем, что доступ к реальным внешним системам ограничен, в ClearTH реализована функциональность, которая позволяет смоделировать поведение всех необходимых внешних систем, взаимодействующих с расчетно-клиринговой системой, сымитировать любой вид активности: от действий пользователя до ответов тестируемой системы (см. Рис. 15). Этот подход позволяет подойти к тестированию с особой тщательностью, позволяет контролировать выполнение тестовых сценариев, максимально точно приблизив к реальности весь процесс.

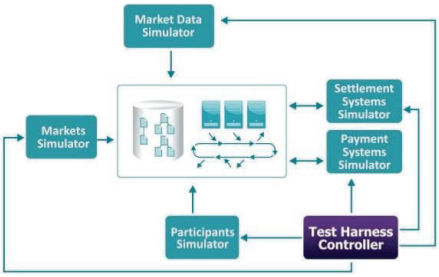


Рис. 15. Симулирование ClearTH реальных внешних систем

Авторы полагают, что с учетом всех особенностей систем проведения расчетов и клиринга ClearTH является крайне удобным и эффективным инструментом для организации тестирования подобных систем за счет того, что при описании подхода к тестированию и проектировании инструмента были учтены основные особенности расчетно-клиринговых систем, а также простоты представления скриптов и использования.

Заключение

С развитием цифровых технологий и, следовательно, увеличением количества транзакций, проведенных через электронные торговые и расчетно-клиринговые системы, стабильность финансовой инфраструктуры будет все больше проистекать от корректной работы систем, обеспечивающих проведение расчета и клиринговых операций. Представленный в статье инструмент для автоматизации тестирования и обеспечения качества систем проведения расчетов и клиринга используется в целях комплексной верификации программных продуктов на различных уровнях, тем самым снижая риск появления дефектов и возникающих с ними затрат.

В статье описаны распространенные методы обеспечения качества программного обеспечения, а также путем анализа сделаны выводы о наиболее рациональном и удобном подходе к тестированию расчетно-клиринговых систем. Рассмотрены также основные особенности и применения инструмента для автоматизированного тестирования систем проведения расчетов и клиринга ClearTH.

Дальнейшие исследования будут направлены на оптимизацию описанного подхода к тестированию и самого инструмента: в частности, на расширение разработанных методов на некоторые торговые, расчетно-клиринговые и другие системы, использующие подобную архитектуру и методы взаимодействий с пользователями, подсистемами и внешними программами.

Источники

[1] Nasdaq Dubai Business Rules [электронный ресурс] // Nasdaq Dubai Ltd. URL: http://www.nasdaqdubai.com/assets/docs/regulation/Nasdaq%20Dubai%20Business%20Rules%20-%20Rulebook%201%20V10.1%20for%20Equities%20(29%20March%202015).pdf (дата доступа: 02.09.2015)

[2] Post-trading/Settlement (SFD, CSDR, T2S) [электронный ресурс] // esma.com. URL: http://www.esma.europa.eu/page/Post-tradingSettlement-SFD-CSDR-T2S2 (дата доступа: 02.09.2015)

[3] Rulebooks [электронный ресурс] // LCH.Clearnet. URL: http://www.lchclearnet.com/rules-regulations/rulebooks (дата доступа: 02.09.2015)

[4] CDP Clearing Rules [электронный ресурс] // SGX.com. URL: http://rulebook.sgx.com/en/display/display_main.html?rbid=3271&element_id=359 (дата доступа: 02.09.2015)

[5] Reengineering Enterprise Wide Legacy BFSI Systems: Industrial case study, Prabhakar Cherukupalli, Y. Raghu Reddy, ISEC '15: Proceedings of the 8th India Software Engineering Conference

[6] Payment, clearing and settlement systems in the United Kingdom [электронный ресурс] // bis.org. URL: http://www.bis.org/cpmi/publ/d105_uk.pdf (дата доступа: 13.09.2015)

[7] Payment, clearing and settlement systems in the United States [электронный ресурс] // bis.org. URL: http://www.bis.org/cpmi/publ/d105_us.pdf (дата доступа: 13.09.2015)

[8] PAYMENT AND SECURITIES SETTLEMENT SYSTEMS IN THE EUROPEAN UNION VOLUME 1 EURO AREA COUNTRIES [электронный ресурс] // EUROPEAN CENTRAL BANK. URL: http://www.ecb.europa.eu/pub/pdf/other/ecbbluebookea200708en.pdf?73f510349d74efaf9d6599048bb32796 (дата доступа: 13.09.2015)

[9] Redesigning business networks: reference process, network and service map, Stefan Reitbauer, Falk Kohlmann, Clemens Eckert, Ken Mansfeldt, Rainer Alt, SAC '08: Proceedings of the 2008 ACM symposium on Applied computing

[10] Singapore Exchange (SGX) [электронный ресурс] // The Global provider of security financial messaging services. URL: http://www.swift.com/products_services/industry_initiatives/SGX/overview?lang=en# (дата доступа: 12.09.2015)

[11] About OeKB [электронный ресурс] // OeKB. URL: http://www.oekb.at/en/about-oekb/pages/default.aspx (дата доступа: 02.09.2015)

[12] SWIFT post-trade services for investment managers [электронный ресурс] // The Global provider of security financial messaging services. URL: http://www.swift.com/assets/swift_com/documents/products_services/SWIFT_post_trade_services_for_investment_managers_56249.pdf (дата доступа: 02.09.2015)

[13] COMMISSION DELEGATED REGULATION (EU) No 148/2013 [электронный ресурс] // Official Journal of the European Union. URL: http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=OJ:L:2013:052:0001:0010:EN:PDF (дата доступа: 25.09.2015)

[14] Comparative tables [электронный ресурс] // bis.org. URL: http://www.bis.org/cpmi/publ/d135a.pdf (дата доступа: 21.09.2015)

[15] О рынке ценных бумаг (с изменениями на 13 июля 2015 года) (редакция, действующая с 1 октября 2015 года) [электронный ресурс] // Электонный фонд правовой и нормативно-технической документации. URL: http://docs.cntd.ru/document/9018809 (дата доступа: 13.09.2015)

[16] Data-driven test cases [электронный ресурс] // boost.org. URL: http://www.boost.org/doc/libs/1_59_0/libs/test/doc/html/boost_test/tests_organization/test_cases/test_case_generation.html (дата доступа: 09.09.2015)

[17] Test Language - Introduction to Keyword Driven Testing [электронный ресурс] // Software Development Magazine - Programming, Software Testing, Project Management, Jobs. URL: http://www.methodsandtools.com/archive/archive.php?id=108 (дата доступа: 17.09.2015)

[18] CESR Technical Advice to the European Commission in the Context of the MiFID Review – Equity Markets Post-trade Transparency Standards [электронный ресурс] // COMMITTEE OF EUROPEAN SECURITIES REGULATORS. URL: http://www.esma.europa.eu/system/files/10_882.pdf (дата доступа: 22.09.2015)

[19] ISO 20022 Securities messages [электронный ресурс] // ISO 20022 Universal financial industry message scheme. URL: https://www.hkex.com.hk/eng/rulesreg/clearrules/ocp/documents/chap15.pdf (дата доступа: 15.09.2015)

[20] TYPHOONS AND RAINSTORMS [электронный ресурс] // OPERATIONAL CLEARING PROCEDURES. URL: http://www.bis.org/cpmi/publ/d135a.pdf (дата доступа: 22.09.2015)

[21] FUNCTIONAL TESTING [электронный ресурс] // hp.com. URL: http://www8.hp.com/us/en/software-solutions/functional-testing-software-testing/index.html?jumpid=hpr_r1002_usen_link1 (дата доступа: 21.09.2015)

[22] Gelperin, D.; B. Hetzel (1988). "The Growth of Software Testing". CACM 31 (6). ISSN 0001-0782

[23] A (Very) Brief History of Test Automation. Geoff Horne [электронный ресурс] // linkedin.com. URL: https://www.linkedin.com/pulse/20141007123253-16089094-a-very-brief-history-of-test-automation (дата доступа: 13.09.2015)

[24] List of Testing Tools [электронный ресурс] // guru99.com. URL: http://www.guru99.com/list-of-testing-tools.html (дата доступа: 18.09.2015)

[25] Test Tools [электронный ресурс] // ministryoftesting.com. URL: http://www.ministryoftesting.com/resources/software-testing-tools/ (дата доступа: 21.09.2015)

[26] 10 REGRESSION/FUNCTIONAL WEB TESTING TOOLS [электронный ресурс] // TECH BLOG. URL: http://www.hurricanesoftwares.com/10-regressionfunctional-web-testing-tools/ (дата доступа: 02.09.2015)

[27] 5 Best Test Automation Tools [электронный ресурс] // automated-360.com. URL: http://automated-360.com/automation-tools/5-best-test-automation-tools (дата доступа: 21.09.2015)

[28] Category:Software testing tools [электронный ресурс] // wikipedia.org. URL:https://en.wikipedia.org/wiki/Category:Software_testing_tools (дата доступа: 11.09.2015)

[29] What is Selenium? [электронный ресурс] // seleniumhq.org. URL: http://docs.seleniumhq.org/ (дата доступа: 21.09.2015)

[30] HP Unified Functional Testing software [электронный ресурс] // Web-site of Hewlett-Packard Company. URL:http://www8.hp.com/h20195/V2/GetPDF.aspx/4AA4-8360ENW.pdf (дата доступа: 21.09.2015)

[31] Test Studio [электронный ресурс] // telerik.com. URL: http://www.telerik.com/teststudio (дата доступа: 14.09.2015)

[32] TestComplete Platform: Testing for Desktop, Mobile, & Web Applications [электронный ресурс] // smartbear.com. URL: http://smartbear.com/product/testcomplete/overview/ (дата доступа: 21.09.2015)

[33] Automated Testing of Desktop. Web. Mobile. [электронный ресурс] // ranorex.com. URL: http://www.ranorex.com/ (дата доступа: 25.09.2015)

[34] Use Jenkins [электронный ресурс] // jenkins-ci.org. URL: https://wiki.jenkins-ci.org/display/JENKINS/Use+Jenkins (дата доступа: 21.09.2015)

[35] Skybot Job Scheduler [электронный ресурс] // helpsystems.com. URL: http://www.helpsystems.com/skybot/products/skybot-scheduler (дата доступа: 08.09.2015)

[36] cURL [электронный ресурс] // haxx.se. URL: http://curl.haxx.se/ (дата доступа: 28.09.2015)

Автоматизированное создание сценариев тестирования финансовых протоколов и подключений

Сергей Павлов
Exactpro Systems
sergey.pavlov@exactprosystems.com

Андрей Соловьев
Exactpro Systems
andrey.soloviev@exactprosystems.com

Александр Бормотин
Exactpro Systems
alexander.bormotin@exactprosystems.com

Анна Торопова
Exactpro Systems
anna.toropova@exactprosystems.com

Иосиф Иткин
Exactpro Systems
iosif.itkin@exactpro.com

***Аннотация*—Современные брокерские, биржевые и клиринговые платформы неотделимы от финансовых протоколов, обеспечивающих интеграцию подсистем и подсоединение пользовательских систем. Верификация корректной работы протоколов и подключений является неотъемлемой частью тестирования любой реализации программного обеспечения для таких платформ. В данной статье рассмотрен комплексный подход к автоматизированной генерации тест-кейсов для тестирования различных финансовых протоколов. Выявлены требования к документации, необходимые для реализации данного подхода, а именно к способам описания словарей сообщений и способам описания состояний и переходов между ними. Рассмотрена реализация стандартного протокола распространения данных о котировках - ITCH. Предложены способы автоматизированной генерации тест-кейсов для его динамической верификации.**

***Ключевые слова*—протокол, тестирование протоколов, protocol testing, connectivity testing, XML, XSD, workflow, конечные автоматы, словарь, dictionary, Finite State Machine, FSM, UML, SCXML, инструменты тестирования, testing tools**

I. Введение

Тестирование программного обеспечения является неотъемлемой частью цикла разработки, существует множество подходов и способов организации тестирования в частности, и цикла разработки в целом. С усложнением программного обеспечения ресурсы, необходимые для его тестирования, также увеличиваются. В особенности, это применимо к современным программным комплексам, обеспечивающим работу финансовых институтов, таких как брокеры, биржи, центральные депозитарии и клиринговые дома, которые являются сложными распределенными системами.

Одним из способов уменьшения затрат на обеспечение качества программного обеспечения является автоматизация процесса тестирования. Кроме того, автоматизация позволяет улучшить качество тестирования, то есть уменьшить количество ошибок, возникающих вследствие человеческого фактора. Существует множество различных инструментов, позволяющих автоматизировать выполнение уже готовых тест-кейсов. Однако, в большинстве случаев написание сценариев тестирования и разработка тест-кейсов остаются полностью ручными процессами.

Необходимо заметить, что финансовые платформы не только являются сложными системами, но и продолжают постоянно развиваться. Следовательно, с добавлением функций тестируемой системы приходится создавать новые тесты. Логичным выводом из этого следует, что стоит попытаться автоматизировать процесс создания тест-кейсов. Но перед этим необходимо принять во внимание два аспекта:

- Будут ли ресурсы, затраченные на создание инструмента и автоматизированную генерацию тест-кейсов, меньше ресурсов, необходимых для их ручного создания;
- Возможность использования созданного инструмента для генерации тестов для других областей или систем.

Тестирование протоколов и соединений являются областями, которые занимают много времени, если тесты создаются вручную. В то же время, принципы, по которым создаётся набор тестов, достаточно просты и применимы к большинству тестируемых систем. Более того, существуют широко распространенные протоколы, такие как FIX и SWIFT[1,2], поддерживаемые и используемые большинством организаций в финансовой индустрии. Поэтому инструмент для тестирования, создающий тест-кейсы можно будет использовать для большинства существующих систем. Именно здесь можно применить автоматизированную генерацию тест-кейсов.

II. Определения

A. *Определения и типы протоколов*

Современные торговые платформы, а так же системы проведения расчетов и клиринга обслуживают большое количество пользователей по различным протоколам.

Протокол передачи данных — это набор правил, позволяющих двум или более объектам осуществлять связь между собой. Правила определяют синтаксис, семантику, способ передачи сообщений и методы обработки ошибок при взаимодействии объектов. Как правило, в научной литературе описывается несколько широко известных парадигм классификаций протоколов. Например, модель OSI[13] состоит из семи упорядоченных уровней, а стек протоколов TCP/IP[17] состоит из четырех уровней. Финансовые протоколы, в большинстве своем, можно классифицировать как относящиеся к прикладному уровню. Однако, популярные классификации имеют широкий охват, и требуют дополнительной структуризации по определенным признакам. Наиболее часто прикладные протоколы дополнительно разделяют по типу представления информации на две группы: текстовые (символьные) и бинарные.

При текстовом представлении информации данные передаются в виде текста (строк), описанных как правило в виде JSON-строк, XML-сообщений (FIXML, SOAP) или тэгерированых сообщений (FIX, SWIFT MT). Несмотря на то, что сообщения в таком формате проще отлаживать и легче воспринимать, они имеют большой процент избыточной информации за счет сравнительно большого объема оболочки.

Бинарные протоколы, в отличие от текстовых, могут использовать различные типы данных (строковые, числовые, битовые поля) для представления информации. Поля, а следовательно и сообщения, могут иметь фиксированную или переменную длину.

Отметим, что возможна ситуация, когда в протоколе используются оба представления.

B. *Технологический процесс (англ. - workflow)*

Под технологическим процессом в данной статье подразумевается совокупность всех возможных состояний, включая начальное и конечное состояния, если таковые имеются, и событий, происходящих внутри рассматриваемой системы или ее части, а также связей между ними.

C. *Спецификация протокола*

Спецификацией протокола в этой статье будем называть набор требований к входящим и исходящим сообщениям. Спецификации протоколов так же могут ссылаться на технические стандарты.

III. Особенности и цели тестирования протоколов

A. *Тестирование соединения*

Большое количество одноадресных протоколов (англ. unicast), к которым, как правило, относятся и финансовые, не позволяет инициировать обмен прикладными сообщениями сразу после установки соединения на транспортном уровне. Перед началом обмена прикладными сообщениями клиент и сервер должны обменяться административными сообщениями, которые кроме установки соединения, осуществляют поддержку соединения, синхронизацию между клиентом и сервером, восстановление сообщений, закрытие соединения и другие функции поддерживаемые протоколом. Прикладные же сообщения используются непосредственно для обмена финансовой информацией. В отличие от прикладных сообщений, которые существенно отличаются в зависимости от конкретной системы, административные сообщения унифицированы. Например, сессионный уровень (административные сообщения) FIX протокола был выделен в отдельный подпротокол FIXT - FIX Transport[1]. В него входят всего 7 сообщений: Logon, Heartbeat, Test Request, Resend Request, Reject, Sequence Reset, Logout.

Тестирование соединения (англ. connectivity testing) относится к функциональному тестированию. В нем проверяются вышеназванные аспекты работы системы. В него входят как позитивные сценарии - например, Logon с правильной комбинацией имени пользователя и пароля, так и негативные, - например, Logon с неправильным паролем для данного имени пользователя.

Как правило, большая часть тестирования протокола происходит после позитивного тестирования подключения в связи с тем, что тестирование протоколов, чаще всего, невозможно без установленного соединения.

B. *Тестирование протоколов*

Под тестированием протоколов, как правило, подразумевается тестирование коммуникации двух или более систем (например, взаимодействие между системами брокера и биржи) с учетом специфики конкретной реализации протокола.

Тестирование протоколов условно можно разделить на две категории: негативное и позитивное тестирование.

1) *Позитивное тестирование протоколов*

Позитивное тестирование представляет собой набор тестов соответствующих штатному поведению системы. Если рассматривать тестирование протоколов, то позитивная часть тестирования, в большинстве случаев, представляет собой комбинацию всех возможных корректных сообщений и ответов на них. Однако, такой подход порождает большое количество тестов и некоторая их часть будет избыточна. При оптимизации процесса тестирования логично использовать широко распространенные подходы, такие как классы

эквивалентности, для определения достаточного набора тестов.

Важно подчеркнуть, что сценарии тестирования, различные с точки зрения общей логики системы (функциональные тесты), могут содержать одни и те же входные и выходные сообщения, то есть быть идентичными с точки зрения тестирования протоколов.

В общем случае представляется, что в любой системе реализация протокола должна быть необходима и достаточна для взаимодействия между клиентом и сервером. Таким образом, при покрытии всей логики системы функциональными тестами, автоматически проверяется позитивное тестирование протоколов, и его отдельная проверка становится ненужной.

Когда же нужно позитивное тестирование протоколов? Мы полагаем, что прежде всего оно необходимо в начале тестирования системы или новой функциональности кем бы то ни было: внутренняя или внешняя команды обеспечения качества программного обеспечения, конечные пользователи. Тестирование протоколов выявляет дефекты на ранних стадиях и уменьшает цену ошибки. Кроме того, оно также является частью тестирования совместимости клиентов.

2) Негативное тестирование протоколов

Негативное тестирование включает себя тесты, которые соответствуют нештатному поведению системы - различные ошибки, крайние состояния, некорректный ввод. Негативное тестирование протоколов подразумевает под собой посыл в систему некорректных сообщений, например, относительно словаря или с неправильным порядком сообщений. Целью таких тестов, как правило, является проверка устойчивости системы к различным входным данным, запись всех событий связанных с этими данными и проверка обработки граничных состояний.

Негативное тестирование протоколов обычно не осуществляется конечными пользователями системы. В особенности это касается пользователей с сертифицированными системами, которые должны посылать только правильные сообщения. Оно часто имеет меньший приоритет для тестировщиков, которые концентрируются на сценариях более вероятных в реальной промышленной системе. Однако возможные последствия неправильной реакции на некорректные сообщения могут быть гораздо серьезнее, чем ошибки при обработке правильного сообщения. Наш опыт показывает, что возможны ситуации, когда при получении неправильных сообщений появляются критические ошибки в системе, влияющие на ее дальнейшую работоспособность.

Разумеется, в большинстве случаев способов послать неправильное сообщение гораздо больше, чем послать корректное сообщение. Поэтому в негативном тестировании протоколов, даже больше чем в позитивном, важна оптимизация набора тестов.

Стоит отметить, что в отличие от позитивного тестирования, негативное должно быть включено в библиотеку регрессионных тестов из-за того, что эти тесты не проверяются другими сценариями.

Подготовка и проведение тестирования протоколов часто отнимает много времени и ресурсов. Но так как принципы, по которым создаётся набор тестов, достаточно просты и применимы к большинству тестируемых систем, именно здесь можно применить автоматизированную генерацию тест-кейсов.

IV. Требования и подходы к описанию словаря и технологического процесса

A. Требования к описанию спецификации протокола

При реализации протокольного соединения между компонентами клиентского и серверного приложений необходимо, чтобы все элементы системы могли четко и однозначно интерпретировать получаемые и отправляемые сообщения. Для этого необходимо описать словарь сообщений.

В настоящее время существует довольно много способов для описания словарей. Спецификация в текстовом формате является наиболее распространенным вариантом. Обычно такой документ поставляется вместе с системой или находится в свободном доступе. В файле описаны типы данных, формат протокола передачи данных, а так же структура сообщений. Пример описания сообщения UnitHeader протокола MITCH[11] можно увидеть в Таблице I.

Таблица I. Пример описания словаря в текстовой спецификации

Field	Offset	Length	Type	Description
Length	0	2	UInt16	Length of the message block including the header and all payload messages.
Message Count	2	1	UInt8	Number of payload messages that will follow the header.
Market Data Group	3	1	Byte	Identity of the market data group the payload messages relate to.
Sequence Number	4	4	UInt32	Sequence number of the first payload message.
Payload	5	Variable	-	One or more payload messages.

Текстовый формат описания словаря позволяет емко и точно описать все детали, но, как правило, имеет свободный формат и сложен для машинной обработки, для которой требуется описание в более формальном виде. Для этого существует ряд языков с простым синтаксисом, удобным для создания и обработки документов, как программами так и людьми. К таким языкам можно отнести XML, XSD, JSON.

В некоторых случаях вместе с системой поставляется описание протокола в одном из вышеперечисленных форматов, но чаще всего предоставляются только текстовые описания.

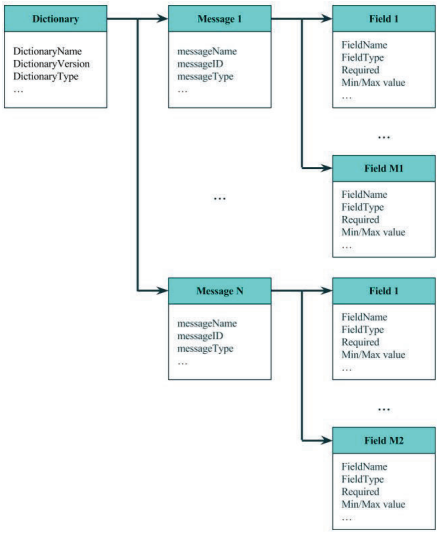


Рис. 1. Схематичное изображение структуры словаря

Несмотря на существование множества различных финансовых протоколов, можно выделить общую структуру словаря (см. Рис. 1) для любого из них:

- Название и атрибуты протокола;
- Список всех сообщений с их атрибутами;
- По каждому сообщению - список полей с их атрибутами.

Наиболее популярным способом описания протокола является язык XML. На верхнем уровне XML словаря описывается структура dictionary. Атрибутами этой структуры являются имя протокола, версия протокола, тип протокола.

Исходя из того, что протокол состоит из набора различных сообщений, структура dictionary включает в себя набор структур message. Атрибутами структуры message являются и идентификатор сообщения. В свою очередь, структура message состоит из набора структур field. Для каждой структуры field определены атрибуты FieldName и FieldType. Каждая структура field имеет набор определенных параметров, например длина поля.

Набор параметров поля варьируется и зависит от конкретной реализации, например существуют поля со сложной структурой, такие как повторяющиеся группы в протоколе FIX.

Описание словаря в формате XML имеет ряд недостатков. В частности, с помощью XML невозможно описать сложные структуры данных и шаблоны данных. Для описания сообщений, в которых применяются сложные поля, можно использовать язык XSD.

Язык XSD, в отличие от XML позволяет описать:

1. Сложный формат строки, используя паттерны, основанные на регулярных выражениях. Например, паттерн:

```
<xs:restriction base="xs:string">
<xs:pattern value="[A-Z]{6,6}[A-Z2-9][A-NP-Z0-9]([A-Z0-9]{3,3}){0,1}"/>
</xs:restriction>
```

2. Поле, которое представлено одним из нескольких вариантов. Например выражение:

```
<xs:complexType name="PartyIdentification36Choice">
<xs:sequence>
<xs:choice>
<xs:element name="AnyBIC" type="AnyBICIdentifier"/>
<xs:element name="PrtryId" type="GenericId19"/>
</xs:choice>
</xs:sequence>
</xs:complexType>
```

означает, что поле “PartyIdentification36Choice” может быть заполнено или значением типа “AnyBICIdentifier”, или значением типа “GenericId19”.

B. Технологический процесс

В отличие от описания словарей, на данный момент мы не видим примеров описания технологического процесса, заданного явным образом для протоколов, используемых торговых и расчетно-клиринговых системах. Чаще всего он описывается неявным образом документами в формате .pdf или .doc, в которых также описываются словарь и сопутствующие бизнес-процессы. Описание может быть представлено в виде комбинации текста, таблиц и различных диаграмм[3, 11].

В [4] отмечается, что типичная реализация FIX протокола имеет технологический процесс, часть которого может быть описана текстом, например:

- В момент времени t система пассивно ждет подключения со стороны клиента;
- В момент времени t2 клиент посылает сообщение Logon;
- В момент времени t3 система получает сообщение Logon и посылает ответное сообщение Logon;
- В момент времени t4 клиент посылает сообщение Heartbeat;
- В момент времени t5 система посылает сообщение;

Вышесказанное также может быть представлено в графическом виде (см. Рис.2)

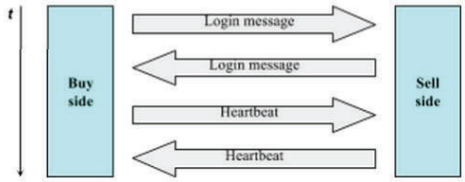


Рис. 2. Графическое представление части технологического процесса протокола FIX

Здесь шкала времени направлена вниз, то есть сообщение, расположенное выше, послано раньше чем сообщение, расположенное ниже.

В обоих случаях представление не является ни точным, ни полным. Очевидная неточность - это последовательность событий. В общем случае система может послать сообщение Heartbeat на доли секунд раньше, чем это сделает клиент, и такое поведение не должно рассматриваться как ошибочное. Таким образом, если мы создадим тест, точно следуя данному описанию технологического процесса, он будет находить ошибку, хотя поведение системы корректно. Очевидно, что представление не является полным, так как не включает, например, случай ошибки в клиентском сообщении Logon - в этом случае система не должна отвечать сообщением Logon.

В случае более сложных сценариев, текстовое описание технологического процесса может быть еще более неточным, неполным и к тому же громоздким, что лишь усложняет задачу написания тестов, так как возникает риск того, что отдельные части спецификации будут противоречить друг другу.

В дополнение к этим общим требованиям, необходимым и для ручного создания сценариев тестирования, автоматизированное создание тест-кейсов также подразумевает, что описание технологического процесса должно иметь формат, который может интерпретировать программа.

Насколько сложные сценарии могут быть указаны в технологическом процессе для тестирования протоколов? В предыдущей главе было сказано про связь системы и используемых ею протоколов. Рассмотрение каждой отдельной комбинации полей сообщений в технологическом процессе неизбежно требует анализа логики системы. Чем сложнее система, тем сложнее протокол. Наш опыт показывает, что для тестирования финансовых протоколов в большинстве случаев имеет смысл использовать самый общий уровень описания событий - ожидание только сообщений. В противном случае процесс написания технологического процесса становится слишком долгим. Таким образом, описание технологического процесса должно отвечать требованию лаконичности.

Итак, можно выделить следующие требования к описанию технологического процесса:

- Точность;
- Полнота;
- Лаконичность;
- Наглядность;
- Гибкость;
- Возможность чтения программой.

Для описания технологического процесса хорошо подходит концепция конечных автоматов (англ. Finite State Machine или FSM). Это вызвано тем, что конечные автоматы хорошо описывают реактивные, то есть реагирующие на события, системы ПО[5]. Программные комплексы, существующие для обеспечения нужд финансовых институтов, таких как биржи, центральные депозитарии, клиринговые системы или во всяком случае их части - торговые шлюзы, являются реактивными по своей сути. Так, FIX шлюз биржи пошлет сообщения клиенту только в ответ на его предыдущие сообщения. Существует множество способов описания FSM[6]: таблица "Состояние/Событие", представимая в виде csv-file, машина Мили (англ. Mealy machine), машина Мура (англ. Moore machine), UML, SDL, SCXML и прочие.

В работе[7] рассматривается создание сценариев тестирования протокола IRC на основе модели расширенного автомата, при этом используется описание в виде машины Мили и таблицы “Состояние/Событие”. Таблица “Состояние/Событие” достаточно наглядна при условии, что она небольшая. Такое представление может быть интерпретировано программой и позволяет легко отслеживать полноту описания технологического процесса и созданных тестов. Однако стоит заметить, что IRC протокол достаточно прост - как видно из статьи, его можно описать конечным автоматом с 4 состояниями и 7 событиями. При увеличении возможных состояний и/или событий, такой способ описания становится абсолютно ненаглядным. Кроме того, если система имеет конфигурируемые параметры, то при изменение параметра таблица должна быть значительно переделана. В случае использования недискретных переменных, например времени, требуется выбор какой-либо модели для тестирования уже на стадии описания.

Финансовые протоколы, как правило, гораздо сложнее, чем упомянутый IRC. Так, при попытке описания существующего технологического процесса реализации FIX протокола с помощью классической таблицы “Состояние/Событие” было обнаружено, что количество состояний увеличивается очень быстро.

Например, согласно спецификации[3] клиент должен быть отключен от системы, если пропустил 6 сообщений Heartbeat. Таким образом, чтобы правильно описать последствия события "время неактивности стало равным Heartbeat interval", нужно ввести 6 состояний "Пропущено 0 сообщений Heartbeat", "Пропущено 1 сообщение Heartbeat" и так далее. В то же самое время, сообщение посланное клиентом, может иметь порядковый номер меньше ожидаемого, равное ожидаемому или больше ожидаемого системой, итого 3 состояния для каждого сообщения. Общее количество сообщений, определенных в протоколе FIX, на данный момент равно 158[1]. Скорее всего, в конкретной системе не все они используются, но число используемых не меньше 10[3]. Параметры “ожидаемый порядковый номер” и “количество пропущенных сообщений Heartbeat” комплементарны по отношению друг к другу, то есть возможны любые комбинации значений этих параметров (стоит отметить, что некоторые события могут влиять на оба параметра). В результате общее количество возможных состояний равно произведению различных комплементарных параметров.

Рассматривая только эти параметры, мы получаем больше 6*3*10 = 180 различных состояний. Кроме того, в реальной системе максимальное количество пропущенных сообщений Heartbeat является конфигурируемым параметром. Поэтому, как было отмечено выше, описание в виде таблицы “Состояние/Событие” потребует значительной переработки в случае изменения параметра.

Многих недостатков, которые присущи представлению в виде таблицы “Состояние/Событие”, лишено представление FSM в унифицированном языке моделирования (англ. Unified Modeling Language - UML) [8, 9]. В нем, кроме прочего, предлагается использовать не состояния, а так называемые "расширенные состояния", которые включают в себя качественный аспект (само состояние) и количественный аспект (переменные "расширенного состояния"). Так, в нашем примере мы можем ввести переменную "количество пропущенных сообщений Heartbeat". Кроме того, можно ввести события, которые будут выполняться только при наступлении определенного условия (называемые “Guard condition” или просто “Guard”). Такое представление гораздо более наглядно и является более гибким.

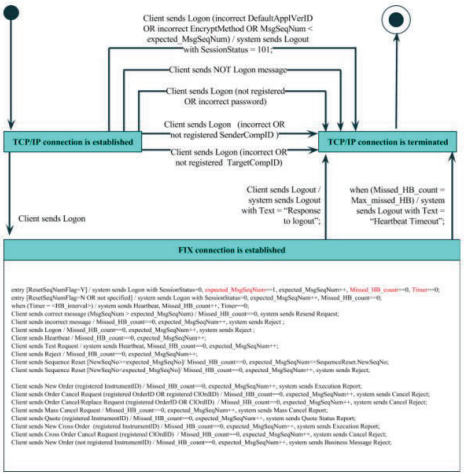


Рис. 3. Представление части технологического процесса реализации FIX протокола на UML-подобном языке

Пример описания технологического процесса реализации FIX протокола с помощью конечных автоматов на UML-подобном языке представлен на Рис. 3. Представление конечных автоматов в таком виде при всей наглядности обладает и недостатками. Полноту описания технологического процесса, представленного в виде UML, проверить сложнее, чем при описании таблицей “Состояние/Событие”. Но для нашей задачи главным недостатком представляется тот факт, что в UML-TP предлагается архитектура, нацеленная больше на ручную разработку тестов[15].

Существует XML спецификация для представления конечных автоматов, она называется SCXML[10]. Однако представление в таком виде, в свою очередь, не так наглядно как в языке UML.

Таким образом, каждый из способов описания конечного автомата имеет свои достоинства и недостатки. В разных ситуациях и для различных задач может понадобиться то или иное описание. Возможность преобразования одного вида описания в другой поможет объединить лучшие стороны каждого представления.

V. Программная реализация генерации сценариев тестирования

A. Общие положения

В тестировании на основании состояний применяются различные критерии покрытия. Слабейший критерий требует, чтобы тесты прошли каждое состояние и проверили каждый переход[14]. В нашей работе мы рассмотрим именно этот критерий. Попробуем описать возможный алгоритм работы приложения, которое, основываясь на технологическом процессе и словаре, создаст набор тестов, проверяющий протокол и каждый переход, указанный в технологическом процессе. Переходы мы разделим на 3 типа:

- Относящиеся к тестированию соединения. Такие переходы возникают при посылке правильных с точки зрения словаря сессионных сообщений, а также в сопутствующих сценариях;
- Относящиеся к позитивному тестированию протоколов. Эти переходы возникают при посылке правильных с точки зрения словаря прикладных сообщений;
- Относящиеся к негативному тестированию протоколов. Данные переходы возникают при посылке несоответствующих словарю сообщений, как административных, так и прикладных.

Желательно, чтобы тест-кейсы были независимы, то есть могли выполняться в любом порядке. Для реализации этого требуется, чтобы в начале тест-кейса система была в начальном состоянии, а в конце тест кейса в конечном. Таким образом, любой тест-кейс должен содержать шаги для перевода системы в конечное состояние, даже если проверяемый переход не приводит к этому.

Отметим, что начальное и конечное состояния зависят от конкретного технологического процесса. Они должны выбираться в зависимости от рассматриваемой области. Для тестирования соединения протоколов прикладного уровня, коими являются все финансовые протоколы, весьма логично рассматривать в качестве начального состояния “TCP/IP connection is established”, то есть TCP/IP соединение установлено, а в качестве конечного - “TCP/IP connection is closed” (TCP/IP соединение закрыто). Если бы мы рассматривали тестирование реализации протокола

TCP, начальное состояние было бы другим - например, “соединение установлено на сетевом уровне”. А для тестирования основной функциональности системы мы, наоборот, использовали бы начальное состояние в виде “соединение установлено на уровне прикладного (финансового) протокола” или еще более “высокие” состояния.

В данной статье описывается подход, где приложение последовательно рассматривает каждый переход, представленный в технологическом процессе, и генерирует тест-кейс или тест-кейсы для него. При этом, в результате использования переходов для вспомогательных целей, а именно перехода в начальное для данного события состояние, и перехода в конечное состояние, некоторые переходы могут проверяться многократно и даже приводить к абсолютному дублированию тестов. Однако, это позволит иметь простые тесты, которые могут служить индикаторами для решения о выполнении более сложных тестов. Часто не имеет смысла выполнять сложный тест, если он уже на первом шаге обнаруживает ошибку. Кроме того, в некоторых случаях это даст уверенность, что результат теста не зависит от пути. К тому же представляется, что такую логику работы программы легче реализовать. Рассмотрение оптимизации данного подхода является предметом будущих исследований.

Нашей конечной целью является определение общих подходов, которые будут применимы к автоматизированному созданию тест-кейсов для любого протокола. Для этого рассмотрим возможную работу приложения на конкретном примере компонента протокола распространения данных о котировках ITCH, отвечающего за восстановление утраченных сообщений[11]. На основе текстовой спецификации нами был сформирован технологический процесс, описывающий большую часть интересующей нас функциональности, а именно необходимой для тестирования соединения и протокола. Описание было выполнено в трех представлениях: UML-подобный язык, таблица “Состояние/Событие” и SCXML.

В соответствии со сказанным выше, начальным состоянием в технологическом процессе является состояние “TCP/IP connection is established” (TCP/IP соединение установлено), а конечным - “TCP/IP connection is closed” (TCP/IP соединение закрыто). Кроме того, в UML-подобном представлении, все события и действия, которые являются событиями, переводящими состояние само в себя, окрашены для удобства в 3 цвета, в зависимости от типа перехода. В этом случае мы сможем сгруппировать тесты, созданные для каждого типа, что в дальнейшем облегчит работу с ними. Мы разберем переходы последовательно и начнем с тех, которые относятся к тестированию соединения, затем рассмотрим те, что относятся к позитивному тестированию и, наконец, разберем переходы, относящиеся к негативному тестированию протокола.

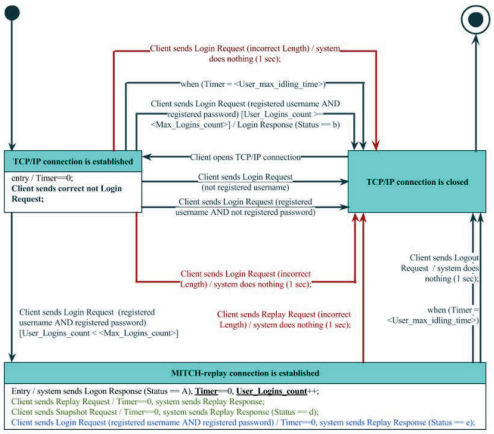


Рис. 4. Представление части технологического процесса реализации ITCH протокола на UML-подобном языке

B. Тестирование соединения

Приложение может выбирать переходы в любом порядке. Начнем, например, с “Client sends Login Request (not registered username)”. Рассмотрим детально описание этого события.

“Client sends” - тестировщик или инструмент для тестирования имитирует действия клиента. Поэтому это выражение означает, что что-то необходимо послать в систему, в отличие от “system sends/does”, где мы будем ждать определенного сообщения или действия от системы.

“Login Request” - название сообщения согласно словарю. Стоит заметить, что, если не указано обратное, сообщение считается корректным с точки зрения протокола.

“(not registered username)” - согласно принятой семантике языка UML в круглых скобках указываются атрибуты перехода.

“username” - это имя поля в сообщении Login Request. В нем, как следует из названия, указывается имя пользователя в системе. Для этого поля, также как для некоторых других полей, кроме деления на категории correct/incorrect, которые определяют правильность поля с точки зрения словаря, следует различать категории registered/not registered. Эти категории определены для полей, правильные значения которых ограничиваются не только протоколом, но и списком возможных значений соответствующего параметра в самой системе.

Поле является “Registered” (зарегистрированным, определенным), если его значение совпадает с одним из возможных значений соответствующего параметра в системе. Соответственно, поле является “Not registered” (противоположность Registered), если его значение не совпадает ни с одним из возможных значений соответствующего параметра в системе. В общем случае,

некоторые из значений параметра, зарегистрированного в системе, могут быть некорректными с точки зрения протокола. Однако, такая конфигурация должна рассматриваться как ошибочная, и потому на практике не должна встречаться. Поэтому в дальнейшем мы будем считать категории registered/not registered относящимися к категории correct. Соответственно, значение, некорректное с точки зрения протокола, автоматически будет считаться неопределенным в системе.

Итак, чтобы реализовать такое событие, в тест-кейсе должно быть послано сообщение Login Request, в котором значения во всех полях соответствуют словарю. Однако, значение поля username является незарегистрированным. Как следует из словаря, в теле сообщения Login Request 4 поля: Length, MsgType, username, password.

Поля Length и MsgType однозначно определяют сообщение, значения в них фиксированы, то есть каждое их них имеет только одно корректное значение.

Поле password имеет тип данных String, и о нем нет деталей в технологическом процессе, поэтому мы можем заполнить его любой правильной строкой. Отсюда следует, что наш инструмент для тестирования должен уметь генерировать соответствующее или несоответствующее словарю значение в зависимости от атрибутов поля, определенных в словаре. Назовем такую функцию #{RandomString(10)}, в скобках - длина строки.

Поле username должно иметь not registered значение. Выше мы обсудили значение понятия "not registered". Однако теперь необходимо подумать, как его задавать. Здесь возможны разные подходы, в том числе автоматическая генерация набора байт, ограниченного типом данных и другими атрибутами поля, заданными словарем. Однако существует ненулевая вероятность, что созданная строка будет являться "registered" полем. Более надежным и простым подходом видится задание "not registered" поля в виде конфигурируемого параметра - переменной. В зависимости от подхода, она может быть локальной (определенной только внутри тест-кейса) или глобальной. Назовем ее not_registered_username, а вызывать ее будем функцией %{not_registered_username}. Недостатком такого подхода является тот факт, что автоматически будет генерироваться только один тест-кейс.

Итак, входные данные готовы. Согласно технологическому процессу на выходе мы не ждем никаких сообщений от сервера, только переход в конечное состояние. Поэтому сгенерированный тест-кейс будет иметь вид наподобие следующего:

- Послать сообщение Login Request с атрибутами Length=18, MsgType=0x01, Password=#{RandomString(10)}, username = %{not_registered_username};
- Подождать некоторое время;
- Проверить, что TCP/IP соединение закрыто;
- Проверить, что количество пришедших сообщений от системы равно нулю за все время теста.

В технологическом процессе не задано время, необходимое для осуществления перехода, а значит теоретически он должен осуществляться мгновенно. В реальном случае время перехода равно совокупной задержке на упаковку, доставку, обработку сообщения шлюзом системы, поэтому мы добавили второй шаг. Это связано с тем, что нашей целью не является проверка времени отклика, и мы не хотим ложных ошибок теста из-за нее. В то же время, слишком большое ожидание увеличит время выполнения тест-кейса. Реальная задержка является функцией многих параметров, поэтому значение времени ожидания должно быть конфигурируемо. Назовем его timeout, а вызывать его будем функцией %{timeout}

Выходное состояние при данном событии совпадает с конечным, поэтому нет необходимости использовать другие переходы.

Переходим к следующему событию. “Client sends Login Request (registered username AND not registered password)”

В атрибутах этого поля есть ключевое слово AND, которое является обычным логическим “И”. Реализация этого события похожа на предыдущую, за исключением того, что поле password должно быть not registered, а username, напротив - registered. Для registered полей будем использовать тот же подход с указанием значения через вызов переменной. Для поля username переменную назовем registered_username. Итого, сгенерированный тест-кейс будет иметь вид наподобие следующего:

- Послать Login Request с атрибутами Length=18, MsgType=0x01, Password = %{not_registered_password}, username = %{registered_username};
- Подождать %{timeout};
- Проверить, что TCP/IP соединение закрыто;
- Проверить, что количество пришедших сообщений от системы равно нулю за все время теста.

Рассмотренные выше события должны происходить при определенных действиях со стороны клиента. Такие события в языке UML называются “событиями вызова” (англ. call event).

Рассмотрим следующее событие “when (Timer=<User_max_idling_time>)”. Это событие означает, что когда значение глобальной системной переменной Timer становится равным заданному системному параметру <User_max_idling_time>, то есть “максимальному времени бездействия”, система должна закрывать TCP/IP соединение с клиентом. Такое событие отличается от рассмотренных выше тем, что оно может произойти без каких-либо действий со стороны клиента. В языке UML такой вид событий называется “событием изменения” (англ. change event). И семантика для его обозначения отличается от событий вызова. Оно начинается с ключевого слова “when", и в скобках указывается булево выражение, которое вызывает событие, когда его значение становится TRUE. В отличие от “Guard condition”, работа которых будет рассмотрена ниже, вычисление булевого выражения производится всегда, а не только когда данное событие вызывается.

Тесты, созданные для проверки этого события, должны проверять оба варианта, когда булево выражение равно

TRUE и когда оно равно FALSE. В обоих случаях система не должна генерировать никаких сообщений. Можно реализовать обе проверки в одном тест-кейсе, он будет содержать пять шагов:

- Подождать время равное <User_max_idling_time>/2;
- Проверить, что TCP/IP соединение открыто;
- Подождать время равное <user_max_idling_time>/2+%{timeout};
- Проверить, что TCP/IP соединение закрыто;
- Проверить, что количество пришедших сообщений от системы за все время теста равно нулю.

Переходим к событию “Client sends Login Request (registered username AND registered password) [User_Logins_count < <Max_Logins_count>]”. Оно является событием вызова, но, кроме указания действия и его атрибутов, содержит квадратные скобки, в которых заключено булево выражение. Это условие называется “Защитник” (англ. Guard). Когда действие происходит, система проверяет, что Защитник имеет значение TRUE и только в этом случае происходит переход и ответная реакция от системы. В нашем случае в условии проверяется, что количество успешно выполненных входов в систему меньше максимально допустимого. Для этого используются системная переменная “User_Logins_count” и системный параметр <Max_Logins_count>. В данном построении мы можем проверить переход, только если значение Защитника равно TRUE, если же его значение равно FALSE, этот переход не будет осуществлен. Должно существовать другое событие, которое будет реализовываться, если Защитник имеет значение FALSE (очевидно, оно также будет содержать Защитника, но с противоположным условием). В нашем случае этим другим событием является “Client sends Login Request [User_Logins_count >= <Max_Logins_count>] / Login Response (Status == b)”, мы рассмотрим его позже. Вернемся к событию “Client sends Login Request (registered username AND registered password) [User_Logins_count < <Max_Logins_count>]”. Предполагая, что условие выполнено, а для этого на этапе предварительной подготовки мы должны удостовериться что User_Logins_count == 0, нам остается выполнить простое действие “Client sends Login Request (registered username AND registered password)”.

Оно переводит нас в состояние “MITCH-replay connection is established”. Это состояние имеет событие, которое должно выполняться как только система переходит в это состояние: “entry / system sends Logon Response (Status == A), Timer==0, User_Logins_count++;”, называемого “действием при входе” (англ. entry action). Указателем на этот факт является ключевое слово “entry”. В соответствии с семантикой языка UML после знака слэш “/” указаны действия, которые должна сделать система. Вообще говоря, они все должны быть проверены. “User_Logins_count++” означает, что система должна увеличить значение переменной “User_Logins_count” на единицу. Проверка этого очевидно связана с обсуждавшимися ранее Защитниками. “Timer==0” указывает, что значение переменной Timer должно быть установлено в ноль. Вообще, для каждой системной переменной, упомянутой в технологическом процессе, должен существовать как минимум один переход, использующий ее значение. Кажется разумным все изменения системной переменной учитывать в тестах, рассматривающих переход, где участвует эта переменная. Таким образом, осталось проверить реакцию системы “system sends Logon Response (Status == A)”. Как было сказано выше, ключевые слова “system sends” означают, что мы должны ждать сообщения от системы. Атрибут “Status == A” сообщает, что следует ожидать значение “A” в поле Status. Ожидаемые значение других полей должны быть ограничены приложением при помощи словаря. Согласно словарю, кроме поля Status сообщение Login Response содержит также поля Length и MsgType. Также как для сообщения Login Request, они могут иметь единственные корректные с точки зрения словаря значения, именно их мы должны ждать в сообщении от системы. Кроме того, состояние “MITCH-replay connection is established” не является конечным, поэтому должны быть добавлены шаги для перевода системы в конечное состояние. Инструмент для тестирования должен выбрать любой из имеющихся путей перехода в конечное состояние. Не уменьшая общности, рассмотрим, что был выбран переход “Client sends Logout Request / system does nothing (1 sec);”. Согласно словарю, Logout Request состоит всего из двух полей, Length и MsgType, с которыми наше приложение уже умеет работать. Способ проверки реакции системы легко представить из сказанного выше. Итоговый тест-кейс может выглядеть следующим образом:

- Послать Login Request с атрибутами Length=18, MsgType=0x01, Password = %{registered_password}, username = %{registered_username}.
- Ожидать в течение времени %{timeout} сообщение Login Response с атрибутами Length=3, MsgType=0x02, Status = A.
- Проверить, что TCP/IP соединение открыто
- Послать Logout Request с атрибутами Length=2, MsgType=0x05
- Подождать время равное ½ секунды
- Проверить, что TCP/IP соединение открыто
- Подождать время равное ½ секунды + %{timeout}
- Проверить, что TCP/IP соединение закрыто
- Проверить, что количество пришедших сообщений от системы за все время теста равно одному.

Рассмотрим далее событие “Client sends Logout Request / system does nothing (1 sec);”. Оно переводит систему из состояния “MITCH-replay connection is established”, не являющегося начальным, в конечное состояние. Поэтому тест-кейс должен содержать шаги для перевода системы из начального состояния в необходимое. Приложение должно выбрать какой-либо путь, в нашем случае для этого существует единственный переход - “Client sends Login

Request (registered username AND registered password) [User_Logins_count < <Max_Logins_count>]". Таким образом, сгенерированный тест будет абсолютно идентичен предыдущему, нет необходимости описывать его снова. О том, что в результате нашего подхода могут появляться абсолютно идентичные тест-кейсы, мы уже упоминали во время обсуждения общих положений.

Событие "Client sends Login Request (registered username AND registered password) / Timer==0, system sends Replay Response (Status == e);" интересно тем, что оно является примером "внутреннего перехода" (англ. internal transition), которое похоже на "переход само-в-себя" (англ. self-transition), с той разницей, что не должны выполняться "действие при входе" и "действие при выходе". Кроме того, в нем генерируется сообщение Replay Response, которое мы еще не рассматривали. Согласно словарю, это сообщение имеет шесть полей: Length, MsgType, MarketDataGroup, FirstMessage, Count, Status. Ожидаемое значение поля Status указано в технологическом процессе, работа с полями Length и MsgType разобрана выше. Значение поля MarketDataGroup не ограничено ни словарем, ни технологическим процессом. Интерес представляют поля FirstMessage и Count, которые, согласно спецификации, должны "иметь значение 0, если Status не равен A". Такое ограничение должно быть задано на уровне словаря, его можно реализовать в языке XSD. Наш инструмент для тестирования должен уметь обрабатывать такие ограничения. В остальном составление тест-кейса для данного события не должно вызвать осложнений, так как остальные его части были разобраны при рассмотрении других событий. Тест-кейс будет состоять из трех частей:

- Переход из начального состояния в состояние "MITCH-replay connection is established";
- Выполнение самого события и ожидание реакции от системы;
- Переход из состояния "MITCH-replay connection is established" в конечное состояние.

Событие "Client sends correct not Login Request;". Это событие является обобщением нескольких событий, объединенных в нашем представлении для удобства. Мы можем это сделать, потому что состояние, в котором происходит событие, и реакция системы одинаковы для всех них. В описании технологического процесса это задается конструкцией "not Login Request", состоящей из булевого оператора NOT и имени сообщения Login Request. Отметим также ключевое слово "correct", означающее что сообщение должно быть корректным с точки зрения словаря. Заметим, что сообщение должно считаться корректным только в том случае, если оно определено для посылки клиентом. Поэтому представляется необходимым, чтобы каждое сообщение в словаре имело атрибут, который бы указывал, определено ли сообщение для посылки клиентом или сервером. Для упрощения можно договориться о том, что если атрибут не указан, считать, что сообщение определено для посылки обеими сторонами. Итак, инструмент тестирования ищет в словаре все сообщения, которые определены для посылки клиентом, но не являются сообщением Login Request, и генерирует их в соответствии только со словарем, так как атрибуты не указаны в технологическом процессе. В нашем случае такими сообщениями являются Logout Request, Replay Request, Snapshot Request. Это событие является внутренним, без какой-либо реакции со стороны системы. Как обычно, каждый тест-кейс должен заканчиваться переходом в конечное состояние. Примеры сгенерированных тест-кейсов:

- Послать сообщение Snapshot Request с атрибутами Length=30, MsgType=0x81, SequenceNumber = #{RandomUInt32}, Segment = #{RandomString(6)}, InstrumentID = #{RandomUInt32}, SubBook = 0, SnapshotType = 1, RecoverFromTime = #{RandomTime}, RequestID = #{RandomUInt32};
- Подождать %{timeout};
- Проверить, что TCP/IP соединение открыто;
- Послать Login Request с атрибутами Length=18, MsgType=0x01, Password = %{registered_password}, username = %{registered_username};
- Ожидать в течение времени %{timeout} сообщение Login Response с атрибутами Length=3, MsgType=0x02, Status = A;
- Проверить, что TCP/IP соединение открыто;
- Послать Logout Request с аттрибутами Length=2, MsgType=0x05;
- Подождать время равное ½ секунды;
- Проверить, что TCP/IP соединение открыто;
- Подождать время равное ½ секунды + %{timeout};
- Проверить, что TCP/IP соединение закрыто;
- Проверить, что количество пришедших сообщений от системы за все время теста равно одному.

В тестировании соединения осталось проверить реализацию событий, включающих в себя работу с системными переменными.

Переход "when (Timer = <User_max_idling_time>);" из состояния "MITCH-replay connection is established" в состояние "TCP/IP connection is closed". В первом тест-кейсе будет проверка самого перехода, она будет состоять из двух частей:

- шаги для перевода системы в состояние "MITCH-replay connection is established";
- шаги для проверки перехода "when (Timer = <User_max_idling_time>);", мы их разбирали выше.

Последующие тест-кейсы должны включать в себя проверки изменения системной переменной в случае событий, описанных в технологическом процессе. Они будут состоять из следующих частей:

- Шаги для перевода системы в состояние "MITCH-replay connection is established";
- Подождать время равное <User_max_idling_time>/2;

- Шаги, осуществляющие событие, изменяющее значение переменной Timer;
- Шаги для проверки перехода “when (Timer = <User_max_idling_time>);”.

В нашем примере все внутренние переходы (их три) внутри состояния “MITCH-replay connection is established” изменяют значение переменной Timer. Таким образом, в дополнении к основному тест-кейсу, мы будем иметь три дополнительных.

Осталось 2 последних перехода в тестировании соединения: “Client sends Login Request (registered username AND registered password) [User_Logins_count >= <Max_Logins_count>] / Login Response (Status == b)” и “Client opens TCP/IP connection”. Они оба связаны с системной переменной User_Logins_count. Второй переход является вспомогательным, он определен только для того, чтобы осуществить первый (нет проблемы в том, что по общей логике работы приложения будет создан отдельный тест-кейс для проверки второго перехода). Общая структура тест-кейса, проверяющего переход с Защитником, будет следующая:

- Шаги для перевода системы в состояние “MITCH-replay connection is established”;
- Шаги для перевода системы в состояние “TCP/IP connection is closed”;
- Переход в начальное состояние “TCP/IP connection is established”;
- Повторить шаги 1)-3) <Max_Logins_count> раз;
- Послать Login Request с registered username и registered password;
- Ожидать сообщения Login Response с Status = b.

Обобщая примеры событий, работающих с системными переменными, следует сказать, что инструмент для тестирования должен уметь проверять работу переменных. Для этого он должен выделять события, использующие переменные (события с Защитником и события изменения), а также события, которые меняют значения переменных, и составлять на их основе комплексные тест-кейсы.

C. Позитивное тестирование протоколов

Проверка событий, относящихся к позитивному тестированию протоколов, похожа на проверку событий для тестирования соединения. Разница в том, что обычно прикладных сообщений, являющихся предметом рассмотрения позитивного тестирования протоколов, больше, и количество полей в них также больше, нежели в административных сообщениях, которые рассматриваются в тестирования соединения. Однако в нашем случае определено всего два прикладных сообщения: Replay Request и Snapshot Request. Поэтому и количество событий равно двум. Мы не будем подробно описывать детально способ генерации тест-кейсов, так как большинство приемов, использующихся в тестировании соединения должны быть использованы и здесь. Остановимся только на особенностях.

“Client sends Replay Request / Timer==0, system sends Replay Response;”. Мы уже посылали сообщение Replay Request, когда проверяли событие “Client sends not Login Request message”, однако в том случае мы удовлетворились одним тест-кейсом для данного сообщения. В данном же случае, напомним, целью является проверка работы системы в случае получения различных комбинаций корректных с точки зрения словаря сообщений. Как было отмечено в главе 3, представляется разумным использовать широко распространенные подходы к тестированию, такие как классы эквивалентности, для определения достаточного набора различных сообщений. Для каждого сообщения должен быть сгенерирован отдельный тест-кейс.

Некоторые примеры различных комбинаций значений:

- MarketDataGroup = <space>, FirstMessage=0, Count=0;
- MarketDataGroup = <NULL>, FirstMessage=0, Count=65535;
- MarketDataGroup = A, FirstMessage = 20000, Count = 5678;
- MarketDataGroup = b, FirstMessage = 1000, Count = 20000;

D. Негативное тестирование протоколов

Как и для позитивного тестирования протокола, рассмотрим лишь особенности.

“Client sends Login Request (incorrect Length) / system does nothing (1 sec);” Ключевое слово “incorrect” в указании события означает, что поле или сообщение некорректно с точки зрения протокола. Если в workflow не уточнено как именно, то, подразумевается, что любой вид ошибки может быть использован. В нашем примере слово “incorrect” указано в атрибуте и означает, что значение в поле Length должно быть неправильным. В остальном сообщение должно соответствовать протоколу. Инструмент тестирования должен уметь генерировать сообщение для каждого возможного неправильного сообщения, подходы могут отличаться в зависимости от конкретного протокола. Для нашего примера может быть использован следующий алгоритм:

- Генерируется корректное сообщение Login Request;
- Изменяется значение в поле Length;
- Если новое значение больше правильного, произвольные байты добавляются после последнего байта сообщения, сгенерированного на первом шаге.

VI. Правила создания сообщений на основе словаря

Инструмент для тестирования должен уметь генерировать корректные и некорректные сообщения на основе словаря. Для этого необходимо определить метод создания позитивных и негативных тестов для каждого атрибута каждого поля, заданного в словаре. Примеры

атрибутов и соответствующие им позитивные и негативные тесты приведены в Таблице II.

Таблица II. Описание набора тестов для разных атрибутов

Аттрибут	Позитивные тесты	Негативные тесты
MinOccurs	Количество появлений поля в сообщении больше либо равно параметру MinOccurs	Количество появлений поля в сообщении меньше параметра MinOccurs
MaxOccurs	Количество появлений поля в сообщении меньше либо равно параметру MaxOccurs	Количество появлений поля в сообщении больше параметра MaxOccurs
Length	Длина поля меньше либо равна параметру Length	Длина поля больше параметра Length
Min Value	Значение поля больше либо равно MinValue	Значение поля меньше MinValue
Max Value	Значение поля меньше либо равно MaxValue	Значение поля больше MaxValue

Большое влияние на количество тест-кейсов оказывает такой обязательный атрибут, как тип данных. Рассмотрим некоторые типы данных и примеры связанных с ними тестов (заметим, что для бинарных протоколов негативные тесты неприменимы):

A. Строки

1) Позитивные:

- Строка заполнена на всю допустимую длину печатными символами, нет пробелов;
- Строка заполнена печатными символами, за которыми следуют пробелы;
- Строка заполнена печатными символами и пробелами случайным образом.

2) Негативные:

- Строка содержит непечатные символы.

B. Целые числа

1) Позитивные:

- Значение в середине диапазона;
- Значение равно максимальному/минимальному для этого типа данных;
- Значение с нулями вначале, например “0100” (специфичен для текстового протокола).

2) Негативные:

- Значение содержащие символы кроме “-” и “0-9”;
- Значение с символом “-”, находящемся не в начале.

C. Числа с плавающей точкой

1) Позитивные:

- Значение в середине диапазона;
- Значение равно максимальному/минимальному для этого типа данных;
- Значение с максимально возможным количеством знаков после запятой;
- Значение с нулями в начале, например “05” (специфичен для текстового протокола);
- Значение с нулями после десятичной части, например “1.12000” (специфичен для текстового протокола);
- Значение с точкой в начале, например “.01” (специфичен для текстового протокола);
- Значение с точкой в конце, например “345.” (специфичен для текстового протокола);

2) Негативные:

- Значения имеющие в себе символы кроме “-”,“.” и “0-9”;
- Значения с символом “-”, находящимся не в начале.

Резюмируя, можно выделить следующие требования к инструменту для тестирования:

- Умение интерпретировать семантику языка представления технологического процесса, в частности интерпретирование атрибутов, таких как registered, correct, send и прочих;
- Наличие алгоритма выбора пути перехода в конечное состояние;
- Умение генерировать соответствующее или несоответствующее словарю поле и/или сообщение;
- Умение проверять работу системных переменных, определенных в технологическом процессе.

Заключение

Инструменты и методы автоматизированного создания тестов как на основе словарей и так и на основе применения конечных автоматов достаточно подробно описаны в научной литературе[7,12,15,16]. В данной статье мы попытались проанализировать совмещение обоих подходов и его применимость к динамической верификации финансовых протоколов. В рамках исследования были рассмотрены компоненты протокола распространения данных о котировках ITCH, отвечающие за восстановление утраченных сообщений. Разработано представление конечного автомата, задающего технологический процесс на UML-подобном языке и на языке SCXML. Уточнены требования к описанию словарей и технологическому процессу, необходимые для автоматизированного создания тестов.

Дальнейшими направлениями работы будут:

- Более глубокий анализ существующих способов описания технологического процесса с помощью конечных автоматов;
- Анализ других финансовых протоколов для уточнения требований к инструменту для тестирования;
- Построение прототипа инструмента для тестирования;
- Проверка подходов, заложенных в требованиях к инструменту и решений для их реализации;
- Нахождение фундаментальных ограничений на применение данного подхода.

Источники

[1] Technical Resources / Specifications [электронный ресурс] // FIX Trading Community. URL: http://www.fixtradingcommunity.org/pg/structure/tech-specs (дата обращения: 15.09.2015)

[2] About ISO 20022 [электронный ресурс] // ISO 20022 Universal financial industry message scheme. URL: http://www.iso20022.org/about_iso20022.page (дата обращения: 21.09.2015)

[3] Technical Specifications - Live Version MIT202 - Trading Gateway (FIX 5.0) Specification - Issue 7.2 [электронный ресурс] // Borsa Italiana. URL: http://www.borsaitaliana.it/borsaitaliana/gestione-mercati/migrazionemillenniumit-mit/mit202-tradinggatewayspecification-issue71_pdf.htm (дата обращения: 10.09.2015)

[4] FIIDL - FIX Interactive Interface Definition Language [электронный ресурс] // J. Greenan. URL: http://blog.alignment-systems.com/2014/11/fix-interactive-interface-definition.html (дата обращения: 13.09.2015)

[5] Harel D., Politi M. Modeling Reactive Systems with Statecharts. NY: McGraw-Hill, 1998

[6] Finite-state machine [электронный ресурс] // Wikipedia.org. URL: https://en.wikipedia.org/wiki/Finite-state_machine (дата обращения: 09.09.2015)

[7] М.В. Жигулин, А.В. Коломеец, Н.Г. Кушик, А.В. Шабалдин. Тестирование программной реализации протокола IRC на основе модели расширенного автомата. Известия Томского политехнического университета. 2011. Т. 318. No 5

[8] UML state machine [электронный ресурс] // Wikipedia.org. URL: https://en.wikipedia.org/wiki/UML_state_machine (дата обращения: 09.09.2015)

[9] UML specification [электронный ресурс] // omg.org. URL: http://www.omg.org/spec/UML/ (дата обращения: 11.09.2015)

[10] State Chart XML specification [электронный ресурс] // W3.org. URL: http://www.w3.org/TR/scxml/ (дата обращения: 25.09.2015)

[11] Technical Specifications - Live Version MIT303 – Market Data Feed (MITCH) Specification - Issue 7.1 [электронный ресурс] // Borsa Italiana. URL: http://www.borsaitaliana.it/borsaitaliana/gestione-mercati/migrazionemillenniumit-mit/mit303-marketdatafeedspecification-issue71_pdf.htm (дата обращения: 10.09.2015)

[12] A. Alexeenko, P. Protsenko, A. Matveeva, I. Itkin, D. Sharov. Compatibility Testing of Protocol Connections of Exchange and Broker Systems Clients, Tools & Methods of Program Analysis 2013

[13] Международный стандарт ISO/IEC 7498-1 [электронный ресурс] // International Organization for Standardization. URL: http://standards.iso.org/ittf/licence.html (дата обращения: 23.09.2015)

[14] Применение методик тестирования. Часть 4 [электронный ресурс] // Инрэко ЛАН. URL: http://inrecolan.ru/blog/viewpost/360 (дата обращения: 01.09.2015)

[15] А.В. Баранцев, И.Б. Бурдонов, А.В. Демаков, С.В. Зеленов, А.С. Косачев, В.В. Кулямин, В.А. Омельченко, Н.В. Пакулин, А.К. Петренко, А.В. Хорошилов. Подход UniTesK к разработке тестов: достижения и перспективы. [электронный ресурс] // Труды Института системного программирования РАН. URL: http://citforum.ru/SE/testing/unitesk/ (дата обращения: 03.09.2015)

[16] Никешин А. В., Пакулин Н. В., Шнитман В. З.. Разработка тестового набора для верификации реализаций протокола безопасности TLS. Труды Института системного программирования РАН том 23 / 201

[17] Дуглас Э. Камер. Сети TCP/IP, том 1. Принципы, протоколы и структура. М.: «Вильямс», 2003. — С. 880

Dynamic verification of input and output data streams for market data aggregation and quote dissemination systems (Ticker Plant)

Alyona Bulda
Exactpro, LLC
alyona.bulda@exactpro.com

Maria Orlova
Exactpro, LLC
maria.orlova@exactpro.com

***Abstract* - Market data aggregation and quote dissemination systems (Ticker Plants) are widely used across the electronic trading industry. A Ticker Plant is responsible for distributing information about multiple execution venues over a normalized protocol. This paper presents a dynamic verification approach for such systems. Based on a set of programs developed by the authors, it allows processing large data sets, including those collected during non-functional testing of trading platforms and using them in real-live production. The paper also outlines benefits and shortcomings of the selected approach for real-time and historical transactions analysis.**

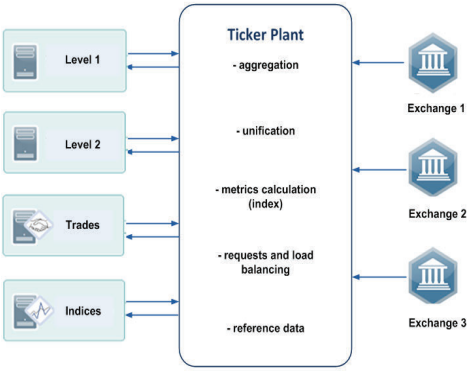
Keywords - market data, ticker plant, trading platform, exchange, non-functional testing, dynamic verification

I. Introduction

It is impossible to imagine modern day trading without up-to-date information about financial instruments, orders, and trades… Electronic trading is characterized by huge amount of data processed by a multitude of systems and algorithms. In its turn such systems and algorithms generate vast amounts of other data disseminated in various forms for a variety of purposes. One of the systems disseminating large amounts of data is a system of quotes aggregation and distribution (Ticker Plant).

A Ticker Plant is a system of aggregating market data information from various electronic trading platforms (or exchanges) and its dissemination. The system provides market data to the traders in a normalized or unified format [6]. Based on accumulated market data, Ticker Plant systems often calculate additional parameters, i.e. they enrich the static data about stocks and derivatives. One other characteristic of Ticker Plant systems is the ability to unite homogeneous values of disseminated quotes - Price Levels, real-time quote data provision based on requests from clients (e.g., such widely used services as Level 1, Level 2 [7], Index [9], T&S, News) - and storing of disseminated market data.

A schematic representation of a Ticker Plant system is provided below as Picture 1.



Picture 1. A Ticker Plant system schema

II. High Level Requirements for Ticker Plant Systems

Based on the description of a Ticker Plant system and its main characteristics noted above, we provide a set of requirements the system should comply with. We will follow this set of characteristics when testing it. A classification of such requirements was provided in [18], and in this paper we are providing a more complete list of such requirements. In order to make the assessment of each of the characteristics easier, we have divided them into functional and non-functional ones. A Ticker Plant system should:

• From a functional standpoint:

1. Collect quotes information from several sources (the suppliers of market data: exchanges, banks);
2. Process reference data provided by the exchanges;
3. Process quotes information disseminated via various data transmission protocols in real-time;
4. Convert the collected information into one format;
5. Aggregate quotes information according to various described methods;
6. Process this data in order to enrich the system's functionality: e.g., provide statistics (VWAP, Turnover, Trade High/Low, 52 week Trade High/Low) [1];

7. Provide data according to clients' requests: Level 1, Level 2, T&S, News, Index, Option chains, etc. [7] [9];
8. Provide recorded historic data about quotes.

• From a non-functional standpoint:

1. Provide fast processing of quotes data streams received from the exchanges in real-time;
2. Provide fast processing of requests received from clients and quotes data depending on the type of a client request (e.g., separately for a traded instrument, or a group of instruments, or the entire market);
3. Provide continuous working efficiency of the system;
4. Provide system operability;
5. Provide the ability of system monitoring (i.e. the availability of applications to monitor the system and operate its components);
6. Provide throughput;
7. Provide latency;
8. Provide fault tolerance.

Having defined a set of functional and non-functional characteristics of a Ticker Plant, we need to understand the process of testing each of the characteristics. The following two approaches to testing such systems become obvious.

In this paper, we would like to focus on the correctness of the logic of order construction in a Ticker Plant system and its reliability under load throughout an extended period of time.

Thus we have a specific task of verifying the outgoing data flow in Ticker Plant systems under load or real-time.

Market Data, Replay and Recovery

Highly loaded exchange and brokerage systems provide market data concerning traded financial instruments by means of their own components called Market Data Feeds. Every financial instrument comprises a fair amount of information generated every second. Therefore the ability to disseminate the entire stream of market data and the speed of disseminating market data for each financial instrument are the main characteristics for this type of components of highly loaded trading systems.

Normally market data includes the following set of parameters that are specific for a certain financial instrument: Ticker Symbol, Last Trade Price, Best Bid & Offer, ISIN, exchange code, Trade Time, Close Price. Depending on the complexity of highly loaded exchange systems, market data can be processed by the electronic exchange's internal components and enriched with additional information: e.g., daily turnover, VWAP, and more detailed information about the stock or derivative, i.e. Reference Data including, for instance, parameters of traders, market, trading sessions, and instruments. Reference Data or Static Data is instrument information, which does not change real-time: e.g., International Securities Identification Number (ISIN), price at the close of previous day's trading session (Close Price), Currency, the parameters of so called "Circuit Breakers" that are normally presented as percentages of last trade price (Dynamic or Static Circuit Breaker Tolerances (%)), and so on [1]. There is a number of standard quote dissemination protocols, such as, for instance, FIX/FAST [2], the so called quote dissemination protocols with fixed length of messages (e.g., ITCH [3]), or coded data dissemination protocols (e.g., HTTPS (HyperText Transfer Protocol Secure) [4], [5]) for providing the above mentioned information.

To transmit quote information, many electronic exchanges use both standard and bespoke protocols. Often traders cannot afford developing computer applications that would collect quote information required for their work. Therefore there is a need of creating systems allowing collection and aggregation of market data from various exchanges disseminated by using different financial data transmission protocols.

Market Data Feeds continue to disseminate data on multicast channels, however these will are active with a primary feed published on a primary network path and an identical secondary feed published on the resilient path. Recipients have access to two identically sequenced multicast feeds: Feed A and Feed B. Recipients may process both feeds and arbitrate between them to minimize the probability of a data loss.

If a gap in sequence numbers is detected on the multicast channel, the recipient should assume that some or all of the order books maintained on its systems are incorrect and initiate one of the recovery processes outlined below.

Replay Channel – The TCP replay channel should be used by clients to recover from a small-scale data loss on a particular channel. It permits to request the retransmission of a limited number of messages already published on the multicast channel. The channel supports the retransmission of the last small amount of messages published [3].

Recovery Channel – The TCP recovery channel should be used by clients to recover from a large-scale data loss. It permits to request a snapshot of the order book for the active instruments in the market data group [3].

Each multicast channel has its own dedicated instance of both a Replay and a Recovery service; each service is accessible by individual TCP sessions to dedicated IP addresses.

Protocol UDP

The two main transport layer protocols are used in the Internet, one of which requires establishing a connection and the other one doesn't. These protocols complement each other. The protocol that does not require establishing a connection is UDP (User Dataram Protocol) [20]. UDP is not a reliable protocol as it does little other than sending the packets between applications allowing them to complement that with their own protocols. UDP is an interface for IP by way of de-multiplication of several processes using ports and optional straight through error detection.

The Main Characteristics of UDP are:

- *UDP is unreliable* – it does not guarantee the delivery of packets. There is no error detection, flow control or re-transmission of lost packets. It just sends them and doesn't care whether they arrive or not.
- *UDP is a connection-less protocol* – Data is just sent and no socket needs to be established first. Data can flow one way (as in a radio broadcast) or both ways (a 2 way phone call). It also allows you to traverse some NAT devices without using port forwarding using a technique called UDP hole punching.
- *UDP is not ordered* – it does not use sequence numbers and therefore it cannot guarantee that they will reconstructed in the right order.
- *UDP is fast* – Because UDP doesn't have the additional overhead as TCP it is a faster protocol ideal for streaming.

For processes that need to drive the flow, control errors and time intervals UDP is ideal. One of the areas where it is especially useful is client-server applications. Often times the client sends a short request to the server and expects a short response. If the request and the response are lost, the client may attempt to send another request after a certain amount of time. This allows for code simplification and the reduction of the amount of required messages compared to protocols that need an initial tuning (such as TCP).

Keeping in mind the main differences between TCP and UDP it is sometimes better to use UDP over TCP. TCP should be used when no packet loss is a requirement and the data must be correct and free from errors. This is obviously useful when viewing web pages, e-mails and most forms of networking communication we are used to. When data is sent over UDP the packets can be lost.

Considering all available channels and all details and peculiarities of these channels, we have developed an approach to verify streams of the data for logical correctness via these channels. The descriptions of this approach will be started from separate parts.

III. Tools used for Ticker Plant dynamic verification

MListener is a program-model based on the mechanism of restoring "lost" updates disseminated by a channel. The program-model was developed due to the specifics of the channels used to disseminate market information. The UDP protocol is a more common way of distributing market information over such channels [20]. The protocol does not guarantee the delivery of the packets to the addressees. [20]. This program-model plays the role of a subscriber to the UDP market information channel. The logic of the program-model is simple – if a packet containing a change in the market flow is lost the program goes into "recovery' mode and re-requests the necessary information via special recovery channels provided by the exchanges. As a result, the program creates up to 2 additional files with market information per each such channel. If examined in their entirety, a complete picture of changes taking place at the exchange can be obtained.

Agglutination tool – is an instrument utility which is a part of MListener program-model. Its main function is to combine real-time market information and restored data if a small amount of real-time information was lost delivered in a queue.

Book Reader – is a program-model allowing to construct a book of quotes according to the logic of order construction in a Ticker Plant system. The implementation logic is based on the information seen by the external user only rather than on Ticker Plant system logic. This allows finding defects in such logic.

JSON (*JavaScript Object Notation*) - is a text format of data exchange. The format is based on JavaScript. There are 2 main advantages of using the format: 1) it is easily human-readable, and 2) it is easily generated and processed by the machines. This provides us with advantages from the standpoints of 1) flow analysis, and 2) the speed of program-models [19].

This method of verification based on program-models the text format of data exchange JSON is used thanks to its lightness (compared to xml for instance). Due to that the format allows for fast serializing / de-serializing and has the advantage of being inter-platform (the libraries exist for all platforms and all programming languages), flexible enough (it is possible to describe the types of data and their structures), compact enough (which is critical for large flows of data), has easily understandable method syntax if there is a need to understand the problems in the interaction of the components.

Book Checker – is a program-model allowing comparative analysis of each level of quotes added to the book in the Book Reader program-model, further pausing the work if differences are found and showing the differences as a result of performing the comparative analysis.

IV. Methods used for Ticker Plant dynamic verification

In this chapter we are going to examine the process of testing a Ticker Plant system by using the method of verifying a system of data aggregation and dissemination based on models.

To achieve better clarity, we are going to separate the testing process into the following stages:

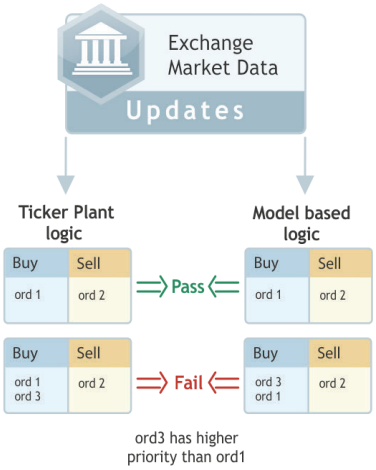
- Collection of market information about quotes from trading platforms (exchanges) delivered via a quote dissemination protocol with fixed length of messages (ITCH [3]). A "test" trading platform which is a full copy of a real electronic Ticker Plant system and MListener program-model receive incoming orders and trades information from the exchanges. At this stage, the "test" exchange [11] processes the quotes

information real-time. The MListener program-model keeps the market information in a queue without processing it.

- The collected historic information about changes allows us to build an order book for each instrument traded at the exchange after each quote change that occurred both at the exchange and the Ticker Plant system. Therefore, we receive 2 fully independent order book flows. The independence of the flows allows us track the defects that can be present in all algorithms (including the Ticker Plant algorithm).

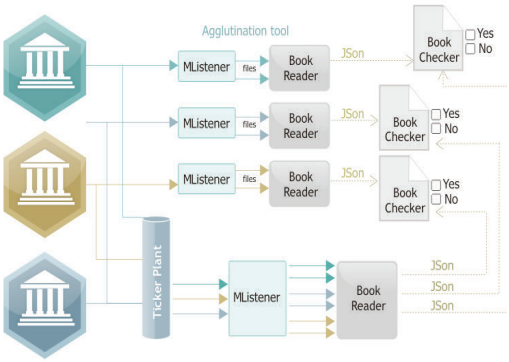
- The algorithm of the outgoing Ticker Plant flow and the outgoing exchange flow processed by program-models. Pic. 2 shows a schematic view of how the program-model works. The received Ticker Plant and exchange flows are verified in such a way that the book flow received by the program-model is the expected result for verification, while the flow built by the Ticker Plant system is the actual result to compare it with the expected one. Further, if a comparison of the two flows finds a difference, an analysis of the succession of changes that could have lead to a defect in an algorithm is performed. If the difference was caused by a defect in the program-model the same analysis is performed as to the logic and correctness of the change. If a defect in the test tool is found, it is easy to fix it and continue processing the flows.

This approach allows verifying a fairly large spectrum of scenarios to check the correctness of a Ticker Plant system operation. Such verification analysis is based on data from quote books. Such verifications include the following: that the values of added quotes are correct (price, quantity, order direction, etc.); that the order change is correct; that the book changes when an order is removed, or in case of an aggressive order change, or in case of executed trades are correct; that the book is constructed correctly in case of multi-level trades; that the priority of constructed orders is correct, that the addition of multi-level orders (10 price levels) is correct.



Pic.2 The algorithm of verifying the outgoing exchange flow processed by program-models.

We are providing a more detailed diagram of Ticker Plant system verification done by using program-models showing all of the steps of going through the program-models explained in this paper in succession.



Pic.3 The verification of Ticker Plant systems by using models.

The test approach is necessary and viable due to the fact that there is a multitude of situations involving various price positions and their changes as well as the large number of exchanges that disseminate quotes from different platforms. All such situations and combinations are structured in their entirety by using the program-models therefore eliminating the

need to spend time on arranging static pre-conditions. It is enough to generate a random set of trading combinations leading to various change combinations in quote aggregation and dissemination systems (Ticker Plant). On the other hand, this can be an emulation of trading cycles including all changes from the production platform, whose outgoing flow is also processed by program-models as described above.

V. Conclusion

This paper provides a description of quote aggregation and dissemination system (Ticker Plant), a list of its basic characteristics, its main functional components that need to be covered by testing. The paper describes the main large functional component for which a verification approach has been developed and the program-models implementing the approach have been built. The proposed method has been successfully used in practice. As the result of its implementation, a large amount of border conditions defects in order book construction has been detected.

References

1. Watson Wheatley Financial Systems, *Reconciliation Best Practice*, http://www.watsonwheatley.com/literature.html
2. M, Cochinwala, V. Kurien, G. Lalk, D. Shasha, "Efficient data reconciliation", *The Journal of Information Science*, vol.137, issue 1-4, pp. 1-15, Sep. 2001
3. Roger Nolan, The Informatica Blog, *Even 'The Most Interesting Man In The World' Won't Do This…* http://blogs.informatica.com/perspectives/2012/03/06/even-the-most-interesting-man-in-the-world-wont-do-this/
4. London Stock Exchange. *How UnaVista Works*: http://www.londonstockexchange.com/products-and-services/matching-reconciliation/how-unavista-works/index.html
5. Official site B2Bits <epam>. // [Electronic resource]. –http://www.b2bits.com/trading_solutions/market-data-solutions.html
6. Official site FIXprotocol. // [Electronic resource]. – Режим доступа http://www.fixprotocol.org/
7. Level1 Market Data Documentation / Official site London Stock Exchange // [Electronic resource]. – http://www.londonstockexchange.com/products-and-services/millennium-exchange/millennium-exchange-migration/mit303.pdf
8. P. Karlton, P. Kocher.: The Secure Sockets Layer (SSL) Protocol Version 3.0 // [Electronic resource]. – Режим доступа http://tools.ietf.org/html/rfc6101
9. Official site FTSE. // [Electronic resource]. – http://www.ftse.com/Indices/Data_Licenses/Real-time_Constituent_Data.jsp
10. Official site B2Bits <epam>. // [Electronic resource]. –http://www.b2bits.com/trading_solutions/market-data-solutions.html
11. Building the Book: A Full-Hardware Nasdaq Itch Ticker Plant on Solarflare's AoE FPGA Board / Sherman,M., Sood,P., Wong,K., Iakovlev,A., Parashar,N. // [Electronic resource]. – http://www.cs.columbia.edu/~sedwards/classes/2013/4840/reports/Itch.pdf
12. Official site Tokyo Stock Exchange // [Electronic resource]. – http://www.tse.or.jp/english/market/mkinfo/mains.html
13. Day Trading. / Official site About.com // [Electronic resource]. – http://daytrading.about.com/od/daytradingmarketdata/a/MarketDataDefin.htm
14. Customer Development Service (CDS). / Official site– London Stock Exchange // [Electronic resource]. – http://www.londonstockexchange.com/products-and-services/technical-library/customer/customerdevelopmentservice/customerdevelopmentservice.htm
15. Trading Floor Architecture / Official site Cisco // [Electronic resource]. – http://www.cisco.com/en/US/docs/solutions/Verticals/Trading_Floor_Architecture-E.html
16. Official site BSE India // [Electronic resource]. – http://www.bseindia.com/markets/MarketInfo/DispNoticesNCirculars.aspx?page=20120531-22&pagecont=0,31/05/2012,31/05/2012,,All,All,All,Scrip%20Name%20/%20Code
17. Official site MICEX // [Electronic resource]. – http://rts.micex.ru/s437
18. Статья К.В.Воронцов (ВЦ РАН), С.Б. Пшеничников (ММВБ): "Имитационное моделирование торгов: новая технология биржевых тренажеров." / Журнал «Индикатор», №2 (42). М. 2002 год // [Electronic resource]. — http://www.forecsys.com/site/about/press/exchange_simulator/
19. Official site Oslo Bors Stock Exchange. // [Electronic resource]. –http://www.oslobors.no/ob_eng/Oslo-Boers/Trading/Delta/Millennium-Exchange/Guide-to-Testing-Services-updated-issue
20. Zverev,A., Bulda,A.: Exchange Simulators for SOR. Algo Testing: Advantages vs. Shortcomings. / Конференция ExTENT // [Electronic resource]. – http://www.slideshare.net/extentconf/exchsims-forsoralgotestingadvantagesvsshortcomings29102011111113011104phpapp02
21. Official site Exactpro Systems. // [Electronic resource]. –http://www.exactprosystems.com/
22. Bulda A., Buyanova O., Zverev A., - Article "Usage of Exchange Simulators and Test Exchanges as Tools for Ticker Plant Systems", https://yadi.sk/d/eKKxTtEGCCeF6
23. Official site JSON. // [Electronic resource]. – http://www.json.org/index.html
24. Andrew S. Tanenbaum, David J. Wetherall.: Computer Networks. // [Electronic resource]. – http://cse.hcmut.edu.vn/~minhnguyen/NET/Computer%20Networks%20-%20A%20Tanenbaum%20-%205th%20edition.pdf

Верификация

Секционные доклады

Implementing the MetaVCG approach in the C-light system

Dmitry Kondratyev
A. P. Ershov Institute of Informatics Systems
Novosibirsk, Russia
Email: apple-66@mail.ru

Alexei Promsky
A. P. Ershov Institute of Informatics Systems
Novosibirsk, Russia
Email: promsky@iis.nsk.su

***Abstract*—Among the problems that can confront a verification system developer, the addition of new axiomatic rules is of great interest. Not only theoretical properties of a Hoare logic (first of all, soundness and completeness) can be endangered by such activity, but also the recoding of verification condition generator (VCG) is required in practice. The error-prone process of manual reprogramming can compromise the very idea of verification system. Can we trust a program verified by (possibly) faulty system? While the self-verified system is still a challenge (though some steps towards it are already taken), as for its VCG-part, there is a method providing a greater level of reliability. In 1980, Moriconi and Schwartz represented the MetaVCG approach, regretfully abandoned. In this article, we would like to describe our efforts to implement this approach in the C-light system.**

Keywords*—*Verification, specification, axiomatic semantics, the C-light language, ACSL, MetaVCG

I. Introduction

The axiomatic semantics serves as a formal basis for deductive program verification [1]. Its implementation usually takes the form of verification condition generator (VCG). When verification is studied for a simple model language, the VCG can be implemented once and for all. But in practice, it may require expansion. What could be the reasons for the expansion?

First, the program language itself may be enriched by new constructions. For example, in our project of the C program verification we introduced a subset, called C-light [8], as a starting point. This subset is quite representative, but still does not cover some low-level aspects of the C language. During the project evolution we added new constructions to the C-light. And yet, new expansions are to come. First of all, the newest standard of the C language is of great interest. Also, the C-light may serve as a basis for the kindred languages (Objective C, C++).

Second, the practical verification can require development of specialized versions of VCG for narrow classes of programs. Compared to the expansion of the language, such specialization (in fact, restriction) is not so obvious. However, it can play crucial role if we try to simplify the verification. Two examples can clarify this point.

As a start, let us consider a pattern code

This research is partially supported by the Russian Foundation for Basic research (project no. 15-01-05974)

$swap(x, y, buf) \equiv$
```
memcpy(buf, x, m);
memcpy(x, y, m);
memcpy(y, buf, m);
```

which can be found in some library routines (`qsort`, for example). Its treatment by general axiomatic rules will involve a triple instantiation of specifications for `memcpy` leading to a cumbersome quantified VC. In the meantime, we can enrich the Hoare system by the following axiom:

$$\{x = x_0 \wedge y = y_0\}\ swap(x, y, buf)\ \{x = y_0 \wedge y = x_0\}$$

Though it works only for those programs which contain a fragment $swap(..)$, its application can simplify the proof considerably.

Another example relates to the linear algebra programs. A Hoare system for this applied area was developed in the Theoretic programming Lab of our institute quite ago [10]. Given, `M` is a two-dimensional matrix and $e(k, i)$ is an expression depending on matrix indices k and i, consider the following Hoare triple:

$\{Q(M \leftarrow rep(M, mat(e_1, e_2, e_3, e_4), e(s, t)))\}$
```
for(k = e1; k <= e2; k++)
   for(i = e3; i <= e4; i++)
      M[k][i] = e(k,i);
```
{Q}

where matrix $rep(M, mat(e_1, e_2, e_3, e_4), e(s, t)))$ results from replacement of all elements corresponding to sub-matrix $mat(e_1, e_2, e_3, e_4)$ by expression e. The usual treatment of loops in axiomatic semantics is based on the loop invariants, which have always been one of the annoying features of this approach. But, as you can see here, two invariants for two nested loops were replaced by application of some logical constructs (rep and mat). A complete logical axiomatization of them can be found in [9].

An interim conclusion here is that the expansion of axiomatic semantics (and its VCG implementation, correspondingly) is inevitable in practice. Moreover, an enormous VCG, which covers all possible classes of programs, is hardly an option. A collection of specialized VCG could be a better solution. Thus, any method that can facilitate the reprogramming of the base VCG or development of the new ones, is of great interest for us.

Fortunately, there is a quite appropriate technique. In 1981, Moriconi and Schwartz [6] proposed a method which forms a

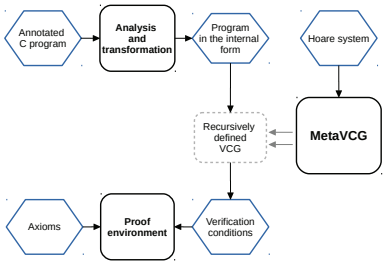


Fig. 1. A "meta-stage" in the verification process

meta verification condition generator (MetaVCG). It takes a Hoare logic as an input and automatically derives a recursively defined VCG. The axiomatic rules must be given in a *normal* form with several constraints. Many axiomatic rules do not satisfy them, so the authors provided an equivalence-preserving transformation from a more liberal *general* form into a normal one. The soundness and completeness were proved for their method, thus providing that a produced VCG is *correct* w.r.t. the original axiomatic definition.

In the presence of this meta-stage, the classical three-block scheme (input analyzer/VCG/prover) of a verification system changes slightly (Fig. 1).

Apart from automatic creation of VCGs, this approach has another benefit for us. An ultimate goal of our project is the development of a self-applicable verification system for the C language [12]. However, at the moment we use the C++ API of the compiler Clang for the "Analysis and transformation" block on Fig. 1. An intermediate translation of Clang AST into the C structures allowed us to implement MetaVCG using the C-light language, thus, promising a more complete self-verification. Some experiments were successfully conducted [13].

In the rest of this paper we give an overview of the MetaVCG approach as well as its implementation in our system.

II. Meta verification condition generation

The method of metageneration proposed by Moriconi and Schwartz [6] consists of two steps. A general axiomatic definition is first transformed into a normal form which, in turn, develops into a recursively defined VCG.

A. Preliminary definitions

Following Moriconi and Schwartz, we will use metavariables $\mathcal{P}$, $\mathcal{Q}$, $\mathcal{R}$, Γ, ... to denote partially interpreted first-order formulas. These formulas can contain uninterpreted predicate symbols P, Q, R, ... and formulas from the underlying theory. For example, $\mathcal{P}$ could denote P, $P \wedge x = 5$, or $x = 5$. We assume that the symbols P, Q, R,... may be instantiated by formulas in the underlying theory.

We also need a binary relation $\Leftarrow$ on uninterpreted predicate symbols. For a Hoare sentence of the form

$$\{\mathcal{P}(P_1, ..., P_m)\}\ S\ \{\mathcal{Q}(Q_1, ..., Q_n)\}$$

where the predicate symbols $P_1, ..., P_m$ and $Q_1, ..., Q_n$ are logically free in $\mathcal{P}$ and $\mathcal{Q}$, respectively, we have

$$P_i \Leftarrow Q_j, \quad \text{for } i \in \{1, ..., m\} \text{ and } j \in \{1, ..., n\}$$

Intuitively, a relation $P \Leftarrow Q$ indicates that the binding of the predicate symbol P depends upon the binding of Q. The relation $\Leftarrow$ is defined with respect to a set of Hoare triples. The notation $\overset{+}{\Leftarrow}$ denotes the *transitive closure* of $\Leftarrow$.

Similarly, for a rule of the form

$$\frac{\{P_1\}\ S_1\ \{\mathcal{Q}_1\}, ..., \{P_n\}\ S_n\ \{\mathcal{Q}_n\}, \Gamma}{\{\mathcal{P}\}\ S\ \{Q\}} \tag{1}$$

the relation $\ll$ defines the dependence of a proof concerning S on proofs of $S_1, ..., S_n$. In particular, we have $S \ll S_i$, for $i \in \{1, ..., n\}$. For a Hoare axiom system, we define the transitive closure $\ll +$ in the obvious manner.

We use the function $FreePreds$ to denote the set of *logically free* predicate symbols. $FreePreds$ applied to a first-order formula denotes its logically free symbols. Then inductively,

$$FreePreds(\{\mathcal{P}\}\ S\ \{\mathcal{Q}\}) = FreePreds(\mathcal{P}) \cup FreePreds(\mathcal{Q})\ ,$$

and for a proof rule R of the form (1)

$$FreePreds(R) = \bigcup_{i=1}^{n} FreePreds(\{P_i\}\ S_i\ \{\mathcal{Q}_i\}) \cup FreePreds(\Gamma) \cup FreePreds(\{\mathcal{P}\}\ S\ \{Q\})$$

We will use the function $FragVars$ to denote the set of "fragment variables" in the program fragment S of a Hoare triple $\{P\}\ S\ \{Q\}$. For example, $FragVars$ applied to "`if` (B) S_1 `else` S_2" has the value $\{B, S_1, S_2\}$. If applied to an entire Hoare rule, $FragVars$ yields a set containing the fragment variables from every Hoare sentence in the rule.

Finally, we define the notion of a bound occurrence of an uninterpreted predicate symbol in a rule. For a rule R, a predicate symbol in $FreePreds(R)$ is *bound* in R iff it is in $FragVars(R)$. Otherwise, the occurrence is said to be *free* in R.

B. The normal form for rules

Let us consider the following

Definition. A *normal form rule* is any instance N of

$$\frac{\{P_1\}\ S_1\ \{\mathcal{Q}_1\}\ , ...,\ \{P_n\}\ S_n\ \{\mathcal{Q}_n\},\ \Gamma}{\{\mathcal{P}\}\ S\ \{Q\}}$$

that satisfies the following constraints:

1) P_1,..., P_n and Q are predicate symbols free in N.
2) $FreePreds(\Gamma) \subseteq FreePreds(N) \cup FragVars(S)$.
3) The fragment variables of each S_i must be bound in S. That is, it must be the case that $\cup_{1 \le i \le n} FragVars(S_i) \subseteq FragVars(S)$.

4) *Dependency ordering*. The Hoare-triple premises of N must satisfy two dependency constraints.

   a. $P_i \overset{+}{\Leftarrow} P_j \supset i < j$

   b. $T \overset{+}{\Leftarrow} U \wedge \neg(\exists R) U \overset{+}{\Leftarrow} R \supset U \equiv Q \vee U$ bound in N

5) *Monotonicity*. Let $\mathcal{P}[P \leftarrow \mathbf{false}, P \in s]$ denote $\mathcal{P}$ with the proper substitution of **false** for each predicate P in the set s. Then, the following constraint on $\mathcal{P}$ must be satisfied:

$$\mathcal{P}[P_1, ..., P_n, Q \leftarrow \mathbf{true}] \vee \\ \forall s \subseteq \{P_1, ..., P_n, Q\}\ \neg\mathcal{P}[P \leftarrow \mathbf{false}, P \in s]$$

This constraint must hold for Γ and for each Q_i.

Two constraints are imposed on a system of the normal form rules: (i) Any terminal string σ in the programming language can be an instance of at most one language fragment S defined by a normal form axiom or an inference rule. (ii) The relation $\ll +$ must be irreflexive. .

Constraint 4 ensures that VCG will be able to compute instantiations for all free uninterpreted predicate symbols in the rules. In particular, constraint 4a requires an ordering of free predicate symbols that is made apparent by the following schema:

$$\frac{\{P_1\}\ S_1\ \{\mathcal{Q}_1(P_2, ..., P_n)\}, ..., \{P_i\}\ S_i\ \{\mathcal{Q}_i(P_{i+1}, ..., P_n)\}, \\ ...,\ \{P_n\}\ S_n\ \{\mathcal{Q}_n\},\ \Gamma}{\{\mathcal{P}(P_1, ..., Q)\}\ S\ \{Q\}}$$

This has the effect of eliminating dependency cycles, such as a premise of the form $\{P\}$ ... $\{P\}$ or a pair of premises of the form $\{P\}$ ... $\{R\}$ and $\{R\}$ ... $\{P\}$. Given this ordering, constraint 4b ensures that the tail of every dependency chain is either expressible as a function of the postcondition Q or is bound in a program fragment.

Constraint 5 is necessary for completeness of a VCG, i.e. it guarantees that VCG is able to compute the weakest precondition $wp(S, Q)$ for given S and Q. It is done by imposing a monotonicity constraint on rules, which eliminates the rules where certain "changes of sign" exist between the preconditions of the premises and the precondition in the conclusion.

C. The general form for rules and its translation into the normal one

The normal form constraints serve two purposes. First, a recursively defined VCG can be built up automatically for the normal rules. Indeed, since the preconditions of premises are individual predicate symbols, they can be substituted by the weakest preconditions for the corresponding programs and postconditions. For a rule of the form (1), the recursive function wp is defined as follows:

$$wp(S, Q) = \mathcal{P}[P_1 \leftarrow wp(S_1, \mathcal{Q}_1), ..., P_n \leftarrow wp(S_n, \mathcal{Q}_n)] \wedge \\ (\forall \overline{v})\Gamma[P_1 \leftarrow wp(S_1, \mathcal{Q}_1), ..., P_n \leftarrow wp(S_n, \mathcal{Q}_n)]$$

where $[P_1 \leftarrow t_1, ..., P_n \leftarrow t_n]$ denotes n subsequent substitutions performed from left to right, and $\overline{v}$ is a set of free logical variables of Γ.

Second, the constraints together with the definition of wp allow us to prove that VCG (as a proof system) is sound and complete w.r.t. the initial Hoare system in the normal form [6].

On the other hand, the normal form constraints narrow the class of admissible Hoare systems. Note that axiomatic semantics for C-kernel [5], which is an intermediate language of our project, does not satisfy these requirements. Moreover, the normal form rules look quite unusual, which is why Moriconi and Schwartz proposed a more liberal *general form* for rules as well as an algorithm of its translation into the normal one. Here we discuss them only briefly. The general form preserves the constraints (1–3) and (4b) (together with a modified monotonicity property). Thus an awkward order on the premises disappears. Further, the preconditions in premises may take more forms: not only singular predicate symbols but also formulas of the underlying theory, as well as the conjunctions of these two variants. The idea of the translation algorithm is as follows: we gather all preconditions that are different from the singular predicate symbols. Instead of them we will use "fresh" predicate symbols. The connection between these new symbols and old formulas is established by some implications where old formulas may be gathered in conjunctions (simultaneously removing duplicates). Finally, the new rule premises must be reordered to satisfy the constraint (4a).

To illustrate, let us consider the proof rules for `if` and `while` statements in the general (2), (4) and equivalent normal (3), (5) form correspondingly:

$$\frac{\{P \wedge B\}\ S_1\ \{Q\}, \quad \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \texttt{if}\ (B)\ S_1\ \texttt{else}\ S_2\ \{Q\}} \tag{2}$$

$$\frac{\{P_1\}\ S_1\ \{Q\}, \quad \{P_2\}\ S_2\ \{Q\}}{\{B \supset P_1 \wedge \neg B \supset P_2\}\ \texttt{if}\ (B)\ S_1\ \texttt{else}\ S_2\ \{Q\}} \tag{3}$$

$$\frac{\{P \wedge B\}\ S\ \{P\}, \quad P \wedge \neg B \supset Q}{\{P\}\ \texttt{while}\ (B)\ S\ \{Q\}} \tag{4}$$

$$\frac{\{P_1\}\ S\ \{P\}, \quad P \wedge \neg B \supset Q, \quad P \wedge B \supset P_1}{\{P\}\ \texttt{while}\ (B)\ S\ \{Q\}} \tag{5}$$

An intermediate conclusion here is that axiomatic semantics for C-kernel fits the requirements of the general form. So, it can be translated in an equivalent normal system which, in turn, can be transformed into a recursive VCG. Thus the MetaVCG approach can be applied in our case.

III. Implementation and experiments

In this Section we discuss the composing parts of our adaptation of the MetaVCG approach. They include the development of the pattern language which is used to express the Hoare rules and axioms. The main (meta)generation algorithm has been written in C-light, thus making its partial verification possible. An example of the code is also presented here.

First of all, let us note the difference between the original idea of MetaVCG and our implementation. Our metagenerator is a two-parameter function and there is *currying* during its work. So, if H is a Hoare system and AP is an annotated program to be verified, then

$$\mathrm{MetaVCG}(H, AP) = \mathrm{VCG}_H(AP)$$

where VCG_H is an ordinary generator built for H. This is not a good solution from the point of view of efficiency since in every verification experiment (the argument AP) we rebuild the generator even if the Hoare system is the same (say, Hoare system for C-kernel). On the other hand, it allows us to verify a single program instead of two, one of which 'appears' prior to any specification. As long as we are concentrated on theoretical studies, this strategy serves our purposes quite well. In future we may use the generator in a usual way as a stand-alone application or a plug-in.

We also do not restrict our MetaVCG to the *weakest precondition* strategy used by Moriconi and Schwartz. The *strongest postcondition* approach can be applied changing the direction of the program tree analysis.

A. The pattern language

A VCG built from a Hoare system in the normal form tries to instantiate those free predicate symbols and fragment variables with specific annotations and program constructs. Since a user provides MetaVCG with axioms and rules in a less restricted general form, we propose a pattern language to express them.

Let us note that the classic way to represent Hoare logics (like in Section II-A) is good enough in theory but it is not so flexible in practice. That is why we do not require strictly that the symbols P, Q, R express predicates while S_i are program fragments. Any symbols can be used, and membership in a specific class is indicated by syntactic wrapping. For example, the construction `any_code(S)` can match any sequence (including empty) of programming language statements, whereas `exists_code(S)` corresponds to a singular construction. The construction `simple_expression` denotes any expression which does not contain function calls and type casts.

To illustrate this, let us consider the assignment axiom

```
{(any_predicate(Q))
     (MD <- upd(MD, loc(val(e, MeM..STD)),
                cast(val(val(e', MeM..STD)),
                     type(e', MeM, TP),
                     type(e, MeM, TP))))
}
    e = simple_expression(e');
{any_predicate(Q)}
```

and the proof rule for the `while` statement

```
{P1} S {INV},
(INV /\ cast(val(val(e, MeM..STD)),
             type(e, MeM, TP), int) = 0)
    => Q,
(INV /\ cast(val(val(e, MeM..STD)),
             type(e, MeM, TP), int) != 0)
    => P1
|-
{any_predicate(INV)}
    while(simple_expression(e)) any_code(S)
{any_predicate(Q)}
```

To save the space, we show them as if they were already transformed from the general form. That is why two logical statements about predicates `Q` and `P1` appear in the `while`-rule premises. Only then the rule satisfies the constraints of the normal form. The names MD, MeM and STD reflect our detailed memory model [5] but they do not alter principally the logical structure of the familiar Hoare sentences.

B. Implementation of MetaVCG

The arguments of metagenerator — Hoare axioms and rules together with an annotated program — are parsed and transformed into the corresponding internal representations. We have already mentioned that on the lower level the C++ API of the compiler Clang was enabled. Thus actually they are passed from the Clang representation into structures compatible with C-light.

As an example, let us consider the datatype `pattern_node` which represents axioms and conclusions of Hoare rules.

```
struct pattern_node
{
    int is_omitted;

    int has_category;
    char* category;

    int has_identifier;
    char identifier[64];

    int has_type;
    char* type;

    int has_value;
    char* value;

    int is_matched;
    int table_length;
    char match_identifiers[2][1000][64];

    int children_count;
    struct pattern_node* children[1000];
};
```

Since we deal with axiomatic semantics, it is obvious that the first and the last node in the children list are a pre- and post-condition, correspondingly. Each node has attributes (category, identifier, type) which contribute to the matching process. In addition, there is a table of correspondence between the program and pattern names which is filled up during the matching. The program tree is based on the datatype `program_node` which, in general, is similar to `pattern_node`.

Thus the metagenerator builds a program tree for an annotated program and a collection of patterns for an applied Hoare system. According to the proof direction, it chooses the leftmost/rightmost program construction and tries to find an appropriate pattern. For a selected pattern it recursively applies to the premises of the corresponding Hoare rule.

The implementation of MetaVCG is quite large, so let us restrict ourselves to two functions in the rest of this Section. As for the main tree comparison algorithm (programs against patterns), at the moment we use a "greedy" algorithm. Such algorithm can be applied successfully thanks to the simplicity

of the C-light language. Perhaps, consideration of the complete C or complex applied program domain will require a more general approach.

C. Verification examples

When comparing the pattern tree node with a node of the program tree we also match the node attributes, including their identifiers. For convenience, we call the current nodes of the corresponding trees simply as `pattern`, and `code`. If both nodes have identifiers, we must check whether the identifier of `pattern` was early matched against any program construction identifier. This comparison is carried out using a scan of table `match_identifiers`. If the validation fails, then the identifier of `pattern` is associated with the identifier of `code`. So, we want to place this information in the table `match_identifiers`, stored in `pattern`. We use the function `add_identifier`, the annotated definition of which looks like[1]

```
/*@
requires \valid(pattern) && \valid(code);
assigns pattern->table_length;
assigns pattern->match_identifiers[0..1]
           [\old(pattern->table_length)]
           [0..\max(strlen(pattern_identifier),
                        63)];
ensures strncmp(pattern->match_identifiers[0]
                        [pattern->table_length],
                 pattern->identifier, 63);
ensures strncmp(pattern->match_identifiers[1]
                        [pattern->table_length],
                 pattern->identifier, 63);
ensures pattern->table_length =
                  \old(pattern->table_length)+1;
*/
void add_identifier(struct pattern_node* pattern,
                    struct program_node* code)
{
    strncpy(pattern->match_identifiers[0]
                         [pattern->table_length],
            pattern->identifier, 63);
    strncpy(pattern->match_identifiers[1]
                         [pattern->table_length],
            code->identifier, 63);
    pattern->table_length++;
}
```

Another case study directly concerns the tree matching process. When comparing the pattern tree node with a node of the program tree we also need to correlate the node attributes, including attribute `category`. The function `compare_categories` implements such comparison. If the `category` fields of the current tree nodes are equal, then this function returns 1. Otherwise, it returns 0.

```
/*@
requires \valid(pattern) && \valid(code) &&
             pattern->has_category == 1;
behavior comparable:
assumes strlen(pattern_category) ==
            strlen(code_category)
            &&
            \forall int i;
            0 <= i <= strlen(code_category)
            ==>
            pattern_category[i] ==
                  code_category[i];
    ensures \result == 1;

behavior incomparable:
    assumes
       \exist int i;
       0 <= i <= \min(strlen(pattern_category),
                      strlen(code_category))
       &&
       pattern_category[i] != code_category[i];
    ensures \result == 0;
*/
int compare_categories(
      struct pattern_node* pattern,
      struct program_node* code)
{
    int result = 1;

    if ((pattern->has_category) &&
        (strcmp(pattern->category,
                code->category)
        != 0))
    {
        result = 0;
    }
    return result;
}
```

Unlike the code nodes, a pattern node may omit the information about syntactic constructions it can be compared to, i.e. its `category` field can be empty. Usually it takes place when a pattern can match any sequence (including empty) of program constructs. However, such situation is handled somewhere on the outer level, thus we implicitly suppose that `pattern->has_category == 1` when `compare_categories` is invoked.

Based upon specifications of string routines `strlen`, `strncpy` and `strcmp` from our earlier work [11] the verification of these two functions is quite straightforward. The other parts of MetaVCG involve loop invariants or recursive function calls and, thus, are bulky to be described here.

IV. Conclusion

In our project, we are pursuing two objectives. First, we need a collection of VCGs for various classes of programs to simplify the verification. This will make verification available not only for theoreticians, but also for ordinary programmers. Second, we would like to guarantee the correctness of our method as much as possible. Apart from theoretical soundness, its implementation also requires validation. The situation when a verification system is written in the target language gives us an opportunity to apply it to itself.

In order to make the creation of such "convenient and verified verifier" more feasible we adapted the MetaVCG approach. First of all, we adapted the metageneration approach and implemented it with some modification using the C-light language. Then the code was supplemented with ACSL annotations. Let us note that they rely deeply on specifications for the Standard C library (mainly string routines) developed

[1] We use ACSL [2] as a specification language.

in our previous works [11]. Finally, a series of experiments was performed in order to verify the MetaVCG.

It is difficult to carry out a qualitative comparison with related works. It appears that the approach of Moricony and Schwartz has not been used by other researchers. And, despite the fact that there are many verification projects, the studies related to the development of a self-applicable verification system are virtually unknown. In many cases researchers use different languages to implement their systems (like the functional O'Caml in WHY [4]). Others are concentrated on verification of different applications (for example, Hyper-V is studied in detail in the VCC project [3]).

We plan to continue our work on specification and verification of the components of our system. At the moment, only a restricted functionality is expressible in a pure C. Perhaps we will return from C++ API of the Clang compiler to the standard C in order to achieve an ultimate goal — the complete self-verification. As for the development of specialized VCG, the method of finite iterations, developed by Prof. V.A. Nepomnyaschy [7], is a high priority candidate for implementation. It resembles the axiomatic semantics for linear algebra, mentioned in Introduction, and relieves us of loop invariants.

References

[1] Apt K.R., Olderog E.R. Verification of sequential and concurrent programs. — Berlin etc.: Springer, 1991. — 450 p.

[2] Baudin P., Filliâtre J.C., Marché C., Monate B., Moy Y., Prevosto V. ACSL: ANSI/ISO C Specification Language http://www.frama-c.cea.fr/download/acsl_1.4.pdf

[3] Cohen E., Dahlweid M., Hillebrand M.A., Leinenbach D., Moskal M., Santen T., Schulte W., Tobies S. VCC: A Practical System for Verifying Concurrent C // Proc. TPHOLs 2009. — LNCS. — 2009. — Vol. 5674. — P. 23-42.

[4] Filliâtre J.C., Marché C. Multi-prover verification of C programs // Proc. ICFEM 2004. — LNCS. — 2004. — Vol. 3308. — P. 15–29.

[5] I.V. Maryasov, V.A. Nepomnyaschy, A.V. Promsky, D.A. Kondratyev. Automatic C Program Verification Based on Mixed Axiomatic Semantics // Proc. Fourth Workshop "Program Semantics, Specification and Verification: Theory and Applications". — Yekaterinburg, Russia, June 24, 2013. — pp. 50–59.

[6] Moriconi M., Schwartz R.L. Automatic Construction of Verification Condition Generators From Hoare Logics // Lect. Notes Comput. Sci. — Berlin etc., 1981. — Vol. 115. — P. 363–377.

[7] Nepomniaschy V.A. Verification of finite iterations over data structures // Programming. — 2002. — N 1. — pp. 3–12. (In Russian)

[8] Nepomniaschy V.A., Anureev I.S., Mikhailov I.N., Promsky A.V. Verification-oriented language C-light // System informatics. — Novosibirsk, SB RAS publishing house, 2004. — Issue 9: Formal methods and informatics models. — P. 51–134. (In Russian)

[9] Nepomniaschy V.A., Ryakin O.M. Applied methods of program verification. — Moscow, 1988. — 256 p. (In Russian)

[10] Nepomniaschy V.A., Sulimov A.A. Verification of the linear algebra programs in the system SPECTRUM // Cybernetics and system analysis. — 1992. — N 5. — pp. 136–144. (In Russian)

[11] A.V. Promsky. C Program Verification: Verification Condition Explanation and Standard Library // Automatic Control and Computer Sciences, 2012, Vol. 46, No. 7, pp. 394–401.

[12] Promsky A.V. Experiments on self-applicability in the C-light verification system // Bull. Nov. Comp. Center, Comp. Science, 35 (2013), 85–99.

[13] A.V. Promsky Experiments on self-applicability in the C-light verification system. Part 2 // Bull. Nov. Comp. Center, Comp. Science, 37 (2014), 93–105.

Расширение метагенерации условий корректности концепцией семантической разметки

Кондратьев Д.А.
Лаборатория Теоретического Программирования
Институт Систем Информатики им. А.П. Ершова
г. Новосибирск, Россия
apple-66@mail.ru

Верификация программ традиционно основана на генерации условий корректности (УК). В проекте C-light по созданию системы дедуктивной верификации Си-программ возникла задача разработки расширенных или специализированных версий генераторов УК. Для решения данной задачи было решено использовать концепцию метагенерации УК. Получая на вход логику Хоара, метагенератор автоматически порождает рекурсивный алгоритм генерации УК. Идея метагенерации позволила органично дополнить систему C-light и концепцией семантической разметки, ориентированной на такую важную задачу, как анализ, трассировку и объяснение самих УК. Основываясь на предложенным Денни и Фишере методе, опишем такое расширение правил Хоара семантическими метками, что само по себе исчисление может использоваться для построения объяснений УК. Объяснения могут быть построены для различных аспектов УК, в этой статье рассмотрим построение объяснений для их структуры и целей. Также опишем отличие нашего метода от метода Денни и Фишера, заключающееся во введении иерархии на метках.

Ключевые слова — верификация; семантическая метка; условие корректности; язык Си; язык C-light; метод метагенерации

I. Введение

Обеспечение надежности программ является одним из важнейших направлений программирования. Наиболее часто используемым методом обеспечения надежности является тестирование, однако это дорогостоящий и сложный процесс, к тому же с его помощью невозможно обнаружить все ошибки и уязвимости программных продуктов. Совершенно иной подход в решении этой проблемы – это формальная верификация программы. Он является актуальным направлением современного программирования и заключается в формальном доказательстве корректности программ в соответствии с описанием их свойств, задаваемых в виде спецификаций программ.

Работа частично поддержана грантом РФФИ № 15-01-05974 «Онтологический подход к формальной семантике языков программирования».

Наличие формальной семантики для языка программирования является необходимым условием разработки метода верификации программ. В лаборатории теоретического программирования ИСИ СО РАН развивается проект по верификации программ на языке C-light [6], который является представительным подмножеством языка Си. Традиционно, наиболее естественным подходом при конструировании семантики является операционный подход. Операционный подход, фактически, задает некоторую абстрактную машину, исполняющую программу. Однако низкий уровень подобного формализма неудобен для верификации, и, как правило, применяется метод дедуктивной верификации, основанный на аксиоматической семантике.

С другой стороны, аксиоматический поход для такого языка, как C-light, может быть слишком громоздким, поэтому был предложен двухуровневый метод [11] верификации программ. На первом этапе C-light транслируется в промежуточный язык C-kernel с целью элиминации некоторых конструкций, сложных для аксиоматической семантики [15]. Для трансляции используется набор формальных правил [13]. На втором этапе для промежуточной C-kernel программы генерируются условия корректности (УК), которые в дальнейшем передаются на блок доказательства [12]. Генератор условий корректности (ГУК) создается по аксиоматической семантике языка C-kernel. Сама аксиоматическая семантика является набором аксиом и правил вывода для языковых конструкций.

В ходе развития проекта важными стали две задачи. Во-первых, необходим способ простого добавления новых языковых конструкций, не требующий значительной переработки генератора. Во-вторых, интерес представляют узкоспециализированные версии генератора, ориентированные на конкретные методы вывода и классы приложений и позволяющие упростить верификацию. Для решения этих задач было решено использовать предложенную Мориконы и Шварцем [7] концепцию метагенерации условий корректности.

Метагенератор принимает на входе правила вывода аксиоматической семантики в специальном формате и автоматически порождает генератор условий

корректности. Заметим, что порождаемый генератор сам задает некоторую семантику языка программирования. Поэтому, встает вопрос о его непротиворечивости относительно исходной аксиоматической семантики. Эта непротиворечивость обеспечивается тем, что на вход метагенератору подаются только ограниченные правила вывода, находящиеся в нормальной форме [7], а сам алгоритм построения генератора строго формализован.

При практическом применении дедуктивной верификации могут возникнуть следующие проблемы: программа может быть не корректна, ее спецификации могут быть не корректны, автоматический доказатель теорем может не обладать достаточной мощностью, теория предметной области может быть не полна. В этих случаях пользователь системы верификации получает набор недоказанных условий корректности, но не получает дополнительной информации о причинах неудачи. Он должен проанализировать такие условия корректности, изучить их составные части, определить, какие именно правила вывода были применены, и соотнести их с соответствующими фрагментами программы.

Опишем предложенный Денни и Фишером метод семантической разметки, ориентированный на такую важную задачу, как анализ, трассировку и объяснение самих УК. Идея метагенерации позволила органично дополнить систему C-light данной концепцией. Основываясь на предложенным Денни и Фишере методе, опишем такое расширение правил Хоара семантическими метками, что само по себе исчисление может использоваться для построения объяснений УК. Метки проходят различные стадии обработки и переводятся в объяснения на естественном языке. Генерация объяснений основывается только на анализе меток, а не на анализе логических значении УК или их доказательств. Объяснения могут быть построены для различных аспектов УК, в этой статье рассмотрим построение объяснений для их структуры и целей. Также опишем отличие нашего метода от метода Денни и Фишера, заключающееся во введении иерархии на метках.

Работа имеет следующую структуру. В разделе 1 даются предварительные сведения, необходимые для понимания дальнейшего материала. В разделе 2 рассмотрены теоретические основы метода метагенерации условий корректности и метода Денни и Фишера. В разделе 3 описан язык представления правил вывода. В разделе 4 описана реализация метагенератора и поддержки семантических меток в проекте C-light. В заключении сформулированы результаты проделанной работы

II. Теоретические основы

Рассмотрим в этом разделе основные понятия аксиоматической семантики, метод метагенерации и метод семантической разметки.

A. Аксиоматическая семантика

В 1964 году Ч. Хоар [1] ввел способ задания аксиоматической семантики, ставший основой метода дедуктивной верификации программ. Подход Хоара заключается в том, чтобы представлять текст программы как особое отношение между утверждениями. Базовыми формулами в рассматриваемом подходе являются тройки Хоара {P} S {Q}, где P — предусловие (логическая формула), S — программа, Q — постусловие (логическая формула). Истинность формулы {P} S {Q} означает, что «если предусловие P истинно перед исполнением фрагмента программы S, и, если исполнение S завершилось, тогда постусловие Q выполняется после его завершения» [7]. Правила вывода задаются в виде

$$\frac{\varphi_1, \ldots, \varphi_n}{\psi},$$

где $\varphi_1, \ldots, \varphi_n$ — посылки правила вывода (набор троек Хоара и логических формул) и ψ — заключение правила вывода (тройка Хоара). Данная нотация означает, что ψ выводимо при гипотезе $\varphi_1, \ldots, \varphi_n$. Семантика простых операторов (например, присваивания) задается, как правило, с помощью набора аксиом, а любого сложного оператора (например, оператора последовательного исполнения) — с помощью правила вывода. Логическая система, содержащая аксиомы и схемы вывода для всех синтаксических форм языка программирования, называется логикой Хоара или аксиоматической семантикой языка.

Генератор условий корректности, фактически, осуществляет вывод в автоматическом режиме по аксиоматической семантике и сводит истинность тройки Хоара {P} S {Q} к корректности некоторого числа лемм, называемых условиями корректности, в предметной области. Доказуемости этих лемм достаточно для корректности исходной аннотированной программы.

Одним из способов реализации генератора условий корректности является рекурсивно определенный преобразователь предиката pre(S, Q), отображающий фрагмент программы S и постусловие Q в предусловие. Функция pre вычисляет утверждение, достаточное, чтобы гарантировать, что Q будет являться постусловием (т. е., что тройка Хоара {pre(S, Q)} S {Q} доказуема). Доказуемости условия корректности $P \supset pre(S, Q)$ в теории, описывающей предметную область, достаточно, чтобы показать, что {P} S {Q} доказуема в любой системе Хоара, содержащей правило для композиции. Функция wdp(S, Q) вычисляет слабейшее предусловие, т.е. такое предусловие, что для любого другого предусловия R должно быть верно $R \supset wdp(S, Q)$ для программы S и постусловия Q. Для доказуемости тройки Хоара {P} S {Q} необходимо и достаточно, чтобы P было эквивалентно wdp(S, Q) [7].

B. Метод метагенерации

Метод метагенерации, предложенный Морикони и Шварцем [7], состоит из двух шагов. На первом из них

обычная аксиоматическая семантика переводится в нормальную форму, по которой затем создается рекурсивно определенный генератор условий корректности. Что касается генератора условий корректности, то для его построения используется метод слабейшего предусловия [1].

Следуя Морикони и Шварцу, мы будем использовать метапеременные P, Q, Г, ... для обозначения частично интерпретированных формул первого порядка. Рассмотрим бинарное отношение $\Leftarrow$ на неинтерпретированных предикатных символах. Для тройки Хоара вида

$$\{P(P_1, ...,P_m)\}\ S\ \{Q(Q_1, ...,Q_n)\}$$

где предикатные символы $P_1,...,P_m$ и $Q_1,..., Q_n$ свободно входят в P и Q соответственно, имеем

$$P_i \Leftarrow Q_j, \text{ для } i \in \{1, ..., m\} \text{ и } j \in \{1, ..., n\}$$

Нотация $\overset{+}{\Leftarrow}$ обозначает транзитивное замыкание $\Leftarrow$.

Аналогично, для правила в форме

$$\frac{\{P_1\}\ S_1\ \{Q_1\}, ..., \{P_n\}\ S_n\ \{Q_n\}, \Gamma}{\{P\}\ S\ \{Q\}},$$

отношение << определяет зависимость выводимости S от выводимости S_1, ..., S_n. В частности, верно $S << S_i$ для $i \in \{1,...,n\}$. Для системы аксиом логики Хоара очевидным образом определим транзитивное замыкание <<+.

Мы используем функцию *FreePreds*, чтобы обозначить множество свободных предикатных символов. Мы будем использовать функцию *FragVars* для обозначения множества "фрагментных переменных" (в терминах Морикони и Шварца) в фрагменте программы S из тройки Хоара $\{P\}\ S\ \{Q\}$.

Определим нотацию для связанного вхождения неинтерпретированного предикатного символа в правило. Для правила R предикатный символ из *FreePreds*(R) является связанным в R, если он входит в *FragVars*(R). Иначе, вхождение будет являться свободным в R.

Определение. Правило в нормальной форме — это любое правило N вида:

$$\frac{\{P_1\}\ S_1\ \{Q_1\}, ..., \{P_n\}\ S_n\ \{Q_n\}, \Gamma}{\{P\}\ S\ \{Q\}},$$

которое удовлетворяет следующим ограничениям:

1) $P_1,..., P_n$ и Q являются предикатными символами, свободно входящими в N.

2) Γ является предложением в предметной области, чьи свободно входящие предикатные символы могут входить только в *FreePreds(N)* или во *FragVars(S)*.

3) Фрагментные переменные из каждого S_i должны быть связанными в S. То есть, это должно означать, что $\cup_{1 \le i \le n} FragVars(S_i) \subseteq FragVars(S)$.

4) Порядок зависимостей. Тройки Хоара из посылок N должны удовлетворять двум ограничениям:

a) $P_i \overset{+}{\Leftarrow} P_j \supset i < j$.

b) $T \overset{+}{\Leftarrow} U \wedge \neg(\exists R) U \overset{+}{\Leftarrow} R \supset U \equiv Q \vee U$ *связан в N*

5) Монотонность. Пусть P$[P \leftarrow false,\ P \in s]$ обозначает P прямое с заменой на false всех предикатов P из множества s. Тогда, следующее ограничение на P должно быть удовлетворено:

$$P[P_1,..., P_n, Q \leftarrow \text{true}] \vee \forall s \subseteq \{P_1,..., P_n, Q\}\ \neg P[P \leftarrow \text{false}, P \in s]$$

Это ограничение должно выполняться для Γ и для всех Q_i.

Два ограничения применяются к набору правил вывода, находящихся в нормальной форме: (i) Любая терминальная строка σ в языке программирования может быть представлена не более чем одним фрагментом S, входящем либо в аксиому, либо в правило вывода. (ii) Отношение <<+ должно быть иррефлексивным.

Необходимо отметить, что ограничения нормальной формы приводят для правил вывода к появлению порядка на посылках и к нетипичному виду предусловий в тройках Хоара.

Для находящегося в нормальной форме правила вывода вида

$$\frac{\{P_1\}\ S_1\ \{Q_1\}, ..., \{P_n\}\ S_n\ \{Q_n\}, \Gamma}{\{P\}\ S\ \{Q\}}$$

слабейшее предусловие (wdp) определяется следующим образом:

$$wdp(S, Q) = P[P_1 \leftarrow wdp(S_1, Q_1), ..., P_n \leftarrow wdp(S_n, Q_n)] \wedge$$
$$(\forall \bar{v})\Gamma[P_1 \leftarrow wdp(S_1, Q_1), ..., P_n \leftarrow wdp(S_n, Q_n)],$$

где $[P_1 \leftarrow t_1, ..., P_n \leftarrow t_n]$ обозначают n подстановок, выполненных последовательно слева направо, и $\bar{v}$ является множеством всех свободных логических переменных в Г.

Напомним, что ограничения нормальной формы приводят к появлению порядка на посылках и к непривычному виду предусловий в тройках Хоара. Для решения данной проблемы Морикони и Шварц ввели менее ограниченную общую форму.

Заметим, что алгоритм построения генератора условий корректности по правилам вывода применяется к нормальной форме. Чтобы избавить пользователя от необходимости перевода правил вывода из общей формы в нормальную, можно позволить ему задавать правила вывода в общей форме. А сам перевод доверить алгоритму, формализованному Морикони и Шварцем [7].

Условия, при которых порождаемый генератор условий корректности является непротиворечивым и полным как система вывода относительно поданной на вход метагенератору аксиоматической семантики были найдены Морикони и Шварцем. Чтобы обеспечить

корректность и полноту порождаемого генератора необходимо и достаточно подавать на вход метагенератору только ограниченные правила вывода, находящиеся в нормальной форме.

C. Метод семантической разметки

Структура УК. После упрощения, УК обычно имеют вид Хорновских дизъюнктов (т.е. $H_1 \wedge \ldots \wedge H_n \supset C$). В данном представлении единственное заключение C можно рассматривать как *цель*. Однако, для более осмысленного описания структуры, мы должны дать более детальную характеристику подформул. Такую информацию нельзя восстановить, исходя только из условий корректности или кода программы, поэтому, ее необходимо задавать явно. Ключевая особенность подхода Денни и Фишера [2] состоит том, что различные подформулы располагаются на специальных позициях в правилах Хоара, и, исходя из этого, ГУК добавляет соответствующие метки к УК.

Концепции. Базовой информацией для генерации объяснений является множество основных концепций, которое зависит от специфических аспектов объясняемых УК. Для объяснения структуры УК большинство концепций отражают *позицию* подформулы в нем. Другие концепции передают информацию о первичной и вторичной *цели* подформулы.

Гипотезы подразделяются на *утверждения* и на *предикаты, управляющие потоком исполнения*. Утверждениями называются вошедшие в УК аннотации из программы. Они включают в себя предусловия и постусловия (метки *ass_pre* и *ass_post*), предусловия и постусловия функций (метки *ass_fpre* и *ass_fpost*) и инварианты циклов. Так как правило вывода для циклов использует инварианты как гипотезы в двух разных позициях и целях, то для подчеркивания различий были введены метки *ass_inv* и *ass_inv_exit*. Управляющими потоком исполнения предикатами называются подформулы в УК, отображающие ход потока исполнения программы. Для оператора *if* и цикла *while* предикаты, управляющие потоком исполнения, используются в формах с отрицанием и без него, приводя к появлению четырех разных меток: *if_ff*, *if_tt*, *while_ff* и *while_tt*.

Заключения передают информацию об основной цели УК. Как и в случае гипотез, инварианты используются в заключениях в двух разных формах: для обозначения входа в цикл (метка *est_inv*) и для обозначения итерации цикла (метка *est_inv_iter*). Заметим, что одна и та же подформула может являться и гипотезой, и заключением, как в разных УК, так и в одном УК.

Уточнители характеризуют и гипотезы, и заключения, передавая информацию о том, как подформула появилась в УК и как она была преобразована. Например, разные концепции *подстановки* отражают подстановки в используемом исчислении Хоара. Концепции *sub* и *upd* передают то, как сказывается эффект от присваивания и обновления массива на УК.

Индуктивные уточнители передают второстепенную цель УК. Они вводятся, когда рекурсивный вызов ГУК для вложенной языковой конструкции порождает УК, цель которых концептуально связана с целью УК для объемлющей конструкции. Например, такая ситуация характерна для цикла, вложенного в другой цикл.

Структура метки. Будем использовать для помеченных термов нотацию $\ulcorner t \urcorner^{l}$, означающую, что терму t сопоставляется метка l или список меток l. Метки имеют вид c(o, n). Здесь c – одна из концепций, описывающая предназначение данного терма, и то, как обрабатывать такую метку. Позиция o передает местоположение в программе связанных с данной меткой конструкций и представляет собой диапазон строк. Список меток n содержит дополнительную уточняющую информацию для рассматриваемой метки. Сначала, для данного вложенного терма список n пуст, но после нормализации он содержит метки, «просочившиеся» из объемлющих термов.

Модифицированные правила Хоара. Обычно, недостаточно просто генерировать объяснение того, как УК было получено. Вместо этого, ГУК должен добавлять метки в тех местах, где это нужно. Также он должен предоставлять удобный интерфейс для доступа к разметке. Это нужно для работы с метками, в том числе, и для генерации по ним объяснений. В модифицированные правила Хоара добавляется семантическая разметка (тип меток и их позиции), нужная для объяснения результата применения правила. Метки добавляются к следующим местам: к постусловию посылки, получившемуся при рекурсивном вызове ГУК, к слабейшему предусловию, к сгенерированному УК или к тройке Хоара. Далее, для ясности изложения, опустим в формулировке правила относящуюся к меткам информацию о позиции, но, полагаем, что ГУК получает ее от программных конструкций и аннотаций и добавляет в метки. Рассмотрим в качества примера находящее в нормальной форме правило вывода для while в модели памяти языка C-kernel:

$$\frac{\begin{array}{l} \ulcorner \{P_1\}\ S\ \{\ulcorner INV \urcorner^{est_inv_iter}\} \urcorner^{pres_inv}, \\ (\ulcorner INV \urcorner^{ass_inv_exit} \wedge \\ \quad \ulcorner cast(val(val(e, MeM..STD)), \\ \qquad type(e, MeM, TP), int) = 0) \urcorner^{while_ff} \supset Q, \\ \ulcorner (\ulcorner INV \urcorner^{ass_inv} \wedge \\ \quad \ulcorner cast(val(val(e, MeM..STD)), \\ \qquad type(e, MeM, TP), int) \neq 0) \urcorner^{while_tt} \supset P_1 \urcorner^{pres_inv} \end{array}}{\{\ulcorner INV \urcorner^{est_inv}\}\ while(e)\ S\ \{Q\}}$$

Рассмотрим структуру данного правила вывода. Чтобы вывести тройку Хоара для цикла while, необходимо использовать индукцию. Ввести индукцию позволяет специальная формула, приписываемая циклу и называющаяся инвариантом цикла. Инвариант цикла – это утверждение, которое истинно перед исполнением цикла, истинно для каждой итерации цикла и обеспечивает корректность на выходе из цикла. Таким образом, посылками правила вывода для цикла с предусловием являются тройка Хоара, соответствующая итерации цикла

без выхода из него, связанная с ней формула, обозначающая итерацию цикла, и формула, соответствующая выходу из цикла. Слабейшее предусловие включает в себя инвариант, отвечающий за вход в цикл, и, поэтому, имеющий метку est_inv. В посылках обособленные подформулы обеих формул, обозначающих выход из цикла и итерацию цикла, имеют соответствующие метки. Также, вся формула в посылке, обозначающая итерацию цикла, имеет метку для обозначения ее второстепенной цели – способствовать сохранению инварианта. В тройке Хоара в посылке постусловие INV должно иметь метку, обозначающую ее цель (т.е. гарантирование выполнения инварианта после каждой итерации цикла) для передачи информации при рекурсивном вызове. Более того, все УК, выведенные от данной тройки, должны иметь метку pres_inv, обозначающую второстепенную цель такого УК. Это обеспечивается тем, что вся рассматриваемая тройка имеет такую метку. Заметим, как одна и та же формула INV имеет четыре различные роли и, поэтому, имеет четыре различные метки. Эта информация следует из контекста, доступного только при применении правила вывода, и не может быть восстановлена простым способом при анализе полученных УК без использования семантической разметки.

Упрощение имеющих метки УК. УК (имеющие метки или нет) обычно являются довольно сложными и, поэтому, они нуждаются в упрощении перед их проверкой на автоматическом доказателе теорем. Правила для упрощения не имеющих меток УК не могут быть просто переиспользованы для имеющих метки УК по следующим причинам:

1) Семантическая разметка изменяет структуру терма, и, соответственно, применимость правила.

2) Метки нужно аккуратно обрабатывать – с одной строны они не могут быть просто распространены на все операторы без нарушения их области действия. С другой стороны, они не могут быть просто подняты на самый верхний уровень УК, так как это может привести к генерации слишком подробных и неточных объяснений.

Правила упрощения имеющих метки УК должны достигать следующих целей:

1) Удаление лишних меток.

2) Минимизация области действия оставшихся меток.

3) Сохранение достаточного числа меток для объяснения любых неожиданных неудач, основываясь на допущении, что большая часть УК может считаться доказанными.

Такие правила подразделяются на пять различных групп. Будем использовать вспомогательную функцию $|\cdot|$ для удаления меток из термов и оператор композиции меток $\otimes$ для добавления внутреннего списка меток к списку меток, вложенному во внешнюю метку, т.е. $c(o, l) \otimes m = c(o, l \bullet m)$, где $\bullet$ – конкатенация списков. Первая группа содержит правила, такие как $\ulcorner true \urcorner^l \to true$ или $P \supset P' \to true$ (если $|P| = |P'|$), удаляющие метки из тривиальных тождественно истинных (под-)формул, потому что их не нужно объяснять. Следующая группа содержит правила, такие как $\ulcorner false \urcorner^l \vee P \to P$, которые выборочно удаляют имеющие метки тождественно ложные подформулы. В такой ситуации информацию для генерации объяснения предоставляет оставшийся контекст. Правила из следующей группы, заменяющие всю формулу на false, например $\ulcorner false \urcorner^l \wedge P \to \ulcorner false \urcorner^l$, должны сохранять метки, так как иначе не остается контекста для объяснения неудачного доказательства. Правила $\ulcorner P \wedge Q \urcorner^l \to \ulcorner P \urcorner^l \wedge \ulcorner Q \urcorner^l$ и $P \supset \ulcorner Q \supset R \urcorner^l \to P \wedge \ulcorner Q \urcorner^l \supset \ulcorner R \urcorner^l$ входят в состав четвертой группы. Они осуществляют «просачивание» меток через конъюнкцию и (вложенную) импликацию, соответственно, область действия меток минимизирована в полученных после упрощения УК. Последняя группа основывается на знаниях, как интерпретировать метки в предметной области. Эта группа также содержит устраняющее вложенность правило $\ulcorner \ulcorner t \urcorner^n \urcorner^m \to \ulcorner t \urcorner^{n \otimes m}$, которое «просачивает» вложенные метки, и, тем самым, позволяет применять другие имеющие или не имеющие метки правила, сохраняя при этом вложенную структуру самих меток. Это обеспечивает правильную вложенность уточнителей и сохранение их первоначальной принадлежности к определенному терму.

Трансляция. Последней стадией работы с (имеющими метки) УК является генерация объяснений для них, т.е. превращение их в понятный для пользователя текст, называемое *трансляцией*. Структуру и текстовое представление объяснений можно задать как грамматику, в которой правая часть каждого правила представляет собой шаблон для их генерации, похожий на форматную строку из языка Си. Эти шаблоны объяснений позволяют удобно и детально задавать текст, который будет содержаться в генерируемых объяснениях. Для генерации по УК текстового представления транслятор выделяет из данного УК метки и сортирует их по номерам строк, получая при этом список меток. Затем, транслятор сопоставляет этому набору шаблон объяснения, используя правила для генерации текстовых представлений. В трансляторе по умолчанию заданы правила генерации текстовых представлений для часто встречающихся концепций, таких как sub и upd. Трансляция включает в себя четыре шага: (i) Нормализация УК с использованием правил их упрощения; (ii) Извлечение меток после применения правила устранения вложенности; (iii) Нормализация меток для сопоставления меткам шаблонов объяснений; (iv) Генерация текста с использованием шаблонов объяснений. На третьем шаге происходит устранение вложенности в параметрах метки, например метка sub(p, sub(q, sub(r))) превращается в список меток ⟨sub(p), sub(q), sub(r)⟩.

III. Язык шаблонов

Напомним, что правила в нормальной форме, поступающие на вход метагенератору, задают собой шаблоны для сопоставления с программными

конструкциями. Классическое графическое представление правил вывода удобно для чтения, но мало подходит для ввода их с клавиатуры. Поэтому, важной задачей является разработка для них некоторого языка. Очевидно, что в основе этого языка лежит язык логики первого порядка и целевой язык программирования. Вместе с тем, в этот язык необходимо добавить некоторые метаобозначения, поскольку правила вывода задают семантику не конкретных программ, а схем программ.

Заметим, что в соответствии с идеей метагенерации, в общем случае разрабатывается схема языка шаблонов. В данном разделе рассмотрим случай применения этой схемы для языка C-light.

A. Язык для написания шаблонов

Основой языка шаблонов для C-light является логика первого порядка и грамматика языка Си [8]. Введение метаобозначений в этот язык фактически означает, что в его выражениях сохраняются некоторые нетерминальные символы (например, неинтерпретированные предикатные символы P, Q, R и фрагментные переменные для обозначения фрагментов кода). Принадлежность метаданных тому или иному классу в языке шаблонов задается в явном виде. Опишем далее способы задания этой информации в языке шаблонов для случая языка C-light вместе с ограничениями на них.

1) Любой идентификатор в шаблоне является метаидентификатором. Таким образом, на место любого идентификатора `ID1` в шаблоне может быть подставлен любой идентификатор `ID2` из программной конструкции на языке Си, которая проверяется на соответствие шаблону. При этом идентификатор `ID2` должен удовлетворять тем же самым семантическим ограничениям, что и идентификатор `ID1`. Например, если из контекста шаблона можно сделать однозначный вывод, что идентификатор `ID1` – это идентификатор функции с двумя аргументами, то и идентификатор `ID2` должен быть идентификатором функции с двумя аргументами. Пусть `pattern_identifier` – любой идентификатор шаблона. Кроме того, пусть `program_identifier` – любой идентификатор программной конструкции. Тогда, если при сопоставлении шаблона, содержащего более одного вхождения `pattern_identifier`, этой программной конструкции одному из вхождений `pattern_identifier` был сопоставлен `program_identifier`, то и всем остальным вхождениям `pattern_identifier` должен быть сопоставлен `program_identifier`. Например, если в шаблоне одному из вхождений идентификатора `i` была сопоставлена переменная `j` из программной конструкции, то и всем остальным вхождениям `i` должна быть сопоставлена только переменная `j`.

2) Помимо идентификаторов к фрагментным переменным в шаблонах относятся и более «крупные» конструкции. Для их задания используются ключевые слова `any_code`, `exist_code` и `simple_expression`. Синтаксическая конструкция `any_code` может быть сопоставлена любому, в том числе и пустому, набору программных конструкций, последовательно стоящих в программе. Пусть `construction` – одна из специальных синтаксических конструкций и пусть `construction_identifier` обозначает уникальный идентификатор. Запись `construction(construction_identifier)` в шаблоне означает, что у данного вхождения construction вводится идентификатор `construction_identifier`. Программная конструкция, сопоставленная такому вхождению construction, будет обозначаться как `construction_identifier`. Введение такого обозначения необходимо для того, чтобы описать некоторые правила вывода. Например такие, у которых программная конструкция, сопоставленная такому вхождению `construction`, используется в их посылках. Синтаксическая конструкция `exist_code` должна быть сопоставлена одной программной конструкции. Синтаксическая конструкция `simple_expression` должна быть сопоставлена выражению, не включающему вызовы функций и операторов приведения.

3) Для метаданных в спецификациях по аналогии используется явный способ задания.

В качестве примера рассмотрим синтаксическую конструкцию `any_predicate`. Ей должно быть сопоставлено одно утверждение из спецификаций. Запись `any_predicate(construction_identifier)` означает, что у данного вхождения конструкции имеется уникальный идентификатор `construction_identifier`. Когда конструкция `any_predicate` имеет уникальный идентификатор, то этот идентификатор можно использовать в спецификациях из посылки правила вывода.

Таким образом, наличие идентификатора у специальных синтаксических конструкций позволяет расширить класс правил вывода, которые можно написать на рассматриваемом языке.

Важное ограничение на язык шаблонов состоит в следующем: на одном и том же уровне вложенности для шаблона и программного фрагмента, у шаблона должна быть либо одна конструкция `any_code`, либо конструкции `any_code` должны быть только последними на данном уровне. Правила вывода для C-kernel в нормальной форме удовлетворяют этому ограничению, что следует из жестких ограничений, наложенных на данный язык. Это ограничение необходимо для применимости «жадного» алгоритма сопоставления программных конструкций и шаблонов.

B. Пример задания аксиомы и правила вывода

Рассмотрим пример аксиомы для C-kernel из [16]. Пусть выражение e' не включает вызовы функций и операторов приведения. Аксиома для операции присваивания имеет вид:

```
{Q(MD ← upd(MD,
            loc(val(e, MeM..STD)),
            cast(val(val(e', MeM..STD)),
                type(e', MeM, TP),
                type(e, MeM, TP))))}
e = e';
{Q}
```

Данная аксиома находится в нормальной форме [7]. Имена MeM, MD отражают специфику модели памяти языка C-kernel. На рассматриваемом языке для написания аксиом и правил вывода данная аксиома записывается так:

```
{(any_predicate(Q))(MD <- upd(MD,
            loc(val(e, MeM..STD)),
            cast(val(val(e', MeM..STD)),
                type(e', MeM, TP),
                type(e, MeM, TP)))))}
e = simple_expression(e');
{any_predicate(Q)}
```

Шаблоном в этой аксиоме является конструкция:

```
e = simple_expression(e');
```

Такая запись означает, что программная конструкция, соответствующая данному шаблону, должна представлять собой операцию присваивания, операндами которой служат `e` и `e'`, и `e'` не включает вызовы функций и операторов приведения.

Рассмотрим пример правила вывода для C-kernel. Правило для while в модели памяти языка C-kernel имеет следующий вид:

$$\frac{\{INV \wedge cast(val(val(e, MeM..STD)), type(e, MeM, TP), int) \neq 0\}\ S\ \{INV\}}{\{INV\}\ while(e)\ S\ \{INV \wedge cast(val(val(e, MeM..STD)), type(e, MeM, TP), int) = 0\}}$$

В нормальной форме эквивалент этого правила выглядит так:

```
{P1} S {INV},
(INV /\ cast(val(val(e, MeM..STD)),
       type(e, MeM, TP), int) = 0) => Q,
(INV /\ cast(val(val(e, MeM..STD)),
       type(e, MeM, TP), int) != 0) => P1
|-
{any_predicate(INV)}
 while(simple_expression(e)) any_code(S)
{any_predicate(Q)}
```

Посылки отделяются от заключения знаком "`|-`", который представляет собой вертикальную черту "`|`" и следующий за ней дефис "`-`". Два косые черты (прямая и обратная) "`/\`" обозначают конъюнкцию.

Шаблоном в этом правиле вывода является конструкция:

```
while(simple_expression(e)) any_code(S)
```

Такая запись означает, что программная конструкция, соответствующая данному шаблону, должна представлять собой цикл, у которого есть условие `e` и может присутствовать тело `S`. Отметим, что `e` и `S` являются фрагментными переменными [7].

C. Расширение языка шаблонов семантическими метками

Для локализации ошибок в проекте C-light, как и у Денни и Фишера [2], используются семантические метки. Семантическая метка приписывается терму в условии корректности и имеет вид c(o), где c – тип метки (характеризующий предназначение метки), o – местоположение относящегося к данной метке программного кода (диапазон строк). В связи с введением меток в проект C-light, в качестве аксиоматической семантики языка стали использоваться размеченные правила вывода, отличающиеся от прежних тем, что в них были добавлены метки. Соответственно, язык описания правил вывода был расширен специальной конструкцией `label`, используемой для описания меток. Конструкция `label` имеет вид `(label t c)`, где `t` – терм, к которому приписана метка, а `c` – строка (тип метки). В качестве примера приведем описание на рассматриваемом языке, расширенном метками, находящегося в нормальной форме правила для while:

```
(label {P1} S {(label INV est_inv_iter)}
 pres_inv),
((label INV ass_inv_exit) /\
 (label
   cast(val(val(e, MeM..STD)),
        type(e, MeM, TP), int) = 0
                        while_ff)) => Q,
(label
 ((label INV ass_inv) /\
  (label
    cast(val(val(e, MeM..STD)),
         type(e, MeM, TP), int) != 0
                        while_tt)) => P1
                               pres_inv)
|-
{(label any_predicate(INV) est_inv}
 while(simple_expression(e)) any_code(S)
{any_predicate(Q)}
```

Конструкция `label` обеспечила переход в проекте C-light от обычных правил Хоара к размеченным.

D. *Задание шаблонов объяснений*

Для генерации текста объяснения УК в нашем подходе, как и в подходе Денни и Фишера, используются шаблоны объяснений. Они задаются для каждой концепции метки с помощью специальной конструкции `label_pattern`, имеющей вид `(label_pattern label format_text)`, где `label` – концепция метки, а `format_text` – форматная строка, задающая текстовый шаблон. Он представляет собой описывающий концепцию метки текст на естественном языке, дополненный специальными конструкциями `%begin` и `%end`, на место которых будут подставлены соответственно начало и конец диапазона строк, в котором записан относящийся к данной метке программный код.

IV. Реализация метагенератора

Схема работы метагенерации состоит в переработке правил вывода в рекурсивную процедуру генерации условий корректности. В этой процедуре происходит сопоставление программных конструкций и шаблонов. Приведем примеры типов данных, которые используются при реализации этой схемы. У нас есть два ключевых типа данных – программа и шаблон. Входными данными являются шаблоны и программные конструкции, которые описываются с помощью соответствующих программных типов данных. В контексте задачи самоверификации отметим, что все эти типы данных вкладываются в язык Си. Реализация состоит в погружении входных данных в программные типы данных и из функций для работы с ними. Рассмотрим некоторые из них.

A. *Структура program_node*

Программа, поступающая на вход метагенератору, представляется в виде дерева. Рассмотрим структуру `program_node` [14], которая задает этот тип данных:

```
struct program_node
{
   char* category;
   int has_identifier;
   char identifier[64];
   int has_type;
   char* type;
   int has_value;
   char* value;
   int children_count;
   struct program_node* children[1000];
};
```

Поскольку для нас важной является задача самоверификации метагенератора, в текущей его реализации есть ряд ограничений. В частности, предполагается, что длина идентификаторов в программе не может быть более 64 символов, и у одного узла не может быть более 1000 потомков, при этом используются статические массивы. В противном случае, необходимо использовать функции для работы с динамической памятью, что усложняет верификацию из-за относительно громоздких спецификаций данных функций [9]. Заметим, что принцип модульной верификации и производительность автоматических доказателей теорем позволяют говорить о разумности этих ограничений.

В поле `category` содержится информация о том, какая именно синтаксическая конструкция хранится в данном узле. Например, для условного оператора поле `category` примет значение "`if_statement`", а для цикла while поле `category` примет значение "`while_statement`". Поле `category` заполнено у каждого узла рассматриваемого дерева.

У некоторых синтаксических конструкций, например, у переменных, имеется идентификатор. Поле `has_identifier` принимает значение `1`, если у синтаксической конструкции, представленной в рассматриваемом узле дерева имеется идентификатор, и принимает значение `0` в обратном случае. Соответственно, поле `identifier` содержит идентификатор в случае его наличия.

По аналогии с предыдущим, синтаксическая конструкция может иметь тип и значение. Поэтому, мы используем поля `has_type` и `type`, `has_value` и `value`. Например, у синтаксической конструкции, отображающей вызов функции, типом будет являться тип возвращаемого значения.

Поле `children_count` содержит количество потомков данного узла. А поле `children` является массивом указателей на потомков данного узла.

B. *Структура label*

В ходе работы метагенератора термам сопоставляются характеризующие их метки. Рассмотрим структуру, задающую этот тип данных:

```
struct label
{
   char* concept;
   int location_begin;
   int location_end;
}
```

В данной структуре поле concept содержит концепцию метки, описывающую предназначение сопоставленного метки терма. Это поле заполняется при синтаксическом анализе аксиом и правил вывода УК. Поля `location_begin` и `location_end` содержат начало и конец диапазона строк, в котором записан относящийся к данной метке программный код. Эти поля заполняет ГУК, так как именно при применении аксиом и правил вывода известна информация о позиции программного кода.

Информация о метках должна содержаться во внутреннем представлении спецификаций. Оно представляет собой дерево, образованное структурой term_node (для краткости изложения не будем приводить определения данной структуры). В ходе работы ГУК терму, имеющему метку, может быть сопоставлена другая метка и т.д. Эти метки хранятся в соответствующем узле дерева спецификаций в виде списка, в котором последним элементом является метка самого верхнего уровня. Данный список представлен в структуре term_node атрибутом labels.

C. *Структура pattern_node*

Метагенератор производит анализ правил вывода, подаваемых ему на вход, и в результате получает множество шаблонов. Шаблоны представляется в виде деревьев. Рассмотрим структуру pattern_node [14], представляющую такое дерево (для краткости изложения не будем приводить определения данной структуры).

В данной структуре поля category, has_identifier, has_type, has_value, value, children_count, children имеют ту же самую семантику, что и соответствующие поля в структуре program_node.

Пусть для языка написания шаблонов construction – одна из специальных синтаксических конструкций и пусть construction_identifier обозначает уникальный идентификатор. У узла дерева шаблона, соответствующего записи construction(construction_identifier), атрибут has_identifier имеет значение 1, а атрибут identifier имеет значение construction_identifier.

Но у узла дерева шаблона, в отличие от узла дерева программы, поле category может быть не заполнено. То есть, узел шаблона может не хранить информацию о том, какому классу синтаксических конструкций он может быть сопоставлен. Поэтому, в структуре pattern_node присутствует поле has_category, которое принимает значение 1, когда поле category заполнено, и принимает значение 0 в обратном случае.

Также в рассматриваемой структуре присутствует поле is_omitted, принимающее либо значение 1, либо значение 0. Поле is_omitted принимает значение 1 тогда и только тогда, когда узел шаблона может быть сопоставлен любому, в том числе и пустому, набору программных конструкций, последовательно стоящих в программе.

Конструкция any_code представляется в дереве шаблона с помощью узла, у которого атрибут is_omitted имеет значение 1. Также у такого узла атрибуты has_category, has_type, has_value имеют значение 0. Кроме того, у такого узла отсутствуют дочерние узлы и атрибут children_count имеет значение 0.

Конструкция exist_code представляется в дереве шаблонов также, как и другие синтаксические конструкции из шаблона. То есть, конструкции exist_code в дереве шаблонов соответствует узел, у которого могут быть заполнены некоторые атрибуты.

Любая структура pattern_node является корнем дерева, хранящем часть шаблона. Поле is_matched содержит информацию о том, удалось ли сопоставить это дерево какой-либо программной конструкции. Если сопоставить удалось, то поле is_matched принимает значение 1, иначе – значение 0.

Шаблон может быть сопоставлен с программной конструкцией только когда идентификаторы, содержащиеся в нем, сопоставлены с идентификаторами, содержащимися в ней. То есть, если в некоторой части шаблона идентификатор ID1 сопоставлен идентификатору ID2 из программной конструкции, то в других частях шаблона идентификатору ID1 может быть сопоставлен только идентификатор ID2.

Для установления такого соответствия между идентификаторами в структуре pattern_node добавлено поле match_identifiers. Фактически это поле представляет собой два массива строк, расположенных по нулевому и первому индексу первого измерения массива match_identifiers. Массив строк, расположенный по нулевому индексу первого измерения массива match_identifiers, – это массив сопоставленных идентификаторов шаблона. А массив строк, расположенный по первому индексу первого измерения массива match_identifiers, – это массив сопоставленных идентификаторов программной конструкции.

Для удобства назовем массив сопоставленных идентификаторов шаблона нулевым массивом, а массив сопоставленных идентификаторов программной конструкции – первым массивом. Эти два массива имеют одинаковую длину, хранящуюся в поле table_length. Если в нулевом массиве некоторый идентификатор ID1 имеет индекс i, то в первом массиве индекс i должен иметь идентификатор ID2, сопоставленный идентификатору ID1. Таким образом хранится информация о сопоставленных идентификаторах.

D. *Анализ шаблонов и программы*

Напомним, что для обозначения программного кода в заключении правила вывода используется целевой язык, совпадающий с языком Си, и имеющий некоторые модификации. Так как шаблоны поступают на вход метагенератору, то необходимо производить их лексический, синтаксический и семантический анализ. В дальнейшем под словом анализ будем понимать прохождение всех этих трех видов анализа. Было принято решение использовать уже существующие инструменты для анализа и построения внутреннего представления шаблонов. Данное решение было принято по следующим причинам:

1) Многие из этих инструментов предлагают довольно удобное API для работы с внутренним представлением программ. Отметим, что эти API совершенствовались в течение многих лет.

2) Многие из этих инструментов имеют большие тестовые базы, на которых они проверялись. Было бы сложно создать такие большие тестовые базы только собственными силами.

В качестве такого инструмента для лексического анализа и построения внутреннего представления было выбрано API на языке программирования C++, предоставляемое компилятором Clang и виртуальной машиной LLVM [6]. Этот инструментарий в полной мере обладает всеми перечисленными выше преимуществами [15] и, кроме того:

1) API на языке программирования C++ позволяет использовать возможности объектно-ориентированного анализа и дизайна при разработке анализатора. Использование объектно-ориентированного анализа и дизайна позволяет значительно облегчить разработку практически любых программных систем. Также использование объектно-ориентированного анализа и дизайна при разработке анализатора дает возможность легко вносить изменения в уже реализованный анализатор.

2) API на языке программирования C++, предоставляемое компилятором Clang, позволяет легко работать с внутренним представлением программ. Например, это API предоставляет возможность достаточно легко обойти всю программу, пользуясь ее внутренним представлением.

3) Задача анализа шаблонов предполагает много работы со строковыми данными. Язык программирования C++, на котором предоставляет свой API компилятор Clang, дает возможность удобно работать со строковыми данными по сравнению со многими другими языками программирования (например, по сравнению с языком программирования Си).

4) Сейчас компилятор Clang активно поддерживается, и нет оснований полагать, что его поддержка в скором времени прекратится.

5) Язык программирования C++, на котором предоставляет свой API компилятор Clang, дает возможность снабдить спецификациями фрагменты собственного кода.

При работе метагенератора на стадии анализа происходит считывание конструкций `label_pattern` и формирование внутреннего представления, в котором типу метки соответствует ее текстовый шаблон. Также, на стадии анализа происходит считывание правил вывода и, соответственно, содержащихся в них конструкций `label`, и формирование внутреннего представления правил вывода. Расширение языка семантическими метками привело к созданию анализатора для конструкций `label`, который в ходе работы создает структуры `label` и добавляет указатели на них их к соответствующим узлам `term_node`.

Так как API Clang имеет встроенный лексический и синтаксический анализатор Си-программ, то использование API Clang позволило не заниматься реализацией этой функциональности. На первом этапе, анализатор с помощью инструментария, предоставляемого API Clang, строит внутреннее представление шаблона. Так как специальные синтаксические конструкции в языке шаблонов являются расширениями языка Си, то для их обработки штатными средствами API Clang необходимо вводить специальные функции, одноименные с рассматриваемыми конструкциями. Тогда специальные синтаксические конструкции попадают во внутреннее представление как вызовы этих специальных функций. На втором этапе анализатор с помощью API Clang строит свое внутреннее представление шаблона, которое более удобно для наших задач и которое основывается на структуре pattern_node. При этом вызовы специальных функций интерпретируются как специальные синтаксические конструкции в языке шаблонов. Таким образом, расширение языка Си специальными синтаксическими конструкциями не вызвало сложностей при анализе языка шаблонов. В результате работы анализатора шаблонов получается внутреннее представление шаблонов – множество деревьев, узлами которых является структуры `pattern_node`.

Аналогично производится и анализ программ, поступающих на вход метагенератору. Так как целевым языком нашей системы является язык Си, то вышеперечисленные преимущества API Clang определили выбор инструментария для решения задачи анализа программ. Он проходит в два этапа, аналогичных этапам анализа шаблонов. Более того, данные этапы являются более простыми, чем аналогичные этапы при анализе шаблонов, так как при анализе программ не приходится обрабатывать специальные синтаксические конструкции, хранящие метаинформацию. В результате работы анализатора программы получается ее внутреннее представление – дерево, узлами которого является структуры `program_node`. Итак, использование API Clang позволило снизить трудоемкость разработки анализатора шаблонов и анализатора программы.

E. Функция match_trees

При генерации условий корректности последовательность программных конструкций просматривается от конца к началу. При этом просмотре на каждом шаге выбирается последняя программная конструкция. Функция `match_trees` проверяет, соответствует ли выбранная программная конструкция определенному шаблону или нет [4]. Рассматриваем набор правил вывода и из набора правил выбираем то, которое позволяет провести унификацию программной

конструкции с шаблоном. Происходит перебор правил вывода, и, соответственно, шаблонов.

Функция match_trees вызывается для каждого из них и для выбранной программной конструкции. Это происходит до тех пор, пока не будет найден шаблон, соответствующий выбранной программной конструкции, или пока не будут перебраны все правила вывода, и, соответственно, все шаблоны.

Функция match_trees принимает два аргумента: шаблон и программную конструкцию. Она возвращает 1, если ее аргументы сопоставимы, и 0 в обратном случае. Первым аргументом функция match_trees принимает переменную pattern – указатель на структуру pattern_node. Переменная pattern указывает на корень дерева, в котором хранится шаблон. А вторым аргументом функция match_trees принимает переменную code – указатель на структуру program_node. Переменная code указывает на корень дерева, в котором хранится программная конструкция.

Таким образом, функция match_trees сопоставляет два дерева: дерево шаблона и дерево программной конструкции. Соответственно, функции match_trees передаются указатели на узлы этих деревьев, являющиеся их корнями. Для удобства будем называть узел дерева шаблона, указатель на который передан функции match_trees, узлом pattern, а узел дерева программной конструкции, указатель на который передан функции match_trees, – узлом code. Атрибуты этих узлов хранятся в полях структур pattern_node и program_node соответственно. Корректное сопоставление синтаксических конструкций any_code, exist_code, simple_expression программным конструкциям обеспечивается с помощью алгоритма, реализованного в функции match_trees.

Отметим, что при сопоставлении используется «жадный» алгоритм. Корректность его применения гарантируется введенным выше дополнительным ограничением на язык написания шаблонов, а также простотой аксиоматической семантики языка C-kernel и ограничениями языка C-light (например, запретом на передачу управления в составные операторы извне).

Сначала функция пытается сопоставить все атрибуты узлов program_node и code, кроме потомков этих узлов. Если эти атрибуты удалось сопоставить, то функция пытается сопоставить потомков узла pattern и потомков узла code. При этом происходят рекурсивные вызовы функции match_trees. В таких вызовах в качестве первого аргумента функции передается один из некоторых потомков узла pattern, а в качестве второго – один из некоторых потомков узла code. Если и потомков узлов pattern и code удалось сопоставить, то можно сделать вывод, что шаблон и программная конструкция сопоставимы, и функция возвращает значение 1.

Тогда в узле pattern атрибут is_matched принимает значение 1, а в атрибуте match_identifiers хранится таблица соответствия идентификаторов шаблона идентификаторам программной конструкции. Использование данной таблицы облегчает применение правила вывода, происходящее после успешного сопоставления. Таким образом, заполнение таблицы match_identifiers в ходе работы функции match_trees является преимуществом рассматриваемого подхода.

Отметим, что в реализации функции match_trees нет привязки к конкретному языку, она подходит к широкому классу языков программирования. Таким образом, данный подход соответствует концепции Морикони и Шварца [7], в которой метагенерация также рассматривается безотносительно от используемого языка программирования.

Замечание. Отметим некоторое отличие нашей реализации от исходной идеи Морикони и Шварца. У них порожденный генератор является отдельной программой, которую можно скомпилировать и запустить. Наш метагенератор имеет два входа – правила вывода с аксиомам и программа. Следовательно, по нашей реализации в каждый сеанс верификации происходит порождение генератора по подаваемой на вход системе Хоара. Хотя эта схема не эффективна, но она позволяет нам верифицировать одну общую программу, а не две – метагенератор и порожденный генератор. Тем более, что по Морикони и Шварцу генератор порождается без спецификаций. Поэтому, с точки зрения создания самоприменимой системы верификации [10], наш подход имеет преимущество перед идеей Морикони и Шварца.

F. Реализация перехода от условия корректности к его объяснению

При работе метагенератора на стадии генерации условий корректности происходит применение правил вывода и, соответственно, переход к тройкам Хоара, чьи шаблоны были описаны в посылках правила вывода. При этом, в узлах деревьев спецификаций вместо метасимволов подставляется то, что им соответствует, а в структуру label записывается диапазон строк. Это не создает дополнительных сложностей, так как при применении правила вывода метагенератор знает, какой оператор рассматривается, и, соответственно, он знает и диапазон строк. В итоге, в тройках Хоара не остается программных фрагментов, и тогда предусловие и постусловие объединяются в одну формулу знаком импликации, образуя условие корректности. Таким образом, два дерева спецификаций объединяются в одно, соответствующее условию корректности. Далее, происходит упрощение УК по описанным выше правилам. При этом, в нашем подходе, в отличие от подхода Денни и Фишера, не применяется четвертая группа этих правил, т.е. группа правил, осуществляющих «просачивание» меток через конъюнкцию и (вложенную) импликацию. Таким образом, метки в дереве УК продолжают быть структурой, тоже представляющей собой дерево. Для

обоснования этого можно ввести отношение предок-потомок на метках, соответствующее такому же отношению на дереве спецификаций, не рассматривая при этом узлы дерева спецификаций, не имеющие меток. Следовательно, в нашем подходе, в отличие от подхода Дении и Фишера [2], метки в условиях корректности образуют некоторую иерархию, что используется при генерации текста на естественном языке, объясняющего смысл данного данного условия корректности. Для объяснения неудачи при доказательстве система верификации будет генерировать такой текст для каждого недоказанного УК. На этой стадии дерево меток обходится в глубину, и для каждой рассмотренной метки к общему тексту добавляется текст ее заполненного номерами строк шаблона. Таким образом в проекте C-light работает система локализации ошибок.

V. Заключение

При работе над проектом C-light возникла задача сделать систему верификации удобной не только для специалистов-теоретиков, но и для обычных программистов. Существенным вкладом в решение этой задачи стала бы возможность локализации и объяснения ошибок при наличии недоказанных УК. Иначе, пользователям системы верификации приходилось бы «вручную» анализировать недоказанные УК. Для решения этой задачи был выбран метод Денни и Фишера. Используемый в системе C-light метод метагенерации позволил удобным образом применить в нашем проекте предложенную Денни и Фишером концепцию семантических меток [2]. Согласно этой концепции полученные после генерации УК несут в себе дополнительную информацию, выраженную в виде семантических меток, которая позволяет локализовать и объяснить ошибки.

Обзор родственных работ. Тематика формальной локализации ошибок мало представлена в исследованиях по сравнению с тематикой дедуктивной верификации программ. Рассмотрим три проекта, основанных на формальном подходе к локализации ошибок. Во-первых, в проекте Centaur [3] УК анализируются для поиска условных выражений из исходных условных операторов и циклов. Во-вторых, в работе Денни и Фишера [2] рассмотрен метод семантической разметки, на котором базируется наш подход. В-третьих, в проекте Лейно [5] базовая логика расширена метками, предоставляющими пригодную для объяснения семантическую информацию. Отметим особенности этих проектов. В первом проекте используются некоторые алгоритмы из области отладки программ. Во втором проекте для порождения объяснений используются только метки, а не сами УК. В третьем проекте метки вводятся на этапе трансляции в промежуточный язык, который обрабатывается стандартным безметочным генератором. Также, во всех этих проектах используются более простые, чем Си, входные языки, что является их недостатком по сравнению с нашим проектом.

В будущем планируется применить систему C-light для верификации программ определенных классов, например программ инженерной математики [17]. Метод метагенерации позволяет пользователям системы верификации использовать произвольный набор аксиом и правил вывода, а также произвольный набор семантических меток. Поэтому, интерес представляет использование узкоспециализированных аксиом и правил вывода для программ определенных классов, а также специфических меток для таких аксиом и правил вывода.

Список литературы

[1] Apt K.R., Olderog E.R. Verification of sequential and concurrent programs. – Berlin etc.: Springer, 1991. – 450 p.

[2] Denney E., Fischer B. Explaining Verification Conditions // Proc. AMAST 2008. – LNCS. – 2008. – Vol. 5140. – P. 145-159.

[3] Fraer R. Tracing the origins of verification conditions. – Rocquencourt, 1996. – 17 p. – (Rapp. / INRIA; N 2840).

[4] Kondratyev D. A., Promsky A. V. Towards the 'verified verifier'. Theory and practice // Proc. Fifth Workshop "Program Semantics, Specification and Verification: Theory and Applications". – Moscow, Russia, June 6, 2014. – P. 68-78.

[5] Leino K.R.M., Millstein T., Saxe J.B. Generating error traces from verification condition counterexamples // Science of Computer Programming. – 2005. – Vol. 55, N 1-3. – P. 209-226.

[6] Maryasov I. V., Nepomniaschy V. A., Promsky A. V., Kondratyev D. A. Automatic C Program Verification Based on Mixed Axiomatic Semantics // Proc. Fourth Workshop "Program Semantics, Specification and Verification: Theory and Applications". – Yekaterinburg, Russia, June 24, 2013. – P. 50-59.

[7] Moriconi M., Schwarts R.L. Automatic Construction of Verification Condition Generators From Hoare Logics // Lect. Notes Comput. Sci. – Berlin etc., 1981. – Vol. 115. – P. 363-377.

[8] Programming languages – C. – ISO/IEC 9899: 1999. – 566 p.

[9] Promsky A.V. C Program Verifcation: Verifcation Condition Explanation and Standard Library // Automatic Control and Computer Sciences, 2012, Vol. 46, No. 7, pp. 394-401.

[10] Promsky A.V. Experiments on self-applicability in the C-light verification system // Bull. Nov. Comp. Center, Comp. Science, 35 (2013), 85-99.

[11] Promsky A.V. Experiments on self-applicability in the C-light verification system. Part 2 // Bull. Nov. Comp. Center, Comp. Science, 37 (2014), 93-105.

[12] Promsky A.V. Towards C-light Program Verification: Overcoming the Obstacles // Proc. International Workshop on Program Understanding, 19-23 June, Altai Mountains, Russia, 2009. – pp. 53-63.

[13] Кондратьев Д. А., Промский А. В. Комплексный подход к локализации ошибок при верификации Си-программ // Системная информатика. 2013. № 1. С. 79-96.

[14] Кондратьев Д. А., Промский А. В. Разработка самоприменимой системы верификации. Теория и практика // Моделирование и анализ информационных систем. Т.21, №6 (2014), 70-81.

[15] Марьясов И.В., Непомнящий В.А., Промский А.В., Кондратьев Д.А. Автоматическая верификация С-программ на основе смешанной аксиоматической семантики // Моделирование и анализ информационных систем. Т.20, №6 (2013), 52-63.

[16] Непомнящий В.А., Ануреев И.С., Михайлов И.Н., Промский А.В. Ориентированный на верификацию язык C-light // Системная информатика. – 2004. – Вып. 9. – С. 51-134.

[17] Непомнящий В.А., Рякин О.М. Прикладные методы верификации программ. – М.: Радио и связь, 1988.

A Need To Specify and Verify Standard Functions

Nikolay V. Shilov
A.P. Ershov Institute of Informatics Systems
Novosibirsk, Russia
shilov@iis.nsk.su

***Abstract* — The problem of validation of standard mathematical functions and libraries is well-recognized by industrial and academic professional community but still is poorly understood by freshmen and inexperienced developers. The paper gives two examples (from author's pedagogical experience) when formal specification and verification of standard functions do help and are needed.**

Keywords — mathematical functions; standard libraries; formal specification; formal program verification

I. π IS 4 BACAUSE OF RAND()

A. *What is π?*

How I want a drink, alcoholic of course,

after the heavy lectures involving quantum mechanics.

James Jeans (1877-1946), British Scientist [14]

Mathematical irrational number π is the ratio of a circle's circumference to its diameter; it implies that the quarter of the ration of the area of the circle to the area of a square built on its diameter is also π. This observation leads to Monte Carlo method for computing approximation of π as follows (Figure 1): to draw a segment of a circle in the first quadrant and the square around it, then randomly drop dots in the square; the ratio of dots inside the circle to the total number of dots should be approximately equal π/4. For example, the series of trials depicted in the figure gives π/4≈8/11, i.e. π≈2,(90).

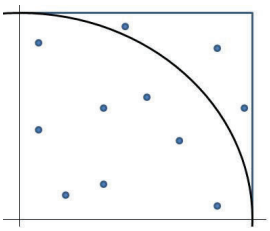


Figure 1: Monte Carlo method to compute π

Of course, the above approximation π≈ 2,(90) is not good[1]. Fortunately, almost everyone remembers much better approximation π≈3.14. Moreover there are many ways to memorize more digits than 3 as above. One way is to memorize a *story* in which the word lengths represent the digits of π: the first word has 3 letters, the second – 1 letter, the third has 4 letters, and so on; in particular, the epigraph of this section is an example of a story to memorize 15 digits of the number.

Some computer languages have a standard function to compute π approximations. For example, the official site `support.office.com` [13] specifies standard PI function and how to use it as shown in Figure 1.

PI function
This article describes the formula syntax and usage of the PI function in Microsoft Excel.

Description

Returns the number 3.14159265358979, the mathematical constant pi, accurate to 15 digits.

Syntax

PI()

The PI function syntax has no arguments

Figure 2: Specification of PI( ) function in MS Excel

B. *Computing π by Monte Carlo*

C-program depicted in Figure 3 implements the above Monte Carlo method to compute an approximation for π. It prescribes to exercise 10 series of 1000000 trials each. This code was developed by a Computer Science instructor to teach first-year students C-loops on base of an interesting and very intuitive algorithm. There were 25 students in the class that used either Code::Blocks 12.11 or Eclipse Kepler IDEs for C/C++ with MinGW environment. Let us refer this program as PiMC (π Monte Carlo) in the sequel.

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
int main(void){
srand(time(NULL));
int i, j, r, n = 10;
float pi_val, x, y;
int n_hits, n_trials=1000000;
for(j = 0; j < n; j++){n_hits=0;
        for(i = 0; i<n_trials; i++){
                r = rand()% 10000000;
                x = r/10000000.0;
                r = rand()% 10000000;
                y = r/10000000.0;
                if(x*x + y*y < 1.0) n_hits++;}
        pi_val = 4.0*n_hits/(float)n_trials;
printf("%f \n", pi_val); } return 0;}
```

Figure 3: C-program PiMC to compute π approximation

[1] The Monte Carlo method isn't adaptive and is very slow compared to other methods to compute π.

C. *π equals 4*

Imagine confuse of the instructor when each of 25 students in the class got 10 time value 4.000000 as an approximation for π. (See Figure 4 for a snapshot.)

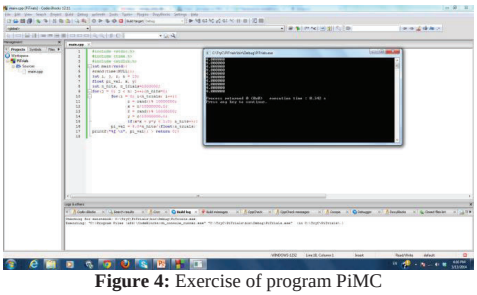


Figure 4: Exercise of program PiMC

But the above outcome of 25 computer experiments was not the last shock for the instructor this day. Because Mathematician that came to run the next class proved that π is really 4. Look at Figure 5 that presents first 3 in a sequence of figures circumscribing a circle with diameter D: each next figure results from the previous one by "cutting corners". The sequence converges to the circle; hence its perimeter converges to πD. But perimeters of all figures in the sequence are constant 4D. Hence π=4.

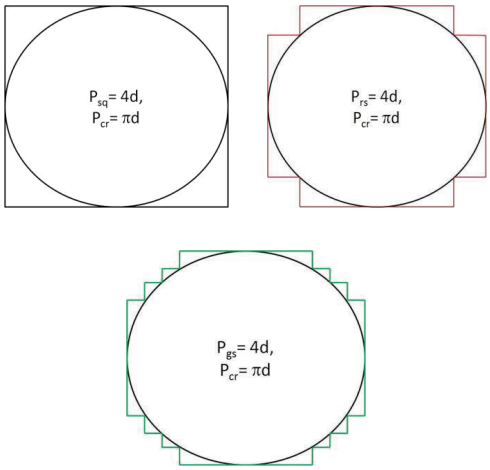


Figure 5: First three figures of a series converging to a circle

So we can summaries: very intuitive Monte Carlo computations and an obvious proof lead us to a paradoxical conclusion that π=4.

D. *Formal Methods as a Rescue*

First let us rule out mathematical "proof" that π=4: presented mathematical arguments don't prove that π=4, but demonstrate that convergence in metrics L_∞ doesn't imply convergence in metrics L_2 [8]: the sequence converges to the circle in metrics L_∞, perimeters of all figures are 4D, but the circumference of the circle is πD≈3.14D.

Next let us try to figure out what is wrong with computer program PiMC. Formal Methods [5] can help this time; in particular let us specify the program in Hoare style by pre- and post-conditions [2].

The pre-condition may be `TRUE` since the program has no input. The post-condition may be `pi_val==4.0` since we know from the program exercise the final value of the variable; but since the real program works with floating point values, maybe it makes sense to use more loose post-condition `3.9<=pi_val<=4.1`. Due to the exercise we may hope that

$$\models [\mathtt{TRUE}]\ \text{PiMC}\ [\mathtt{3.9<=pi_val<=4.1}],$$

i.e. that the total correctness assertion is valid.

But if we try to apply verification methodic from [2] to generate verification conditions and prove the above assertion then we encounter a problem of formal semantics of the function `rand()` in the assignment

```
r = rand()% 10000000;
```

that has 2 instances in the program. The standard rule to generate verification condition for assignment reads

$$\frac{\varphi(x)\rightarrow\psi(t)}{[\varphi(x)]\ \ x=t\ \ [\psi(x)]}\ ;$$

for function `rand()` it leads to

$$\frac{\varphi(x)\rightarrow\psi(\mathtt{rand}())}{[\varphi(x)]\ \ x=\mathtt{rand}()\ \ [\psi(x)]}\ .$$

But, unfortunately, we know too little about properties of this function to prove any non-trivial verification condition! In particular, C reference portal [15] provides just a loose information about this function (Figure 6).

So it is possible to conclude: the cause of wrong π approximation by the program PiMC is use of the standard function `rand()`, its poor specification in the language standard and no verification in MinGW.

rand
C Numerics Pseudo-random number generation
Defined in header <stdlib.h>
int rand();
Returns a pseudo-random integral value between 0 and RAND_MAX (0 and RAND_MAX included).
srand() seeds the pseudo-random number generator used by rand(). If rand() is used before any calls to srand(), rand() behaves as if it was seeded with srand(1). Each time rand() is seeded with srand(), it must produce the same sequence of values.
rand() is not guaranteed to be thread-safe.
Parameters
(none)
Return value
Pseudo-random integral value between 0 and RAND_MAX, inclusive.
Notes
There are no guarantees as to the quality of the random sequence produced. In the past, some implementations of rand() have had serious shortcomings in the randomness, distribution and period of the sequence produced (in one well-known example, the low-order bit simply alternated between 1 and 0 between calls). rand() is not recommended for serious random-number generation needs, like cryptography.
POSIX requires that the period of the pseudo-random number generator used by rand is at least 2^{32}
POSIX offered a thread-safe version of rand called rand_r, which is obsolete in favor of the drand48 family of functions.

Figure 6: Specification of function rand from C reference

II. What is sqrt?

A. *Solving Quadratic Equations*

An error becomes an error when born as truth.

Stanisław Jerzy Lec (1909-1966),

Polish poet and aphorist [20]

A very popular (but a vulgar for *professional* education) approach to teach standard input/output, floating point data type, sequencing and branching control flow is to program solving of quadratic equations. (Please check [6][17][18] for instance.) Below in Figure 7 one can find a variant of this vulgar code that "solves" quadratic equations in the form $ax^2 + bx + c = 0$.

```
#include <stdio.h>
#include <math.h>
int main(void){
float a, b, c, d, x;
printf("Input coefficients a, b and c and type
'enter' after each:");
scanf("%f%f%f", &a, &b, &c);
d=b*b -4*a*c;
if (d<0) printf("No root(s).");
else {x= (-b + sqrt(d))/(2*a);
      printf("A root is %f.", x);}  return 0;}
```

Figure 7: A vulgar code to "solve" quadratic equation

We put the verb "solves" to quotation marks because non of conventional computers can find root of a simple equation $x^2 - 2 = 0$ (i.e. $\sqrt{2}$) due to irrational nature of the number but finite size all numeric data types in every implementation of C. Surprisingly, but even C reference [16] says that conventional computers must be able to compute $\sqrt{2}$ (refer to Figure 8).

sqrt, sqrtf, sqrtl
C Numerics Common mathematical functions
Defined in header <math.h>
float sqrtf(float arg);
(1) (since C99)
double sqrt(double arg);
(2)
long double sqrtl(long double arg);
(3) (since C99)
Defined in header <tgmath.h>
#define sqrt(arg)
(4) (since C99)
1-3) Computes square root of arg.
4) Type-generic macro: If arg has type long double, sqrtl is called. Otherwise, if arg has integer type or the type double, sqrt is called. Otherwise, sqrtf is called. If arg is complex or imaginary, then the macro invokes the corresponding complex function (csqrtf, csqrt, csqrtl).
Parameters
arg - floating point value
Return value
If no errors occur, square root of arg ($\sqrt{\arg}$), is returned.
If a domain error occurs, an implementation-defined value is returned (NaN where supported).
If a range error occurs due to underflow, the correct result (after rounding) is returned.
Error handling
Errors are reported as specified in math_errhandling.
Domain error occurs if arg is less than zero.
If the implementation supports IEEE floating-point arithmetic (IEC 60559),
If the argument is less than -0, FE_INVALID is raised and NaN is returned.
If the argument is $+\infty$ or ± 0, it is returned, unmodified.
If the argument is NaN, NaN is returned

Figure 8: Specification of `sqrt`-family functions from C reference

But conventional computers can't compute in finite time any irrational number in general and the square root of 2 in particular. It implies that the above specification of the standard function `sqrt` says nonsense and that the function shouldn't be used to solve quadratic equations.

Instead the exact value of irrational square root of 2 a conventional computer can find an approximation of the root with some precision or (it would be better to say) *accuracy*. These approximation and accuracy may be formalized in different ways discussed in the next section.

B. *Alternatives for sqrt*

For instance, it makes sense to introduce another function with two arguments SQR(Y, E) where Y stays for the argument and E stays for accuracy, that can be formally specified by any (or both) of the following two clauses:

- if Y≥0 and E>0 then SQR(Y, E) differs from $\sqrt{Y}$ less than E, i.e. $|\mathrm{SQR(Y,E)} - \sqrt{\mathrm{Y}}| < E$;
- if Y≥0 and E>0 then $(\mathrm{SQR(Y,E)})^2$ differs from Y less than E, i.e. $|\mathrm{SQR(Y,E)} \times \mathrm{SQR(Y,E)} - \mathrm{Y}| < E$.

Let us fix the first specification for this paper. Then let us select a computation method to compute an approximation. One can select Newton-Raphson Method [7] as a very

intuitive: first guess an initial approximation for the root; then compute arithmetic mean between the guess and the number (whose square root you want to obtain) divided by the initial guess; let this mean to be a new guess for another go-around while the difference between the next and the previous guesses is bigger than the half of the accuracy. The method is easy to implement (see Figure 9).

```
float ab(float X)
{if (X<0) return(-X); else return(X);}

float SQR(float Y, float E)
{float X, D;
X=Y;
do {D=(Y/X-X)/2; X+=D;} while (ab(D)>E/2);
return X;
}
```

Figure 9: Floating-point function to compute an approximation of square root

Both functions in this implementation are easy to specify formally in Hoare style:

- [X is float] ab(X) [returned value is |X|],
- [Y and E are positive floats]

SQR(Y,E) [$|\mathrm{SQR}(Y,E) - \sqrt{Y}| < E$].

If to prove these specifications, than SQR may be a good alternative to the standard function `sqrt`. Unfortunately, it isn't easy to prove automatically and formally [4] due to several reasons. The major one is axiomatization of computer floating-point arithmetic [1][19].

Even a manual verification of SQR algorithm SQR (assuming precise arithmetic for real numbers) isn't a trivial exercise. Below in Figure 10 one can see a flowchart of (a little bit modified) algorithm of function SQR. (Let us refer the algorithm in the sequel by SQR also.)

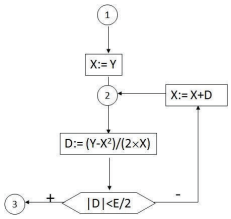


Figure 10: Flowchart of the algorithm implemented in SQR

If to specify the algorithm in line with the function then we need to prove Hoare triple

[Y>0 & E>0 & Y,E∈ℝ] SQR [$|\mathrm{SQR}(Y,E) - \sqrt{Y}| < E$].

In case when Y>1 the corresponding partial correctness assertion can be proved by Floyd method [2] with the following loop invariant (the correctness assertion for the control point 2 in the figure): Y>1 & E>0 & $\sqrt{Y} < X \le Y$. Halting (termination) of the algorithm can be proved using observation that every time the absolute value of D is twice less (at least) than in the previous iteration.

III. Concluding Remarks

It worth to remark that a need of better specification and validation of standard functions is well-recognized by industrial and academic professional community as well as the problem of conformance of their implementation with the specification [4][9][10][11][12]. Paper [4] addresses the formal verification of some low-level mathematical software for the Intel® Itanium® architecture; in particular it presents details of the verification of a square root algorithm with aid of HOL Light theorem prover. Next two papers [9][10] address formal specification and testing of standard mathematical functions. The last two papers [10][12] present formal specification and verification of some standard memory management and input-output functions.

But a very serious obstacle for formal verification of standard mathematical functions is a need of axiomatization of floating point arithmetic [1][19]. Maybe interval analysis approach [3] and formalization of interval arithmetic may help to tackle the problem for functions like `sqrt` (but not for functions like `rand`).

Unfortunately, the problem (or a pitfall) of poorly specified and verified standard functions and libraries still is poorly understood by freshmen and inexperienced developers. Better education, specification and verification are needed to solve the problem (and avoid the catch of poor libraries).

References

[1] A. Ayad, C. Marché. Multi-prover verification of floating-point programs. Proceedings of Fifth International Joint Conference on Automated Reasoning. Lecture Notes in Artificial Intelligence, 2010, Vol. 6173, p.127-141.

[2] D. Gries, The Science of Programming. Springer-Verlag, 1981.

[3] M.W. Gutowski, Power and beauty of interval methods. arXiv:physics/0302034 [physics.data-an]. Visited October 7, 2015.

[4] J. Harrison, Formal Verification of Square Root Algorithms. Formal Methods in System Design, 2003, Vol.22(2), p.143-153.

[5] C. A. R.. Hoare, The Verifying Compiler: A Grand Challenge for Computing Research. Perspectives of Systems Informatics (PSI'2003), SpringerVerlag, Berlin, LNCS., no. 2890, pp. 1-12, 2003.

[6] S.G. Kochan, Programming in C: A Complete Introduction to the C Programming Language. Exercise #8 at p.162. Sam's Publishing, 2005 (3rd Edition).

[7] S.G. Kochan, Programming in C: A Complete Introduction to the C Programming Language. Functions Calling Functions at p.131. Sam's Publishing, 2005 (3rd Edition).

[8] A.N. Kolmagorov, S.V. Fomin Elements of Funcions Theory and Functional Analysis. Nauka Publishers, 1976 (4th ed., in Russian)

[9] V. Kuliamin, Standardization and Testing of Mathematical Functions Programming and Computer Software, 2007, Vol. 33 (3), p.154-173.

[10] V.V. Kuliamin, Standardization and Testing of Mathematical Functions in floating point numbers. Proceedings of Int. Conf. Perspectives of Systems Informatics PSI-2009. Lecture Notes in Computer Science, 2010, Vol. 5947, p. 257-268.

[11] A.V. Promsky, C Program Verification: Verification Condition Explanation and Standard Library. Automatic Control and Computer Sciences, 2012, Vol. 46, No. 7, p. 394–401.

[12] A.V. Promsky, Experiments on self-applicability in the C-light verification system. Bull. Nov.Comp. Center, Comp. Science, Vol.35, 2013, p.85-99.

[13] Pi Function. https://support.office.com/en-us/article/PI-function-264199d0-a3ba-46b8-975a-c4a04608989b. Visited October 7, 2015.

[14] Pi. Memorizing digits. https://en.wikipedia.org/wiki/Pi#Memorizing_digits. Visited October 7, 2015.

[15] C reference. Rand. http://en.cppreference.com/w/c/numeric/random/rand. Visited October 7, 2015.

[16] C refernce. Sqrt, sqrtf, sqrtl. http://en.cppreference.com/w/c/numeric/math/sqrt. Visited October 7, 2015.

[17] How to make a program that solves the quadratic formula. http://www.youtube.com/watch?v=15NbFrBUdu0. Visited October 7, 2015.

[18] Write a C++ program that solves quadratic equation to find its roots. http://www.cplusplus.com/forum/general/36313/. Visited October 7, 2015.

[19] Hisseo. http://hisseo.saclay.inria.fr/index.html. Visited October 7, 2015.

[20] Stanislaw Jerzy Lec Quotes. http://www.azquotes.com/author/8631-Stanislaw_Jerzy_Lec. Visited October 7, 2015.

Формальные методы в робототехнике

Д.А. Мордвинов
Санкт-Петербургский государственный университет
Кафедра системного программирования
Email: mordvinov.dmitry@gmail.com

Ю.В. Литвинов
Санкт-Петербургский государственный университет
Кафедра системного программирования
Email: y.litvinov@spbu.ru

***Аннотация*—В статье дается обзор применения формальных методов в контексте робототехники. Рассматриваются недавние работы, посвященные спецификациям поведения роботов в терминах темпоральных логик, применению идей подхода model checking к таким системам. Также рассматриваются применения формальных методов анализа сетей Петри и моделирования поведения робототехнических систем с их помощью. Отдельное внимание уделяется верификации гибридных систем, применению хоаровского исчисления процессов для спецификации поведения параллельных систем, а также использованию других подходов для верификации программ поведения роботов.**

I. Введение

Разговор о формальных методах чаще всего сопряжен с понятием опасности отказа системы или некорректного ее поведения, особенно когда речь идет о дорогостоящих системах и ошибках, приводящих к причинению вреда людям. Чаще всего формальные методы находят применение в военных, космических, медицинских, промышленных технологиях, протоколах общения узлов в компьютерных сетях, о чем написано немало литературы.

Естественным образом аналогичные проблемы появляются и в области робототехники: отказ дорогостоящих робототехнических систем может привести к весьма неприятным последствиям (особенно если речь идет о боевых, медицинских или космических роботах). Неудивительно, что применению формальных методов посвящены целые секции на крупнейших конференциях по робототехнике (таких как ICRA[1] и IROS[2]), на каждом из которых ежегодно публикуются десятки работ. Однако стоит отметить, что далеко не всегда они касаются тематики безопасности поведения роботов. Проблемы, решаемые авторами таких статей, чаще относятся к методам планирования действий и рассуждений, симуляции, управления роботами, а также их групповой координации.

Многие формализмы, построенные в контексте применения формальных методов в робототехнике, выходят за рамки апробации на компьютерных симуляторах, их воплощения работают на дорогостоящих роботах. Например, Verifiable Robotics Research Group[3], авторы одного из наиболее обсуждаемых в настоящее время подходов к планированию действий робота для задачи, поставленной в терминах темпоральных логик, применяют результаты своих работ на одном из крупнейших робототехнических состязаний DARPA Challenge[4] в составе команды Vigir[5]. Будут также рассмотрены случаи применения формальных методов в реабилитационной робототехнике, к созданию автономных промышленных робототехнических систем, робофутболу и т.д.

Задачей данной статьи является обзор и классификация применения формальных методов к проблемам робототехники. Будут рассмотрены как самые известные работы середины девяностых годов, положившие начало целым областям исследований, так и работы с последних робототехнических конференций вплоть до 2015 года. Авторы не претендуют на полноту обзора как со стороны формальных методов, так и со стороны задач робототехники, тем не менее, многие из самых обсуждаемых идей последних годов, а также самые разработанные за два десятилетия области будут здесь рассмотрены.

II. Темпоральные логики и синтез конечных систем переходов

В 1995 году в статье [1] был предложен новаторский подход к синтезу управления шагающей системой. В работе строится модель искусственной ноги робота в виде конечного автомата с начальным состоянием («start») и состояниями «под нагрузкой» («load»), «толчок» («drive»), «без нагрузки» («unload»), «переносится вперед» («recover») и «соскользнула» («slipped»). Далее, рассматривая *полное произведение* (*shuffle product*) четырех автоматов, авторы получают модель четырехногой шагающей системы (состоящей из 1296 состояний и 5184 переходов). Очевидно, что пространство состояний результирующей

[1] IEEE International Conference on Robotics and Automation, URL: http://icra2015.org (дата обращения: 30.09.2015)

[2] IEEE/RSJ International Conference on Intelligent Robots and Systems, URL: http://www.iros2015.org (дата обращения: 30.09.2015)

[3] URL: http://verifiablerobotics.com (дата обращения: 30.09.2015)

[4] URL: http://www.theroboticschallenge.org (дата обращения: 30.09.2015)

[5] URL: http://www.teamvigir.org (дата обращения: 30.09.2015)

системы содержит множество некорректных состояний (например, для двуногой системы не может быть состояния, когда обе ноги переносят свой вес в одно и то же время, или во время переноса одной ноги вторая не может перейти в состояние «без нагрузки»). Для спецификации таких инвариантов (свойств живучести системы, если выражаться в терминологии [2]) авторы используют темпоральную логику ветвящегося времени (CTL, [3]). Исторически это первый пример успешного использования темпоральной логики в контексте робототехнических систем. Техники model checking были применены в работе в своем «стандартном» виде: по заданной модели поведения системы и ее инварианту была проведена формальная верификация соответствия поведения системы требованиям.

Авторы рекомендуют читателю быть ознакомленным с [4] или с [5] для понимания материала данного раздела. Введем понятия и обозначения, используемые далее в статье. Для спецификации формул темпоральной логики мы будем использовать стандартную нотацию: $\bigcirc\phi$ для оператора *next*, $\Box\phi$ для оператора *always*, $\Diamond\phi$ для оператора *eventually* и $\phi\mathcal{U}\psi$ для оператора *until*. Множество формул темпоральной логики линейного времени (LTL, [6]) вида

$$(\Box\Diamond p_1 \wedge \cdots \wedge \Box\Diamond p_m) \Rightarrow (\Box\Diamond q_1 \wedge \cdots \wedge \Box\Diamond q_n)$$

будем называть, в соответствии с [7], классом *General Reactivity (1)*, или *GR(1)*. Класс $GR(1)$ часто используется в контексте решения задачи *синтеза*, целью которой является построение программного формализма, удовлетворяющего какой-либо формальной спецификации (в нашем случае, спецификации в терминах темпоральной логики) [8]. По представителю класса $GR(1)$ ϕ можно синтезировать автомат (при условии его существования), все траектории которого удовлетворяют ϕ, за $O(n^3)$, где n — количество вхождений атомарных предикатов в ϕ (размер формулы), в то время как для общего случая известны лишь алгоритмы, делающие это с двойной экспоненциальной сложностью [8]. Структурой Крипке [9] будем называть пятерку (S, s_0, R, AP, L), где S — непустое конечное множество состояний, $s_0 \in S$ — начальное состояние, $R \subseteq S \times S$ — тотальное отношение на S, AP — конечное множество атомарных предикатов, $L : S \to 2^{AP}$ — функция пометок.

В 2005, 2007 и 2009 году авторы из лаборатории GRASP Пенсильванского Университета представили работы, определившие текущее состояние области применения формальных методов в робототехнике. В последние 3 года (с 2013 по 2015) более половины работ, публикуемых на секциях формальных методов крупнейших робототехнических конференций (таких как ICRA и IROS), так или иначе затрагивают обсуждаемую ниже область, дополняя ее теоретическую базу или проводя эксперименты с применением теоретических наработок этой области.

В статье [10] 2005 года обсуждается задача планирования траектории движения мобильного робота, формально удовлетворяющей спецификации LTL. Предполагается, что робот представляет собой точку в плоской области, ограниченной многоугольником — окружении робота, внешней среде. Окружение разбивается на n многоугольников (ячеек, регионов), некоторые из которых считаются *интересными*. Интересные регионы могут символизировать комнаты в здании, либо препятствия, с которыми робот не должен сталкиваться. Разбиение геометрического пространства на регионы превращает непрерывную задачу планирования движения в дискретную (*дискретизирует* ее). Каждый регион получает в соответствие свой номер, а также вводится множество предикатов $\Pi = \{\pi_1, \pi_2, \ldots \pi_n\}$, где π_i означает «робот находится внутри региона $i \in \{1..n\}$». «Карта» окружения в таком случае будет представлять собой структуру Крипке $D = (S, s_0, R, \Pi, L)$, где $S = \{s_1, s_2, \ldots, s_n\}$ — множество состояний, соответствующих регионам разбиения, s_0 — состояние, соответствующее региону, в котором робот находится в начальный момент времени, R — отношение перехода, $(s_i, s_j) \in R$ тогда и только тогда, когда регион i геометрически смежен региону j, $L(s_i) =^{def} \pi_i$.

Основная идея работы — постановка задачи роботу декларативно в виде LTL-спецификации. К примеру, формула $\phi = \Diamond(\pi_2 \wedge \Diamond(\pi_3 \wedge \Diamond(\pi_4 \wedge (\neg\pi_2 \wedge \neg\pi_3) \cup \pi_1)))$ для стартового состояния π_1 может означать «посетить регион π_2, затем π_3, затем π_4 и, наконец, вернуться в регион π_1, избегая регионов π_2 и π_3». Для обнаружения траектории, удовлетворяющей спецификации ϕ на структуре Крипке D окружения, достаточно построить контрпример для свойства $\neg\phi$ на модели D. В статье проводятся эксперименты с использованием верификаторов NuSMV [11] и SPIN [12]. Полученная дискретная траектория робота затем преобразуется в непрерывное управление при помощи методов теории управления, описанных в [13].

Подход, описанный в [10] может показаться непрактичным и чересчур усложненным для решаемой задачи, однако идеи, на основе которых он строится, легли в основу действительно полезного метода. Речь идет о статье 2007 года [14]. Развитие подхода, предложенное в ней, в основном, происходит в двух направлениях: работа с датчиками и мультиагетность. Множество атомарных предикатов теперь имеет вид $AP = \Pi \cup X$, где Π — предикаты принадлежности робота региону, а $X = \{x_1, \ldots, x_m\}$ — конечное множество предикатов, отражающих информацию с датчика о внешнем мире. В такой модели становится возможным построение поведения с реакцией на внешний мир, например, такая неформальная спецификация: «Стартуя в регионе 1, проверять, не плачет ли ребенок в регионах 2 или 3. Если плачущий ребенок обнаружен, необходимо отыскать родителей в регионах 4, 5 или 6». Расширение модели на мультиагентные системы происходит естественным

образом, так как каждый робот представляет собой часть окружения другого.

Очевидно, что дискретное управление системой в случае такой модели уже не описывается одной траекторией на «карте» окружения, а представляет собой автомат, учитывающий информацию с датчиков и описывающий все возможные траектории робота, удовлетворяющие LTL-спецификации. Авторы предлагают проводить синтез автомата, описывающего всевозможные поведения робота, удовлетворяющие LTL-спецификации из класса $GR(1)$. Структура Крипке, описывающая «карту» окружения, в данном случае становится не нужна и заменяется описанием в терминах LTL: в общую спецификацию системы добавляются утверждения о геометрически смежных регионах вида $\Box(\pi_1 \Rightarrow (\bigcirc\pi_2 \wedge \bigcirc\pi_3))$, а также утверждения о состоятельности окружения, например, что в один момент времени робот должен находиться точно в одном регионе. Общий вид LTL-требования имеет вид $\phi = \phi_e \Rightarrow \phi_s$, где ϕ_e — требования к поведению окружения, ϕ_s — требования к системе (группе роботов). Говоря неформально, такая спецификация подразумевает, что если среда «играет не по правилам», т.е. в случае нарушения ожидаемых условий окружения, система не обязана вести себя в соответствии с требованиями к ней.

Интересна трактовка синтеза дискретного управления как на игры в математическом смысле между средой и системой роботов. Начиная в стартовом состоянии, среда и система по очереди делают ходы. Среда делает ход первой. Задача среды — опровергнуть спецификацию ϕ (поэтому она должна «играть по правилам» для того, чтобы импликация была ложной), задача системы — удовлетворять спецификации вне зависимости от того, что делает среда. Если система выигрывает, то синтез требуемого дискретного управления может быть произведен, в случае если выигрывает среда — полностью корректной стратегии не существует.

В статье 2009 года [15] те же авторы производят синтез непрерывного управления роботом с обратной связью по дискретной модели (т.е. с учетом реакции системы на значения датчиков) с использованием подхода, описанного в [16]. Другой результат, полученный в [15] — формализация действий робота в общем виде. Множество атомарных предикатов расширяется множеством $\mathcal{A} = \{a_1, a_2, \ldots, a_k\}$ действий, которые могут выполняться роботом. Каждый из них принимает значение истины в определенном состоянии тогда и только тогда, когда робот выполняет действие, находясь в этом состоянии. Передвижение робота из региона в регион является теперь элементом множества $\mathcal{A}$ вместе с такими действиями, как, например, захват предмета клешней или включение/выключение камеры.

Достоинства описанного подхода очевидны: по декларативной спецификации требований к системе в достаточно выразительном формализме LTL автоматически синтезируется математически корректное «по построению» непрерывное управление группой роботов, которое гарантированно приведет к решению задачи. Тем не менее, ограничений у такого подхода достаточно много. Во-первых, система может функционировать только в досконально известной среде, которая ведет себя только так, как ожидается при постановке требований. Во-вторых, хоть сложность синтеза полиномиальна от размера пространства состояний, сам размер пространства состяний может зависеть экспоненциально от количества входов и выходов (датчиков и действий). Однако в результате получается готовый алгоритм поведения системы, т.е. синтез можно произвести лишь один раз.

Шаги к решению как минимум второй проблемы были проделаны в работе [17] 2013 года, главный вклад которой заключается в описании подхода синтезирования дискретной стратегии робота в онлайн-режиме, «на лету». Получая на вход все ту же LTL-спецификацию в форме $GR(1)$, описывающую как «карту» окружения, так и ограничения и цели системы, алгоритм выдает на выходе не автомат, описывающий корректное поведение робота, а величину *горизонта*. Интуитивно *горизонт* — это глубина, на которую нужно просчитывать всевозможные состояния, получаемые из текущего, иначе говоря, это высота дерева игры, количество ходов, которые нужно просчитать в контексте математической игры между средой и роботом для того, чтобы локальные решения, принятые по такому неполному обсчету, гарантированно приводили бы к глобально корректному поведению. В худшем случае трудоемкость принятия решения на определенном шагу сравнится с трудоемкостью алгоритма построения автомата синтеза полной стратегии, применяемым раннее. Работоспособность такого подхода объясняется следующим фактом. В статье [7] построение автомата по LTL-спецификации производилось за два шага: итеративное построение множества «выигрышных» для системы состояний (вычисление неподвижной точки состояний с инвариантом «выигрышности»), и далее сам синтез автомата по результатам каждого шага вычисления неподвижной точки. Подсчет неподвижной точки оперирует пространствами состояний, допуская применение т.н. *символических* структур данных, которые показали себя на практике очень лаконично представляющими множества состояний. Интуитивно обсуждаемый алгоритм принятия решения в реальном времени использует только вычисление неподвижной точки, не проводя самого синтеза автомата, что и дает выигрыш во времени.

Схема работы системы теперь принимает следующий вид. В каждый момент времени робот производит наблюдение за поведением среды, т.е. определяет «ход» оппонента. Это может быть сдвиг передвигающегося препятствия или сигнал «батарея почти разряжена». Среди всех «выигрышных» состояний, полученных в

результате вычисления неподвижной точки, выбирается наиболее близкое в смысле пути на графе состояний с одной эвристикой исключения уже посещенных состояний, чтобы не случилось «зацикливания» системы в среде. Такая эвристика должна быть проигнорирована, если среда «нарушает правила», это повлечет за собой повторные попытки выполнения задачи роботом.

Кроме возможности планирования поведения роботом в режиме реального времени, такой подход позволяет системе принять полностью реактивный характер. Другими словами, система получает возможность функционирования в изменяющемся со временем окружении (но только если характер изменений среды полностью известен на момент синтеза управления). В частности, становится возможным объезд движущегося препятствия, что раннее было слабо затронуто в статьях на тему планирования поведения по LTL-спецификации. Работа [7] содержит результаты экспериментов по синтезу поведения робота при наличии подвижного препятствия.

Существует множество статей с экспериментами, поставленными на теоретической основе обсуждаемого подхода. Например, в [18] рассказывается о применении LTL-спецификаций для планирования движений манипулятора робота-бармена за прилавком суши-ресторана. Применение именно LTL-подхода дало значительный прирост в «интеллектуальности» системы. Например, робот «понимал», что предметы, которые лежат на месте других или не дают перенести банку на другое место, должны быть временно отодвинуты в другую сторону, что раньше достигалось только методом *бэктрекинга*, описанном в статье [19] (однако бэктрекинг работал не во всех ситуациях и был крайне неэффективен во временном и ресурсном отношении). Работа [20] описывает планирование для мультиагентной системы раздельного сбора отходов. В работе [21] обсуждается планирование поведения робота-спасателя, исследующего место катастрофы с целью тушения пожаров и предоставления первой помощи выжившим.

Следует отметить, что количество опубликованных работ на тему планирования поведения робота при помощи LTL-спецификации на момент 2015 года имеет порядок сотен, поэтому авторы считают уместным ограничиться здесь лишь рассмотренными статьями, так как они дают общую характеристику подхода. Актуальные статьи на данную тему можно найти, в частности, в списках докладов секций формальных методов конференций ICRA и IROS.

III. Сети Петри

Один из самых старых и некогда один из самых популярных методов формального анализа программ — сети Петри. Впервые предложенные Карлом Петри в 1962 году, они предназначались прежде всего для моделирования и анализа параллельных и распределённых систем. Они интересны тем, что имеют наглядную графическую нотацию, и многие их свойства, имеющие важное практическое значение, можно установить аналитически, зная топологию и начальное состояние сети. Сети Петри быстро стали популярны, например, в статье [22], опубликованной в 1991 году, приводится библиография из 4099 статей, рассматривающих сети Петри или их применения. Мы здесь рассмотрим лишь некоторые применения сетей Петри в робототехнике.

Сетью Петри мы, следуя [23], будем называть тройку (P, T, ϕ), где P — конечное множество позиций, T — конечное множество переходов, а $\phi : (P \times T) \cup (T \times P) \to N$ — функция потока. Маркировкой называется отображение $\mu : P \to N$, ставящее в соответствие позиции количество маркеров, которые находятся в данной позиции. Графически сеть Петри представляется как двудольный граф, в котором позиции обозначаются кругами (и маркеры — точками в позициях), переходы — прямоугольниками, функция потока рисуется как рёбра графа (каждое ребро может быть входным или выходным и иметь вес, который пишется над ребром, см. рис. 1).

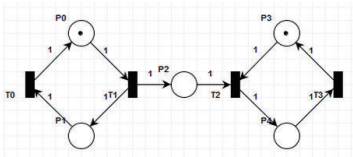


Рис. 1: Визуализация сети Петри.

Переход $t \in T$ называется активным при маркировке μ, если $\forall p \in P\ \phi(p, t) \leq \mu(p)$. Любой активный переход сети может сработать, в этом случае из каждой входной позиции p перехода удаляются $\phi(p, t)$ маркеров и для каждой выходной позиции перехода p' в неё добавляются $\phi(t, p')$ маркеров. Можно записать срабатывание перехода как $\mu \xrightarrow{t} \mu'$, где $\mu'(p) = \mu(p) - \phi(p, t) + \phi(t, p)\ \forall p \in P$.

Практический смысл приведённых определений можно представить себе так: позиция — это состояние системы, либо предусловие для выполнения некоторого действия. Переход — это действие, выполняемое системой. Функция потока показывает поток управления или поток данных в системе, маркировка показывает состояние, в котором в определённый момент времени находится система (при этом маркер можно понимать как указатель на текущую исполняемую инструкцию, хотя, конечно, это далеко не всегда так), активный переход — действие, которое может быть исполнено, тот факт, что может сработать любой активный переход в сети — неопределённость, возникающую при параллельном исполнении (гонки), срабатывание перехода — исполнение действия и передачу управления (или данных) дальше.

Про сеть Петри с заданной топологией и маркировкой можно аналитически установить следующие свойства:

- **Достижимость**: существует ли последовательность срабатываний переходов из маркировки μ в маркировку μ'. Практический смысл — может ли система в принципе решить поставленную задачу, или наоборот, оказаться в нежелательном состоянии.
- **Ограниченность**: существует наперёд заданное число k такое, что в любой достижимой маркировке сети в каждой позиции находится не более k маркеров. Практический смысл — в системе никогда не произойдёт переполнения буфера.
- **Безопасность**: в каждой позиции сети в каждой достижимой маркировке находится не более одного маркера. Практический смысл — возможность не использовать буферы вообще, проверка корректности моделирования (например, движения собираемой детали по конвейеру).
- **Живость**: достижимость из данной маркировки, в которой заданный переход может сработать. Живость вводится как для конкретного перехода, так и для сети в целом, при этом определяется 6 уровней живости, от 0-го (переход не может сработать никогда) до 5-го (в любой маркировке, достижимой из заданной, существует последовательность переходов, в которой переход сработает хотя бы один раз). Практический смысл — анализ системы на возможность тупиковых ситуаций, поиск мёртвого кода.

Существует несколько аналитических методов анализа, которые будут лишь упомянуты здесь в силу ограниченности объёма статьи:

- **Анализ уравнения состояния** — сеть Петри представляется в виде матрицы инцидентности, тогда текущее состояние выражается в виде линейного уравнения, решения которого показывают количество срабатываний переходов, необходимых для перевода сети из начального состояния в некоторое заданное. Если решений нет, заданное состояние недостижимо, но если решения есть, это ещё не значит, что заданное состояние достижимо. Методы, связанные с уравнением состояния, имеют низкую вычислительную сложность, но некоторые важные свойства установить не могут.
- **Анализ графа (или дерева) достижимости** — вершинами такого графа являются достижимые состояния сети, рёбрами — срабатывающие переходы. Такие методы позволяют установить любое из перечисленных свойств сети, но для неограниченных сетей граф будет бесконечным. Для анализа неограниченных сетей вводится граф покрытия, в котором потенциально бесконечное количество маркеров обозначается специальным символом ω, но в таком случае теряется информация о переходах сети и, например, достижимость установить уже не удастся. В любом случае, граф достижимости (покрытия) имеет размер, в общем случае экспоненциально зависящий от количества позиций в сети, так что алгоритмы анализа, как правило, имеют экспоненциальную трудоёмкость.
- **Структурный анализ** — исследование топологии сети в поиске структур с известными свойствами. Этот метод тоже не позволяет установить все свойства.

Как видно, в общем случае доказательство практически важных свойств сети Петри имеет большую временную трудоёмкость, но, тем не менее, алгоритмически разрешимо. Поэтому сети Петри имеют меньшую выразительную силу, чем машины Тьюринга, это их недостаток (поскольку сетью Петри нельзя смоделировать произвольную программу) и достоинство (для машин Тьюринга перечисленные выше свойства в общем случае алгоритмически неразрешимы). Вводятся различные расширения сетей Петри, такие как ингибиторные сети, цветные сети, сети с приоритетами, временные сети и т.д., большинство из них тьюринг-полны, так что не поддаются формальному анализу. Они, тем не менее, интересны тем, что имеют формальную семантику и могут быть использованы для динамической верификации путём симуляции сети. На этом теоретическое введение заканчивается, подробнее про сети Петри можно прочитать в статье [24], оценки трудоёмкости анализа сетей приводятся в статье [23], подробный обзор работ о разрешимости некоторых проблем в контексте сетей Петри до 1994 года можно найти в [25], хорошее введение в сети Петри с точки зрения их приложений в промышленности приводится в [26], пример работы о применении в технологическом процессе — [27].

В робототехнике сети Петри используются разными способами. В работах [28] и [29] сеть Петри используется для моделирования роботов-футболистов и их окружения. При этом инструментальные средства, предлагаемые авторами, позволяют сначала оценить качественные свойства системы (например, возможно ли вообще в данных условиях забить мяч в ворота), рассматривая модель как классическую сеть Петри, затем, с помощью симуляции, количественные свойства (такие как вероятность забить гол и оценка времени для решения этой задачи), рассматривая модель как временную сеть. Аналогично, статья [30] описывает симулятор для разных видов сетей Петри (временных, ингибиторных, с приоритетами), который используется для верификации и оптимизации разных технологических процессов (например, добычи руды в шахте с использованием полуавтономных погрузочно-транспортных машин). Для верификации робототехнических систем расширенные сети Петри используются и в работах [31] и [32], причём

в последнем случае предлагается новое расширение сетей Петри, и для их формального анализа используются техники model checking.

Следующий важный класс применений сетей Петри в робототехнике — в качестве языка программирования. Все такие работы используют расширенные сети, что делает невозможным их формальный анализ, но наличие чёткой семантики и наглядность упрощают разработку программ и написание для них интерпретаторов. Примеры таких работ — [33] (визуальный язык со вставками на языке Python для программирования мультиагентных систем), [34] (интерпретация ингибиторных сетей Петри для управления группой роботов, с визуальным редактором сетей и посылкой команд роботам по радиоканалу), [35] (программирование одновременно нескольких роботов с помощью расширенной сети, где задания для каждого робота «нарезаются» из общей программы, с приложением опять-таки в робофутболе).

Сети Петри также используются для планирования и распределения задач между роботами в группе, хороший пример такой работы — [36], здесь действия каждого робота в группе моделируются ингибиторной сетью, определяется, может ли один робот в группе заменить другого для решения той или иной подзадачи, предлагается алгоритм построения оптимального с точки зрения времени выполнения плана работы группы.

Несколько менее типичный пример использования сети Петри — работа [37], где сеть используется для представления знаний робота при обучении по демонстрации. Человек-учитель перемещает предметы в поле зрения робота, при этом стационарные состояния предметов становятся позициями сети, а сами перемещения — переходами. После этого робот анализирует граф достижимости сети, чтобы найти кратчайшую последовательность перемещений, которые привели бы предметы из одного состояния в другое.

IV. Communicating Sequential Processes

Речь в данном разделе пойдет о формальном языке описания параллельных систем CSP (Communicating Sequential Processes), языке occam[6], построенном на формализме CSP и о некоторых работах, посвященных их применению в робототехнике. CSP принадлежит к классу математических теорий под общим названием *алгебр процессов* (*process algebras*) [38], [39]. В простейшем виде алгебра процессов для множества атомарных действий A определяется как замыкание A относительно операций альтернативы («+»), последовательной композиции («·») и параллельной композиции («||»).

Алгебра процессов CSP состоит из множества атомарных событий и элементарных процессов STOP и SKIP. Первый — процесс над пустым алфавитом, т.е. процесс, не взаимодействующий с другими процессами и окружением и означающий аварийную остановку работы (иначе его называют *тупиком*). Второй означает нормальное завершение работы системы. Каждое событие является атомарной сущностью, соответствующей какому-либо явлению моделируемой предметной области. Все события в модели CSP являются мгновенными, при этом не важен интервал времени, разделяющий события, важна лишь упорядоченность событий.

Для подробного ознакомления с формализмом мы отсылаем читателя к оригинальным работам [40], [41], [42]. Здесь будут неформально определены основные операции в алгебре CSP.

- **Префикс** («→»). Выражение $Q = a \to P$ конструирует новый процесс Q, который ждет события a, а затем ведет себя как P. В качестве примера приведем простейший бесконечный процесс работы часов: $CLOCK = tick \to CLOCK$. Такая запись может быть развернута бесконечной подстановкой выражения для $CLOCK$ в правую часть в последовательность $tick \to tick \to tick \to \ldots$. Запись $Q = (x : B \to P(x))$ предлагает выбрать любое событие x из множества событий B, и далее процесс ведет себя как $P(x)$. В таком случае B называют *меню* процесса P.
- **Недетерминированный выбор** («⊓»). Выражение $P = Q \sqcap R$ конструирует процесс, который ведет себя либо как Q, либо как R, при этом выбор имеет внутреннюю природу, которая не определяется, и среда не может влиять на то, в чью пользу будет сделан этот выбор. Другими словами, процесс $(a \to Q) \sqcap (b \to R)$ может отказаться принимать a или b, и природа этого отказа носит внутренний характер.
- **Детерминированный выбор** («□»). $P = (a \to Q)\Box(b \to R)$ конструирует процесс P, который ведет себя как P или Q в зависимости от события, выработанного окружением. В случае, когда окружение произвело событие, принимаемое обоими операндами, выбор между ними будет произведен недетерминировано.
- **Параллельная композиция по пересечению** («||»). Данный оператор позволяет процессам исполняться параллельно, но только в случае, когда оба процесса к этому готовы. Более формально,

$$(x : B \to P(x)) \mid\mid (y : C \to Q(y)) = (z : (B \cap C) \to (P(z) \mid\mid Q(z))). \quad (1)$$

  Оператор параллельной композиции по пересечению часто применяется для описания синхронизации процессов в формализме CSP.
- **Параллельная композиция чередованием** (*interleaving*, «|||»). В отличие от композиции по пересечению, чередование позволяет исполняться параллельно независимо друг от друга, т.е. каждое

[6] Страница occam на сайте WoTUG, URL: http://www.wotug.org/occam (дата обращения: 30.09.2015)

событие требует участия одного процесса вместо двух. Формально, если B — меню процесса P, а C — меню процесса Q, то

$$P \mid\mid\mid Q = (x : B \to (P(x) \mid\mid\mid Q)) \\ \Box\, (y : C \to (P \mid\mid\mid Q(y))), \quad (2)$$

т.е. события P и Q «чередуются» во времени, откуда и произошло название операции.

- **Последовательная композиция** («;») позволяет вести себя процессу $(P; Q)$ как процессу P до тех пор, пока тот не закончит свою работу, а затем как Q. В частности,

  ```
  SKIP;P = P,
  STOP;P = STOP.
  ```

- **Сокрытие** (concealment, «\»). Процесс $P \setminus b$ ведет себя как P, за исключением того, что все вхождения b удаляются из всех траекторий исполнения P. В частности, $(b \to P) \setminus b = P \setminus b$.
- **Образ** и **прообраз**. Пусть f — тотальное отношение из множества событий в множество событий, причем прообраз каждого события конечен. Тогда $f(P)$ определяет процесс, который выполняет $f(a)$ тогда и только тогда, когда P выполняет a, $f^{-1}(P)$ — процесс, который выполняет а тогда и только тогда, когда P выполняет $f(a)$.

Существует множество инструментов для анализа CSP, например, FDR2[7] и PAT[8]. FDR2, применения которого мы рассмотрим ниже, является коммерческим инструментом проверки моделей для CSP (*refinement checker*), позволяющим формально доказать выполнимость свойств процесса, описанного на входном языке FDR2, который практически полностью совпадает с версией языка CSP самого Ч.Э.Хоара (автора CSP) из [42]. Также CSP повлиял на такие языки программирования, как *occam*-π [43] и Go[9].

Перейдем к рассмотрению работ применения описанного формализма к задачам робототехники. В 1998 году была опубликована статья [44], в которой обсуждается применение формальных методов в цеях безопасности на примере реабилитационной робототехники. Работа ведется в контексте бременского автономного кресла — интеллектуальной системы для передвижения людей преклонного возраста и людей с ограниченными возможностями. Предлагается использование подхода с моделированием дерева опасностей [45], [46] посредством CSP. Дерево опасностей представляет собой иерархическое представление всех рисков, которым подвергается система, начиная от самого абстрактного описания в корневых узлах до самого конкретного в листьях. Узлы-дети конкретизируют опасности родителя, объединяясь либо логическим И (родительская опасность случается тогда и только тогда, когда случаются все опасности, соответствующие узлам-детям), либо логическим ИЛИ (родительская опасность случается, когда случается хотя бы одна дочерняя опасность). Полное дерево опасностей для бременского автономного кресла имеет 170 узлов и охватывает опасности от угроз столкновения (активного и пассивного, т.е. произошедших по вине самого кресла и по причине внешних обстоятельств) до угрозы потери связи между программными и аппаратными компонентами кресла. Дерево опасностей (или его поддеревья) далее описывается на входном языке FDR2, происходит моделирование окружения и системы со случайными выработками событий опасностей. Утверждается, что такой подход более выразителен, чем просто постановка требований в терминах LTL (см. раздел II), т.к. в общем случае дерево опасностей не выражается в LTL. Наконец, производится верификация сетевых протоколов взаимодействия компонент кресла с блокировками и двойной буферизацией, а также обсуждаются другие возможные применения для подхода с деревом опасностей.

Работа [47] интересна в контексте образовательной робототехники, а также в контексте визуального моделирования. В ней обсуждается разработка визуального языка для программирования роботов Surveyor SRV-1 в целях обучения студентов Университета Кента параллельному программированию, для чего раннее использовался язык *occam*-π [43], основанный на формализмах π-исчисления [43] и CSP.

В работе [48] обсуждается применение языка *occam*-π для построения роевой [49] системы роботов сбора отходов. Для реализации системы выбрана архитектура Брукса [50], в которой система представляется как набор уровней ответственности, при этом работоспособность верхних уровней напрямую зависит от нижних, которые могут, например, контролировать работу датчиков или обеспечивать передвижение мобильного робота, избегая при этом столкновений с препятствиями. Характерной чертой такой архитектуры является то, что отказ работы верхних уровней не влечет отказа всей системы, робот все еще в состоянии делать более «примитивные» действия. Использование *occam*-π позволило представить каждый уровень ответственности как отдельный процесс, что оказалось довольно удобным при программировании системы сбора отходов группой роботов.

Работа [51] изучает подход к планированию передвижения роботов с использованием алгебр процессов. Общая схема подхода примерно такая же, как у описанного в разделе II данной работы, однако вместо спецификаций LTL используются алгебры процессов.

[7] Домашняя страница FDR2, URL: http://www.fsel.com (дата обращения: 30.09.2015)

[8] Домашняя страница PAT, URL: http://pat.comp.nus.edu.sg (дата обращения: 30.09.2015)

[9] Домашняя страница Go, URL: https://golang.org (дата обращения: 30.09.2015)

V. Гибридные системы

Робототехническая система, действующая в реальном мире, всегда является *гибридной системой* в том смысле, что совмещает дискретную логику поведения, отражающую внутреннее состояние системы, с непрерывной динамикой (управление скоростями моторов, углом соединения деталей, длина тормозного пути и т.д.). При этом класс гибридных систем не ограничивается одними лишь роботами. Их использование довольно естественно в автомобиле-, авиа- и судостроении, а также различных отраслях науки (включая такие, как экология хищников-жертв).

На тему теории гибридных систем выходят целые сборники (например, [52], [53]), создано множество инструментов анализа, включая HyTech [54], PHAVer [55], KeYmaera [56]. Мы рекомендуем работу [57] для ознакомления с общими принципами работы верификаторов гибридных систем. Среди приложений теории гибридных систем к робототехнике авторам не известны какие-либо «обособленные» работы, которые не могли бы быть применены для других классов инженерных систем, поэтому ограничимся здесь лишь приведенным общим описанием области.

VI. Марковские модели

Еще одной активно развивающейся областью применения математического аппарата к задачам робототехники является теория марковских процессов принятия решений. Для подробного ознакомления с теорией марковских процессов мы отсылаем читателя к [58].

Марковским процессом принятия решений (Markov Decision Process, MDP) назовем шестерку $M = (S, A, D_0, P, R, \gamma)$, где

- S — конечное множество состояний;
- A — конечное множество действий;
- $D_0 : S \to \mathbb{R}$ — начальное распределение вероятности по состояниям;
- $P : S \times A \times S \to [0, 1]$ — функция вероятности перехода. По данным $s, s' \in S$ и $a \in A$ величина $P(s, a, s')$ означает вероятность перехода из состояния s в состояние s' при совершении системой действии a;
- $R : S \times A \times S \to \mathbb{R}$ — функция награды. Значение $R(s, a, s')$ есть величина награды, полученной системой немедленно после перехода из состояния s в состояние s' при выполнении действия $a(s, s' \in S, a \in A)$;
- $\gamma \in [0, 1]$ — фактор скидки, означающий важность ближайших наград относительно будущих наград.

Политикой процесса M назовем отображение $f : S^* \to D(A)$, где S^* — множество конечных кортежей, все элементы которого принадлежат S, $D(X)$ — множество функций распределения вероятности над множеством X. Политика f возвращает по «истории» передвижения по состояниям марковского процесса распределение вероятностей очередного действия системы. *Детерминированная политика* — политика марковского процесса, отображающая конечный путь в одно конкретное действие. Очевидно, что при фиксированном марковском процессе M и политике f, комбинация (M, f) ведет себя в точности как марковская цепь.

В классическом виде задачей анализа MDP является поиск политики f, максимизирующей потенциальную награду над бесконечным горизонтом, иначе говоря, ставится экстремальная задача

$$\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \to max,$$

где действие a_t выбирается по состоянию s_t в соответствие с политикой f. Значения γ, близкие к 0, заставляют систему пренебрегать более удаленными во времени наградами, близкие к 1 — учитывать их.

MDP применяются для моделирования поведения дискретных систем, поведения которых носят стохастический характер, поэтому часто применяются в робототехнике. С их помощью моделируется поведение роботов и их частей (вероятности переходов в таком случае означают, вероятности успешного выполнения задачи, например, захвата предмета клешней), элементов окружения и даже поведения человека [59], [60], [61] (к примеру, когнитивного состояния оператора робота или человека как движущегося препятствия).

В контексте формальных методов интересны расширения марковских процессов множеством атомарных предикатов AP и функцией пометок $L : S \to 2^{AP}$. Такая структура носит название *марковского процесса принятия решений с метками (Labeled Markov Decision Drocess, LMDP)*. Можно проследить аналогию между LMDP с метками и структурами Крипке: первые соответствуют, вторым, но переходы между состояниями носят вероятностный характер и могут не произойти. Детерминированная политика LMDP будет в таком случае соответствовать траектории на структуре Крипке. Мы заимствуем терминологию и определения из раздела II и отсылаем читателя к тем же работам для лучшего понимания.

Как известно, по формуле LTL можно построить соответствующий автомат Бюхи. Напоминаем, что в общей схеме алгоритма model checking для LTL по структуре Крипке K (точнее, по соответствующему ей автомату Бюхи B_K) и автомату Бюхи, распознающего неправильное поведение $B_{\neg\phi}$, строится синхронная композиция автоматов $B = B_M \otimes B\neg\phi$, в которой ищется допускающий проход, который, если найден, является контрпримером к требованию корректности ϕ. Аналогично для марковского процесса принятия решений M и формуле LTL ϕ может быть построена их композиция $M' = M \boxtimes A_\phi$, где A_ϕ — детерминированный автомат Рабина, соответствующий ϕ. Формальное определение этой композиции лишь усложнит содержание данной статьи и может быть найдено, к примеру,

в [62]. Интуитивно, такая композиция аналогична синхронному произведению автоматов, оставляя вероятности и награды на переходах соответствующими LMDP M, но само множество переходов между состояниями редуцируется до присутствующих в автомате A_ϕ. Полученное произведение позволяет рассуждать о синтезе политик, максимизирующих награду над бесконечным (или конечным) горизонтом и удовлетворяющих свойству корректности ϕ.

В [62] приводится интересное приложение описанной теории к задаче *гибридного управления* роботом (*shared autonomy robot*). Для робота, у которого есть как подсистема операторского контроля, так и подсистема автономного поведения, строится такая модель поведения системы, которая находится в состоянии парето-оптимума между максимизацией вероятности выполнения задания, поставленного LTL-спецификацией, и стоимостью работы оператора. Для синтеза такой стратегии поведение робота и когнитивное состояние оператора представляется марковским процессом.

Другая часто применяемая в контексте робототехники математическая структура — частично наблюдаемые марковские процессы принятия решений (*Partially Observable Markov Decision Process, POMDP*). От обычных MDP POMDP отличаются тем, что состояние, в котором находится система, неизвестно, и система может только поддерживать набор вероятностей нахождения в определенном состоянии, который она получает из *наблюдений* о поведении окружающей среды в ответ на выполнение какого-либо действия. Формально, частично наблюдаемый марковский процесс принятия решений — кортеж $(S, A, D_0, P, R, Z, O, \gamma)$, где S, A, D_0, P, R и γ соответствуют аналогичным элементам из определения MDP, Z — конечное множество наблюдений, $O : S \to Z$ — отображение, сопоставляющее состоянию наблюдение. Очевидно, что в реальном мире часто приходится сталкиваться с ситуациями, описывающимися POMDP, что подтверждается большим числом работ, посвященных применению POMDP в робототехнике.

Формальный анализ POMDP разделяется на *количественный* и *качественный*. Задачей качественного анализа POMDP является нахождение политики, удовлетворяющей цели с вероятностью 1. Количественный анализ задается вопросом об удовлетворении цели с вероятностью $\lambda \in (0, 1)$. В [63] доказывается неразрешимость задачи количественного анализа. В [64] приводится конструктивное доказательство разрешимости EXPTIME-полноты качественного анализа POMDP.

Отметим, что область анализа POMDP является на данный момент мало исследованной. Одной из самых интересных на данный момент работ на тему применения качественного анализа POMDP в контексте робототехники, на взгляд авторов, является статья [65] 2015 года, где исследовано применение качественного анализа POMDP совместно с LTL-требованиями для классических задач планирования поведения робота.

VII. Другие используемые формализмы

A. Process Algebra for Robot Schemas

Существуют формализмы, которые не попали ни в одну из категорий выше. Один из таких формализмов — ещё один вариант алгебры процессов, PARS (Process Algebra for Robot Schemas), описанный в работах [66] и [67]. Этот подход рассматривает модель программы робота, аппаратного обеспечения робота и среды как набор процессов, каждый из которых имеет набор входных и выходных переменных, преобразует входные переменные в выходные, используя данные, которые он может читать из входных портов, при этом процесс может писать в выходные порты. Каждый процесс может завершиться в двух состояниях — «stop» и «abort», или не завершиться вовсе. Процессы комбинируются с помощью операторов «;» (последовательное выполнение), «|» (параллельное выполнение до тех пор, пока все процессы не завершатся) и «#» (параллельное выполнение до тех пор, пока хоть один из процессов не завершится). Последовательное выполнение передаёт управление следующему процессу, только если предыдущий завершился в состоянии stop, что дает возможность моделировать условный оператор:

$$T = (Eq\langle a, b\rangle; P \mid Neq\langle a, b\rangle; Q),$$

где Eq — процесс, который заканчивается в состоянии «stop», если его входные переменные a и b равны, и в состоянии «abort» в противном случае, Neq — аналогично моделирует неравенство. Циклическое исполнение моделируется через хвостовую рекурсию:

$$T\langle a\rangle\langle b\rangle = P\langle a\rangle\langle b\rangle; T\langle b\rangle.$$

Таким образом, выразительной мощности алгебры процессов достаточно, чтобы выразить любую программу. Временной аспект вводится в модель посредством процесса $Delay\langle t\rangle$, который останавливается в состоянии «stop» по истечении времени t, коммуникации между параллельно исполняемыми частями системы — через процессы $In\langle p\rangle\langle x\rangle$ и $Out\langle p, x\rangle$.

Ограничение на систему вводится как выражение на PARS, дополненное логическим условием на переменные, используемые процессами в выражении. Для того, чтобы моделировать неопределённость, имеющуюся в реальном мире, переменные рассматриваются как случайные величины с некоторой функцией распределения. Вводится процесс $Ran\langle\Phi\rangle\langle v\rangle$, возвращающий случайную величину v с распределением Φ.

Целью введения такого формализма было подавление комбинаторного взрыва, которому подвержены многие другие методы анализа. Достигается это так: рассматриваются только системы, представляющиеся в виде совокупности параллельных циклических процессов (выражаемых с помощью хвостовой рекурсии), для них считается функция потока, показывающая, как входные параметры преобразуются в выходные за одну

итерацию. Это не накладывает серьёзных ограничений на моделируемые системы, поскольку представление в требуемом виде во многих случаях может быть построено автоматически (причём, полиномиальным относительно количества процессов алгоритмом). Далее переменные, используемые процессами, «пропускаются» через динамическую байесовскую сеть, чтобы определить вероятность удовлетворения ограничения. Таким образом, можно априори определить вероятность успешного выполнения миссии. Авторы поставили более сотни экспериментов, в результате которых установлено, что реальные значения вероятности выполнения миссии статистически соответствуют предсказанным.

B. Integrated Behavior-Based Control

Ещё один интересный формализм, используемый для программирования роботов, — Integrated Behavior-Based Control (i2BC), формализм и архитектура, описанные в статье [68]. Архитектура базируется на известной в среде робототехников работе [50] и представляет программу в виде сети взаимодействующих *поведений*. Поведение преобразует входные данные в выходные, кроме того, имеет входы *активации* и *ингибиции*. Входы принимают сигнал в виде числа от 0 до 1, при этом смысл их таков: если ингибиция равна 1, то поведение не должно работать, если 0, то должно работать полностью. Соответственно, если активация равна 1, то наоборот, поведение должно работать, если 0, то не должно (обычно их комбинируют как $a*(1-i)$, где a — это активация, i — ингибиция). Помимо выхода для данных есть ещё выходы активности и целевого рейтинга. Это тоже числа от 0 до 1, активность показывает, насколько поведение активно и готово влиять на состояние системы (с учётом активации и ингибиции), целевой рейтинг — насколько поведение «удовлетворено» тем, что происходит. Наличие такой структуры позволяет разделить поток данных и поток управления и реализовать сеть из влияющих друг на друга поведений без нужды в централизованной координации. Благодаря активностям и ингибициям поведения могут перехватывать управление (например, подсистема уклонения от столкновений может подавлять управляющее воздействие от поведения более высокого уровня, обрабатывающего действия оператора пульта дистанционного управления). Возможна и более сложная логика комбинирования поведений, для этого применяется *fusion behavior*, специальный вид поведения, имеющий несколько входов данных, активаций и ингибиций, и выдающий результат согласно некоторой внутренней функции комбинирования этих сигналов.

Как формализм i2BC интересна тем, что сводима к набору конечных автоматов, для которых оказываются применимы методы model checking, как описано в работе [69]. Каждое поведение представляется в виде набора автоматов, связанных друг с другом каналами для сообщений. Автоматное представление системы может быть получено автоматически по модели. Авторы [69] проводили эксперименты по оценке влияния отказов датчиков на свойства системы управления мобильным роботом, в результате чего выяснилось, что некоторые свойства доказываются относительно эффективно (порядка секунд на обычном настольном компьютере), но некоторые вообще не доказываются за «разумное» время (за два дня на процессоре Intel XEON 2.7 ГГц с 1024 Гб оперативной памяти).

C. Формальная логика

Помимо верификации уже существующих систем, формальные методы применяются при нахождении корректных «по построению» и оптимальных решений. Ещё один пример такого применения (помимо описанных в разделе II) — решение задачи о нахождении оптимального плана реконфигурации модульных роботов путём сведения её к задаче SAT методами матлогики [70]. Оптимальный (в смысле количества шагов соединения-рассоединения) план для групп размером примерно в 400 роботов находился существующими решателями SAT за время порядка десятков минут.

VIII. Динамическая верификация

Рассмотренные ранее методы предназначаются прежде всего для верификации программы до запуска на реальном роботе или создания корректных «по построению» программ, но в реальных условиях этого оказывается недостаточно. Даже если удалось полностью верифицировать модель, необходимо ещё убедиться, что модель соответствует реальному миру, в котором будет функционировать робот. Доказать в общем случае невозможно в силу непредсказуемости реального мира, но можно проверить во время выполнения и активировать аварийный механизм поведения, если данные о реальном мире перестали соответствовать предсказанным моделью.

Пример такой работы — средство ModelPlex, описанное в работе [71]. Верифицированная модель робота, программы управления и окружающей среды здесь уже дана, цель работы — сгенерировать код верификатора, который бы запускался на роботе каждый раз, когда вырабатывается новое управляющее воздействие, проверял, что параметры соответствуют предсказанным моделью и запускал бы аварийный режим управления, если нет. Авторы формально доказывают корректность и своевременность перехода на аварийный режим работы при условии ограниченности возможных отклонений реальной среды от модели.

Менее формальный метод обеспечения корректности предложен для системы ROS[10] в работе [72]. Система ROS представляет программу в виде модулей (узлов), связанных каналами, по которым узлы обмениваются сообщениями. В работе предлагается вставлять между

[10]Robot Operating System, URL: http://www.ros.org/ (дата обращения: 30.09.2015)

двумя взаимодействующими узлами монитор, который бы верифицировал сообщения, передающиеся по каналу, и в случае, если сообщения нарушают заданные ограничения, инициировать аварийное поведение. Монитор описывается на специальном языке, после чего генерируется код для ROS. В статье приводится пример с контролем угла наклона пейнтбольной пушки на роботе, чтобы подавить сигнал о выстреле, если робот может попасть в себя.

IX. Заключение

В статье приведён обзор использования формальных методов в робототехнике. Обзор проводился по материалам конференций ICRA и IROS последних лет, а также данным Scopus и Google Scholar. Наиболее популярный и исследуемый подход на данный момент — постановка задач посредством LTL-спецификаций, также активно ведутся исследования применений сетей Петри и марковских моделей. Есть и ряд альтернативных направлений, призванных избежать вычислительной сложности более популярных подходов, либо сложности специфицирования систем в таких формализмах. Также в контексте робототехники актуальна задача динамической верификации, когда корректность поведения робота контролируется в процессе его работы в реальном времени.

Следует отметить, что часто формальные методы применяются не для верификации, а для синтеза программы поведения робота по заданной модели среды и желаемым результатам. Это позволяет задавать поведение робота очень декларативно, и результирующая программа будет корректна по построению, однако такое построение часто требует довольно больших вычислительных ресурсов.

Работа выполнена в рамках «Межвузовской проектной лаборатории робототехники» — совместного проекта компаний JetBrains и КиберТех Лабс.

Список литературы

[1] Antoniotti Marco, Mishra Bud. Discrete event models+ temporal logic= supervisory controller: Automatic synthesis of locomotion controllers // Robotics and Automation, 1995. Proceedings., 1995 IEEE International Conference on / IEEE. T. 2. 1995. C. 1441–1446.

[2] Alpern Bowen, Schneider Fred B. Defining liveness // Information processing letters. 1985. T. 21, № 4. C. 181–185.

[3] Clarke Edmund M., Emerson E. Allen, Sistla A. Prasad. Automatic verification of finite-state concurrent systems using temporal logic specifications // ACM Transactions on Programming Languages and Systems (TOPLAS). 1986. T. 8, № 2. C. 244–263.

[4] Карпов Юрий Глебович. MODEL CHECKING. Верификация параллельных и распределенных программных систем. БХВ-Петербург, 2010.

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

[6] Emerson E Allen. Temporal and modal logic. // Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B). 1990. T. 995, № 1072. C. 5.

[7] Piterman Nir, Pnueli Amir, Sa'ar Yaniv. Synthesis of reactive (1) designs // Verification, Model Checking, and Abstract Interpretation / Springer. 2006. C. 364–380.

[8] Pnueli Amir, Rosner Roni. On the synthesis of a reactive module // Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages / ACM. 1989. C. 179–190.

[9] Kripke Saul A. Semantical analysis of modal logic i normal modal propositional calculi // Mathematical Logic Quarterly. 1963. T. 9, № 5-6. C. 67–96.

[10] Fainekos Georgios E, Kress-Gazit Hadas, Pappas George J. Temporal logic motion planning for mobile robots // Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on / IEEE. 2005. C. 2020–2025.

[11] Nusmv 2: An opensource tool for symbolic model checking / Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia [и др.] // Computer Aided Verification / Springer. 2002. C. 359–364.

[12] Holzmann Gerard J. The SPIN model checker: Primer and reference manual. Addison-Wesley Reading, 2003.

[13] Belta Calin, Habets Luc CGJM. Constructing decidable hybrid systems with velocity bounds // Decision and Control, 2004. CDC. 43rd IEEE Conference on / IEEE. T. 1. 2004. C. 467–472.

[14] Kress-Gazit Hadas, Fainekos Georgios E, Pappas George J. Where's Waldo? Sensor-based temporal logic motion planning // Robotics and Automation, 2007 IEEE International Conference on / IEEE. 2007. C. 3116–3121.

[15] Kress-Gazit Hadas, Fainekos Georgios E, Pappas George J. Temporal-logic-based reactive mission and motion planning // Robotics, IEEE Transactions on. 2009. T. 25, № 6. C. 1370–1381.

[16] Composition of local potential functions for global robot control and navigation / David C Conner, Alfred Rizzi, Howie Choset [и др.] // Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on / IEEE. T. 4. 2003. C. 3546–3551.

[17] Livingston Scott C, Murray Richard M. Just-in-time synthesis for reactive motion planning with temporal logic // Robotics and Automation (ICRA), 2013 IEEE International Conference on / IEEE. 2013. C. 5048–5053.

[18] Towards manipulation planning with temporal logic specifications / Keliang He, Morteza Lahijanian, Lydia E Kavraki [и др.] // Robotics and Automation (ICRA), 2015 IEEE International Conference on / IEEE. 2015. C. 346–352.

[19] Combined task and motion planning through an extensible planner-independent interface layer / Sanjeev Srivastava, Eugene Fang, Lorenzo Riano [и др.] // Robotics and Automation (ICRA), 2014 IEEE International Conference on / IEEE. 2014. C. 639–646.

[20] Raman Vasumathi, Kress-Gazit Hadas. Synthesis for multi-robot controllers with interleaved motion // Robotics and Automation (ICRA), 2014 IEEE International Conference on / IEEE. 2014. C. 4316–4321.

[21] Vasile Cristian Ioan, Belta Calin. Reactive sampling-based temporal logic path planning // Robotics and Automation (ICRA), 2014 IEEE International Conference on / IEEE. 2014. C. 4310–4315.

[22] Plünnecke Helmut, Reisig Wolfgang. Bibliography of Petri nets 1990 // Advances on I'rtri Net.s 1991, Lecture Notes in Cornputer Science. 1991. T. 524. C. 312–572.

[23] Yen Hsu Chun. Introduction to Petri net theory // Studies in Computational Intelligence. 2006. T. 25. C. 343–373.

[24] Murata Tadao. Petri nets: Properties, analysis and applications // Proceedings of the IEEE. 1989. T. 77, № 4. C. 541–580.

[25] Esparza J, Nielsen M. Decidability Issues for Petri Nets-a Survey // Bulletin of the European Association for Theoretical Computer Science. 1994. T. 52. C. 245–262.

[26] Zurawski Richard, Zhou MengChu. Petri nets and industrial applications: A tutorial // IEEE Transactions on Industrial Electronics. 1994. T. 41, № 6. C. 567–583.

[27] Мартынов Виктор Геннадьевич, Масягин Василий Борисович. Применение сетей Петри при моделировании управления технологическими процессами сборочного производ-

ства // Омский научный вестник. 2014. № 1 (127). С. 134–137.
[28] Costelha Hugo, Lima Pedro. Robotic Tasks Modeling and Analysis Based on Petri Nets. 2003.
[29] Costelha Hugo, Lima Pedro. Petri net robotic task plan representation: Modelling, analysis and execution // Autonomous Agents. 2010. С. 65–91.
[30] Конюх В.Л. Опыт применения сетей Петри для имитации поведения систем // «Имитационное моделирование. Теория и практика» ИММОД-2009, пленарный доклад. 2009. С. 27–37.
[31] Aguiar Adriano Jose Cunha, Villani Emilia. Petri Nets and Graphic Simulation for the Validation of Collaborative Robotic Cells in Aircraft Industry // ABCM Symposium Series in Mechatronics. 2010. Т. 4. С. 364–373.
[32] Fu Yujian, Drabo Mebougna. Formal Modeling and Verification of Dynamic Reconfiguration of Autonomous Robotics Systems // Proceedings of the International Conference on Embedded Systems and Applications (ESA). 2014.
[33] Simon Jochen, Moldt Daniel. PyTri, a Visual Agent Programming Language // Algorithmen und Werkzeuge f{\"u}r Petrinetze. 2010. Т. 643. С. 106—-111.
[34] Kashima Hideharu, Masuda Ryosuke. Cooperative Control of Mobile Roots Based on Petri Net // The 10th international conference on advanced robotics (ICAR'2001). 2001. С. 399–404.
[35] A Robotic Soccer Passing Task Using Petri Net Plans (Demo Paper) / F Palamara, V a Ziparo, L Iocchi [и др.] // Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems: demo papers. 2008. С. 1711–1712.
[36] Kotb Y T, Beauchemin S S, Barron J L. Petri Net-Based Cooperation In Multi-Agent Systems // Computer and Robot Vision, 2007. CRV'07. Fourth Canadian Conference on. 2007. С. 123—-130.
[37] Chang Guoting Jane, Kulić Dana. Robot Task Learning from Demonstration Using Petri Nets // 2013 IEEE RO-MAN: The 22nd IEEE International Symposium on Robot and Human Interactive Communication Gyeongju, Korea. 2013. С. 31–36.
[38] Baeten Jos CM. A brief history of process algebra // Theoretical Computer Science. 2005. Т. 335, № 2. С. 131–146.
[39] Fokkink Wan. Introduction to process algebra. Springer Science & Business Media, 2013.
[40] Hoare Charles Antony Richard. Communicating sequential processes // Communications of the ACM. 1978. Т. 21, № 8. С. 666–677.
[41] Brookes Stephen D, Hoare Charles AR, Roscoe Andrew W. A theory of communicating sequential processes // Journal of the ACM (JACM). 1984. Т. 31, № 3. С. 560–599.
[42] Hoare C. A. R. Communicating Sequential Processes. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1985.
[43] Welch Peter H, Barnes Frederick RM. Communicating mobile processes // Communicating Sequential Processes. The First 25 Years. Springer, 2005. С. 175–210.
[44] Lankenau Axel, Meyer Oliver. Formal methods in robotics: Fault tree based verification // In: Proc. of Quality Week / Citeseer. 1999.
[45] Fault tree handbook: Tech. Rep.: / William E Vesely, Francine F Goldberg, Norman H Roberts [и др.]: DTIC Document, 1981.
[46] Hansen Kirsten Mark. Linking safety analysis to safety requirements: exemplified by railway interlocking systems. Institut for Informationsteknologi, DTU, 1996.
[47] Simpson Jonathan, Jacobsen Christian L. Visual Process-Oriented Programming for Robotics. // CPA. 2008. С. 365–380.
[48] Process-Oriented Subsumption Architectures in Swarm Robotic Systems. / Jeremy C Posso, Adam T Sampson, Jonathan Simpson [и др.] // CPA. 2011. С. 303–316.
[49] Şahin Erol. Swarm robotics: From sources of inspiration to domains of application // Swarm robotics. Springer, 2005. С. 10–20.
[50] Brooks R. A robust layered control system for a mobile robot // IEEE Journal on Robotics and Automation. 1986. Т. 2, № 1. С. 14–23.
[51] Varricchio Valerio, Chaudhari Pratik, Frazzoli Emilio. Sampling-based algorithms for optimal motion planning using process algebra specifications // Robotics and Automation (ICRA), 2014 IEEE International Conference on / IEEE. 2014. С. 5326–5332.
[52] Vaandrager Frits W, van Schuppen Jan H. Hybrid Systems: Computation and Control: Second International Workshop, HSCC'99, Berg en Dal, The Netherlands, March 29-31, 1999 Proceedings. № 1569. Springer Science & Business Media, 1999.
[53] Lynch Nancy, Krogh Bruce. Hybrid Systems: Computation and Control: Third International Workshop, HSCC 2000 Pittsburgh, PA, USA, March 23-25, 2000 Proceedings. Springer Science & Business Media, 2007.
[54] Henzinger Thomas A, Ho Pei-Hsin, Wong-Toi Howard. HyTech: A model checker for hybrid systems // Computer aided verification / Springer. 1997. С. 460–463.
[55] Frehse Goran. PHAVer: Algorithmic verification of hybrid systems past HyTech // Hybrid Systems: Computation and Control. Springer, 2005. С. 258–273.
[56] Platzer André, Quesel Jan-David. KeYmaera: A hybrid theorem prover for hybrid systems (system description) // Automated Reasoning. Springer, 2008. С. 171–178.
[57] The algorithmic analysis of hybrid systems / Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs [и др.] // Theoretical computer science. 1995. Т. 138, № 1. С. 3–34.
[58] Beranek William. RONALD A. HOWARD "Dynamic Programming and Markov Processes," // Technometrics. 1961. Т. 3, № 1. С. 120–121.
[59] Pentland Alex, Liu Andrew. Modeling and prediction of human behavior // Neural computation. 1999. Т. 11, № 1. С. 229–242.
[60] Rothkopf Constantin A, Ballard Dana H. Modular inverse reinforcement learning for visuomotor behavior // Biological cybernetics. 2013. Т. 107, № 4. С. 477–490.
[61] McGhan Catharine LR, Nasir Ali, Atkins Ella M. Human intent prediction using markov decision processes // Journal of Aerospace Information Systems. 2015. С. 1–5.
[62] Fu Jie, Topcu Ufuk. Pareto efficiency in synthesizing shared autonomy policies with temporal logic constraints // arXiv preprint arXiv:1412.6029. 2014.
[63] Baier Christel, Größer Marcus, Bertrand Nathalie. Probabilistic ω-automata // Journal of the ACM (JACM). 2012. Т. 59, № 1. С. 1.
[64] Chatterjee Krishnendu, Chmelik Martin, Tracol Mathieu. What is Decidable about Partially Observable Markov Decision Processes with {\ omega}-Regular Objectives // arXiv preprint arXiv:1309.2802. 2013.
[65] Qualitative analysis of pomdps with temporal logic specifications for robotics applications / Krishnendu Chatterjee, Martin Chmelík, Raghav Gupta [и др.] // arXiv preprint arXiv:1409.3360. 2014.
[66] Performance Verification for Behavior-based Robot Missions / D Lyons, R Arkin, S Jiang [и др.] // IEEE Transactions on Robotics. 2015. Т. 31, № 3. С. 619 – 636.
[67] A Software Tool for the Design of Critical Robot Missions with Performance Guarantees / Damian M Lyons, Ronald C Arkin, Paramesh Nirmal [и др.] // Procedia Computer Science. 2013. Т. 16. С. 888–897.
[68] Proetzsch Martin, Luksch Tobias, Berns Karsten. The Behaviour-Based Control Architecture iB2C for Complex Robotic Systems. 2007.
[69] Kiekbusch Lisa, Armbrust Christopher, Berns Karsten. Formal Verification of Behaviour Networks Including Hardware Failures // Proceedings of the 13th International Conference on Intelligent Autonomous Systems (IAS-13). 2014.
[70] Gorbenko A. A., Popov V. Yu. Programming for modular reconfigurable robots // Programming and Computer Software. 2012. Т. 38, № 1. С. 13–23.
[71] Mitsch Stefan, Platzer Andre. ModelPlex: Verified Runtime Validation of Verified Cyber-Physical System Models // Runtime Verification. 2014. С. 199–214.
[72] ROSRV: Runtime Verification for Robots / Jeff Huang, Cansu Erdogan, Yi Zhang [и др.] // Runtime Verification. 2014. С. 247–254.

Верификация функциональных программ методом построения диаграмм состояний

Андрей Миронов

ФИЦ "Информатика и Управление" Российской Академии Наук

amironov66@gmail.com

Аннотация

В настоящей статье рассматривается проблема верификации функциональных программ (ФП) над символьными строками, где спецификации свойств ФП определяются другими ФП, и ФП Σ_1 удовлетворяет спецификации, определяемой ФП Σ_2, если композиция функций, определяемых ФП Σ_1 и Σ_2, принимает значение 1 на всех аргументах. Мы вводим понятие диаграммы состояний ФП, и сводим проблему верификации ФП к проблеме анализа диаграмм состояний ФП. Предложенный подход иллюстрируется примером верификации ФП сортировки.

Ключевые слова: функциональная программа, диаграмма состояний, верификация.

1 Введение

Проблема верификации программ является одной из основных проблем теоретической информатики. Для различных классов программ используются различные методы верификации. Например, для верификации последовательных программ используются метод индуктивных утверждений Флойда [1], логика Хоара [2], и т.д. Для верификации параллельных и распределённых программ используются методы, основанные на исчислении взаимодействующих систем (CCS) и π-исчислении [3], [4], теории взаимодействующих последовательных процессов (CSP) и его обобщений [5], [6], темпоральной логике и методе model checking [7], процессной алгебре [8], сетях Петри [9], и т.д.

Методы верификации функциональных программ (ФП) развиты не так удовлетворительно, как методы верификации последовательных и параллельных программ. Одними из основных методов верификации ФП являются вычислительная индукция и структурная индукция [10]. Недостаток этих методов связан с трудностью построения доказательств правильности программ. Среди других методов верификации ФП следует отметить методы, основанные на рассуждениях с типами данных и абстрактной интерпретации через уточнение вывода типов [12], метод model checking для верификации ФП [13], [14], методы основанные на потоковом анализе [11], методы основанные на понятии мультипараметрического преобразователя деревьев [15].

В настоящей статье мы рассматриваем ФП как системы алгебраических уравнений над символьными строками, вводим понятие диаграммы состояний таких ФП и излагаем метод верификации, основанный на использовании диаграмм состояний ФП. Главные преимущества нашего подхода по сравнению со всеми вышеперечисленными подходами к верификации ФП заключаются в том, что наш подход позволяет представлять доказательства корректности ФП в виде простых свойств их диаграмм состояний.

Основная идея нашего подхода заключается в следующем. Мы предполагаем, что спецификация свойств верифицируемой ФП Σ_1 определяется другой ФП Σ_2, вход которой является выходом Σ_1, т.е. мы рассматриваем ФП $\Sigma_1 \circ \Sigma_2$, которая является последовательной композицией Σ_1 и Σ_2. Мы будем говорить, что Σ_1 корректна относительно спецификации Σ_2, если функция $f_{\Sigma_1 \circ \Sigma_2}$, которая соответствует $\Sigma_1 \circ \Sigma_2$ (т.е. композиция функций, определяемых Σ_1 и Σ_2) принимает значение 1 на всех аргументах. Мы сводим проблему доказательства соотношения $f_{\Sigma_1 \circ \Sigma_2} = 1$ к проблеме анализа диаграммы состояний ФП $\Sigma_1 \circ \Sigma_2$.

Предложенный метод верификации ФП иллюстрируется примером верификации ФП сортировки. Сначала мы излагаем полное доказательство корректности ФП сортировки методом структурной индукции (это сделано для сравнения сложности метода структурной индукции и сложности предложенного метода). Затем мы излагаем доказательство корректности ФП сортировки методом построения диаграммы состояний. Доказательство вторым методом существенно короче, и кроме того, оно может быть порождено автоматически. Это свидетельствует о преимуществах предложенного метода верификации ФП по сравнению с "ручной" верификацией, основанной на построении доказательства с применением метода математической индукции.

2 Основные понятия

2.1 Термы

Мы предполагаем, что заданы множества

- $\mathcal{D}$ **значений**, представляющее собой объединение $\mathcal{D}_{\mathbf{C}} \cup \mathcal{D}_{\mathbf{S}}$, причём
  - элементы $\mathcal{D}_{\mathbf{C}}$ называются **символами**, и
  - элементы $\mathcal{D}_{\mathbf{S}}$ называются **строками символов** (или просто **строками**), каждая строка из $\mathcal{D}_{\mathbf{S}}$ представляет собой конечную (возможно пустую) последовательность элементов $\mathcal{D}_{\mathbf{C}}$,
- $\mathcal{X}$ **переменных по данным** (называемых также просто **переменными**),
- $\mathcal{C}$ **констант**,
- $\mathcal{F}$ **функциональных символов (ФС)**, и
- Φ **функциональных переменных**,

причём каждому элементу m какого-либо из вышеперечисленных множеств сопоставлен некоторый **тип**, обозначаемый записью $type(m)$, и

- если $m \in \mathcal{D} \cup \mathcal{X} \cup \mathcal{C}$, то $type(m) \in \{\mathbf{C}, \mathbf{S}\}$,
- если $m \in \mathcal{F} \cup \Phi$, то $type(m)$ является записью вида $t_1 \times \ldots \times t_n \to t$, где $t_1, \ldots, t_n, t \in \{\mathbf{C}, \mathbf{S}\}$.

Если $d \in \mathcal{D}_{\mathbf{C}}$, то $type(d) = \mathbf{C}$, и если $d \in \mathcal{D}_{\mathbf{S}}$, то $type(d) = \mathbf{S}$.

Каждой константе $c \in \mathcal{C}$ соответствует элемент множества $\mathcal{D}_{type(c)}$, называемый **значением** этой константы. Запись ε обозначает константу типа $\mathbf{S}$, значением которой является пустая строка. Мы будем предполагать, что ε является единственной константой типа $\mathbf{S}$.

Каждому ФС $f \in \mathcal{F}$ сопоставлена частичная функция вида $\mathcal{D}_{t_1} \times \ldots \times \mathcal{D}_{t_n} \to \mathcal{D}_t$, где

$$type(f) = t_1 \times \ldots \times t_n \to t.$$

Данная функция обозначается тем же символом f.

Ниже мы перечисляем некоторые ФС, входящие в $\mathcal{F}$. Рядом с каждым ФС мы указываем (через двоеточие) его тип.

1. $head : \mathbf{S} \to \mathbf{C}$. Функция $head$ определена только на непустых строках, она сопоставляет каждой непустой строке её первый символ.
2. $tail : \mathbf{S} \to \mathbf{S}$. Функция $tail$ тоже определена только на непустых строках, она сопоставляет каждой непустой строке u строку, получаемую из u удалением первого символа (называемую **хвостом** u).
3. $conc : \mathbf{C} \times \mathbf{S} \to \mathbf{S}$. Для каждой пары $(a, u) \in \mathcal{D}_{\mathbf{C}} \times \mathcal{D}_{\mathbf{S}}$ строка $conc(a, u)$ представляет собой строку, получаемую приписыванием символа a спереди к строке u.
4. $empty : \mathbf{S} \to \mathbf{C}$. Функция $empty$ сопоставляет пустой строке символ 1, и каждой непустой строке – символ 0.
5. $=: \mathbf{C} \times \mathbf{C} \to \mathbf{C}$. Значение функции $=$ на паре (u, v) равно 1, если $u = v$, и 0 – иначе.
6. $\leq: \mathbf{C} \times \mathbf{C} \to \mathbf{C}$. Мы предполагаем, что на $\mathcal{D}_{\mathbf{C}}$ задано отношение линейного порядка, и значение функции $\leq$ на паре (u, v) равно 1, если $u \leq v$, и 0 – иначе.
7. Булевы ФС: $\neg : \mathbf{C} \to \mathbf{C}$, $\wedge : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$, и т.д., соответствующие им функции действуют стандартным образом на аргументы, состоящие из 0 и 1 (т.е. $\neg(1) = 0$, и т.д.) и не определены на других аргументах.
8. $if_then_else : \mathbf{C} \times t \times t \to t$, где $t = \mathbf{C}$ или $\mathbf{S}$ (т.е. записью if_then_else обозначены два ФС), и функции, соответствующие обоим ФС, определяются одинаково:

$$if_then_else\ (a, u, v) \stackrel{\text{def}}{=} \begin{cases} u, & \text{если } a = 1 \\ v, & \text{иначе.} \end{cases}$$

Понятие **терма** определяется индуктивно. С каждым термом e связан некоторый тип $type(e) \in \{\mathbf{C}, \mathbf{S}\}$. Каждая переменная по данным или константа является термом, тип которого совпадает с типом этой переменной или константы. Если $e_1, \ldots, e_n$ – список термов, и g – ФС или функциональная переменная, причем

$$type(g) = \ type(e_1) \times \ldots \times type(e_n) \to t$$

то запись $g(e_1, \ldots, e_n)$ является термом типа t.

Мы будем записывать термы

$$head(e),\ tail(e),\ conc(e_1, e_2),\ empty(e),$$
$$= (e_1, e_2),\ \leq (e_1, e_2),\ if_then_else\ (e_1, e_2, e_3)$$

в виде

$$e_h,\ e_t,\ e_1 e_2,\ e = \varepsilon,\ e_1 = e_2,\ e_1 \leq e_2,\ e_1\ ?\ e_2 : e_3$$

соответственно. Термы, содержащие булевы ФС, записываются так, как это принято в математических текстах (т.е. в виде $e_1 \wedge e_2$, и т.п.). Термы вида $e_1 \wedge \ldots \wedge e_n$ могут записываться также в виде $\{e_1, \ldots, e_n\}$.

2.2 Понятие функциональной программы над символьными строками

Функциональная программа над символьными строками (называемая ниже просто **функциональной программой (ФП)**) – это совокупность Σ

функциональных уравнений вида

$$\begin{cases} \varphi_1(x_{11},\ldots,x_{1n_1}) = e_1 \\ \ldots \\ \varphi_m(x_{m1},\ldots,x_{mn_m}) = e_m \end{cases} \tag{1}$$

где $\varphi_1,\ldots,\varphi_m$ – различные функциональные переменные, и для каждого $i = 1,\ldots,m$ $\varphi_i(x_{i1},\ldots,x_{in_i})$ и e_i – термы одинакового типа, причём

$$X_{e_i} = \{x_{i1},\ldots,x_{in_i}\}, \quad \Phi_{e_i} \subseteq \{\varphi_1,\ldots,\varphi_m\}.$$

Мы будем обозначать записью Φ_Σ совокупность всех функциональных переменных, входящих в Σ.

ФП (1) определяет список

$$(f_{\varphi_1},\ldots,f_{\varphi_m}) \tag{2}$$

функций, соответствующих функциональным переменным из Φ_Σ, являющийся наименьшим (в смысле порядка на списках частичных функций, описанного в [10]) решением системы функциональных уравнений (1) (данный список называется **наименьшей неподвижной точкой (ННТ) ФП** (1)). Значения этих фунций м.б. вычислены стандартной рекурсией. Мы будем предполагать, что для каждой рассматриваемой ФП все компоненты ННТ, соответствующей этой ФП, являются тотальными функциями. Первая функция из списка (2) (т.е. f_{φ_1}) обозначается записью f_Σ и называется **функцией, вычисляемой ФП** Σ. Если ФП Σ имеет вид (1), то $type(\Sigma)$ обозначает тип $type(e_1)$.

3 Пример спецификации и верификации ФП

3.1 Пример ФП

Рассмотрим в качестве примера ФП, определяющую функцию сортировки строк. Эта ФП имеет вид

$$\begin{array}{l} \mathbf{sort}(x) = (x = \varepsilon)?\ \varepsilon : \mathbf{insert}(x_h, \mathbf{sort}(x_t)) \\ \mathbf{insert}(a, y) = (y = \varepsilon) \quad ?\ a\varepsilon \\ \qquad\qquad\qquad\qquad\qquad : (a \le y_h) \quad ?\ ay \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad : y_h\ \mathbf{insert}(a, y_t) \end{array} \tag{3}$$

В данной ФП определяются две функции:

- $\mathbf{sort} : \mathbf{S} \to \mathbf{S}$ – основная функция, и
- $\mathbf{insert} : \mathbf{C} \times \mathbf{S} \to \mathbf{S}$ – вспомогательная функция, которая отображает пару $(a, y) \in \mathbf{C} \times \mathbf{S}$ в строку, получаемую вставкой символа a в строку y, так чтобы если строка y упорядочена, то строка $\mathbf{insert}(a, y)$ тоже была бы упорядоченной. (мы называем строку упорядоченной, если ее компоненты образуют неубывающую последовательность)

3.2 Пример спецификации ФП

Одно из свойств корректности ФП, изложенной в предыдущем пункте, заключается в следующем: для каждой строки $x \in \mathbf{S}$ строка $\mathbf{sort}(x)$ упорядочена. Это свойство можно формально выразить следующим образом. Рассмотрим ФП, определяющую функцию **ord** проверки упорядоченности строки:

$$\begin{array}{l} \mathbf{ord}(x) = \\ = (x = \varepsilon) \quad ?\ 1 \\ \qquad\qquad : (x_t = \varepsilon) \quad ?\ 1 \\ \qquad\qquad\qquad\qquad : (x_h \le (x_t)_h) \quad ?\ \mathbf{ord}(x_t) \\ \qquad\qquad\qquad\qquad\qquad\qquad : 0 \end{array} \tag{4}$$

Функция **ord** позволяет выразить вышеупомянутое свойство корректности в виде следующего математического утверждения:

$$\forall\, x \in \mathbf{S} \quad \mathbf{ord}(\mathbf{sort}(x)) = 1 \tag{5}$$

3.3 Пример верификации ФП

Задача верификации свойства корректности ФП, описанного в предыдущем пункте, заключается в построении формального доказательства утверждения (5). Данное утверждение может быть доказано как обычная математическая теорема, например с использованием метода математической индукции. Доказательство этого утверждения может иметь следующий вид.

Если $x = \varepsilon$, то, согласно первому уравнению системы (3), верно равенство $\mathbf{sort}(x) = \varepsilon$, и поэтому

$$\mathbf{ord}(\mathbf{sort}(x)) = \mathbf{ord}(\varepsilon) = 1$$

Пусть $x \ne \varepsilon$. Докажем равенство (5) для этого случая методом математической индукции. Предположим, что верно равенство, получаемое из равенства в (5) заменой x на любую строку, длина которой меньше длины x. Докажем, что в этом случае равенство в (5) также будет верным.

Равенство в (5) можно переписать в виде

$$\mathbf{ord}(\ \mathbf{insert}(x_h, \mathbf{sort}(x_t))) = 1 \tag{6}$$

По индуктивному предположению, верно равенство

$$\mathbf{ord}(\mathbf{sort}(x_t)) = 1$$

из которого следует (6) по нижеследующей лемме.

Лемма.
Имеет место импликация

$$\mathbf{ord}(y) = 1 \quad \Rightarrow \quad \mathbf{ord}(\mathbf{insert}(a, y)) = 1 \tag{7}$$

Доказательство.
Доказываем лемму индукцией по длине y.

Если $y = \varepsilon$, то правая часть в (7) имеет вид

$$\mathbf{ord}(a\varepsilon) = 1$$

что верно по определению **ord**.

Пусть $y \neq \varepsilon$, и для каждой строки z, длина которой меньше длины y, верна импликация

$$\mathbf{ord}(z) = 1 \quad \Rightarrow \quad \mathbf{ord}(\mathbf{insert}(a, z)) = 1 \qquad (8)$$

Обозначим $c \stackrel{\text{def}}{=} y_h$, $d \stackrel{\text{def}}{=} y_t$.
(7) имеет вид

$$\mathbf{ord}(cd) = 1 \quad \Rightarrow \quad \mathbf{ord}(\mathbf{insert}(a, cd)) = 1 \qquad (9)$$

Для доказательства импликации (9) нужно доказать, что при условии $\mathbf{ord}(cd) = 1$ верны импликации

(a) $a \leq c \quad \Rightarrow \quad \mathbf{ord}(a(cd)) = 1$,

(b) $c < a \quad \Rightarrow \quad \mathbf{ord}(c\,\mathbf{insert}(a, d)) = 1$.

(a) верно потому, что из $a \leq c$ следует

$$\mathbf{ord}(a(cd)) = \mathbf{ord}(cd) = 1.$$

Докажем (b).

- $d = \varepsilon$. В этом случае правая часть в (b) имеет вид

$$\mathbf{ord}(c(a\varepsilon)) = 1 \qquad (10)$$

(10) следует из $c < a$.

- $d \neq \varepsilon$. Обозначим $p \stackrel{\text{def}}{=} d_h$, $q \stackrel{\text{def}}{=} d_t$.

В этом случае надо доказать, что при $c < a$

$$\mathbf{ord}(c\,\mathbf{insert}(a, pq)) = 1 \qquad (11)$$

1. Если $a \leq p$, то (11) имеет вид

$$\mathbf{ord}(c(a(pq))) = 1 \qquad (12)$$

Т.к. $c < a \leq p$, то (12) следует из равенств

$$\mathbf{ord}(c(a(pq))) = \mathbf{ord}(a(pq)) = \mathbf{ord}(pq) = \\ = \mathbf{ord}(c(pq)) = \mathbf{ord}(cd) = 1$$

2. Если $p < a$, то (11) имеет вид

$$\mathbf{ord}(c(p\,\mathbf{insert}(a, q))) = 1 \qquad (13)$$

Поскольку по предположению

$$\mathbf{ord}(cd) = \mathbf{ord}(c(pq)) = 1$$

то $c \leq p$, и поэтому (13) можно переписать в виде

$$\mathbf{ord}(p\,\mathbf{insert}(a, q)) = 1 \qquad (14)$$

При $p < a$

$$\mathbf{insert}(a, d) = \mathbf{insert}(a, pq) = p\,\mathbf{insert}(a, q)$$

поэтому (14) можно переписать в виде

$$\mathbf{ord}(\mathbf{insert}(a, d)) = 1 \qquad (15)$$

(15) следует по индуктивному предположению для леммы (т.е. из импликации (8), в которой $z \stackrel{\text{def}}{=} d$) из равенства

$$\mathbf{ord}(d) = 1$$

которое обосновывается цепочкой равенств

$$1 = \mathbf{ord}(cd) = \mathbf{ord}(c(pq)) = \qquad (\text{т.к. } c \leq p) \\ = \mathbf{ord}(pq) = \mathbf{ord}(d). \quad \blacksquare$$

Из рассмотренного примера мы видим, что даже для простейшей ФП из нескольких строк доказательство ее корректности представляет собой довольно нетривиальное математическое рассуждение, его сложно проверить и гораздо сложнее построить. Кроме того, в настоящее время не существует математического подхода на основе которого данное доказательство могло бы быть порождено автоматически.

Ниже мы излагаем другой метод для верификации ФП, основанный на построении диаграмм состояний функциональных программ, и иллюстрируем данный метод путем доказательства на его основе утверждения (5). Это доказательство существенно короче доказательства, приведённого в настоящем пункте. Кроме того, данное доказательство может быть порождено автоматически, что является убедительным свидетельством преимущества предложенного метода для верификации ФП.

4 Диаграммы состояний ФП

4.1 Понятия и обозначения, связанные с термами

Ниже будут использоваться следующие обозначения.

- Мы будем обозначать записью $\mathcal{E}$ множество всех термов, и записью $\mathcal{E}_0$ – множество всех термов, не содержащих функциональных переменных.
- Для каждого $e \in \mathcal{E}$ записи X_e и Φ_e обозначают множество всех переменных по данным и функциональных переменных соответственно, входящих в e.
- Если $e \in \mathcal{E}$, $x_1, \ldots, x_n$ – список различных переменных, и $e_1, \ldots, e_n$ – термы, причём $\forall\, i = 1, \ldots, n$ $type(e_i) = type(x_i)$, то запись

$$e(e_1/x_1, \ldots, e_n/x_n) \qquad (16)$$

обозначает терм, получаемый из e заменой для каждого $i \in \{1, \ldots, n\}$ каждого вхождения x_i в e на терм e_i.

- Если e и e' – термы, то для каждого терма e'', тип которого равен типу терма e', запись $e(e''/e')$ обозначает терм, получаемый из e заменой всех вхождений e' в e на терм e''.

- Если $X \subseteq \mathcal{X}$, то **означиванием** переменных из X называется соответствие ξ, сопоставляющее каждой переменной $x \in X$ некоторое значение $x^\xi \in \mathcal{D}_{type(x)}$. Множество всех означиваний переменных из X будем обозначать записью $X^\bullet$.

- Для каждого $e \in \mathcal{E}_0$, каждого $X \supseteq X_e$ и каждого $\xi \in X^\bullet$ запись e^ξ обозначает объект, называемый **значением** e на ξ и определяемый стандартным образом (т.е. если $e \in \mathcal{C}$, то e^ξ равно значению константы e, если $e \in \mathcal{X}$, то e^ξ равно значению означивания ξ на переменной e, и если $e = f(e_1, \ldots, e_n)$, то $e^\xi = f(e_1^\xi, \ldots, e_n^\xi)$).

- Мы будем считать термы $e_1, e_2 \in \mathcal{E}_0$ равными, если для каждого $\xi \in (X_{e_1} \cup X_{e_2})^\bullet$ верно равенство $e_1^\xi = e_2^\xi$, которое мы понимаем в следующем смысле: значения e_1^ξ и e_2^ξ либо оба не определены, либо оба определены и совпадают.

- Терм $e \in \mathcal{E}_0$ называется **формулой**, если все переменные из X_e имеют тип **C** и $\forall\, \xi \in X_e^\bullet \quad e^\xi \in \{0, 1\}$. Символ $\mathcal{B}$ обозначает множество всех формул. Символы $\top$ и $\bot$ обозначают формулы, принимающие на произвольном означивании значение 1 и 0 соответственно.

4.2 Понятие состояния ФП

Пусть задана ФП Σ.

Обозначим записью $\mathcal{E}_{conc}$ множество термов из $\mathcal{E}_0$, в которых нет других ФС кроме $conc$, и записью $\mathcal{E}_\Sigma$ – множество термов, каждый из которых либо является переменной, либо имеет вид $\varphi(u_1, \ldots, u_n)$, где $\varphi \in \Phi_\Sigma$ и $u_1, \ldots, u_n \in \mathcal{E}_{conc}$.

Будем называть **присваиванием** запись вида

$$u := e \tag{17}$$

где $u \in \mathcal{E}_{conc}, \;\; e \in \mathcal{E}_\Sigma, \;\; type(u) = type(e)$.

Состоянием ФП Σ называется запись s вида

$$b.u\,(\theta_1, \ldots, \theta_m) \tag{18}$$

компоненты которой определяются следующим образом:

- b – формула из $\mathcal{B}$, называемая **условием** s,
- u – терм из $\mathcal{E}_{conc}$, называемый **значением** s, и
- $\theta_1, \ldots, \theta_m$ – присваивания.

Будем использовать следующие обозначения.

- S_Σ обозначает совокупность всех состояний Σ.

- Если состояние $s \in S_\Sigma$ имеет вид (18), то мы будем обозначать через b_s, u_s, Θ_s и $type(s)$ формулу b, терм u, последовательность присваиваний (которая м.б. пустой) в (18), и тип $type(u)$ соответственно.

  Если $b_s = \top$, то формулу b в (18) мы будем опускать.

- Если $s \in S_\Sigma$, то
  - X_s обозначает множество всех переменных по данным, входящих в s,
  - переменные из X_s, входящие в левую часть какого-либо присваивания из Θ_s, называются **внутренними переменными** s, все остальные переменные из X_s называются **входными переменными** s,
  - $s^\bullet$ обозначает множество всех $\xi \in X_s^\bullet$, таких, что $b_s^\xi = 1$, и $\forall\, (u_i := e_i) \in \Theta_s$
    - если $e_i \in \mathcal{E}_{conc}$, то $u_i^\xi = e_i^\xi$, и
    - если $e_i = \varphi(v_1, \ldots, v_n)$, то

      $$u_i^\xi = f_\varphi(v_1^\xi, \ldots, v_n^\xi),$$

      где f_φ – соответствующая компонента ННТ ФП Σ.

Состояние $s \in S_\Sigma$ называется **терминальным**, если Θ_s не содержит функциональных переменных.

Пусть задана пара состояний $s_1, s_2 \in S_\Sigma$. Будем обозначать записью $s_1 \subseteq s_2$ следующее утверждение: множества входных переменных s_1 и s_2 совпадают, и

$$\forall\, \xi_1 \in s_1^\bullet\; \exists\, \xi_2 \in s_2^\bullet : u_{s_1}^{\xi_1} = u_{s_2}^{\xi_2}.$$

Наряду с состояниями ФП мы будем рассматривать также **псевдосостояния**, которые отличаются от состояний ФП лишь тем, что присваивания в них имеют вид $u := e$, где $u \in \mathcal{E}_{conc}$, $e \in \mathcal{E}$. Для каждого псевдосостояния s записи b_s, u_s и Θ_s имеют тот же смысл, что и для состояний ФП.

4.3 Раскрытия состояний ФП

Пусть заданы ФП Σ, состояние $s \in S_\Sigma$, и присваивание $\theta \in \Theta_s$, причём θ имеет вид $u := \varphi(v_1, \ldots, v_n)$, и уравнение в ФП Σ, соответствующее φ, имеет вид $\varphi(x_1, \ldots, x_n) = e_\varphi$.

Обозначим записью s^θ множество, называемое **раскрытием** состояния s относительно θ, и определяемое процедурой его построения, которая состоит из перечисляемых ниже действий.

Действие 1.

s^θ полагается равным одноэлементному множеству, которое состоит из псевдосостояния, получаемого из s заменой θ на присваивание

$$u := e_\varphi(v_1/x_1, \ldots, v_n/x_n).$$

Действие 2.

(Данное действие может выполняться несколько раз, до тех пор пока есть возможность его выполнять.)

Если все элементы множества s^θ являются состояниями из S_Σ, то выполнение данного действия завершается, иначе s^θ модифицируется следующим образом.

Выберем произвольный элемент $s' \in s^\theta$, который не является состоянием из S_Σ, и обозначим записью θ' первое из присваиваний, входящих в $\Theta_{s'}$, которое имеет вид $u := e$, где $e \notin \mathcal{E}_\Sigma$. Рассмотрим все возможные варианты вида терма e, и для каждого из этих вариантов укажем правило модификации множества s^θ, соответствующее этому варианту. Ниже словосочетание "новая переменная" означает "переменная, не имеющая вхождений в рассматриваемое псевдосостояние".

- $e \in \mathcal{C}$, в этом случае
  - если $u = e$, то удаляем θ' из s',
  - если $u \in \mathcal{X}$, то заменяем все вхождения u в s' на e, и удаляем θ' из s',
  - иначе удаляем s' из s^θ.
- $e = e'_h$, в этом случае заменяем θ' на присваивание
  - $u := e_1$, если e' имеет вид $e_1 e_2$,
  - $ux := e'$, где x – новая переменная, иначе.
- $e = e'_t$, в этом случае заменяем θ' на присваивание
  - $u := e_2$, если e' имеет вид $e_1 e_2$,
  - $xu := e'$, где x – новая переменная, иначе.
- $e = e_1 e_2$, в этом случае
  - если $u = u_1 u_2$, то заменяем θ' на пару присваиваний $u_1 := e_1$, $u_2 := e_2$,
  - если $u \in \mathcal{X}$, то заменяем все вхождения u в s' на терм xy (где x и y – новые переменные), и θ' – на пару присваиваний $x := e_1$, $y := e_2$,
  - иначе – удаляем s' из s^θ.
- $e = (e_1 = \varepsilon)$, в этом случае
  - добавляем к s^θ копию состояния s' (обозначим её s''),
  - заменяем
    - θ' в s' на пару $u := 1$, $\varepsilon := e_1$, и
    - θ' в s'' – на пару $u := 0$, $xy := e_1$, где x и y – новые переменные.
- $e = (e_1 = e_2)$, $e = (e_1 \leq e_2)$, $e = (e_1 \wedge e_2)$ и т.п., в этом случае
  - заменяем θ' на пару $x_1 := e_1$, $x_2 := e_2$, где x_1, x_2 – новые переменные, и
  - добавляем к $b_{s'}$ коньюнктивный член $u = e'$, где e' получается из e заменой e_i на x_i $(i = 1, 2)$.
- $e = (e_1 ? e_2 : e_3)$, в этом случае добавляем к s^θ копию s' (обозначим её s''), и заменяем все вхождения
  - θ' в s' на пару $1 := e_1$, $u := e_2$,
  - θ' в s'' на пару $0 := e_1$, $u := e_3$.
- $e = \varphi(e_1, \ldots, e_k)$, $\exists\, i : e_i \notin \mathcal{E}_{conc}$, в этом случае заменяем e_i в θ' на новую переменную x, и добавляем $x := e_i$ перед θ'.

Действие 3.

Для каждого $s' \in s^\theta$

- если в $\Theta_{s'}$ есть пара вида $u := x$, $v := x$, где $x \in \mathcal{X}$, и u, v имеют вид $u_1 \ldots u_n$, $v_1 \ldots v_m$ соответственно, то выполняется алгоритм, состоящий из следующих шагов:
  (в результате выполнения каждого из этих шагов вид данных присваиваний меняется, но мы будем обозначать изменившиеся присваивания теми же записями что и исходные присваивания):
  - если $n < m$, то в случае $u_n \in \mathcal{X}$ каждое вхождение переменной u_n в s' заменяем на терм $v_n \ldots v_m$, а в случае $u_n = \varepsilon$ удаляем s' из s^θ,
  - аналогично – при $m < n$,
  - $\forall\, i = 1, \ldots, n$:
    - если $u_i \in \mathcal{X}$, то заменяем все вхождения u_i в s' на v_i, а если $u_i \notin \mathcal{X}$, но $v_i \in \mathcal{X}$, то заменяем все вхождения v_i в s' на u_i,
    - если $u_i \neq v_i$, то удаляем s' из s^θ,
  - удаляем одно из рассматриваемых присваиваний,
- если $b_{s'} = \{b', x = u\}$, где $x \in \mathcal{X}$, $u \in \mathcal{X} \cup \mathcal{C}$, то $b_{s'}$ заменяется на b', и все вхождения x в s' заменяются на u,
- $b_{s'}$ упрощается путем
  - замены подтермов без переменных на равные им константы, и

– применения упрощающих преобразований, связанных булевскими тождествами и свойствами отношений равенства и линейного порядка,

- если $b_{s'} = \bot$, то s' удаляется из s^θ.

Теорема 1.
Описанная выше процедура построения множества s^θ всегда завершается. ■

Состояние $s \in S_\Sigma$ называется **противоречивым**, если оно нетерминально, и $\exists\, \theta \in \Theta_s$: либо $s^\theta = \emptyset$, либо все состояния из s^θ противоречивы.

4.4 Подстановка состояний в термы

Пусть заданы ФП Σ терм $e \in \mathcal{E}$, список $x_1, \ldots, x_n$ различных переменных из $\mathcal{X}$, и $s_1, \ldots, s_n$ – список состояний из S_Σ, причём $\forall\, i = 1, \ldots, n \quad type(s_i) = type(x_i)$. Будем обозначать записью

$$e(s_1/x_1, \ldots, s_n/x_n) \tag{19}$$

состояние $s_e \in S_\Sigma$, определяемое индукцией по структуре e:

- если $e = x_i \in \{x_1, \ldots, x_n\}$, то $s_e \stackrel{\text{def}}{=} s_i$,
- если $e \in \mathcal{X} \setminus \{x_1, \ldots, x_n\}$ или $e \in \mathcal{C}$, то $s_e \stackrel{\text{def}}{=} e\,(\,)$,
- если $e = g(e_1, \ldots, e_k)$, где $g \in \mathcal{F} \cup \Phi$, и состояния $s_{e_1}, \ldots, s_{e_k}$ вида (19), соответствующие термам $e_1, \ldots, e_k$, уже определены, то s_e определяется следующим образом:
  - внутренние переменные состояний s_{e_i} заменяются на новые переменные стандартным образом, так, чтобы все внутренние переменные этих состояний были различными, обозначим получившиеся состояния записями $b_i.u_i(\Theta_i)$ $(i = 1, \ldots, k)$,
  - s_e является результом применения действий 2 и 3 из пункта (4.3) к состоянию

$$\{b_1, \ldots, b_k\}.g(u_1, \ldots, u_k)\,(\Theta_1, \ldots, \Theta_k).$$

Терм (19) обозначается записью $e(s_1, \ldots, s_n)$, в том случае, когда список переменных $x_1, \ldots, x_n$ ясен из контекста.

4.5 Понятие диаграммы состояний ФП

Пусть Σ – ФП, и левая часть первого уравнения в Σ имеет вид $\varphi(x_1, \ldots, x_n)$.

Диаграммой состояний (ДС) ФП Σ называется граф G с выделенной вершиной n_0 (называемой **начальной вершиной**), удовлетворяющий следующим условиям.

- Каждой вершине n графа G сопоставлено некоторое состояние $s_n \in S_\Sigma$, причём s_{n_0} имеет вид

$$y\,(y := \varphi(x_1, \ldots, x_n)), \quad \text{где } y \notin \{x_1, \ldots, x_n\}.$$

- Для каждой вершины n графа G верно одно из следующих утверждений.
  1. Из n не выходит ни одного ребра, и s_n терминально.
  2. Из n выходят два ребра, и состояния s', s'', соответствующие концам этих рёбер, обладают следующим свойством: $\exists\, x \in X_{s_n}$: $type(x) = \mathbf{S}$, в Θ_{s_n} нет присваиваний вида $u := x$, и s', s'' получаются из s_n путём
     - замены всех вхождений x на константу ε и на терм yz соответственно (где y и z – переменные, не входящие в X_{s_n}), и
     - если x не входит в левую часть ни одного присваивания из Θ_{s_n}, то – добавления присваиваний $\varepsilon := x$ и $yz := x$ к $\Theta_{s'}$ и $\Theta_{s''}$ соответственно.
  3. $\exists\, \theta \in \Theta_{s_n}$: множество состояний, соответствующих концам рёбер, выходящих из n, совпадает с множеством всех непротиворечивых состояний из s_n^θ.
  4. u_{s_n} имеет вид $u_1 u_2$, и из n выходит одно ребро с меткой $tail$, конец n' которого удовлетворяет условию: $tail(s_n) \subseteq s_{n'}$.
  5. Из n выходит одно ребро, с меткой $<$, конец n' которого удовлетворяет условию:
     - $\exists\, n_1, n_2$: G содержит ребро из n_1 в n_2 с меткой $tail$, и
     - $\exists\, e \in \mathcal{E}_\Sigma$, $\exists\, x \in X_e$:

$$s_n \subseteq e(tail(s_1)/x), \quad e(s_2/x) \subseteq s_{n'}.$$

Опишем неформальный смысл понятия ДС. Каждое состояние s можно рассматривать как описание процесса вычисления значения u_s на конкретных значениях входных переменных этого состояния (путем выполнения присваиваний из Θ_s, проверки условия b_s и вычисления значения терма u_s на вычисленных значениях переменных, входящих в этот терм). Если из состояния n выходят рёбра без меток, то концы этих ребер соответствуют возможным вариантам вычисления значения u_{s_n} (путем различной детализации структуры значения некоторой переменной из X_{s_n}, или путем эквивалентного преобразования какого-либо присваивания из Θ_{s_n}). Если из состояния n выходит ребро с меткой $tail$ в состояние n', то это ребро выражает сведение задачи вычисления хвоста значения u_{s_n} к вычислению значения $u_{s_{n'}}$. Если из состояния n выходит ребро с меткой $<$ в состояние n', то это ребро выражает сведение задачи вычисления значения u_{s_n} к вычислению значения $u_{s_{n'}}$ на аргументах меньшего размера.

Будем говорить, что ФП Σ **имеет конечную ДС**, если существует ДС ФП Σ, множество вершин которой является конечным.

Теорема 2.
Пусть Σ_1 и Σ_2 имеют конечные ДС, $\Phi_{\Sigma_1} \cap \Phi_{\Sigma_2} = \emptyset$, и левые части первых уравнений в Σ_1 и Σ_2 имеют вид $\varphi_1(x_1, \ldots, x_n)$ и $\varphi_2(y_1, \ldots, y_m)$ соответственно, причём $type(\Sigma_1) = type(y_1)$. Тогда ФП Σ, первое уравнение в которой имеет вид

$$\varphi(x_1, \ldots, x_n, y_2, \ldots, y_m) = \\ = \varphi_2(\varphi_1(x_1, \ldots, x_n), y_2, \ldots, y_m)$$

а совокупность остальных уравнений равна $\Sigma_1 \cup \Sigma_2$, тоже имеет конечную ДС. ■

Мы не даем описание алгоритма построение ДС для Σ ввиду ограничений на размер статьи. Отметим лишь, что данная ДС представляет собой объединение ДС для Σ_1, ДС для Σ_2, и ДС, являющейся декартовым произведением двух предыдущих ДС.

Теорема 3.
Пусть ФП Σ имеет конечную ДС Σ, причем значения состояний, соответствующих тем терминальным вершинам этой ДС, которые достижимы из начального состояния, равны 1. Тогда f_Σ принимает значение 1 на всех своих аргументах. ■

Сформулированные выше теоремы являются теоретическим обоснованием метода применения ДС для верификации функциональных программ. Суть данного метода заключается в попытке построения конечных ДС для анализируемой функции, описываемой ФП Σ_1, и проверочной функции, описываемой ФП Σ_2. Если для этих ФП удалось построить конечные ДС, то, согласно теореме 2, для ФП, соответствующей суперпозиции этих функций, тоже можно построить конечную ДС. Если эта ДС обладает свойством, указанным в теореме 3, то суперпозиция функций, соответствующих Σ_1 и Σ_2, принимает значение 1 на всех своих аргументах. В следующем параграфе мы приводим пример использования этого метода.

Для построения ДС используется метод обоснования соотношений вида $s_1 \subseteq s_2$, который мы не излагаем здесь ввиду ограничений на размер статьи. Отметим лишь, что, этот метод использует понятие унификации термов.

При изображении ДС ФП мы будем использовать следующее соглашение: если состояние s, сопоставленное какой-либо вершине, имеет вид $b.u(\theta_1, \ldots, \theta_n)$, то данная вершина обозначается в виде овала, над которым нарисована запись $b.u$ (или u, если $b = \top$), внутри которого перечислены компоненты списка Θ_s, и слева от которого м.б. нарисован символ, являющийся идентификатором этой вершины.

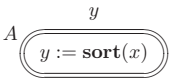
5 Пример верификации ФП методом построения ДС

В этом пункте мы иллюстрируем изложенный выше метод верификации ФП примером верификации ФП сортировки, в данном случае $\Sigma_1 = (3)$ и $\Sigma_2 = (4)$. Будем использовать следующее соглашение: если в ДС вершины n_1 и n_2 таковы, что n_2 можно получить из n_1 путем выполнения последовательности действий 2 и 3 из определения ДС, то из n_1 в n_2 можно нарисовать ребро без метки (т.е. рёбра без меток в новом понимании ДС соответствуют путям из рёбер без меток в исходном понимании ДС).

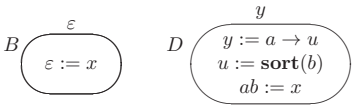
5.1 Диаграмма состояний ФП сортировки

В этом пункте мы описываем процесс построения ДС ФП (3). Термы вида $\mathbf{insert}(a, y)$ мы будем обозначать записью $a \to y$.

Начальная вершина ДС ФП (3) (обозначим эту вершину идентификатором A) имеет вид



Из этой вершины можно нарисовать два ребра без меток (соответствующих замене x на ε и ab, и раскрытию одного присваивания) с концами в вершинах



Из D тоже можно нарисовать два ребра без меток (соответствующих замене переменной u на константу ε и на терм cd): конец первого ребра имеет метку

$$y\,(y := a \to cd, cd := \mathbf{sort}(b), ab := x), \qquad (20)$$

а конец второго ребра имеет метку

$$y\,(y := a\varepsilon, \varepsilon := \mathbf{sort}(b), ab := x). \qquad (21)$$

Из вершины с меткой (20), согласно пункту 3 в определении ДС (раскрытие первого присваивания), можно нарисовать два ребра с концами

$$\begin{array}{ll} C: & \{a \le c\}.acd\,(cd := \mathbf{sort}(b), ab := x), \\ G: & \{c < a\}.cz\,(z := a \to d, cd := \mathbf{sort}(b), ab := x). \end{array}$$

Из C можно провести ребро с меткой $tail$ в начальную вершину (мы не приводим описание метода, на основании которого устанавливается существование такого ребра, отметим лишь, что в данном случае существование такого ребра усматривается непосредственно).

Из вершины с меткой (21), согласно пункту 2 в определении ДС (замена b на ε и на pq) можно получить две вершины, одна из которых терминальна и имеет вид

$$E: \quad a\varepsilon\ (a\varepsilon := x),$$

а вторая вершина будет противоречивой (что можно установить путем дополнительных раскрытий, которые мы здесь не приводим).

Из G можно нарисовать два ребра без меток (соответствующих замене b на ε и на pq), первая вершина будет противоречивой, а вторая имеет метку

$$\{c < a\}.\, cz \begin{pmatrix} z := a \to d, \\ cd := p \to w, \\ w := \mathbf{sort}(q), \\ apq := x \end{pmatrix}. \tag{22}$$

Из (22) можно нарисовать два ребра без меток (соответствующих замене w на ε и на ij):

- из конца первого из этих ребер можно нарисовать несколько ребер без меток, но среди концов этих ребер непротиворечивой будет только вершина с меткой

$$\{c < a\}.\, cz \begin{pmatrix} z := a \to \varepsilon, \\ c := p, \\ d := \varepsilon, \\ ap\varepsilon := x \end{pmatrix},$$

  из которой существует единственное ребро без метки в терминальную вершину

$$H: \quad \{c < a\}.\, ca\varepsilon\ (ac\varepsilon := x),$$

- конец второго из этих ребер имеет метку

$$\{c < a\}.\, cz \begin{pmatrix} z := a \to d, \\ cd := p \to ij, \\ ij := \mathbf{sort}(q), \\ apq := x \end{pmatrix}. \tag{23}$$

Из (23) можно нарисовать пару ребер, концы которых имеют метки

$$F: \quad \{c < a, c \le i\}.\, cz \begin{pmatrix} z := a \to ij, \\ ij := \mathbf{sort}(q), \\ acq := x \end{pmatrix},$$

$$I: \quad \{c < a, c < p\}.\, cz \begin{pmatrix} z := a \to d, \\ d := p \to j, \\ cj := \mathbf{sort}(q), \\ apq := x \end{pmatrix}.$$

Из F можно провести ребро с меткой $tail$ в начальную вершину (существование такого ребра усматривается непосредственно).

Нетрудно заметить, что пара вершин (D, I) связана с парой вершин (A, G) следующими соотношениями:

$$tail(I) = e(tail(G)/h), \quad D = e(A/h) \tag{24}$$

где $e = a \to h$. Другими словами, метки вершин вершин I, D можно получить из меток вершин G, A путем приписывания дополнительного присваивания сверху. Данный факт может быть использован для обоснования существования ребра с меткой $tail$ из G в A. Мы не излагаем детального описания метода обоснования существования такого ребра, опишем лишь схему такого обоснования. Обозначим записью $\rho(x)$ частичную функцию, обладающую следующим свойством: если ρ определена на некотором значении α переменной x, то она преобразует α в такую строку β, которая обладает свойством

$$u_{tail(G)}^{x \mapsto \alpha} = u_A^{x \mapsto \beta}.$$

Из соотношения (24) непосредственно выводится следующее свойство функции ρ:

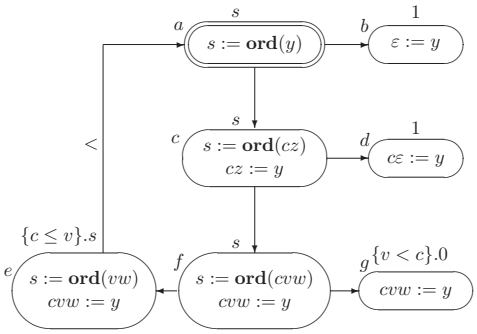
$$x \ne \varepsilon \ \Rightarrow\ \rho(x) \sqsupseteq x_h \rho(x_t) \tag{25}$$

где неравенство $\sqsupseteq$ понимается как отношение порядка на множестве частично определенных функций: если для некоторого значения x правая часть (25) определена, то левая часть на этом значении x тоже определена и значения обоих частей совпадают. Тотальность функции ρ обосновывается неравенством (25) и анализом уже построенного фрагмента ДС для (3). Отметим, что это обоснование может быть порождено автоматически. Доказательство корректности данного обоснования использует понятие унификации пар состояний, его описание занимает большой объем и мы в данном тексте его не приводим.

Построенная ДС для ФП (3) представлена на рис. 1, её можно упростить до ДС на рис. 2.

5.2 Диаграмма состояний ФП проверки упорядоченности строки

Фрагмент ДС для ФП Σ_2 (см. (4)) (состоящий из вершин, достижимых из начального состояния) имеет вид



5.3 Диаграмма состояний для суперпозиции ФП сортировки и ФП проверки упорядоченности строки

Существует алгоритм, основанный на теореме 3, который можно применить к ДС для (3) и (4), и получить ДС, изображенную на рис. 3. Данная ДС имеет две терминальные вершины, метки обоих этих вершин имеют значение 1. Согласно теореме 3, отсюда следует утверждение о том, что функция $\mathbf{ord} \circ \mathbf{sort}$ принимает значение 1 на всех своих аргументах.

В заключение отметим, что несмотря на сложность всех вышеприведенных преобразований и обоснований, все они могут быть порождены автоматически. Попытка обоснования существования ребер с метками $tail$ и $<$ производится автоматически для каждой пары вершин, возникающих в процессе построения ДС. Как видно из рассмотренного примера построения ДС для программы сортировки, процесс построения ДС заканчивается достаточно быстро.

6 Заключение

Мы предложили понятие диаграммы состояний функциональных программ (ФП) и основанный на этом понятии метод верификации ФП. Одна из проблем для дальнейших исследований, связанных с понятием ДС, имеет следующий вид: найти необходимое и достаточное условие (или как можно более сильное достаточное условие), которому должна удовлетворять ФП, чтобы эта ФП имела конечную ДС.

Список литературы

[1] R.W. Floyd: Assigning meanings to programs. In J.T. Schwartz, editor, Proceedings Symposium in Applied Mathematics, Mathematical Aspects of Computer Science, pages 19-32. AMS, 1967.

[2] C. A. R. Hoare: An axiomatic basis for computer programming. Communications of the ACM, 12(10): 576–580, 583, October 1969.

[3] R. Milner: A Calculus of Communicating Systems. Number 92 in Lecture Notes in Computer Science. Springer Verlag, 1980.

[4] R. Milner: Communicating and Mobile Systems: the π-Calculus. Cambridge University Press, 1999.

[5] Hoare, C. A. R.: Communicating sequential processes. Communications of the ACM 21 (8): 666–677, 1978.

[6] Separation Logic: A Logic for Shared Mutable Data Structures. John C. Reynolds. LICS 2002.

[7] Clarke, E.M., Grumberg, O., and Peled, D.: Model Checking. MIT Press, 1999.

[8] J.A. Bergstra, A. Ponse, and S.A. Smolka, editors: Handbook of Process Algebra. North-Holland, Amsterdam, 2001.

[9] C.A. Petri: Introduction to general net theory. In W. Brauer, editor, Proc. Advanced Course on General Net Theory, Processes and Systems, number 84 in LNCS, Springer Verlag, 1980.

[10] Z. Manna: Mathematical Theory of Computation. McGraw-Hill Series in Computer Science,1974.

[11] N. D. Jones and N. Andersen. Flow analysis of lazy higher-order functional programs. Theoretical Computer Science, 375:120–136, 2007.

[12] Ranjit Jhala, Rupak Majumdar, Andrey Rybalchenko: HMC: Verifying Functional Programs Using Abstract Interpreters, `http://arxiv.org/abs/1004.2884`

[13] N. Kobayashi and C.-H. L. Ong. A type theory equivalent to the modal mu- calculus model checking of higher-order recursion schemes. In Proceedings of LICS 2009. IEEE Computer Society, 2009.

[14] C.-H. L. Ong. On model- checking trees generated by higher order recursion schemes. In Proceedings 21st Annual IEEE Symposium on Logic in Computer Science, Seattle, pages 81–90. Computer Society Press, 2006.

[15] N. Kobayashi, N. Tabuchi, and H. Unno. Higher-order multiparameter tree transducers and recursion schemes for program verification. In POPL, pages 495–508, 2010.

Рис. 1:

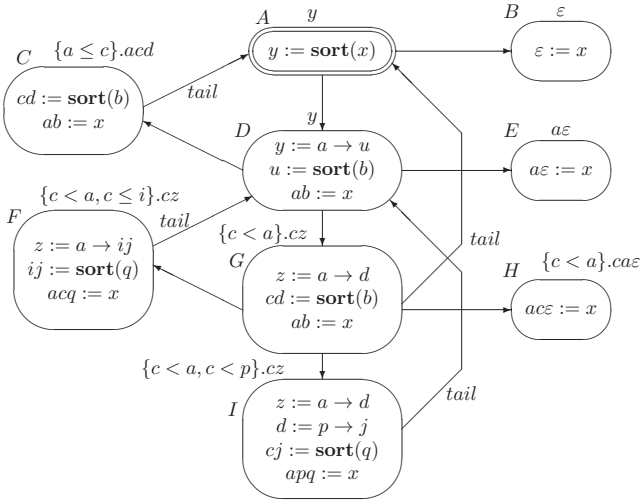


Рис. 2:

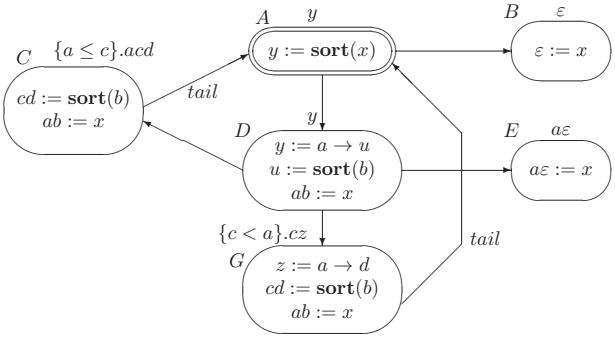
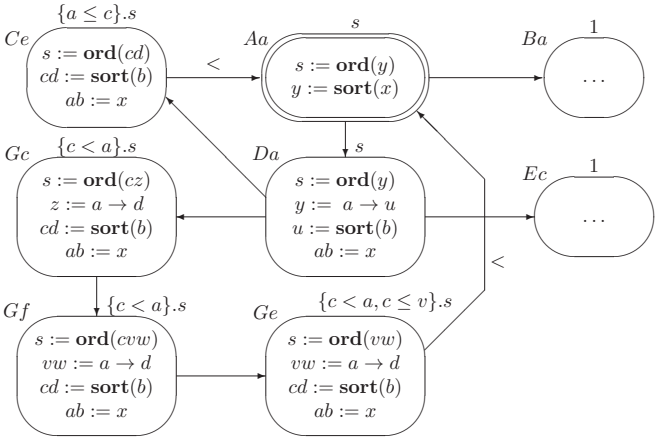


Рис. 3:



Towards a usable defect prediction tool: crossbreeding machine learning and heuristics

Vladimir Kovalenko
IntelliJ Labs / JetBrains
St. Petersburg, Russia
Vladimir.Kovalenko@jetbrains.com

Galina Alperovich
IntelliJ Labs / JetBrains
St. Petersburg, Russia
Galina.Shchekotova@jetbrains.com

ABSTRACT

A traditional supervised learning approach to defect prediction has been evaluated in numerous research papers. The biggest drawback of plain machine learning prediction models is their demand for labeled data. It sets a high restriction on applicability of defect prediction tools based on such models, and requires considerable time and resources to train the algorithms.

Building a defect prediction tool that can be used in real-world applications requires a simpler model that does not require as much human effort to train. The issue is confronted by heuristic approaches, such as the Google Bug Prediction Score, however studies show that the accuracy of heuristic models is often too low to justify their implementation.

We introduce a combined approach: by using a set of simple heuristics to identify bug fixes in version control and generating training instances based on bugfix data and language independent features, our combined algorithm performs significantly better than Google's heuristic, while still remaining fully automated and language independent. As traditional classification quality metrics are not applicable in our case due to lack of precise labeling, we use an empirical method to compare relative prediction quality based on issue tracker activity.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

defect prediction, machine learning, heuristics, bug prediction tool, VCS data mining

1. INTRODUCTION

A typical scenario of modern software development process includes a geographically distributed team working with a huge – up to hundreds of thousands VCS commits and millions of lines of code – code repository. As the development processes speed up, the techniques for efficient quality assurance and software defect management become more and more important. Even though in general it is harder to maintain a larger project, some common patterns about the way the developers work, and, in particular, about the way they introduce and fix defects in code, get clearer as the codebase grows.

Over the last few years the amount of research concerning making use of data mined from VCS repositories has grown significantly. One of the most popular research topics, which addresses the code quality problem, is development of various defect prediction models. [1]

Despite a number of successful research experiments, which show that in most cases it is possible to build a model that is capable of predicting some of the hotspots in code with quite a high precision [2], no tools based on prediction models are known to be used in industry software development environments. The reason is such tools, easy to set up and use, probably don't exist.

According to research papers, the most efficient defect prediction models are based on machine learning algorithms. The obvious drawback of these models is their demand for labeled data about past defects and fixes. We believe that a usable defect prediction tool should not require a lot of human effort to train.

A well-known example of an experiment with usage of a bug prediction tool in a real development environment is the Google case study [3]. The algorithm used in Google was based on a purely heuristic approach, so the hotspot prediction process was fully automated. In this study we have used the heuristic score described in [3] as one of comparison points for the prediction quality.

2. PROBLEM STATEMENT

There are few challenges that we came across while planning the bug prediction tool prototype development. The very first one was how to define what the "suspicious" code is. After some discussion, we have decided that suspicious, i.e. buggy, code is the code that is likely to be affected by a

bugfix in the future.

There are several techniques known for finding the suspicious code. They can be divided in two groups: techniques based on the heuristics and (more advanced) techniques based on machine learning.

The former techniques' disadvantage is the fact that they can not always be generalized for multiple projects and in general require building separate models for different projects. The advantages are simplicity of implementation and relatively low computational complexity.

The latter approaches' common drawback is the fact that they require human markup for machine learning process for each project and also complexity of implementation. Among advantages are higher precision due to ability to implicitly find potentially more sophisticated dependencies between features and dependent variable than any pre-defined heuristic is capable of making use of.

In the final implementation we combine these two approaches with the goal to get the best out of each and develop a new technique that is flexible and does not require human training.

Also, opposing our work in a way to many research papers, where the applicability scope of the algorithm becomes clear after the final quality and performance results are evaluated, we decided to start from the other end and keep the usability of the future tool in mind from the very beginning.

After discission on implementation restrictions we think that a *usable* tool for prediction of the code affected by bugfixes should satisfy the following criteria:

- It should be able to process data from any version control system repository
- It should be programming language independent (therefore the lowest granularity unit of the suspicious code will be file level)
- The tool algorithm should not use human interaction like it is often the case in the learning stage of ML-based algorithms.
- The code analysis and prediction of suspicious spots should be near realtime.
- As a result the tool should produce a list of the most suspicious files that are most likely to be affected by a bugfix in the future.

In the following sections we briefly describe various aspects of implementation of the prototype of a tool designed to meet the criteria mentioned above.

3. IMPLEMENTATION

A simple prediction model based on Weka library was implemented. Also a plugin for TeamCity continuous integration server[1], that was responsible for collecting and processing all the data the model needs, was built. The TeamCity API gave us the ability to write VCS-agnostic code with no bias to exact VCS implementation used, may it be Git, Subversion, Mercurial, Perforce or other.

[1]http://jetbrains.com/teamcity

3.1 Training data

As mentioned above, human training should not be heavily used in a tool. Also in this study we could not afford to collect enough of manually labeled data for prediction quality evaluation. Instead, an attempt to find a way to automatically detect bugfixes among VCS changes was undertaken.

To make it possible, we have manually labeled pieces of VCS history for several projects, marking if a change was a critical change for a bug fix or not. After that, for every change a set of simple per-commit and per-file metrics related to relative and absolute change size, number of files affected by commit, the presence of certain keywords like "fix" in the commit message, and a few others, was calculated. Then, using one of Weka clustering algorithms, we have composed a set of weighted rules based on these metrics, which a bugfix change is likely to comply to. After the optimization of the rules and the weights, the ruleset was capable of telling if a given set of metrics represents a bugfix change or not. Experimenting with the threshold value, we were able to settle the precision and recall of the ruleset to values close to 0.7.

Finally, a minimalistic tool that finds bugfix-candidate changes in a given code repository using the heuristic ruleset was built. *This heuristically generated data is later used for classifier learning.*

3.2 Learning process

During the learning stage, the input of the model are VCS commit data and the bugfix markup generated by the tool described above.

For every file in commit, the following metrics are calculated:

- Change frequency over a certain time period
- Number of authors
- Number of previous modifications
- Time since last change
- File age
- Google heuristic score

In addition, an indicator metric, showing if the commit is related to a bug tracker issue of type "Bug" or "Exception", is calculated.

The model keeps track of modifications for every source file in the project. If the commit is marked as a bugfix, the model looks up for the sets of metrics representing the states of every file in this commit right after their previous changes. These states are believed to be "buggy", as the files are fixed later, so these states are used as training instances for Weka classifier.

We used two different Weka classifier implementaions: NaiveBayesUpdateable and HoeffdingTree. As we found later, the exact choice of classifier implementation does not change the prediction quality metric dramatically.

During the prediction phase, we calculate the same metrics for every file in commit and classify the instances using the trained Weka classifiers.

The output of the classifiers we used is a floating-point number representing a class probability. Due to imperfection of the training data, the absolute value of classifier output does not make much sense and should not be shown to the user. Instead, the model marks as "suspicious" a given number of files with the highest output value among all files recently changed in the project. After the quality evaluation method described in the next section was established, we found that the optimal number of files in the model output was between 5 and 20. We used the value of 5 in the final quality evaluation.

4. EVALUATION OF QUALITY

After the model was implemented, we needed to evaluate the prediction quality. Even though we are lucky to have quite a few developers among our colleagues, we decided not to implement a process of manual prediction quality evaluation. The main reason for such decision is that the process of integration of a tool into the company's development environment could not be quick and seamless. Moreover, we were not sure if we could collect a significant amount of feedback in reasonable time at all. Instead we have automated the process of evaluation.

4.1 Method

A simple tool, which accesses the internal issue tracker and figures out if the file that was marked as suspicious by the model was affected by a bugfix change in the future, was developed. In such case we considered a single prediction for a file "successful", and vice-versa. By calculating the fraction of successful predictions among all of them, we get a value that can be interpreted as *bugfix probability* for files the model called buggy, which we consider the main prediction quality metric.

Though it is a synthetic metric and has very little in common with standard classification quality metrics like precision and recall, it does not seem to be a problem in our case. First reason is we don't have a precisely labeled train set to evaluate the model prediction precision and recall against. The other reason is, even if we had the labeled data, using precision and recall metrics could only let us evaluate the performance of machine learning algorithms used, and not quality of the tool in general. Also it is important to note that in the projects we used for quality evaluation the issue tracker is used very consistently, so we believe that the bugfixes retrieved from issue tracker do represent real bugfixes very well, so the future bugfix probability calculated in the way described at least does correlate with human-evaluated precision.

Using a synthetic quality metric, we cannot make any conclusions out of its absolute value. To see if predictions our model makes are informative at all, we have implemented two more simple models to compare the quality metric against. First of these two models is a *random model*, which for every revision marks as "dangerous" a given number of randomly chosen files from all files modified in previous revisions. The other model is *Google Score top model*. Its output consists of a given number of files with the highest values of Google heuristic score among all files modified previously.

Table 1: Bugfix probability for various models, project A, 12 months

Random	Google score top	Naive Bayes	Decision Tree
0.49	0.56	0.71	0.70

Table 2: Bugfix probability for various models, project B, 24 months

Random	Google score top	Naive Bayes	Decision Tree
0.50	0.63	0.86	0.87

4.2 Results

We have evaluated the model quality using the repository history and issue tracker data for two JetBrains' products. The sample quality metric values are shown in the Tables 1 and 2.

Project A is a very mature project, with the development history of over 5 years and over 50000 commits in the source repository.

Project B is smaller, yet still large. It has been developed for more than 3 years so far, and its repository contains over 20000 commits.

Both projects are written mostly in Java, though both projects' codebases contain a significant amount (over 10%) of code in other languages.

As the quality evaluation method described above is imperfect also in a way that some of predictions are made by the model that is learned with the dataset that includes data for the time period *after* a prediction was made, it was important to make it safe to assume that the repository and code effort structure does not change significantly over the period studied. It was the reason why we did not use the whole projects' change history for prediction quality measurements. Instead, we used smaller chunks no longer than two years each. The data we show for Project A in the Table 1 is for the 12 months long history chunk. The data in Table 2 is calculated using the 24 months long chunk of Project B's change and issue tracker history.

The prediction quality evaluation results shown in Tables 1 and 2 show that our approach performs significantly better than Google's heuristic, while still not requiring human training thanks to automated heuristic training data generation method. It is important to note that the Google score is one of the features used for classification, so the quality improvement actually comes from the information implicitly contained in other features' values.

5. CONCLUSION

During the study described, we built a prototype of a defect prediction tool implementing a novel approach to defect prediction with strong accent on applicability. The automated quality evaluation method shows that the tool performs better than Google's heuristic approach while possessing the same advantage of being automated.

One of the authors of this paper is currently working on integration of the algorithm described into the company's development environment to collect the feedback from developers and make the final conclusion on applicability of the approach

We would like to thank our colleagues, and personally Evgeniy Koshkin, for productive discussions and help with numerous technical issues we faced during the study.

6. REFERENCES

[1] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 31–41. IEEE, 2010.

[2] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *Software Engineering, IEEE Transactions on*, 34(4):485–496, 2008.

[3] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead. Does bug prediction support human developers? findings from a google case study. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 372–381. IEEE, 2013.

Задача Верификации Выполнения QoS Требований с Учетом Случайного Поведения Вычислительных Систем

С.Л Френкель
ИПИ ФИЦ ИУ РАН
fsergei51@gmail.com

В.Н.Захаров
ИПИ ФИЦ ИУ РАН
VZakharo@ipiran.ru

Аннотация— В докладе анализируются возможные подходы к оценке выполнения требований Quality of Service (QoS), под которыми в современной литературе понимают различные характеристики качества и производительности проектируемой вычислительной системы сетевого типа (телекоммуникация, сервис-ориентированные системы, например, веб-сервисы, и т.д.) с точки зрения пользователя. Рассматриваются возможности применения традиционных подходов к обеспечению требований к QoS, использующие целочисленное линейное программирование (ILP) для систем, для которых необходим учет стохастичности выполнения тех или иных сервисов. Показывается, что включение в QoSs вероятностных характеристик (например, вероятностей безошибочной передачи запросов, пакетов данных и т.д.) усложняет использование традиционных приближенных подходов к ILP. В предлагаемом докладе авторы, на основе анализе большого числа появившихся в последние годы публикаций по разработке сервис-ориентированных архитектур, рассматривают возможность сочетания традиционных оптимизационных методов с методами формальной верификации QoSs, в частности, Probabilistic Model Checking.

Keywords—сервис-ориентированные системы, Probabilistic Model Checking, моделирование, Quality of Service

I. Введение

В настоящее время при проектировании и эксплуатации телекоммуникационных и компьютерных сетей значительное внимание уделяется обеспечению так называемых требований *Quality of Services (***QoS***)* - обобщенных характеристик качества и производительности системы с точки зрения пользователя. К QoS-требованиям могут относиться, например, время ответа, задержки при передаче пакетов, частота возникновения ошибок (или сбоев), пропускная способность и т.д.

В частности, задача обеспечения требуемых QoS является одной из важнейших при проектировании систем, называемых в современной литературе “сервис-ориентированными архитектурами” (Service-oriented architecture (SOA)), к которым относят всевозможные программно-аппаратные системы с динамически выбираемыми сервисами.
Например, сервис-ориентированная архитектура поддерживает совместное использование *веб-сервисов* - он-лайн систем, с помощью которых возможен заказ, получение и коммуницирование с различными услугами (рисунок 1).

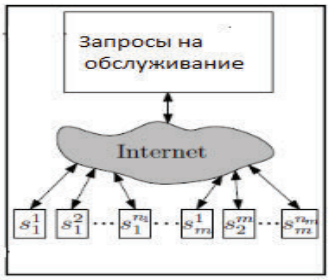


Рисунок 1. Возможный пример сервис-ориентированной системы как веб-сервиса. Здесь s^i_j - сервисы, где верхний индекс - возможные сервисы для j-того запроса (задачи).

При этом выбор сервисов и их комбинаций должен выполняться с учетом обеспечения требуемых QoS.
Пользователи могут обращаться к сервисам без знании их реализационной платформы. SOA используются во

Работа выполнена при частичной поддержке грантов РФФИ 15-07-05316, 14-07-00090,13-07-00579

многих критически важных системах, таких как системы здравоохранения, управления движением и т.д.

Проблема обеспечения QoS крайне актуальна для пользователей облачных систем, также рассматриваемых в рамках концептуальной модели SOA, для которых QoS означает производительность и надежность выполнения приложений, и при этом требуется выбирать компромисс между соответствующими QoS и доступными ресурсами (operational cost) [1].
Очевидно, что вопросы обеспечения QoS для проектируемых SOA должны решаться на всех этапах проектирования, Ввиду того, что основной концептуальной моделью SOA является сетевая структура, представляющая возможную связь сервисов при решении той или иной задачи, проблема обеспечения QoS формулируется как проблема выбора маршрута (последовательности вызовов и выполнения сервисов), обеспечивающего решение целевых функциональных задач с минимальным затратой ресурсов, при выполнении заданных требований к QoSs.

В настоящее время в литературе рассмотрено значительное число различных математических подходов к решению данной задачи, которые условно можно разделить на “синтетические”, в основном связанные с решением оптимизационных задач для автоматического построения некоторых структурных решений [2], и “аналитические”, состоящие в оценке возможных значений QoS в реализованной системе, с последующей коррекцией полученных проектных решений (“синтез через анализ”) [3,4].

Для оптимизации основном используются методы целочисленного линейного программирования [2], минимизирующие некоторые обобщенные ресурсные затраты (“стоимость”) выполнения требуемых сервисов (например, стоимость оборудования и обслуживания) когда ограничения, соответствующие QoSs (например, задержки в передаче запросов на обслуживание, характеристики надежности), задаются линейными функциями параметров, хотя известны попытки использования и нелинейных методов (в случаях, когда критерии являются нелинейными (квадратичными, в частности) функциями параметров сети и запросов) [5]. При этом решением является, как правило, оптимальная (в указанном смысле) маршрутизация или размещение сервисов [2,5]. Вторая группа методов (“анализ”) использует такие традиционные подходы как имитационное моделирование, тестирование, модели теории очередей (при учете случайного характера обработки сервисов как случайных, например, ввиду возможных потерь сообщений при обмене сетевыми сервисами) [6]. Кроме того, информация о предполагаемом функционировании проектируемой системы на ранних этапах проектирования, как правило, точно неизвестна, поэтому задачи разумно формулировать относительно вероятностей удовлетворения соответствующих требований.
Ввиду того, что сервис-ориентированные системы (особенно для указанных выше критических приложений) должны быть *гарантированно* надежными, безопасными, доступными, рассматриваются способы формальной верификации, в основном Probabilistic Model Checking (PMC), ввиду указанного выше случайного поведения сервис-ориентированных систем [7].

Очевидно, что предпочтительным для разработчика является автоматическое построение оптимального решения, гарантирующего выполнения QoSs. Однако в общем случае такие математические методы как целочисленное программирование, лежащие в основе указанной оптимизации, являются переборными, и поэтому для многих приложений приходится использовать различные приближенные методы, решения которых нуждаются в анализе и уточнении. Для решения этой задачи требуется использовать те или иные методы программной поддержки, например, для этого можно использовать методы второй (“аналитической”) группы, в частности PMC. При этом требования к QoS формулируются как вероятности выполнения формальных свойств, описанных на языке Probabilistic Linear Temporal Logic (PLTL), или Probabilistic Computational Tree Logic (PCTL) [7].
В современной литературе, однако, указанные синтетические и аналитические методы фактически рассматриваются независимо друг от друга [6], и специфика их совместного использования, в частности, для уточнений/улучшений их решений, практически не изучена .

В предлагаемом докладе авторы, основываясь на анализе большого числа появившихся в последние годы публикаций по разработке SOA, рассматривают возможность сочетания традиционных оптимизационных методов с методами формальной верификации QoSs. При этом рассматривается специфика использования оптимизационных методов, методов теории очередей, Probabilistic Model Checking. На основании данного анализа предлагается методология комбинирования двух групп методов обеспечения QoS-требований.

II. Проблема обеспечения QoS в контексте общей проблемы проектирования сервис-ориентированных систем: возможные решения

В настоящее время в литературе рассматриваются как детерминистские, так и вероятностные подходы к анализу QoS [2,6].

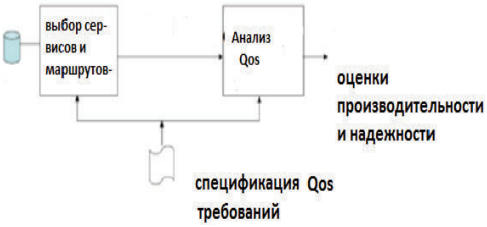


Рисунок 2. Общая схема верификации и обеспечения требований к QoS

На сравнительно ранних этапах разработки систем с сервис-ориентированной архитектурой разработчик системы должен быть уверен, что проект системы (на данном этапе проектирования), например, сетевой структуры обмена сервисами, отвечает требованиям к QoS (рисунок 2), что может, например, давать уверенность, что определенный набор сервисов, вызываемых в соответствующей рабочей среде выполнения интересующих приложений, обеспечит заданные требования заказчика, например, производительность и надежность.
Для этого разработчик SW ("software architect") должен описать функциональные и временные требования для указанных приложений, например, в терминах следующих переменных:

r_j - надежность выполнения j-ого сервиса s_j;

ServiceExecutionTime(j) - математическое ожидание времени вызова s_j;

c_j - стоимость (затраты - например, потребление энергии, запрашиваемая память и т.д.) выполнения s_i.

Начальное описание (формализация) системы, как правило, выполняется в терминах того или иного графа, задающего возможные связи между сервисам, или программами/устройствам, реализующими данные сервисы, как правило, с двунаправленными дугами (поскольку направление взаимодействия может быть также задачей выбора), с явно заданными входными и выходными вершинами, и аннотацией дуг существенными для данного проекта параметрами выполнения сервисов, например, стоимостью (затратами тех или иных ресурсов), временной задержкой, вероятностью успешного выполнения и т.д.
Например, в графе на рисунке 3 из [2] входной является вершина 1, выходом - вершина 8. Цифры (c,d) на дугах - стоимости и задержки соответственно. Требование к QoS будет задаваться относительно суммарной задержки, например, D < 15. Путь, который удовлетворяет указанным ограничениям, называется *допустимым* (feasible) [8].

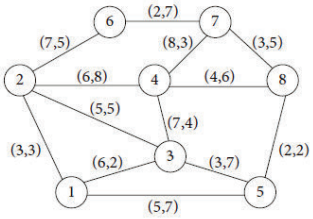


Рисунок 3. Пример графа сети сервисов

Соответственно, проблему обеспечения QoSs проектируемой системы решают в рамках решения задачи выбора состава используемых сервисов и последовательности их выполнения, обеспечивающих требуемые QoSs при минимальной/допустимой стоимости выполнения.

В настоящее время, с точки зрения проблемы *синтеза* структуры, отвечающей заданным требованиям к QoS, данная задача решается или как задача поиска кратчайшего пути, обеспечивающего QoS требования с минимальными затратами ресурсов (constrained shortest path (CSP)), или как задача поиска т.н. k-Disjoint path, т.е. k путей в графе, ни один из которых не имеет общих вершин (или дуг) с другими.
Обе эти задачи решаются либо с использованием целочисленного линейного программирования (Integer Linear Programming (ILP) [8], часто относительно булевых переменных (говоря в этом случае о *0-1 ILP-задаче*)), либо того или иного алгоритма поиска кратчайшего пути (например, алгоритмов Дейкстры), причем обе эти задачи являются NP-полными.
В настоящее время решение QoS проблемы как поиска k-Disjoint путей является наиболее распространенной, поскольку целый ряд требований, например, надежности, быстродействия, можно обеспечить только при рассмотрении как минимум двух путей.
Обеспечение Disjoint QoS routing [2] представляет собой эффективную стратегию достижения устойчивости, нагрузочной способности (сбалансированности), приемлемого уровня возможных помех (congestion), пропускной способности. Также маршрутизация по нескольким путям рассматривается как эффективный способ обеспечения устойчивости сети к ошибкам соединений (самовосстановление за счет реконфигурации маршрутов).

В [9] показано, что даже задача нахождения раздельных оптимальных (в терминах задачи линейного программирования) двух возможных путей, обеспечивающих требуемые для каждого найденного маршрута QoSs, является NP-полной.

Задача 0-1 ILP нахождения k оптимальных раздельных путей может быть представлена как [2]:

$$Min_x \Sigma_{i=1,k}(f(c^i,x^i))$$
$$Ax^i \le b, \qquad (1)$$

где k- заданное число искомых раздельных (Disjoint) путей (маршрутов) в графе, f - линейная функция компонентов вектора стоимостей c^i выполнения операций по i-му пути, образованному дугам графа сети, x^i –вектор, компоненты которого x^i_j представляют собой булевы переменные, показывающие, выбрана ли j-я вершина в графе (при соответствующей нумерации вершин; при этом каждой j-й вершине приписана стоимость, задаваемая j-м компонентом c^i_j вектора c^i), A-матрица, числовых значений на дугах графа (значений задержек, в рассмотренном примере), размерность которой согласована с размером вектора x^i, b - ограничения, представляющие собой QoS-требования (b = D = 15 – ограничение на задержку для сети на рис.3) Соответственно, решением должен быть вектор нулей и единиц, задающих оптимальные пути, удовлетворяющие ограничениям.

Ввиду того, что методы целочисленного линейного программирования не являются целью нашей статьи, мы не будем подробно рассматривать построение указанных векторов и матриц, что легко найти в публикациях [2,9]. В сфере нашего внимания в настоящей статье находятся только вопросы технической реализации и вычислительная сложность применяемых в задачах QoS маршрутизации методов, и трудности достижения приемлемой сложности .

Полиномиальное время решения стараются обеспечить за счет использования “облегченного” (relaxing) требования к решению задачи ILP [4]. Это означает, что вместо задачи ILP решают задачу линейного программирования (LP) без требований к целочисленности решения. LP решается за полиномиальное время, а процедура округления нецелочисленного оптимального решения LP-задачи дает возможность получать приближенное оптимальное целочисленное решение. Однако не всегда получаемое решение является *допустимым (feasible)*[4], где под допустимым понимают путь, сумма значений QoS (задержки, в рассматриваемом примере) по которому не превышает заданной величины (D = 15 для примера рис.3). Причина состоит в том, что оптимальные решения, получаемые при решении LP (“облегченной” относительно ILP) могут быть вещественными (а не целыми, тем более - булевыми), и их округление до ближайшего целого значения может не отвечать указанным ограничениям (подробные примеры этого явления можно найти в [8, 10]).

Рассмотрим на примере графа на рисунке 4 возможные решения 0-1 ILP-задачи для графа рис.3 [2].

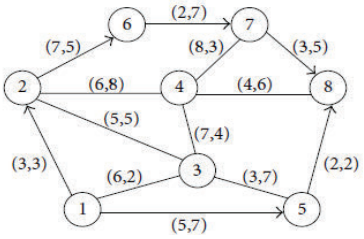


Рисунок 4. Оптимальные пути для сети рис. 3

Рисунок 4 показывает два разъединенных пути, получаемых как оптимальные решения в результате решения “облегченной” (relaxed) LP- задачи, из которых (1-5-8) имеет задержку 9 < 15, и, следовательно, является допустимым, другой (соответствующий пути k=2 в (1)), 1-2-6-7-8 имеет задержку 20, и следовательно, не является допустимым (infeasible).

Таким образом, использование приближенных методов оптимизации может приводить к необходимости дополнительного анализа полученных решений.

В случае, когда нельзя игнорировать случайность поведения системы, к условиям оптимизации (1) должны добавляться также условия на те или иные вероятностные характеристики, в частности, надежность, которая определяется как вероятность того, что пакет данных в сети или запрос в задаче оптимизации веб-сервисов достигнет получателя без потерь, и является одной из важнейших QoS. Прежде всего, в этом случае проблема оптимизации перестает быть линейной, поскольку надежности отдельных путей мультиплицируются ($r = 1-\Pi(1-x^i r_i)$, где r_i - надежности по каждому i-му пути среди k путей (1)). Но и при решении данной нелинейной задачи, как показано в [5], могут существовать конфликты между различными требованиями, например, к обеспечению допустимой задержки передачи запроса и надежностью (вероятностью) успешного выполнения запрошенного сервиса. Если требования к задержке таковы, что не существует ни одного простого пути, обеспечивающего требования данного QoS, то и использование одновременно нескольких путей не обязательно приводят к его выполнению, в то время как с точки зрения требований к надежности, увеличение числа путей (число k в (1)) решает задачу. Это означает, что все пути, получаемые в результате решения системы неравенств (1) (дополненными очевидными ограничениями на требования к надежности), могут быть неприемлемы (*infeasible*) с точки зрения указанного мульти-критерия (например, суммарной задержки выполнения запросов, и

вероятности их выполнения в течение некоторого времени, понимаемого в литературе как надежность данной SOA системы).

Если для некоторой SOA направления взаимодействия сервисов или передачи потоков данных определены (в том числе и в результате нахождения направления некоторых путей при решении оптимизационной задачи (1)), проектируемую систему можно представить в виде “графов задач” или различных “диаграмм активностей” [11], вершины которых интерпретируют как состояния системы, соответствующие выполняемому действию, порождаемому запросами (например, “поиск товара”, “выбор товара”, ”оплата“). Каждая вершина соответствует одному состоянию, а ребра между вершинами соответствуют переходам между состояниями. Разветвлениям путей в вершинах графа Рис.3 соответствуют ветвления в решающей вершине (соответствующей вершине “ромб”, рисунок 5 [11]).

Например, на рис.5 представлена архитектура системы для покупки книг он-лайн [11], где используются следующие сервисы, необходимые для решения задачи заказа и покупки:

- поиск книги (сервис LocateBook)

- заведение аккаунта (сервис CreateAcct)

- ввод данных карты (Signin)

- загрузка данных покупателя (LoadUserProfile)

- указание метода оплаты (SpecifyPaymentMethod)
- указание деталей доставки книги (SpecifyDeliverDetails)
- завершение покупки (FinalyzeBuy)

Ромб означает процесс проверки наличия книги, а прямоугольник PutinCard задает две возможности задания информации о кредитной карте.

Дуги переходов также могут быть аннотированы различными параметрами (временем выполнения, вероятностями выполнения, затратами энергии на переход и т.д.). Пример анализа данного графа приведен в разделе IV.

Для таких структур (моделей проектируемых систем) задача обеспечения оптимальных QoS также может быть сформулирована как задача оптимизации ILP [17], а именно, минимизации суммарной стоимости выполнения выбранных сервисов при требуемых условиях (ограничениях) на QoS.

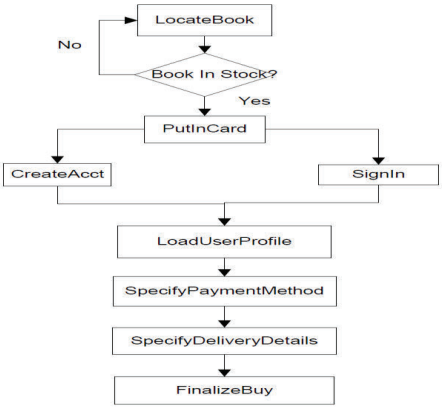


Рисунок 5. Граф задачи (диаграмма) “заказ покупки книги”

Ввиду отмеченных трудностей решения оптимизационных задач, рассмотрим возможность обеспечения требований к QoS в системе со стохастическим поведением на основе оценки QoSs для текущего варианта проектируемой системы, с учетом необходимости обеспечения разработчика информацией о возможных причинах невыполнения тех или иных требований для соответствующей модификации проекта.
В последние годы активно изучаются подходы к обеспечению QoS, основанные на формальной верификации выполнения соответствующих функциональных и временных свойств [6,11,12], и соответственно, внесении изменений в проект систем, в случае их невыполнения (“синтез через анализ”), и, соответственно, вместо получения точного решения задачи ILP-оптимизации, вычисляют вероятности достижения требуемых значений QoS и стараются обеспечить приемлемые значения этих вероятностей.
При этом, как будет показано в разделе III, по графическим структурам, аналогичным рисункам 3, 5 и другим видам диаграмм сервисов, для которых, как отмечалось, задачи обеспечения QoS также решают как задачи ILP, можно строить цепи Маркова, моделирующие функционирование сервис-ориентированных архитектур. Это обстоятельство, как будет продемонстрировано ниже, и позволяет совместно использовать два этих подхода к обеспечению QoS.

III. Оценка QoS требований с использованием Probabilistic Model Checking

Полезность использования формальной верификации требований QoS состоит в возможности исключения неоднозначности и противоречивости в задании

указанных требований, например, отмеченных выше проблем с одновременным обеспечением быстродействия и надежности.

Для решения нашей задачи обеспечения требований к QoS с учетом случайного поведения тех или иных компонентов системы использование формальных методов может состоять в том, чтобы на основе данной формальной спецификации требований получить оценку достижимого уровня QoS с заданной вероятностью. Использование формальной спецификации в проектировании предполагает наличие соответствующих программных средств поддержки языка спецификации (компиляции, интерпретации, моделирования).

Наиболее распространенными и естественными средствами для этого является вероятностный (Probabilistic) Model Checking (PMC). Применение Probabilistic Model Checking (PMC) основано на модели проектируемой системы как системы помеченных переходов, с заданными вероятностями этих переходов [7].

Специфицируемые QoS свойства задаются формулами временной вероятностной логики *Probabilistic CTL* (PCTL), которые являются расширением формального языка временной логики CTL (Computational Tree Logic) [13], описывающей отношения следования ("раньше-позже") на дереве возможных переходов исследуемой системы, либо Probabilistic LTL-логики, аналогично являющейся расширением *линейной временной логики,* задающей отношения следования переходов на линейном пути (последовательности) посредством квантификации CTL/LTL-формул значениями вероятностей их выполнения.

Специфицируемые PCTL/PLTL свойства рассматриваются как свойства переходов некоторой Цепи Маркова (конкретизирующей указанную выше систему помеченных переходов).

Например, оператор *F* (future), используемый в выражениях CTL/LTL в виде Fϕ, означающий (в обеих логиках), что в конце концов на одном из переходов системы формула ϕ, задающая некоторое условие для состояний системы (например, что некоторое состояние s=1) будет истинно, в вероятностной логике трансформируется в оператор $F^{\leq k}_{\geq p}\phi$, означающий, что условие, заданное формулой выполнится (формула станет истинной) не более, чем через k переходов соответствующей цепи Маркова с вероятностью не меньшей p. Аналогично, оператор временной логики G, означающий истинность формулы в любой момент времени в вероятностной логике представляется как $G^{\leq k}_{\geq kp}\phi$, как вероятность того, что формула ϕ будет истинна в течении k шагов с вероятностью не ниже p.

Поскольку мы говорим о вероятностных характеристках качества сервисов (надежности, в самом общем случае), мы можем специфицировать указанные свойства в терминах PLTL как вероятности того, что начав в заданном начальном состоянии (например, соответствующего запросу данного сервиса), сервис будет корректно выполнен и приведет к желаемому пользователем результату. Например, можно специфицировать свойство "вероятность выполнения запроса в течении 15 единиц времени (QoS "время выполнения") с вероятностью 0.95", и проверить его выполнение в данной структуре с помощью программы Probabilistic Model Checking, например, PRISM [7]. Существенно, что различные программные инструменты, поддерживающие Probabilistic Model Checking, обеспечивают получение информации, позволяющей оценить причины невыполнения тех или иных свойств при данной конфигурации сети посредством анализа формируемого программой контрпримера (о проблемах, возникающих при использовании контпримеров, построенных при реализации PMC, подробно в [14,15,16]).

Probabilistic Model Checking можно определить для системы, моделируемой цепями Маркова, в частности, цепью Маркова (ЦМ) с дискретным временем (Discrete Time Markov Chain (DTMC).

DTMC определяют как тройку $D = (S,P,L)$, где S – конечное множество состояний, $P : S\times S \rightarrow [0,1]$ – матрица вероятностей переходов, $\Sigma_{s'\in S} P(s,s') = 1$ для всех $s\in S$, и $L : S \rightarrow 2^{AP}$ – разметка системы переходов, представляющая собой множество логических атомарных предложений (*atomic propositions* (AP), т.е. неделимых на более мелкие предложения), которые каждому состоянию *s* сопоставляют те или иные атомарные предложения, верные в данном состоянии s. AP определяются над множеством переменных (например, *x, s)*, числовых (0, 1, 2,…) или символьных (например, "true" ,"false") констант, функций (скажем, *max*(), min()) и тех или иных предикатов (например, $x = 2$, $x <= 0$), которые необходимы для описаний свойств проектируемой системы. 2^{AP}, как обычно, означает множество подмножеств множества *AP*.

Пусть, например, некоторая архитектура моделируется дискретной цепью Маркова на рисунке 6, для которой качество обслуживания может характеризоваться завершением обслуживания сервиса s4 с формированием признака подтверждения выполнения задачи за требуемое время d с заданной надежностью p, например, 0.99 (рабочий цикл начинается с вершины s0, проверяющей наличие признака выполнения ack, и запускающей сервис s1 в случае его отсутствия). Пусть все времена выполнения переходов t целочисленные, одинаковы и фиксированы, а все сервисы надежны кроме s2, обращение к которому может оказаться в данный момент невыполнимым (например, из-за перегрузки данного узла по другим каналам, не рассматриваемым в данном представлении), и в этом случае запрос от сервиса s1 должен быть повторен. Пусть эта ситуация возникает с вероятностью 0.1.

Тогда выполнение указанных QoS-требований (время выполнения и вероятность p) можно проверить через верификацию вероятностного свойства, задаваемого формулой $F^{\leq d/t}_{\geq 0.99}\phi$, где логическую формулу ϕ можно задать как: s4=true.

Если использовать для верификации широко распространенную систему PRISM, то получим, что с вероятностью 0.99 сервис 4 может быть выполнен за 6 переходов, и если 6*t<d, то оба Qos-требования обеспечены.

Заметим, что это решение получается как вектор вероятностей состояния, в которое цепь Маркова рис.6 переходит за 6 шагов при векторе

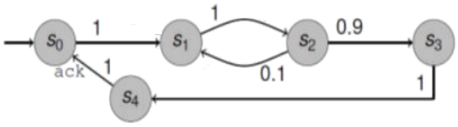


Рисунок 6. Цепь Маркова для оценки времени выполнения и надежности выполнения сервисов.

вероятностей начальных состояний P_0=(1,0,0,0,0), вычисляемый как $P_6 = P_0A^6$

$$A = \begin{array}{c} \\ s0 \\ s1 \\ s2 \\ s3 \\ s4 \end{array} \begin{array}{c} \begin{array}{ccccc} s0 & s1 & s2 & s3 & s4 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0.1 & 0 & 0.9 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

Матрица A^6:

$$\begin{pmatrix} 0 & 0 & 0.0100 & 0 & 0.9900 \\ 0 & 0.0010 & 0 & 0.0090 & 0.9900 \\ 0 & 0 & 0.0010 & 0 & 0.9990 \\ 0 & 0 & 0 & 0 & 1.0000 \\ 0 & 0 & 0 & 0 & 1.0000 \end{pmatrix}$$

Соответственно, вектор вероятностей состояний, в которую система попадет за 6 шагов:

$$P_6 = (0,0,0.01,0,0.99)$$

т.е. выполнение сервиса s4 за 6 шагов произойдет с вероятностью 0.99.

Таким образом, для того, чтобы использовать PMC, должны быть средства построения цепи Маркова исследуемой системы по тому или иному исходному описанию проекта разрабатываемой сервис-ориентированной системы. Данные средства должны строить программную модель, оценивающую вероятность выполнения специфицированных на языках PCTL/PLTL формальных свойств по входному описанию как структуры системы, так и требований к ее функционированию.

IV. Общая схема проектирования с учетом QoS-требований

Как отмечалось во Введении, самый предпочтительный путь проектирования состоит в описании задачи как оптимизационной, и нахождении конфигурации сети, обеспечивающей выполнение указанных требований, поскольку это позволило бы исключить многочисленные внесения изменений в проект. Этому, как отмечалось, могут помешать, во-первых, возможное стохастическое поведение системы, что может привести к указанной выше необходимости усложнения задачи линейного программирования, и, во-вторых, отсутствие приемлемых решений оптимизационной задачи за приемлемое время, когда решение для какого-то из QoS может просто не существовать, например, как это было продемонстрировано в Разделе II. При наличии таких проблем разработчик может использовать рассмотренные выше вероятностные модели и формальные методы анализа для обеспечения QoS, выполняя либо вероятностную верификацию конфигурации сети, полученной в результате решения оптимизационной задачи (возможно, без учета требований к надежности), либо выполняя формальную вероятностную верификацию некоторого принятого варианта сети, с последующей доработкой в случае невыполнения требований к QoS. При этом, как это видно из рассмотренного выше примера, оказывается возможным проверка как временных, так и надежностных свойств.

На рисунке 7 схематично показаны возможные действия разработчика SOA по обеспечению требования к QoS. Здесь (P)MC означает, что может быть использован как Probabilistic, так и "детерминистский" Model Checking (для верификации функциональной корректности выполнения сервисов). Заметим, что проверка функциональной корректности может быть необходимой даже если удалось решить задачу (1) и получить допустимые пути (при этом может быть применен предложенный в [16] авторами подход к комбинации вероятностной и детерминированной верификации).

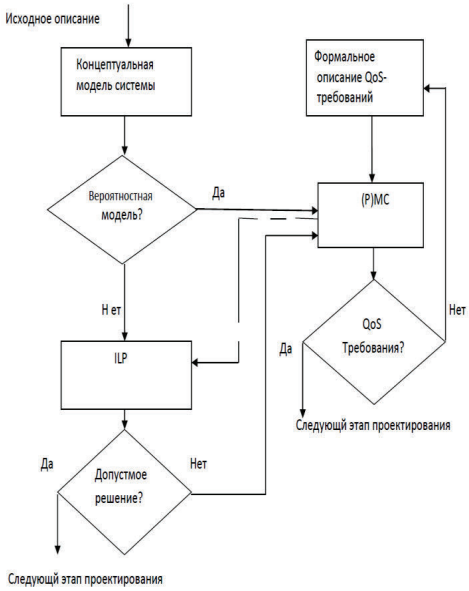


Рисунок 7. Обобщенная схемы обеспечения требований QoS на верхних уровнях моделирования

Разрывная линия означает, что разработчик может проверить функциональную и временную корректность текущего варианта проектируемой системы перед решением задачи оптимизации

Схема формальной верификации QoS показана на рисунке 8.
Здесь:

GD (graphical description) - исходное графическое описание системы,

PM (probabilistic model) - вероятностная модель системы, ориентированная на используемую программу верификацию (VT), например, PRISM,

QoS R - требования к QoS, трансформируемые (автоматически или вручную) в формулы PCTL (или PLTL),

NDS – следующий этап проектирования.

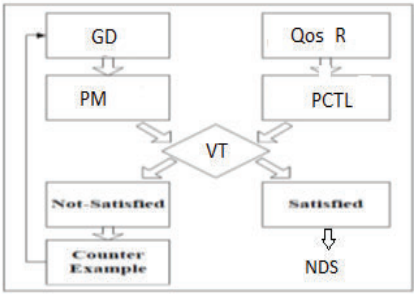


Рисунок 8. Схема верификации выполнения требований к QoS

Если требования (представленные формулами PCTL, как в примере рисунка 6) выполняются, то переходим к следующему этапу проектирования, если же нет - программа верификации формирует контр-пример, и разработчик формирует другое решение (т.е. другой набор используемых сервисов, марщрутов вызова и передачи запросов), также анализируемое на выполнение требований QoS.

В случаях, когда на проектируемую систему не накладываются требования к надежности, функциональная верификация проекта, т.е. проверка выполнения функциональной корректности, обычно выполняется с использованием Model Checking, что, очевидно, предполагает наличие соответствующих программных средств. Однако часто эту задачу можно решить также и с использованием только программного средства PMC.
Так, для SOA на рисунке 5, предполагая, что выбор каждой ветви процесса оформления покупки происходит

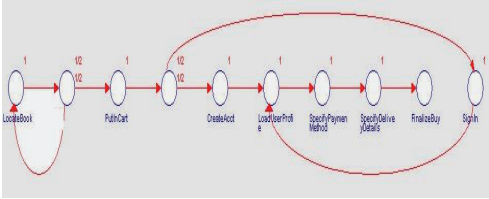


Рисунок 8. Цепь Маркова для графа задачи заказа и покупки книги (рисунок 5).

равновероятно, т.е с вероятностью 1/2 в данном примере, можно построить цепь Маркова (DTMC), представленную на рис.8 [11].
Тогда проверка свойства “живучести” (*liveness)* [13], обязательно рассматриваемого при формальной верификации посредством Model Checking, т.е. *“если*

что-то желательное должно произойти, то оно обязательно произойдет, независимо от того, что происходит в настоящий момент", может быть специфицировано и проверено в терминах вероятностной временной логики как единичная вероятность, того, что состояние FinalizeBuy будет в конце концов достигнуто. Аналогично свойство "если не выбрано ни одной книги, то не достигается и состояние PutinCard" (не запрашивается сервис определения порядка считывания с карты) можно определить выражением $P_{<=0}$[FPutInCard], т.е. условием нулевой вероятности того, что данная ситуация возникнет [11].

Таким образом, по сути мы имеем инструментарий, позволяющий строить программные модели, используемые как для Model Checking, так и Probabilistic Model Checking.

Взаимодействие ILP и PMC методов при обеспечении QoS возможно благодаря тому, что в обоих подходах, QoS выражаются в терминах путей между двумя узлами, которые интерпретируются или как узел обработки информации (в ILP), или состояния DTMC.

Заметим, что говоря о вероятностной оценке QoS мы ничего не говорили о проблеме обеспечения требуемой стоимости, о которой говорили в описании ILP-формулировки задачи. Но дело в том, что при решении оптимизационной задачи мы получаем минимальную стоимость, которая, однако, даже при допустимом решении для QoS, может оказаться выше допустимых значений, что потребует внесение изменений в проект.
Поэтому при рассматриваемом итерационном проектировании, основываясь на имеющейся марковской модели поведения проектируемой SOA, мы также можем оценивать стоимость для каждого полученного варианта обеспечения требуемых QoS (например, как в[18]), и сравнить с допустимым значением, с тем, чтобы принять решении о завершении процесса проектирования (на уровне SOA).

V. Заключение

Обеспечение требований к QoS является актуальной задачей проектирования вычислительных систем, как коммерческого (e-Buy) назначения, так и телекоммуникационных сетей (обеспечение передач потоков той или иной информации), и сетей управления и обмена сервисами.
Наиболее естественным способом решения данной проблемы является оптимизация сети путем решения задачи целочисленного программирования. Однако целый ряд факторов, таких, как вычислительная сложность, возможность получения не реализуемых решений, явно стохастическая природа проектируемой системы, может помешать реализации этого метода. Поэтому актуальной является анализ дополнения данного метода другими широко используемыми средствами проектирования, такими как Model Checking, Probabilistic Model Checking. Методологию совместного использования этих двух подходов в одном цикле проектирования и анализируется в предлагаемой статье.
Как известно, одним из важных аспектов снижения трудоемкости проектирования является универсализация использования данных и вычислений, т.е возможность использования одних и тех же данных и результатов вычисления для решения разных подзадач проектирования. Например, для решения задачи обеспечения QoS может потребоваться решение задачи оптимизации (ILP), Model Checking, Probabilistic Model Checking.
Возможность использования графического описания проектируемой системы, являющегося входом для системы решения 0-1 ILP проблемы (раздел II), а также одного и того инструмента для Model Checking и Probabilistic Model Checking как при формальном анализе условий выполнения QoS требований, так и верификации функциональной корректности, очевидно, снижает общую трудоемкость проектирования.

Литература

1. Danilo Ardagna1, Giuliano Casale, Michele Ciavotta, Juan F Perez and Weikun Wang, "Quality-of-service in cloud computing: modeling techniques and their applications", Journal of Internet Services and Applications, pp. 5:11, Dec. 2014,
2. Zhanke Yu, FengMa, Jingxia Liu, Bingxin Hu, and Zhaodong Zhang, "An Efficient approximate algorithm for disjoint QoS routing", Mathematical Problems in Engineering, Volume 2013, Article ID 489149, 9 pages http://dx.doi.org/10.1155/2013/489149.
3. Q. Sun, S. Wang, H. Zou, F. Yang, *QSSA:* "A QoS-aware service selection approach, International Journal of Web and Grid Services", pp.147 - 169, May 2011.
4. D. Ardagna and B. Pernici, Global and local QoS guarantee in web service selection. In Proc. of Business Process Management Workshops, pp. 32- 46, 2005.
5.Xiaoxia Huang · Yuguang Fang, "Multiconstrained QoS multipath routing in wireless sensor", Networks Wireless Networks, no. 4, vol., pp. 14:465–478, August 2008.
6. Giuseppe Cicotti, Luigi Coppolino, Salvatore D'Antonio, Luigi Romano, "Big Data analytics for QoS prediction through Probabilistic Model Checking" , arXiv 1405.0327v1, 2 May 2014.
7. M. Kwiatkowska, "Quantitative verification: Models, techniques and tools", ESEC/FSE'07, ACM Press, pp. 449–458, 2007.
8. L. A. Wolsey, Integer Programming, John Wiley & Sons, NewYork, NY, USA, 1998.
9. M. J. Blesa and C. Blum, "Finding edge-disjoint paths in networks: an ant colony optimization algorithm," Journal of

Mathematical Modelling and Algorithms, vol. 6, no. 3, pp. 361–391, 2007.
10. Javier Larrosa, Albert Oliveras, Enric Rodrıguez-Carbonell, “Mixed Integer Linear Programming”, *https://www.cs.upc.edu/~erodri/webpage/**rpar**/s6.pdf*
11. G. Oghabi, “Verifying web services using Probabilistic Model Checking”, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.453.5234&rep=rep1&type=pdf.
12.Radu Calinescu, Lars Grunske, Marta Kwiatkowska, Raffaela Mirandola, and Giordano Tamburrelli, “Dynamic QoS management and optimization in service-based systems”, IEEE Trans. On Software Eng., vol. 37, no 3, pp.387-409, May/June 2011.
13. E.M. Clarke, O. Grumberg, and D. A. Peled, Model Checking, MIT Press, 2000.
14. H. Aljazzar and S. Leue, “Extended directed search for probabilistic timed reachability”, Tech. Report soft-06-03, University of Konstanz, 2003.
15. M. Schmalz, H. Völzer, and D. Varacca, “Counterexamples in probabilistic LTL model checking for Markov chains”, / Technical Report 627, ETH Zürich, Switzerland, 2009, researcher.watson.ibm.com/researcher/files/zurich-hvo/SVV09.pdf.
16. Френкель С. Л., Захаров В. Н., Ушаков В. Г. “Вероятностная верификация при проектировании вычислительных систем”, Инструменты и методы анализа программ (TMPA-2014): Материалы международной научно-практической конференции, стр.148-155, Кострома, 14–15 ноября 2014.
17.Ulrich Lampe, Dieter Schuller, Julian Eckert, and Ralf Steinmetz, Optimizing service selection for probabilistic QoS attributes, Proceedings of ACT4SOC-2010
18. Vibhu Saujanya Sharma, “Markov Chains and Queueing Network”, CS 797 Independent Study Report, sece.ut.ac.ir/Classpages/S86/ECE462/slides/MarkovChains.pdf

Инструменты тестирования и отладки

Секционные доклады

Мультиплатформенный метод обратной отладки виртуальных машин

Довгалюк П. М., Климушенкова М. А., Дмитриев Д. С., Макаров В. А.
Новгородский государственный университет им. Ярослава Мудрого
В. Новгород, ул. Б. С.-Петербургская, 41
Emails: {pavel.dovgaluk, maria.klimushenkova, denis.dmitriev, vladimir.makarov}@ispras.ru

Аннотация—Отладка и прототипирование новых аппаратных платформ, периферийных устройств и операционных систем осложняется недетерминированностью их поведения, воздействием отладчика на систему, длительностью процесса воспроизведения ошибок, отказами всей системы и другими подобными факторами. В работе представлен метод детерминированного воспроизведения работы виртуальных машин и показано как он позволяет бороться со сложностями, возникающими при отладке. На основе разработанного подхода авторами создан мультиплатформенный полносистемный обратный отладчик. С помощью этого отладчика становится возможным изучение сбоев без воздействия на их проявление. Можно воспроизводить и многократно изучать состояние всей виртуальной машины и всех ее периферийных устройств. В работе представлен обратный отладчик, поддерживающий целевые платформы i386, x86-64, MIPS, PowerPC и ARM. Он протестирован для гостевых ОС Windows и GNU/Linux. Отладчик можно использовать при моделировании периферийных устройств и отладке пользовательского и системного кода. Влияние на процесс отладки ограничено лишь вносимым при записи работы системы замедлением. Это замедление незначительно влияет на работу гостевой системы и позволяет отлаживать критичные ко времени приложения.

1. Введение[1]

Прототипирование новых аппаратных платформ, периферийных устройств, операционных систем и драйверов устройств требуют значительных затрат на симуляцию, тестирование и отладку [1]. Эти процессы можно упростить и ускорить с помощью применения технологий виртуализации [2].

Отладку усложняют ошибки в ядре ОС, состояния гонки и сбои оборудования — каждый раз, когда происходит критическая ошибка, разработчик должен перенастроить окружение и перезапустить отлаживаемую систему. Это приводит к потерям времени, а кроме того, в результате сбоев могут быть повреждены данные в системе. Сбои могут проявляться каждый раз по-разному или не проявляться вообще [3].

1. Работа была выполнена при частичной поддержке РФФИ, проект 14-07-00411 а

Если используется интерактивный отладчик, то задержки при остановке программы могут вызвать сбои из-за таймаутов оборудования или удаленных систем. Также отладчик может повлиять на работу программы из-за состояний гонки, неинициализированной памяти и т. п.

Традиционная итеративная отладка требует, чтобы программа при каждом запуске работала одинаково. Но при взаимодействии с внешней аппаратурой становится сложнее обеспечить это условие, так как начальное состояние устройств должно быть одним и тем же. Разработчику приходится затрачивать дополнительное время для инициализации и настройки устройств и окружения.

Обратная отладка позволяет уменьшить влияние подобных проблем на процесс разработки драйверов и моделей оборудования. Вместо повторного выполнения программы можно записать ее работу и изучать ошибку многократно с помощью воспроизведения этой записи.

Обратная отладка нацелена на изучение прошедших состояний системы, включая значения ячеек памяти и переменных в программе [4]. Например, можно установить точку останова и «выполнить» программу назад, чтобы узнать, когда это точка срабатывала ранее. Основное преимущество обратной отладки — это возможность отслеживания источников значений данных. Кроме того, можно отделить этап анализа или отладки программы от этапа ее выполнения. Таким образом динамический анализ не будет влиять на работу программы [5].

Наша цель — это упрощение отладки операционных систем, драйверов, моделей и прототипов периферийных устройств. Для этого используется обратная отладка на основе виртуальных машин. Воспроизведение работы всей виртуальной машины позволяет изучать код ядра операционной системы, состояние всех виртуальных устройств и взаимодействие с внешними устройствами. Многоплатформенный механизм воспроизведения позволяет упростить протопипирование виртуальных устройств, подключаемых к различным платформам, переносимых операционных систем, а также использовать платформо-независимые методы динамического анализа программ. Многоплатформенность позволяет многократно использовать код для анали-

за программ без затрат времени на его портирование под разные платформы, если это потребуется.

Те немногие существующие разработки, которые доступны для использования или описаны в публикациях, обычно поддерживают только одну или две гостевые платформы. Simics, единственное по-настоящему многоплатформенное средство для обратной отладки, работает слишком медленно, чтобы можно было отлаживать критичные по времени программы (например, сетевые приложения) или взаимодействие с периферией [6].

В настоящей работе представлен подход к созданию мультиплатформенного средства обратной отладки. Это средство может быть использовано для отладки пользовательского и системного кода, а также моделей аппаратных устройств. В работе представлены следующие результаты:

- Метод воспроизведения работы системы, который не зависит от аппаратной и программной гостевых платформ. Новизна метода воспроизведения заключается в использовании платформо-независимых средств для внедрения недетерминированных событий.
- Реализация обратной отладки на основе симулятора QEMU и отладчика GDB. Отладчик работает с современыми аппаратными платформами и операционными системами, и не требует их модификации.
- Оценка производительности симулятора со включенным механизмом воспроизведения программ и сравнение его с другими существующими реализациями.

Работа состоит из следующих разделов. В разделах 2 и 3 описаны особенности предлагаемого метода и его реализации. В разделе 4 представлена реализация обратной отладки на основе симулятора QEMU, совместимая с интерфейсом отладчика GDB. В раздел 5 приведены результаты измерения производительности симулятора при записи и воспроизведении работы виртуальной машины. Раздел 6 содержит обзор других исследований в области обратной отладки виртуальных машин и их сравнение с предлагаемым подходом. Разделы 7 и 8 содержат заключение и описание направлений продолжения исследований.

2. Платформо-независимая обратная отладка

Необходимость создания нового реверсивного отладчика продиктована тем, что не существует других средств обратной отладки, которые могли бы работать со всеми современными платформами, отлаживать взаимодействие с внешними устройствами и были бы достаточно быстрыми для отладки критичных ко времени приложений.

Для полносистемной отладки и поддержки различных платформ отладчик может быть реализован на основе монитора виртуальных машин. Поэтому при его реализации мы использовали симулятор QEMU. Он поддерживает множество гостевых систем, например x86, x86-64, ARM, MIPS и PowerPC [7]. QEMU транслирует гостевой бинарный код в код хозяйской машины, а затем исполняет уже его. За счет кэширования ранее транслированных блоков QEMU работает быстрее, чем программные интерпретаторы.

Для создания механизма обратной отладки мы использовали идею детерминированного воспроизведения, когда во время выполнения сценария с исследуемой ошибкой записываются все недетерминированные данные в системе. Такие данные могут поступать от пользователя, удаленных компьютеров, часов реального времени и внешних устройств. Аппаратные прерывания также должны быть записаны, так как они возникают в случайные моменты времени.

Наш подход основывается на воспроизведении состояния центрального процессора, памяти и всех виртуальных периферийных устройств, подключенных к системе. Для этого перехватываются все входы к этим компонентам системы. Данные, поступающие на эти входы, записываются (а затем и воспроизводятся) в журнал недетерминированных событий, как показано на рисунке 1.

Новизна нашего подхода заключается в создании аппаратно-независимого слоя внутри симулятора для захвата недетерминированных данных. Мы используем платформо-независимый счетчик инструкций процессора, в то время как в других подходах для этого используются платформо-зависимые регистры [8], [9]. Эти свойства позволяют нам создать средство обратной отладки, поддерживающее множество аппаратных платформ. Добавляемые в симулятор платформы автоматически получают возможность использования обратной отладки без их модификации.

На основе этого подхода мы реализовали платформо-независимые запись и воспроизведение работы виртуальной машины, а также обратную отладку для нее. Все функции, связанные с воспроизведением, работают с абстрактными данными и событиями. Можно записывать работу всей виртуальной машины, а затем воспроизводить все программы и состояния виртуальных устройств или внешних моделей устройств (например, моделей SystemC [1]) с целью отладки или анализа их работы.

3. Реализация детерминированного воспроизведения

Наш механизм записи и воспроизведения основан на сохранении недетерминированных событий (например, ввода с клавиатуры) и симуляции детерминированных (например, чтения жесткого диска или

Рис. 1. Мультиплатформенное воспроизведение работы виртуальных машин на основе QEMU.

гостевой памяти) [10]. Сохранение только высокоуровневых событий значительно уменьшает объем журнала событий по сравнению с реализациями, сохраняющими обращения процессора к портам ввода-вывода (например, [8]), что в конечном итоге ускоряет процесс симуляции и позволяет использовать обратную отладку для критичных по времени приложений.

В журнал сохраняются следующие данные от периферийных устройств: ввод с клавиатуры и мыши, сетевые пакеты, ввод от аудио контроллера, пакеты USB, ввод с последовательного порта и показания аппаратных часов реального времени. Ввод из виртуальных устройств, гостевой памяти, программные прерывания и выполнение отдельных инструкций не сохраняются в журнал, потому что они детерминированные и могут быть воспроизведены с помощью симуляции работы виртуальной машины, если начинать ее с одного и того же стартового состояния.

Для реализации воспроизведения работы виртуальной машины нам нужно было решить три задачи: запись недетерминированных событий, воспроизведение событий и проверка того, что поведение виртуальной машины при воспроизведении не изменилось.

Для этого необходимо было привязать внешние события к моменту выполнения гостевой программы, чтобы воспроизводить их в нужное время. Мы задаем эти моменты времени с помощью подсчета инструкций, выполнившихся между каждой парой последовательных событий.

Чтобы убедиться, что работа виртуальной машины соответствует работе во время записи выполнения, мы проверяем, что внутренние события машины (обращения к аппаратным часам, прерывания, исключительные ситуации) считываются из журнала в правильном порядке. Если прочитано событие, которое не может произойти в текущем состоянии симулятора, то воспроизведение останавливается с сообщением об ошибке.

3.1. Подсчет инструкций

Для подсчета количества инструкций между внешними событиями мы используем встроенный в QEMU независимый от гостевой платформы счетчик icount. Этот счетчик работает в отдельном режиме работы icount. Он позволяет детерминированно выполнять программу в отсутствие внешних воздействий. Мы расширили этот режим, чтобы была возможна запись и воспроизведение работы виртуальной машины со всеми внешними взаимодействиями.

QEMU использует динамическую трансляцию гостевого кода. Каждая гостевая инструкция преобразуется в последовательность команд хозяйской машины. Инструкции объединяются в блоки трансляции — непрерывные последовательности инструкций. Блоки трансляции всегда выполняются от начала до конца целиком, за исключением случаев, когда возникают исключительные ситуации.

С помощью icount можно узнавать точное количество выполненных инструкций. Но счетчик инструкций увеличивается не после каждой инструкции, а лишь в начале каждого блока трансляции. Если же возникает исключительная ситуация, то блок транслируется повторно для того, чтобы вычислить сколько инструкций успело выполниться с начала блока до возникновения исключения.

Мы используем icount, чтобы управлять восстановлением последовательности недетерминированных событий. Число инструкций между событиями записывается в журнал. А при воспроизведении счетчик icount останавливается на том месте, где должно быть загружено очередное событие.

Использование icount позволяет контролировать выполнение недетерминированных событий, но при этом уменьшает производительность симулятора. Код для подсчета инструкций добавляется к коду блока трансляции, как показано на рисунке 2. Знаком “+” помечены операции, выполняющие подсчет инструкций. Оценка влияния этого кода на производительность приведена в разделе 5.

Выполняемый блок трансляции

```
0x000fd1fe:  mov    %eax,%gs
0x000fd200:  mov    %ecx,%eax
0x000fd202:  jmp    *%edx
```

Промежуточное представление блока трансляции

```
  ld_i32 tmp12,env,$0xffffff4
  movi_i32 tmp13,$0x0
  brcond_i32 tmp12,tmp13,ne,$0x0
+ ld_i32 loc14,env,$0xffffffe8
+ movi_i32 tmp12,$0x3
+ sub_i32 loc14,loc14,tmp12
+ movi_i32 tmp12,$0x0
+ brcond_i32 loc14,tmp12,lt,$0x1
+ st16_i32 loc14,env,$0xffffffe8
  ---- 0xfd1fe
  mov_i32 tmp0,eax
  movi_i32 tmp3,$0xfd1fe
  st_i32 tmp3,env,$0x20
  mov_i32 tmp6,tmp0
  movi_i32 tmp12,$0x5
  call load_seg,$0x0,$0,env,tmp12,tmp6
  ---- 0xfd200
  mov_i32 tmp0,ecx
  mov_i32 eax,tmp0
  ---- 0xfd202
  mov_i32 tmp0,edx
  st_i32 tmp0,env,$0x20
  ----
  exit_tb $0x0
  set_label $0x0
  exit_tb $0x5c8033b
+ set_label $0x1
+ exit_tb $0x5c0045a
```

Рис. 2. Трансляция кода в режиме icount.

3.2. Часы реального времени

QEMU использует часы реального времени хозяйской машины (real time clock, RTC) для использования в гостевой машине или для внутренних таймеров. С помощью таймеров функции обратного вызова для различных подсистем QEMU вызываются в заданные моменты времени. Есть несколько типов часов для таймеров:

- Часы реального времени (real time). Используется для функций, которые не меняют состояния гостевой системы. Поэтому эти часы не влияют на работу записи и воспроизведения.
- Виртуальные часы (virtual). Эти часы идут только во время эмуляции. В режиме icount их показания вычисляются на основе счетчика инструкций. В результате они становятся детерминированными и их показания не нужно записывать в журнал событий.
- Хозяйские часы (host). Они используются моделями устройств, которые эмулируют источники реального времени (например, микросхему часов). Поэтому они являются одним из источников недетерминизма и их показания записываются в журнал при считывании времени.
- Часы реального времени для icount. Это те же самые часы, что и первые, но они используются для увеличения показаний виртуальных часов, когда гостевой процессор не работает. Эти изменения влияют на работу гостевой системы и поэтому записываются в журнал событий.

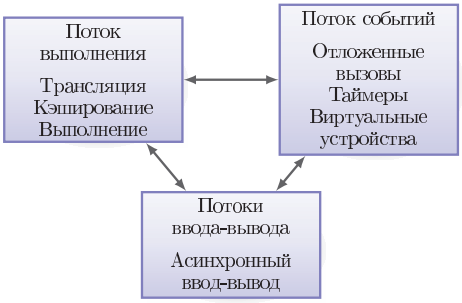


Рис. 3. Структура потоков управления QEMU.

Для тех часов, чьи показания записываются в журнал, мы создали функции-обертки вокруг обращений к ним. В режиме записи эти функции записывают возвращаемое время в журнал. В режиме воспроизведения реального обращения к часам не происходит, а выполняется чтение из журнала.

3.3. Контрольные точки симуляции

Работа с часами или периферией в большинстве случаев выполняется синхронно с основным процессором, так как эти события инициируются гостевыми инструкциями, которые все работают в одном потоке выполнения. Но кроме этого потока в QEMU работает еще несколько потоков для обеспечения асинхронного ввода-вывода.

Симулятор выполняется в нескольких потоках, как изображено на рисунке 3. Поток выполнения работает в цикле и вызывает модуль исполнения гостевого кода. Модуль исполнения выполняет трансляцию очередного блока или берет его из кэша, а затем выполняет. В потоках событий и ввода-вывода выполняются отложенные или асинхронные задания. Таким образом чтение данных с диска может производиться параллельно с выполнением кода. Кроме того, несмотря на то, что недетерминированные показания часов и записываются в журнал, обращения к этим часам могут происходить в произвольные моменты времени, когда симулятор проверяет не истекли ли таймеры.

Такая асинхронная архитектура усложняет детерминированное воспроизведение работы гостевой машины. Кроме недетерминированного ввода от периферийных устройств, появляется еще недетерминированное межпотоковое взаимодействие и обработка таймеров.

Так как мы хотим поддерживать состояние всех устройств в корректном состоянии как во время записи, так и во время воспроизведения, нужно поддерживать относительный порядок таймерных вызовов друг для друга, если они привязаны к одному моменту в потоке инструкций гостевой системы. Для этого мы добавили в симулятор контрольные точки. Каждый раз, когда выполняется очередная контрольная точка, в журнал событий записывается информация об этом. Контрольные точки позволяют обрабатывать таймеры в том же порядке при воспроизведении, в котором они выполнялись при записи.

Контрольые точки также позволяют корректно управлять изменением счетчика времени, когда процессор виртуальной машины не выполняет код. За это отвечает функция qemu_clock_warp, вызовы которой привязываются к контрольной точке. Тогда эта функция вызывается в одни и те же моменты времени выполнения гостевой программы и при записи, и при воспроизведении ее работы.

3.4. Начальное состояние системы

Перед воспроизведением записанного сценария работы системы нужно привести ее в начальное состояние, соответствующее начальному состоянию при записи работы. И так как виртуальная машина выполняет тот же код, а все недетерминированные события записаны в журнал, ее состояния будут при воспроизведении меняться так же, как и менялись при записи.

3.5. Асинхронный ввод-вывод

Дисковый ввод и вывод оперирует одними и теми же данными и при записи, и при воспроизведении сценария работы системы. Но из-за того, что операции чтения и записи работают в отдельных потоках, может произойти рассинхронизация с работой процессора.

Для того, чтобы сделать ввод-вывод детерминированным, мы синхронизировали эти операции с контрольными точками симулятора. Когда создается задание на ввод-вывод, оно добавляется в очередь. При прохождении контрольной точки все задания из очереди выполняются, а их список сохраняется в журнале. Когда сценарий работы воспроизводится, содержимое очереди сопоставляется с содержимым журнала. Если в журнале нужного задания нет, то оно ожидает следующей контольной точки.

3.6. Внешние устройства

QEMU поддерживает подключение реальных USB-устройств к виртуальной машине. С помощью этого можно прототипировать и отлаживать программное обеспечение, работающее с USB. QEMU работает с USB версий 1.1–3.0. Наш механизм записи и воспроизведения работает в абстрактном слое оборудования QEMU, поэтому он поддерживает воспроизведение работы с реальными USB-устройствами, подключаемыми к симулятору.

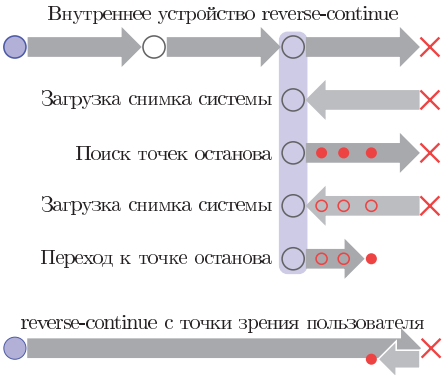


Рис. 4. Обратная отладка с помощью детерминированного воспроизведения.

4. Обратная отладка через интерфейс GDB

Обратная отладка — это прохождение процесса выполнения программы в обратном направлении с целью изучения ее состояния в разные моменты времени [11]. Обратная отладка позволяет использовать точки останова и находить те из них, которые должны были бы сработать в прошлом, если бы были установлены.

QEMU поддерживает обычную отладку через интерфейс GDB для удаленной отладки. Этот интерфейс поддерживает точки останова, точки наблюдения, пошаговое выполнение и другие традиционные возможности отладки. Но так как обратная отладка в QEMU реализована не была, нам пришлось расширить QEMU, чтобы сделать обратную отладку доступной через интерфейс GDB. Мы добавили поддержку операций «шаг назад» (reverse step) и «продолжить назад» (reverse continue). Выполнение этих операций становится возможным в режиме воспроизведения работы системы. «Шаг назад» переходит к предыдущему записанному шагу (инструкции), а «продолжить назад» находит самую позднюю из предшествующих точек останова.

Для обеих этих команд используются заранее сделанные снимки состояния системы, как показано на рисунке 4. Снимки делаются во время записи сценария работы системы. Первый снимок делается при старте, а остальные через каждые N секунд (где N — опция командной строки).

После загрузки состояния из сохраненного снимка симулятор воспроизводит сценарий работы, чтобы найти заданную точку (для шага назад) или для поиска сработавшей точки останова (для команды продолжить назад). После этого для продолжения назад нужно сделать еще один дополнительный проход. После достижения точки откуда команда была подана, нужно снова вернуться назад и перейти на найденную точку останова.

Пользователю для отладки в обратном направлении нужно лишь подать команду симулятору через отладчик. Мы реализовали эти команды так, что все действия, связанные с загрузкой состояний системы и несколькими проходами воспроизведения, выполняются автоматически.

5. Оценка производительности записи и воспроизведения

В этом разделе приводятся оценки замедления симулятора, возникающего при записи и воспроизведении работы виртуальной машины по отношению к обычной симуляции. Для этого мы использовали несколько тестов. В первом тесте мы загружали Windows XP на виртуальной машине с процессором x86. В остальных тестах загружалась операционная система Debian Wheezy в системах на x86, ARM, PowerPC и MIPS. У всех виртуальных машин было по 128 Мб памяти.

Симулятор запускался на компьютере, оснащенном процессором Intel Core i7 с 8 ядрами на частоте 3.40GHz и 8Gb ОЗУ под управлением 64-битной версии Ubuntu 14.04.

Каждый тест выполнялся в 4 режимах:

- regular Симуляция в QEMU без использования механизма icount для подсчета инструкций. Это недетерминированный режим и приводится для сравнения с режимом детерминированного выполнения.
- icount Включен подсчет инструкций с помощью опции -icount shift=7. Этот режим необходим для детерминированного выполнения инструкций и хода виртуальных часов. Внешние воздействия остаются недетерминированными.
- icount+record Режим записи работы системы. Все внешние устройства подключены к виртуальной машине, а поступающие от них данные записываются в журнал событий.
- icount+replay Работа системы воспроизводится по ранее записанному журналу. Внешние устройства не подключены, а данные поступают из журнала.

Мы измеряли время выполнения каждого теста в каждом из режимов. Результаты представлены в таблице 1. Кроме времени в секундах в таблице приведены два нормализованнх значения для каждого теста. Первое нормализованное значение показывает увеличение времени работы симулятора по отношению к режиму обычного выполнения. Второе нормализованное значение позволяет оценить время, затрачиваемое на запись журнала событий и его воспроизведение.

Таблица 1. Производительность детерминированного воспроизведения

Режим работы	Время выполнения, с	Отношение к обычному времени	Отношение ко времени с icount
Загрузка Windows (x86)			
regular	75	1.00	—
icount	89	1.19	1.00
icount+record	98	1.31	1.10
icount+replay	228	3.04	2.56
Загрузка Debian Wheezy (x86)			
regular	130	1.00	—
icount	177	1.36	1.00
icount+record	183	1.41	1.03
icount+replay	418	3.22	2.36
Загрузка Debian Wheezy (ARM)			
regular	124	1.00	—
icount	157	1.27	1.00
icount+record	164	1.32	1.04
icount+replay	457	3.69	2.91
Загрузка Debian Wheezy (PowerPC)			
regular	74	1.00	—
icount	71	0.96	1.00
icount+record	153	2.07	2.15
icount+replay	очень долго	—	—
Загрузка Debian Wheezy (MIPS)			
regular	160	1.00	—
icount	179	1.12	1.00
icount+record	182	1.14	1.02
icount+replay	428	2.68	2.39

Включение icount делает работу системы детерминированной. Виртуальное время в системе вычисляется на основе числа выполненных инструкций. Поэтому скорость симуляции не будет влиять на работу системы. Виртуальная машина может обнаружить замедление лишь при взаимодействии с внешним миром (сетью или USB-устройствами). Если замедление будет слишком велико, это может повлиять на удобство использования системы во время записи сценария ее работы. В нашем случае замедление из-за подсчета инструкций невелико и составляет от 12 до 36%

Замедление из-за записи журнала также может исказить работу системы. Чтобы оценить это замедление, мы разделили время выполнения с записью сценария ко времени работы со включенным icount. Полученные значения представлены в четвертой колонке таблицы 1. В наших тестах замедление записи находится в пределах от 3 до 10% (за исключением PowerPC).

Замедление при воспроизведении не влияет на работу гостевой машины, а только на удобство пользователя. Замедление в наших тестах незначительно влияет на удобство работы и находится в пределах

Таблица 2. Скорость роста журнала для разных платформ.

Платформа	Размер журнала, байт	Выполнено инструкций	Байт на 1000 инструкций
x86	139M	6346M	21.9
ARM	80M	4469M	17.9
PowerPC	2258M	4790M	471.4
MIPS	257M	3408M	75.4

от 204 до 269%.

Кроме замедления мы также измерили объем дискового пространства, используемого для хранения журнала событий. Журнал растет на 10 Кб/с, когда ОС находится в режиме ожидания, до 715 Кб/с, если происходит много прерываний и дисковых операций. Скорость роста журнала достаточно небольшая для того, чтобы записывать длительные сценарии работы, вплоть до нескольких дней. В таблице 2 приведена скорость роста журнала для случаев загрузки Debian на разных платформах.

Симулятор платформ на процессоре PowerPC показал заметно отличающиеся от остальных результаты. Симулятор со включенным подсчетом инструкций работает быстрее, чем с выключенным подсчетом, а воспроизведение записанного сценария работает на порядки медленнее, чем обычное его выполнение. Скорость роста журнала также значительно больше, чем для других платформ. В дальнейшем мы планируем разобраться с этим случаем, чтобы исправить работу записи и воспроизведения для PowerPC.

6. Обзор полносистемной обратной отладки

Существует множество работ, описывающих подходы и реализации полносистемной обратной отладки [5], [8], [9], [12], [13]. Некоторые из них направлены на работу только с одной гостевой платформой, другие слишком медленные или не поддерживают работу с периферийными устройствами. Мы также не будем рассматривать здесь подходы, направленные на отладку отлдельных программ, так как сфера их применения отличается от полносистемных отладчиков.

PANDA — симулятор на основе QEMU, поддерживающий запись и воспроизведение для нескольких аппаратных архитектур [8]. Во время воспроизведения PANDA восстанавливает только состояние ОЗУ и центрального процессора. В журнал записываются читаемые процессором данные из портов, произошедшие прерывания и DMA-транзакции.

PANDA использует платформо-зависимый способ определения того, когда нужно воспроизводить очередное событий из журнала. Для x86 моменты возникновения событий определяются с помощью счетчика инструкций (EIP), количества выполненных инструкций и значения неявной переменной цикла (регистра ECX).

PANDA не может продолжить нормальное выполнение после воспроизведения записанного журнала, так как восстанавливаются только состояния ЦП и ОЗУ. Поэтому пользователь не может изучать состояние виртуальных устройств при их моделировании. Наша реализация записи и воспроизведения позволяет продолжать работу в любой момент воспроизведения сценария работы, потому что все виртуальные устройства всегда находятся в согласованном состоянии. Замедление записи работы для PANDA составляет 85%, а воспроизведения — 257%.

Simics — это мультиплатформенный симулятор от компании Wind River [6]. Поддержка обратной отладки в нем появилась в 2005 году. Обратная отладка возможна для целой виртуальной машины и даже для нескольких машин одновременно. Simics симулирует множество платформ (включая x86-64, ARM, MIPS и PowerPC) и операционных систем. Все это делает Simics одним из лучших коммерческих обратных отладчиков. С другой стороны, из-за такого обширного набора возможностей, стоимость Simics очень высока, что делает его недоступным для большинства разработчиков.

В то же время Simics работает слишком медленно для комфортной отладки приложений, взаимодействующих с пользователем и реальным оборудованием. В работе [14] приведены данные о том, что Simics выполняется в 40 раз медленнее, чем QEMU.

7. Заключение

В работе представлен полносистемный легковесный платформо-независимый метод записи и воспроизведения работы виртуальных машин. Наш метод был реализован на основе мульти-платформенного симулятора QEMU, а исходные тексты были опубликованы на github[2]. Изменения в симулятор вносились только платформо-независимые модули, поэтому запись и воспроизведение автоматически становятся доступными для всех аппаратных платформ, поддерживаемых в QEMU. Мы протестировали запись и воспроизведение для платформ i386, x86-64, MIPS, PowerPC и ARM.

Механизм записи поддерживает работу с сетевыми картами, аудио адаптером, последовательным портом и USB-устройствами. Его работа была протестирована с рядом современных USB-устройств, работающих на разных скоростях и в разных режимах.

Обратная отладка, добавленная в QEMU, позволяет изучать состояние всей системы, возвращаясь в любой момент времени записанного сценария. Механизм обратной отладки поддерживает переход на шаг назад и поиск точек останова назад во времени. Отлаживать работу с реальными устройствами можно без их подключения, записав однажды сценарий работы.

Реализованный механизм записи и воспроизведения работы программ можно использовать для от-

2. https://github.com/Dovgalyuk/qemu/tree/rr-17

ладки драйверов, кода операционных систем и пользовательских приложений, не затрачивая времени на повторное воспроизведение ошибок и настройку системы. Воспроизведение виртуальной машины можно использовать при прототипировании новых платформ и устройств. Разработанную модель устройства, не существующего «в железе», можно изучать в режиме «медленной пермотки» благодаря детерминированному воспроизведению работы системы.

8. Направления будущей работы

Разработанный механизм детерминированного воспроизведения выложен в открытый доступ. Сейчас мы дорабатываем его для включения в основную ветку симулятора QEMU. Это сделает обратную отладку доступной для всех разработчиков, использующих QEMU. Также мы планируем доработать запись и воспроизведение для ускорения эмуляции платформ на PowerPC в этих режимах. Детерминированное воспроизведение может быть полезным и при использовании других методов анализа программ помимо обратной отладки. Анализ уже записанного сценария может быть сколь угодно «тяжелым» и при этом не влиять на работу приложения. Например, анализ помеченных данных может добавлять замедление до 600% [15]. Перенос этого замедления с фазы выполнения программы на фазу ее воспроизведения может сделать результаты анализа более достоверными. А сам анализ можно ускорить за счет распараллеливания на нескольких компьютерах [14].

Список литературы

[1] M. Monton, A. Portero, M. Moreno, B. Martinez, and J. Carrabina, "Mixed sw/systemc soc emulation framework," in Industrial Electronics, 2007. ISIE 2007. IEEE International Symposium on, June 2007, pp. 2338–2341.

[2] M. Baklashov, "An on-line memory state validation using shadow memory cloning," in Proceedings of the 2011 IEEE 17th International On-Line Testing Symposium, ser. IOLTS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 186–189. [Online]. Available: http://dx.doi.org/10.1109/IOLTS.2011.5993837

[3] J. Gray, "Why do computers stop and what can be done about it?" 1985.

[4] J. Engblom, "A review of reverse debugging," in in S4D, 2012.

[5] J. Chow, T. Garfinkel, and P. M. Chen, "Decoupling dynamic program analysis from execution in virtual environments," in USENIX 2008 Annual Technical Conference on Annual Technical Conference, ser. ATC'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 1–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=1404014.1404015

[6] J. Engblom, D. Aarno, and B. Werner, "Full-system simulation from embedded to high-performance systems," in Processor and System-on-Chip Simulation, R. Leupers and O. Temam, Eds. Springer US, 2010, pp. 25–45. [Online]. Available: http://dx.doi.org/10.1007/978-1-4419-6175-4_3

[7] F. Bellard, "Qemu, a fast and portable dynamic translator," in Proceedings of the Annual Conference on USENIX Annual Technical Conference, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247360.1247401

[8] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, "Repeatable reverse engineering for the greater good with panda," Oct. 2014.

[9] H. Liu, H. Jin, X. Liao, and Z. Pan, "Xenlr: Xen-based logging for deterministic replay," in Proceedings of the 2008 Japan-China Joint Workshop on Frontier of Computer Science and Technology, ser. FCST '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 149–154. [Online]. Available: http://dx.doi.org/10.1109/FCST.2008.31

[10] P. Dovgalyuk, "Deterministic replay of system's execution with multi-target qemu simulator for dynamic analysis and reverse debugging," in Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering, ser. CSMR '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 553–556.

[11] B. Boothe, "Efficient algorithms for bidirectional debugging," SIGPLAN Not., vol. 35, no. 5, pp. 299–310, May 2000. [Online]. Available: http://doi.acm.org/10.1145/358438.349339

[12] S. S. Chia-Wei Hsu, "Free: A fine-grain replaying executions by using emulation," ser. The 20th Cryptology and Information Security Conference (CISC 2010), 2010.

[13] D. Jacobowitz and B. P., "Reversible debugging," ser. GCC Developer's Summit, 2007.

[14] M. Rittinghaus, K. Miller, M. Hillenbrand, and F. Bellosa, "Simuboost: Scalable parallelization of functional system simulation," in Proceedings of the 11th International Workshop on Dynamic Analysis (WODA 2013), Houston, Texas, Mar. 16 2013.

[15] A. Henderson, A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin, "Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform," in Proceedings of the 2014 International Symposium on Software Testing and Analysis, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 248–258. [Online]. Available: http://doi.acm.org/10.1145/2610384.2610407

Трассировка многомодульного приложения в среде операционной системы z/OS

Ефремов Р.С.
математико-механический факультет
СПбГУ
Санкт-Петербург, Россия
slava.efremov1994@mail.ru

***Аннотация*—Динамический анализ бинарного кода - методика, позволяющая с помощью информации о ходе выполнения программы на различных данных, получать данные о том, как она работает. Задача динамического анализа особенно актуальна для операционной системы z/OS, так как для нее существует множество программ, написанных десятки лет назад, которые до сих пор поддерживаются и развиваются. Mainframe - специфичная бизнес платформа, и публикации, освещающие в ее контексте некоторые проблемы достаточно редки. В частности, это относится к исследованиям в области динамического анализа в среде операционной системы z/OS. В данном исследовании рассмотрена эта проблема, а также определены основные методики получения графа динамических вызовов по результатам исполнения исследуемой программы. Кроме того, произведено сравнение их производительности, уровня воздействия на окружение исследуемой программы. Дополнительно в статье описан реализованный автором прототип трассировщика, который строит таблицу переходов для исследуемой программы. В ходе исследования оказалось, что решение центральной задачи динамического анализа: трассировки, сопряжено с рядом трудностей, связанных со специфичностью платформы. Тем не менее, выявлено два наиболее предпочтительных метода, которые используются в таких популярных коммерческих отладчиках как TDF и z/XDC. Эти подходы позволяют сохранить реентерабельность приложения, имеют слабые ограничения на окружение, являются достаточно производительными. Результаты исследования открывают возможности для разработки своих продуктов, осуществляющих динамический анализ, создания отладчиков. Кроме того, в работе рассматриваются некоторые слабо освещенные вопросы относительно архитектуры операционной системы z/OS.**

Ключевые слова—Mainframe; граф динамических вызовов; z/OS;

I. Введение

Динамический анализ бинарного кода - методика, позволяющая с помощью информации о ходе выполнения программы на различных данных, делать выводы о том, как она работает. Задача динамического анализа особенно актуальна для операционной системы z/OS. Имеется много старого кода, который сейчас поддерживают, развивают и дополняют новым функционалом. Динамический анализ требует информации о переходах внутри модулей и между модулями. Перехват и трассировка прыжков между модулями является более сложной и общей задачей, которая решена в ряде коммерческих продуктов, неизвестным нам образом, так как их код закрыт. Справимся с этой задачей - с большой вероятностью будем иметь решение и для переходов внутри.

Целью данной работы было исследование различных методов трассировки переходов между модулями, сравнение их производительности и реализация тестовой программы для отслеживания простых прыжков типа BC в программах 31-битного режима в primary address space mode для операционной системы z/OS, укомплектованных информацией о компиляции (ADATA) и скомпилированных с опцией THREAD. Во время трассировки должна сохраняться реентерабельность исследуемого приложения.

II. Существующие аналоги

A. IDF - Interactive Debug Facility [1]

IDF - стандартная утилита IBM для отладки, которая поставляется вместе с HLASM Toolkit. В IDF User's Guide указано, что для отладки она использует либо SVC 97 (то есть SVC hooking) либо ESTAE/ESPIE в зависимости от указанных опций [1]. Из этого сразу следуют существенные ограничения в работе отладчика, которые будут более подробно описаны в разделе 3. В частности, они привели к низкой популярности IDF среди системных программистов.

B. z/XDC - Extended Debugging Controller [2]

z/XDC - пожалуй самый мощный инструмент для отладки и сбора трассировочной информации. Он позволяет отлаживать приложения, исполняемые в практически любом окружении операционной системы z/OS: SRB, SVC, RTM, а также использующие разные типы адресных пространств, различные уровни привилегированности. Хотя статей о том как работает z/XDC мало, по Release Guide и анализу памяти приложения в момент работы отладчика (предоставленной коллегами из компании EMC) можно сделать вывод о том, что точно используются следующие методы перехвата: ESTAE, SVC hooking [2]. Кроме того, в

целом неизвестно, какими средствами обеспечивается связь отладчика и исследуемой программы, и в каких случаях замена команд происходит до запуска исследуемой программы. Так как код отладчика закрыт, и он является платным, изучение этих вопросов затруднительно. Далее в статье рассмотрена самостоятельная реализация SVC hooking для трассировки многопоточного приложения.

C. *TDF - Trap Debug Facility [3]*

TDF - сравнительно новая программа, основанная на технологии TRAP, о чем явно указано в презентации продукта [3]. Из этого автоматически следуют ограничения в ее работе: нельзя исследовать приложения secondary address space mode и программы, использующие TRAP механизм. Также интерес представляет то, как устанавливается TRAP для исследуемой программы, и как эта программа запускается. В статье далее показано, как теоретически можно обойти эти ограничения, а также приведено сравнение скорости работы TRAP и SVC hooking технологий.

III. Подходы к перехвату переходов между модулями

В целом к построению графа динамических переходов можно подойти с разных позиций:

A. *Создание (или модификация) виртуальной машины*

Создание виртуальной машины - весьма трудоемкая задача. Однако для Mainframe уже существует открытый бесплатный эмулятор Hercules. Его можно модифицировать для отслеживания определенных событий. Однако динамический анализ продуктов использующих возможности Mainframe, которые не поддерживаются в Hecules, будет просто невозможен. Например, это могут быть FICON или нестандартное hardware [4].

B. *Использование стандартных аппаратных средств организации отладки*

Изучение документации показало, что платформа Mainframe содержит аппаратное средство для реализации отладчиков, и называется оно PER - Program Event Recording [5]. Его использует стандартная часть операционной системы z/OS - SLIP Trace

Однако использование PER приводит в нашей ситуации к следующим проблемам:

- Придется обрабатывать все "прыжки" вне зависимости от того: внутренние они или внешние
- Все события перехода для выделенного куска памяти будут записываться в один dump. Таким образом, например, могут перемешаться записи о "прыжках" из разных subtask. Их нужно будет разделять
- Если прыжок происходит в область памяти, которая не входит в заданный диапазон адресов (а он может быть только один), то невозможно получить адрес точки назначения перехода. Значит, надо выбирать память таким образом, чтобы она включала в себя инструкцию перехода и адрес точки назначения. Но по сути это и является задачей: определением перехода. Можно попробовать выбрать память так, чтобы в ней лежали все модули интересующего нас приложения, но нет никакой гарантии, что между ними не окажется, например, кода, исполняющегося в другом потоке. Возникает идея выделить address space целиком, но это приводит к катастрофическому снижению производительности, так как система не справляется с обработкой переходов всех task, subtask, SRB, потому что в крупных системах их очень много
- Невозможность установки своего обработчика PER: ESPIE (сервис для установки обработчиков прерываний) позволяет назначить обработчик только для прерываний с кодами 0-15, 17 [6], а PER имеет код 80 [5]. Другими доступными методами установка также невозможна, так как IH - Interrupt Handler поддерживает область real memory с обработчиком прерываний от z/OS. Это FLIH - First Level Interrupt Handler [7]. Написать свой FLIH не представляется возможным из-за отсутствия доступа к определенной технической документации

Другое стандартное средство организации трассировки - инструкция TRAP [5]. В системе команд Mainframe есть две команды: TRAP2 и TRAP4. Они изначально были предназначены для решения проблемы 2000 года но нашли применение и в организации отладки, так как предоставляют быстрый и удобный сервис для перехода в некоторую подпрограмму с почти полным восстановлением контекста. Поскольку команда TRAP2 занимает 2 байта, её можно записать поверх любой другой инструкции и попасть в TRAP подпрограмму.

Однако при использовании команды TRAP возникают пять проблем:

- Окружение восстанавливается не полностью [5]
- В secondary address space mode команду TRAP вызвать нельзя [5]
- TRAP не может восстановить home space mode, если процессор не находится в привилегированном режиме (supervisor state) [5]
- Установка TRAP окружения для дочернего потока сопряжена с определенными трудностями: в родительском потоке мы не имеем доступа к структурам данных, где нужно выставить необходимые флаги и адреса

- При анализе многопоточного приложения придется устанавливать Trap-окружение для каждого subtask

C. Внесение ошибок и их перехват

Внесение ошибок и их перехват - очень распространенный метод организации отладки и перехвата переходов. Команда перехода заменяется, например, на код X'0000'. В z/OS имеется система для обработки прерываний (в том числе по ошибкам) - IH - Interrupt Handler, и система исключительно для обработки ошибок - RTM - Recovery Termination Manager [6]. Важно что IH вызывается первым в случае прерывания по ошибке.

z/OS предоставляет широкий спектр инструментов для перехвата ошибок (ESTAE, ESTAI, FRR) [8], макрос ESPIE [8] для перехвата прерываний.

Механизм ESTAE/ESTAI позволяет назначить двухступенчатый обработчик ABEND, состоящий из пользовательских подпрограммы восстановления и подпрограммы возврата [6].

При возникновении исключения управление передается таким образом: программа->RTM->подпрограмма восстановления->RTM->подпрограмма возврата [6]. В общем говоря, механизм ESTAE/ESTAI создан не для перехвата ошибок в чужих программах, что делает его не очень удобным инструментом для реализации трассировщика. Кроме того, лишь ESTAI официально позволяет обрабатывать ошибки в другом потоке. Хотя, есть "хакерский" способ назначить для другого потока и обработчик ESTAE. Стоит заметить, что при восстановлении после обработки ошибки ESTAI подпрограммой нельзя перейти в режим супервизора, если исследуемая программа исполнялась в режиме задачи. Этот факт приводит к следующим проблемам в подпрограмме возврата, а значит и в точке восстановления:

- Address space mode будет тот же что и у программы, вызвавшей ATTACH. Исследуемая программа же может изменить свой ASC, который должен не измениться после обработки перехода. В противном случае исследуемая программа может завершиться с ошибкой, при попытке работы, например, с secondary address space [6]
- Всегда разрешены IO interrupts и external interrupts. Запрет или разрешение IO прерывания могут быть важны, например, если исследуемая программа работает с канальной подсистемой [6]
- Режим адресации тот же что при вызове ATTACH , если не определен иной с помощью макроса SETRP [6]. Дело в том, что исследуемая программа может динамически изменить режим адресации и работать с 64-битными адресами. Внезапное переключение в 31-битную адресацию может плохо сказаться на ее работе
- Состояние флага CC (condition code) в PSW не определено [6]. Может нарушиться логика условных переходов в программе
- Состояние флага R в PSW не определено (PER enable) [6]. Программа может быть чувствительна к прерыванию PER
- Состояние флага T в PSW не определено (DAT enable) [6]. Программа может отключить DAT, а потом обнаружить его включенным, что нарушит логику ее работы

D. Замена SVC

Метод получил распространение в последнее время и в контексте организации трассировки называется HOOKING. В некотором смысле он является развитием метода внесения и перехвата ошибок. SVC - supervisor call - двухбайтная инструкция для осуществления системных вызовов [5].

Можно написать свою подпрограмму обработки SVC [6] и заменять в исследуемой программе все необходимые инструкции на команду ее вызова. Это в целом является достаточно удобным и быстрым механизмом.

Минусы данного подхода:

- SVC нельзя вызывать из SRB [5]
- SVC нельзя вызывать в cross memory mode [5]
- Существует проблема установки соединения с трассировщиком, что, впрочем, решаемая задача

Итак, выявлены основные подходы к трассировке переходов между модулями. С некоторыми ограничениями пригодны SVC, ESTAI, и комбинированные схемы SVC+ESPIE/SVC+TRAP. Комбинированные схемы используют SVC для установки ESPIE или TRAP окружения следующим образом:

- Заменяют обработчик SVC 242
- Заменяют команду в entry point исследуемой программы на инструкцию SVC 242
- В обработчике SVC 242 устанавливают TRAP/ESTAE/ESPIE и исполняют оригинальную команду

Это позволяет работать в том числе с приложениями secondary address space mode, например, с помощью ESPIE или ESTAE. Дополнительный статический анализ перед запуском приложения позволит определить конкретные методы пригодные для трассировки разных участков кода приложения.

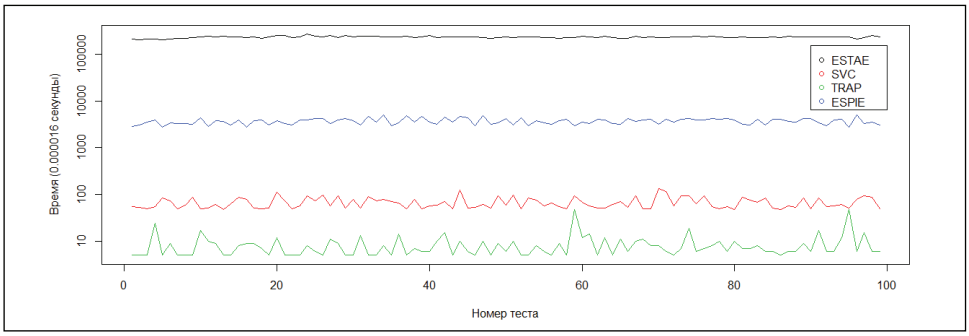


Рис. 1. График производительности, построенный по ста повторениям теста "применения" каждого метода трассировки на пятидесяти командах (единица измерения времени 0.000016 секунды)

IV. Оценка производительности предложенных методик

Для оценки производительности разработано приложение, реализующее четыре типа перехвата переходов:

- Внесение ошибки и ее перехват с помощью ESPIE
- Внесение ошибки и ее перехват с помощью ESTAI
- Замена целевой инструкции на TRAP2
- Замена целевой инструкции на SVC

Разработанное приложение состоит из пяти модулей:

- TESTSYS - модуль, написанный на языке C, предназначен для вызова тестов, форматирования полученных данных и вывода их в набор данных
- SVCUPD1 - модуль, написанный на HLASM, устанавливает корректное окружение для тестирования (ESTAI, ESPIE SVC,TRAP) и замеряет время работы APPRTEST
- APPRTEST - сам тестирующий модуль, написанный на HLASM, вызывает некоторое количество раз либо ошибочную команду, либо TRAP2, либо SVC
- Модуль, написанный на HLASM и содержащий TRAP ROUTINE
- Модуль, написанный на HLASM и содержащий SVC ROUTINE

Стократное повторение теста "применения" метода трассировки на пятидесяти командах показало, что самым производительным подходом является использование TRAP facility или SVC, это показано на рис. 1. Стоит отметить, что это основной метод в продукте TDF. Затем идут ESPIE и ESTAI. Реализация TRAP подхода сопряжена с определенной трудностью: отсутствием легкого доступа к привилегированному режиму. Это затрудняет организацию коммуникации между программой - монитором и генератором трассировочных событий. У SVC механизма такой проблемы нет, а реализация трассировщика с его использованием позволит в дальнейшем добавить поддержку TRAP.

V. Разработка трассировщика

Предполагается, что система имеет доступ к ADATA - информации о компиляции исследуемой программы.

Система состоит из двух программ:

- Анализатора исследуемой программы, использующего ADATA. Он выбирает команды перехода для их замены в дальнейшем на команду SVC и генерирует требуемую таблицу для AMASPZAP и программы-монитора
- Программы-монитора, которая устанавливает окружение, запускает исследуемую программу, собирает и сохраняет трассировочную информацию

Информация от анализатора монитору передается через временный набор данных. Анализатор строит две таблицы команд: одну для AMASPZAP, другую для программы-монитора. В первой содержатся смещения команд, которые нужно заменять, а во второй еще и их оригиналы. Для запуска системы написана JCL процедура. Она делает следующее:

1. Вызывает анализатор
2. Создает резервную копию библиотеки с исполняемыми файлами
3. Вызывает AMASPZAP, который меняет требуемые команды
4. Запускает программу-монитор

5. Восстанавливает библиотеку из резервной копии

Таким образом, чтобы реализовать перехват прыжков типа BC в программах 31-битного режима в primary address space mode для операционной системы z/OS система:

1. Найдет все вхождения команды BC в бинарном файле исследуемой программы путем просмотра ADATA и заменит их с помощью AMASPZAP на SVC 241
2. Установит свой обработчик SVC 241
3. Запустит исследуемую программу с подходящим набором входных данных
4. По окончании исполнения исследуемой программы сохранит трассировочную информацию в наборе данных

Этапы разработки:

- Анализатор ADATA
- Программа-монитор
- Обработчик SVC 241

В ходе разработки возникло две существенные проблемы:

- Получение монитором информации о модуле для организации прозрачной трассировки
- Установка связи обработчика SVC с монитором для сохранения и вывода трассировочных записей

Первая проблема может быть решена двумя способами:

- Изучение системных структур данных показало, что можно получить всю необходимую информацию о модуле самостоятельным просмотром системных структур данных: TCB, RB, LLE, CDE
- Оказалось, что существует макрос CSVINFO, который может дать лишь часть необходимой информации по имени модуля. Например, нельзя определить точку загрузки [9]. Поэтому первый способ оказался предпочтительнее

Вторая проблема решается с помощью использования специальных полей системной структуры DUCT [10] (она уникальна для каждой подзадачи), которые помечены как "For use by programming".

Предлагается следующий алгоритм:

1. Программа-монитор устанавливает обработчик SVC 241 и сохраняет адрес структуры данных, используемой для хранения смещений и оригиналов "испорченных" команд, в теле SVC рутины
2. Программа-монитор делает ATTACH [11] исследуемой программы
3. При вызове SVC241 проверяется, поднят ли TRAP флаг:
   a. Если нет, то создается (или расширяется под новый поток) структура для хранения трассировочных записей. Ее адрес хранится в памяти, содержащей код SVC, что затем позволяет легко организовать вывод собранной информации в датасет в конце работы трассировщика. Поднимается TRAP флаг. Адрес служебной области данных сохраняется в поле "For use by programming" системной структуры данных DUCT
   b. Если TRAP флаг поднят, то вызывается процедура добавления трассировочной записи. Адрес процедуры заранее сохранен в теле SVC рутины. В конце эмулируется исполнение оригинала команды, и управление возвращается

В некоторых ситуациях использование SVC невозможно, например, внутри SRB или если исследуемая программа исполняется в cross memory mode. В таких случаях можно заменять команды перехода на TRAP. А в случаях, которые не подходят и для SVC и для TRAP, например, если исследуемая программа работает в режиме secondary address space, можно использовать ESTAI. Это обеспечит наибольшую скорость и наиболее полный охват имеющихся вариантов. Однако это пока не реализовано.

Процедура добавления трассировочной записи определяет по имени модуля, адресу в памяти и ADATA координаты точки перехода и точки назначения - это векторы вида (MODULE NAME, CSECT NAME, OFFSET). Для увеличения скорости определения координат адреса модулей, их имена и имена CSECT хранятся в структуре данных, которая автоматически синхронизируется с системными структурами, как только в ней по какому-то запросу не нашлась соответствующая запись (на самом деле надо учитывать еще "внешние" воздействия на цепочку RB, но это уже более сложная задача).

Для трассировки многопоточного приложения используется один монитор на все потоки приложения и раздельные области памяти для сохранения трассировочных записей. Создан механизм инициализации, включающий в себя потокобезопасную процедуру выделения памяти под записи для нового потока. Указатель на данные специфичные для каждой конкретной подзадачи хранится в поле "use for programming" системной структуры DUCT, которая уникальна для каждого потока. Это позволяет получать к ним доступ из обработчика SVC без блокировок, мьютексов и просмотров очередей.

```
********************************* TOP OF DATA *********************************
THREAD=START====================
 DEPARTURE                      DESTINATION
 MDL NAME | CSECT NM | OFFSET | MDL NAME | csect NM | OFFSET |

SMPL1M1   | SMPL1M1  |      8 | SMPL1M1  | SMPL1M1  |     14 |

SMPL1M1   | SMPL1M1  |     34 | SMPL1M1  | SMPL1M1  |     80 |

SMPL1M2   | SMPL1M2  |      8 | SMPL1M2  | SMPL1M2  |     14 |

SMPL1M3   | SMPL1M3  |      8 | SMPL1M3  | SMPL1M3  |     14 |

SMPL1M4   | SMPL1M4  |      8 | SMPL1M4  | SMPL1M4  |     14 |

SMPL1M4   | SMPL1M4  |     5E | SMPL1M3  | SMPL1M3  |     5C |

SMPL1M3   | SMPL1M3  |     76 | SMPL1M2  | SMPL1M2  |     5C |

SMPL1M2   | SMPL1M2  |     76 | SMPL1M1  | SMPL1M1  |     B0 |

SMPL1M1   | SMPL1M1  |     CA |          |          |      0 |
THREAD=END======================

THREAD=START====================
 DEPARTURE                      DESTINATION
 MDL NAME | CSECT NM | OFFSET | MDL NAME | csect NM | OFFSET |

SMPL1M6   | SMPL1M6  |      8 | SMPL1M6  | SMPL1M6  |     14 |

SMPL1M6   | SMPL1M6  |     50 | SMPL1M6  | SMPL1M6  |     58 |

SMPL1M6   | SMPL1M6  |     58 | SMPL1M6  | SMPL1M6  |     54 |

SMPL1M6   | SMPL1M6  |     54 | SMPL1M6  | SMPL1M6  |     5C |

SMPL1M6   | SMPL1M6  |     84 |          |          |      0 |
THREAD=END======================

********************************* BOTTOM OF DATA ******************************
```

Рис. 2. Пример результата работы прототипа трассировщика на пятимодульном двухпоточном приложении

VI. Об использовании трассировщика

Система PARSE+SPYM позволяет трассировать команды BC многопоточного приложения 31-битного режима в primary address space mode для операционной системы z/OS. На вход системе подается библиотека с загрузочными модулями приложения (DD BACKUP.SYSUT1) и библиотека с соответствующей ADATA информаций (DD BLDTBL.ADATA). В параметре MAIN процедуры SPYMRUN указывается имя главного модуля.

На основе ADATA программа PARSE принимает решение о замене определенных команд на SVC 241, и генерирует два набора данных: один для AMASPZAP, другой для SPYM.

SPYM по набору данных строит таблицу команд, устанавливает SVC 241 и вызывает главный модуль исследуемой программы. По окончании ее работы информация записывается в DD GO.SPYM$OUT. Трассировочная запись представляет собой два вектора вида (MODULE NAME, CSECT NAME, OFFSET): один для точки перехода другой для точки назначения. Пример итоговой таблицы на рис. 2. Таблица позволяет построить некоторое покрытие кода, что может быть полезно для определенных задач динамического и статического анализа.

VII. Результаты

Разработано два приложения:

- Программа для оценки скорости различных методик трассировки
- Разработана система PARSE+SPYM, позволяющая трассировать команды BC в 31-битном режиме в primary address space mode для многопоточных приложений операционной системы z/OS с сохранением реентерабельности исследуемого приложения

Система реализована с использованием GCCMVS, HLASM, JCL и является open source. Кроме того, для использования с GCCMVS в среде z/OS реализована библиотека dirent.h (работа с библиотеками наборов данных) и создан ряд макросов (установка TRAP окружения, использование DYNAMIC ALLOCATION).

Система испробована на нескольких тестовых программах. При разработке каждая функция системы была протестирована в отдельном модуле с заданным окружением.

Возможна дальнейшая модификация системы, а следующие факторы облегчат развитие и поддержку:

- Использование C и HLASM, а не чистого ассемблера
- Реализация SVC hooking позволяет развивать комбинированные схемы трассировки (SVC+ESPIE или SVC+TRAP)
- Программа для оценки скорости трассировки реализует все основные методики, которые

фактически можно брать и переносить в PARSE+SPYM

Путем возможной замены модуля PARSE, расширения возможностей по эмуляции исполнения команд и добавления постпроцессинга трассы, подобная система может быть, например, использована в следующих случаях:

- Поиск состояния гонок по SLIP трассе изменений памяти и трассе работы примитивов синхронизации, полученной с помощью системы
- Модификация и отладка приложений с помощью трассы исполнения
- Построение профайлера для сбора информации о медленно работающих участках кода

Репозиторий: https://github.com/aton4eg/emc-spbsu-coop-mainframe

Литература

[1] IBM. Toolkit Feature Interactive Debug Facility User's Guide (GC26-8709-07). 2013

[2] David B. Cole. z/XDC® RELEASE GUIDE z/XDC®Release z1.13 for z/OS // ColeSoft Marketing, Inc. | No Guesswork. See the Code. 2011. www.colesoft.com URL: http://www.colesoft.com/documentation/z1d/pdf/z-xdc%20release%20guide%20%28z1.13%29.pdf (дата обращения 10.10.2014)

[3] Trap Diagnosis Facility Fact Sheet // Trap Diagnostic Facility. URL: http://www.arneycomputer.com/tdf/tdff.pdf (дата обращения 20.10.2014)

[4] Hercules Version 3: Frequently-Asked Questions. URL: http://www.hercules-390.org/hercfaq.html (дата обращения 29.10.2015)

[5] IBM. z/Architecture IBM Principles of Operation (SA22-7832-03). 2004

[6] IBM. z/OS MVS Programming: Authorized Assembler Services Guide (SA22-7608-15). 2010

[7] IBM. ABCs of z/OS System Programming, Том 10 (5694-A01). 2012

[8] IBM. MVS Programming: Authorized Assembler Services Reference, Volume 2 (EDT-IXG) (SA22-7610-18). 2010

[9] IBM. MVS Programming: Assembler Services Reference ABE-HSP (SA23-1369-00). 2013

[10] IBM // IBM: DUCT. URL: http://www.vm.ibm.com/pubs/cp510/DUCT.HTML (дата обращения 15.10.2014)

[11] IBM. MVS Programming: Authorized Assembler Services Reference, Volume 1 (ALE-DYN) (SA22-7608-15). 2010

Тестирование среды программирования роботов

Д.А. Мордвинов
Санкт-Петербургский государственный университет
Кафедра системного программирования
Email: mordvinov.dmitry@gmail.com

Ю.В. Литвинов
Санкт-Петербургский государственный университет
Кафедра системного программирования
Email: y.litvinov@spbu.ru

***Аннотация*—В статье представлен опыт тестирования достаточно сложной визуальной среды программирования роботов, написанной на C++ с использованием библиотеки Qt. Рассматриваются три аспекта тестирования среды — модульное тестирование, тестирование пользовательского интерфейса и функциональное тестирование посредством интерпретации программ, написанных в среде, на имитационной модели с заданными ограничениями на состояние модели. Приводится описание языка задания ограничений. Описываются другие задачи, помимо тестирования, где представленные в статье решения оказались полезны — код для тестирования пользовательского интерфейса переиспользован для реализации режима обучения, код для функционального тестирования переиспользован для проверки заданий учащихся онлайн-курса. Статья может быть полезна специалистом, занимающимся разработкой приложений на C++ с Qt и людям, занимающимся тестированием с использованием имитационных моделей физических объектов, которыми управляет программа.**

I. Введение

С 2012 года ведётся разработка робототехнического конструктора ТРИК[1] и графической среды программирования для него TRIK Studio [1]. Среда программирования появилась на базе наших исследований в области визуального моделирования ([2], [3], [4]) и их приложения к робототехнике ([5], [6], [7], [8]), так что на данный момент она представляет собой достаточно большую и богатую функциональными возможностями систему, имеющую несколько тысяч пользователей по всей стране и в мире. Среда предназначается прежде всего для обучения школьников и студентов программированию и робототехнике. Система имеет открытый исходный код, кроссплатформенная[2] (на момент написания статьи работает под управлением операционных систем Windows, Linux и Mac OS X), реализована на языке C++ с использованием библиотеки Qt 5[3], содержит как развитый пользовательский интерфейс, так и сложную внутреннюю логику работы, включающую в себя управление реальными роботами и исполнение программ на имитационной модели. Это создаёт некоторые сложности в организации автоматического тестирования системы. В этой статье будет рассказано, как эти сложности преодолевались и как вообще было организовано автоматическое тестирование, что может быть полезно читателю как с чисто технической точки зрения (тестирование приложений с пользовательским интерфейсом, реализованных с использованием библиотеки Qt), так и с научной (тестирование среды программирования роботов посредством проверки корректности исполнения программ на имитационной модели).

Среда TRIK Studio представляет собой средство визуального программирования роботов, то есть среду, где программы представляются в виде набора графических символов (или блоков), реализующих элементарные команды роботу (включить моторы, дождаться срабатывания датчика и т.д.). Блоки соединены стрелками, показывающими передачу управления. Такой подход довольно типичен для данной предметной области (ближайшие аналоги TRIK Studio — среды Robolab [9], NXT-G[4], MRDS[5]). Одна и та же нарисованная программа может быть исполнена на компьютере с посылкой команд реальному роботу, сгенерирована в программу на JavaScript, F# или C и передана роботу для исполнения в автономном режиме, или исполнена на имитационной модели, где симулируется робот и его окружение.

Автоматическое тестирование системы TRIK Studio состоит из трёх частей — модульного тестирования, тестирования пользовательского интерфейса и функционального тестирования посредством имитационной модели. Первые две части типичны для практически любых приложений с пользовательским интерфейсом, третья же специфична для предметной области робототехники — среда симулирует поведение реального робота и исполняет тестовые программы на нём, контролируя состояние среды симуляции посредством

[1] Домашняя страница конструктора ТРИК, URL: http://www.trikset.com/ (дата обращения: 17.08.2015)

[2] Репозиторий проекта на GitHub, URL: http://github.com/qreal/qreal/ (дата обращения: 17.08.2015)

[3] Домашняя страница библиотеки Qt, URL: http://www.qt.io/ (дата обращения: 17.08.2015)

[4] Среда программирования NXT-G, URL: http://www.lego.com/ru-ru/mindstorms/learn-to-program (дата обращения: 07.02.2015)

[5] Microsoft Robotics Developer Studio, домашняя страница, URL: https://msdn.microsoft.com/en-us/library/bb648760.aspx (дата обращения: 07.02.2015)

задания ограничений на её состояние, зависящих от времени и происходящих событий.

В статье будет рассказано про каждый из упомянутых выше этапов автоматического тестирования. Каждый этап представляет собой применение знаний из весьма обширной области, поэтому в данной работе не ставится цель подробно рассказать про модульное тестирование или тестирование пользовательского интерфейса вообще (это было бы слишком амбициозной задачей, к тому же уже решённой в многочисленных статьях и книгах, например, [10], [11]). Цель данной работы — представить опыт применения некоторых методик тестирования в реальном промышленном активно развивающемся проекте, представить и пояснить принятые нами решения.

II. Модульное тестирование

Для C++ существует множество библиотек модульного тестирования и единого стандарта не существует. Для выбора конкретной библиотеки мы руководствовались такими критериями, как скорость и простота написания теста, умение библиотеки работать с исключениями, наличие развитого механизма проверок, наличие поддержки создания mock-объектов.

Библиотека Qt, на основе которой разрабатывается TRIK Studio, имеет в своём составе библиотеку модульного тестирования Qt Test[6], но, несмотря на хорошую интеграцию с Qt, нам она не подошла по причине отсутствия встроенной поддержки работы с исключениями и mock-объектами. Также были рассмотрены библиотеки CPPUnit, Boost.Test, CppUnitLite, Unit++, CxxTest, Google C++ Testing Framework, подробное сравнение этих библиотек можно найти в работе [12]. В результате была выбрана библиотека Google C++ Testing Framework[7] и её расширение для написания mock-объектов Google C++ Mocking Framework[8].Эти инструменты удовлетворяют нашим критериям, кроме того, оказались просты в использовании, в том числе и в кроссплатформенном окружении.

При написании тестов мы пытались поддерживать соответствие «один исполняемый файл с тестами — один бинарный файл системы (разделяемая библиотека или исполнимый файл)», структура тестов примерно повторяет структуру основного проекта. При этом возникла проблема, что операционная система Windows не позволяет передавать в командной строке слишком длинные пути (более 260 символов), поэтому пришлось несколько упростить в тестах иерархию каталогов, иначе возникали трудности со сборкой. Модульные тесты не раз доказали свою эффективность, помогая искать ошибки на ранних этапах разработки и упрощая рефакторинг.

III. Тестирование пользовательского интерфейса

Одно лишь модульное тестирование, по крайней мере в нашем случае, не может покрыть все аспекты системы. Пользовательский интерфейс также подлежит проверке корректности. Данный раздел содержит описание результатов поиска подходящего решения, подхода к тестированию пользовательского интерфейса в TRIK Studio, а также обсуждение некоторых прикладных сторон технологии, полученной в результате данного исследования.

A. Существующие подходы

Тестирование графических интерфейсов — область, возраст которой составляет четверть века. За это время было опубликовано множество статей, проведена их подробная классификация (см. [10]), разработано большое количество подходов и инструментов, реализующих эти подходы. Данная работа не затрагивает фундаментальных теоретических оснований данной области, а скорее является практическим результатом на пересечении технологий, о которых будет рассказано в этом и последующих разделах.

TRIK Studio обладает сложным и постоянно изменяющимся интерфейсом, созданным на основе инструментария Qt Widgets[9]. Кроссплатформенность системы предъявляет важное требование к технологии тестирования интерфейса — одни и те же тесты должны иметь работать на различных операционных системах, оконных менеджерах, установленных разрешениях экрана и т.д. Открытость кода и свободное распространение среды делает непригодными коммерческие системы тестирования (например, Squish[10], который пригоден по всем другим параметрам). Приведем обзор технологий, с которыми мы столкнулись в процессе поиска конечного решения задачи тестирования интерфейсов.

1) Библиотека Qt Test: Инструментарий Qt содержит набор средств автоматизации взаимодействия с пользовательским интерфейсом приложения, предоставляя библиотеку Qt Test в базовом некоммерческом модуле. Данный инструмент позволяет автоматизировать взаимодействие с графическим интерфейсом посредством задержек и эмулирования низкоуровневых событий мыши и клавиатуры. Высокоуровневые операции поиска и взаимодействия с определенными элементами управления, инструменты их поиска и определения отсутствуют в библиотеке Qt Test, тем не менее

[6]Документация по Qt Test, URL: http://doc.qt.io/qt-5/qttest-index.html (дата обращения 17.08.2015)

[7]Домашняя страница Google C++ Testing Framework, URL: https://code.google.com/p/googletest/ (дата обращения: 07.02.2015)

[8]Домашняя страница Google C++ Mocking Framework, URL: https://code.google.com/p/googlemock/ (дата обращения: 07.02.2015)

[9]Документация по Qt Widgets, URL: http://doc.qt.io/qt-5/qtwidgets-index.html (дата обращения 17.08.2015)

[10]Домашняя страница Squish, URL: http://www.froglogic.com/squish/gui-testing/ (дата обращения 17.08.2015)

содержатся в том или ином виде в различных местах инструментария для создания самих интерфейсов.

В результате тест с использованием данной библиотеки представляет собой модуль на C++, требующий самостоятельной реализации частых операций (например, создание операции drag-and-drop из одного места в другое с последующим использованием результатов этой операции), а также перекомпиляции тестов при изменении пользовательского интерфейса, что затрудняет их поддержку. Сами по себе эти факторы не кажутся слишком проблематичными, однако на практике показали себя довольно затрудняющими процесс создания и поддержки тестов, код получался громоздким.

Тем не менее, формально данная технология удовлетворяет всем нашим требованиям, тесты не зависят от платформы, изменение интерфейса требует сопоставимых изменений тестов.

2) Windows Automation API: Для тестирования пользовательского интерфейса Windows-приложений часто используются возможности операционной системы Windows, а именно Windows Automation API[11]. Это набор функций, доступных как из .NET, так и для программ на C++, с помощью которых можно получить доступ к элементам пользовательского интерфейса, таким как кнопки, поля для ввода, списки и т.д., и симулировать события ввода. В составе Windows SDK[12] поставляются программы Inspect и AccScope, использующие Automation API для того, чтобы показывать структуру и свойства элементов управления приложений. Существуют и полноценные библиотеки GUI-тестирования, такие как White[13] или UI Automation Verify[14], скрывающие от программиста сложность использования Automation API и позволяющие удобно писать тестовые сценарии на .NET, работая в терминах элементов пользовательского интерфейса.

Основной недостаток подобного подхода в нашем случае — отсутствие кроссплатформенности и некоторая ориентированность на .NET (сам Automation API может использоваться и из кода на C++, но большинство библиотек к нему написаны на C#), что хоть и не делает невозможным, но усложняет использование такого подхода.

3) Sikuli: Sikuli [13] — это свободный инструмент, разрабатываемый в MIT для скриптования пользовательских действий на основе изображений отдельных областей экрана. Тест представляет собой скрипт на языке Python, где в качестве примитива может быть использовано изображение какого-либо фрагмента экрана. Инструмент предоставляет удобную среду программирования таких скриптов Sikuli IDE, которая позволяет быстро создавать и менять изображения прямо в коде теста.

Удобство написания простых тестов намного выше, чем при использовании библиотеки Qt Test, а также исчезает ряд проблем взаимодействия с операционной системой, однако данный инструмент не подошел нам из-за требований кроссплатформенности тестов. Очевидно, что на различных платформах и даже в разных версиях одной и той же платформы интерфейс приложения может выглядеть совершенно по-разному. При незначительной модификации интерфейса зачастую приходилось полностью изменять содержимое тестов. В итоге, несмотря на удобство и легкость написания простых тестов, данное обстоятельство делает Sikuli непригодным в нашем случае.

B. Скриптовый программный интерфейс

В результате так и не было найдено готового решения, способного удовлетворить всем поставленным ограничениям. Наиболее близкой к желаемому оказалась библиотека Qt Test, которая представляет собой скорее основу для технологии тестирования интерфейсов, чем саму технологию. Опыт использования описанных решений позволил нам сформировать представление о платформе, обладающей всеми достоинствами Qt Test в виде кроссплатформенности и возможности создания самой изощренной логики тестирования и простотой и лаконичностью скриптового подхода наподобие Sikuli.

В нашем представлении подходу с использованием Qt Test не хватает «скриптовости» и высокоуровневости в смысле работы в терминах элементов управления, а не событий от манипуляторов. Эти идеи были учтены при реализации системы тестирования интерфейсов TRIK Studio.

В качестве основного языка написания тестов был выбран Qt Script. Это реализация стандарта ECMAScript[15] (наиболее популярной реализацией которого является язык JavaScript), расширенная механизмами интеграции с Qt-приложениями. Скриптовое окружение расширено инструментами работы с пользовательским интерфейсом системы в терминах самого интерфейса. Работа этого окружения осуществляется посредством библиотек Qt Test и Qt Widgets, таким образом, сохраняются независимость тестов от платформы и легкость доступа ко всем элементам управления, при этом решаются проблемы громоздкости кода тестов и низкоуровневости терминов, в которых они работают.

[11] Документация Windows Automation API на MSDN, URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ff486375(v=vs.85).aspx (дата обращения 17.08.2015)

[12] Домашняя страница Windows 10 SDK, URL: https://dev.windows.com/en-us/downloads/windows-10-sdk (дата обращения 17.08.2015)

[13] Страница проекта White на GitHub, URL: https://github.com/TestStack/White (дата обращения 17.08.2015)

[14] Домашняя страница UI Automation Verify, URL: https://uiautomationverify.codeplex.com/ (дата обращения 17.08.2015)

[15] Домашняя страница ECMAScript, URL: http://www.ecmascript.org/ (дата обращения 17.08.2015)

Окружение предоставляет доступ ко всем частям интерфейса и методам работы с ними и содержит вспомогательную функциональность, специфичную только для TRIK Studio. К примеру, существует возможность скриптования создания визуальных диаграмм, рисования модели мира, в котором будет производиться имитационное моделирование робота, работы с жестами мышью, изменения свойств примитивов, из которых строится программа, работы с настройками приложения, управления состоянием вспомогательных панелей и т.д. Реализованы также методы произвольного доступа к элементам управления интерфейса по структурному описанию (идентификаторам и/или положению в дереве элементов управления) и координатам на экране. Управление эмуляторами устройств ввода (например, навигация и действия виртуальной мышью или указание клавиатурного фокуса и последовательностей нажатий на кнопки клавиатуры) тесно интегрировано с различными видами элементов управления. Например, присутствуют отдельные функции прокрутки и работы с выпадающими списками. При этом остаются доступными методы синтезирования низкоуровневых событий манипуляторов.

Кроме всего описанного, скриптовый интерфейс системы содержит методы показа информационных сообщений, подсвечивания отдельных элементов управления и рисования стрелок-указателей на них, анимации виртуального курсора мыши и тому подобных «показательных» функций, надобность которых будет обсуждаться в разделе III-D. Тем не менее, остаются открытыми некоторые проблемы, каждая из которых носит чисто технический характер. Одна из них — управление системными диалоговыми окнами. Дело в том, что на каждой платформе системные диалоговые окна (такие как диалоги выбора файлов, печати, выбора цвета и т.д.) свои и не поддаются контролю системы синтезирования событий, используемой в библиотеке Qt Test. Сейчас эта проблема обходится работой с этими диалогами как с «черными ящиками» — тесты задают для себя результаты выбора сущностей, редактируемых этими диалогами, не прибегая к работе с ними, «обходя», таким образом, их на уровень выше. Другая проблема — отсутствие в Qt Test возможности эмулирования ввода с клавиатуры на разных языках, что делает невозможным полноценное тестирование локализации среды. Все эти проблемы являются, скорее, проблемами библиотеки Qt Test, в то время как с нашей стороны единственной на данный момент проблемой является стоимость поддержки кодовой базы, что, безусловно, являясь значимым, не будет обсуждаться в рамках этой статьи.

Полученная технология оказалась практически полностью удовлетворяющей нашим потребностям в удобстве написания тестов и их переносимости. Важно, что разработкой автоматизированного тестирования интерфейса смогли заниматься люди, практически не имеющие отношения к разработке и поддержке самой системы и не имеющие опыта программирования на C++.

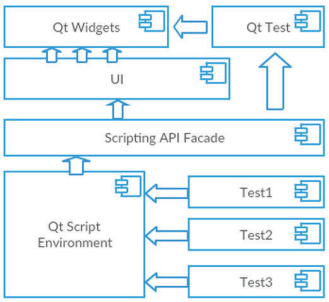


Рис. 1: Архитектура системы тестирования интерфейсов TRIK Studio.

C. Инфраструктура

В данном разделе будет кратко описана структура технологии тестирования, описанной в разделе III-B. Общая архитектура всего комплекса представлена на рисунке 1. Основная идея состоит в создании новой «прослойки» системы, стоящей над всей технологией, интерфейса программирования поведения пользовательского интерфейса. Этот уровень не модифицирует кодовую базу системы, тем не менее агрегируя в себе все ее отдельные компоненты. Здесь реализуются все операции, специфичные только для данной платформы в удобном для использования в скриптах виде, а также происходит «оборачивание» операций библиотеки Qt Test. Интерпретатор Qt Script, занимающийся исполнением кода самих тестов, импортирует символы скриптового API. Очевидно, что необходимости в Qt Script в такой архитектуре нет, можно создавать тесты и на компилируемых языках, компонуясь с интерфейсом программирования, в этом случае такой новый уровень можно рассматривать просто как инструмент повторного использования кода в тестах.

D. Использование технологии для обучения пользователей

Полученная технология обладает одним интересным приложением. На базе одной и той же инфраструктуры могут быть получены как тесты пользовательского интерфейса, так и автоматизированные демонстрации процесса работы со средой, встроенные прямо в нее. Это способ обучения пользователя возможностям среды, используемый, в основном, в компьютерных играх — про возможности продукта рассказывает сам продукт (в противоположность, например, таким подходам как текстовая справка или видео уроки).

Возможно, такой подход будет полезен разработчикам приложений с развитым, нестандартным графическим интерфейсом. Написанием скриптов тестирования и скриптов демонстрации могут заниматься одни и те же люди, даже не участвующие в написании кода самого приложения.

E. Обсуждение

Дальнейшее развитие этой технологии предполагает генерацию скриптов симуляции поведения пользователей. В самом простом случае тесты графического интерфейса и встроенные демонстрации возможностей среды можно создавать, записывая действия пользователя, работающего со средой. Если же требуется более сложная логика работы сценария, её можно описать с помощью специального визуального языка, по которому будет автоматически сгенерирован код скрипта тестирования. На данный момент эти технологии реализованы в системе TRIK Studio и проходят апробацию, но в силу того, что окончательный результат ещё не получен, они не будут подробно обсуждаться в рамках данной работы.

Разумеется, создание специального визуального языка для нужд тестирования пользовательского интерфейса «с нуля» было бы неоправданно трудозатратно. Однако такой подход полностью соответствует парадигме предметно-ориентированного моделирования (Domain-Specific Modeling [14]), а визуальный язык и инструментальные средства к нему можно быстро реализовать с помощью так называемых DSM-платформ ([15], [16]). Проект, на котором основывается TRIK Studio, QReal — это пример такой DSM-платформы (на самом деле, сам язык программирования роботов и редактор к нему были созданы с помощью QReal в течение нескольких часов). Создать ещё один визуальный язык для тестирования пользовательского интерфейса TRIK Studio не представляло особой сложности.

Идея записи пользовательских действий с последующей генерацией теста уже находила свою реализацию, например, в проекте TPTP[16] тестирования приложений, созданных в среде разработке Eclipse[17]. Более того, существуют работы, в которых эта методика расширяется добавлением элементов искусственного интеллекта, например [17], [18]. Интеграция подобных подходов в наше решение является интересной темой дальнейших исследований.

IV. Функциональное тестирование

A. Краткое описание возможностей среды

Далее в этом разделе пойдёт речь о тестировании функциональности, специфичной для TRIK Studio (и аналогичных систем, предназначенных для управления некоторым объектом), поэтому следует описать подробнее функциональность, которую требуется тестировать.

TRIK Studio представляет собой среду программирования роботов на визуальном языке. Система поддерживает программирование робототехнических конструкторов Lego Mindstorms NXT 2.0, Lego Mindstorms EV3 и ТРИК, в разработке находится поддержка платформы Arduino. Для каждой из поддерживаемых платформ имеется возможность отладки визуальной программы на компьютере с посылкой команд роботу, генерации кода на каком-либо текстовом языке программирования (например, C, JavaScript, F#, ШАЯ) с последующей загрузкой кода на робота по различным интерфейсам (Bluetooth, USB, Wi-Fi) для автономного исполнения. Существует также возможность отладки программы на двумерном виртуальном исполнителе. Общий вид интерфейса среды представлен на рисунке 2

Программа представляется в виде набора блоков, каждый из которых представляет собой элементарную команду роботу или элементарный шаг вычисления. Команды роботу могут быть как простыми (например, включить мотор на таком-то порту с такой-то мощностью), так и довольно сложными (например, получить данные датчика линии, использующего видеокамеру и алгоритмы распознавания изображений), сложные алгоритмы, как правило, реализуются на самом роботе и среда лишь вызывает их и обрабатывает результаты. Во всех блоках можно использовать математические выражения на текстовом языке, представляющем собой подмножество языка Lua 5.3[18], синтаксический анализатор и интерпретатор которого были специально реализованы в TRIK Studio (готовые решения нас не устраивали тем, что невозможно было использовать их абстрактное синтаксическое дерево для генерации кода на других языках программирования, кроме того, над Lua пришлось реализовать свою систему типов, потому что некоторые целевые языки генерации, такие как C, статически типизированы). В выражениях можно использовать текущие значения датчиков робота, доступные как специальные переменные, они же отображаются в виде таблицы или графиков при запущенной в режиме интерпретации программе. В режиме генерации и автономного исполнения обращения к таким переменным транслируются в команды чтения значений датчиков и их показания на компьютере не отображаются (чтобы дать возможность программе исполняться автономно).

Двумерная модель позволяет исполнять большую часть программ, которые могут быть исполнены на реальном роботе. Пользователь может нарисовать в области редактирования двумерной модели стены, цветные

[16] Страница TPTP GUI Recorder, URL: http://www.eclipse.org/tptp/test/documents/userguides/Intro-Auto-GUI.html (дата обращения 17.08.2015)

[17] Домашняя страница проекта Eclipse, URL: https://eclipse.org/ (дата обращения 17.08.2015)

[18] Домашняя страница языка Lua, URL: http://www.lua.org/ (дата обращения 17.08.2015)

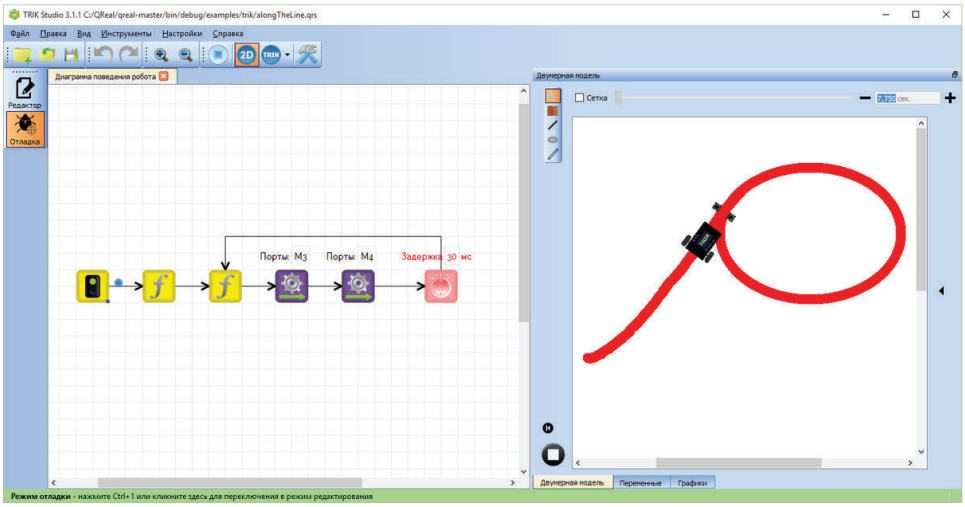


Рис. 2: Интерфейс TRIK Studio.

линии или области на полу, разместить самого робота и его датчики, задать им направление. Датчики будут взаимодействовать со стенами и цветными линиями (для ТРИК поддержаны инфракрасные и ультразвуковые датчики расстояния, датчик освещённости и детектор линии по видеокамере, для Lego NXT или Lego EV3 — все датчики, входящие в стандартный набор: касания, света, цвета и ультразвуковой датчик расстояния). Часть функциональности, доступная только на реальном роботе, имитационной моделью не поддерживается, например, на данный момент на имитационной модели может быть только один робот, поэтому в поддержке функциональности общения между роботами нет смысла. Тем не менее, имитационная модель позволяет решать большинство задач из курсов по робототехнике для Lego NXT (например, [19]) и ТРИК[19], и существенно упрощает отладку, давая возможность отладить алгоритм без реального робота, а потом лишь подобрать коэффициенты для исполнения той же программы в реальном мире.

Отдельно стоит отметить наличие достаточно развитых средств обеспечения удобства пользовательского интерфейса, что связано с образовательной направленностью среды. Например, при рисовании диаграмм можно пользоваться механизмом распознавания жестов мышью ([20]) — с зажатой правой кнопкой рисуется схематичная фигура, которая распознаётся средой и на месте нарисованной фигуры создаётся соответствующий ей блок, что позволяет не искать нужный блок в палитре и сделать использование среды интереснее для детей. Для TRIK Studio проводились исследования по юзабилити и вносились правки в интерфейс по их результатам ([21], [22]).

Генерация кода для автономного исполнения на данный момент реализована для Lego NXT в язык C (с использованием библиотеки ECRobot[20] и ОСРВ nxtOSEK[21]) и для ТРИК в языки JavaScript ([23]) и F# ([24]). Для ТРИК также реализованы библиотеки поддержки времени выполнения на C++ и F#, которые работают непосредственно на роботе и, собственно, реализуют функции, вызываемые программами, сгенерированными в TRIK Studio.

B. Общая схема функционального тестирования

Поскольку имитационная модель покрывает достаточно большой процент функциональности среды и ввиду сложности использования реального робота в автоматических тестах, именно имитационная модель используется нами для функционального тестирования. Обычно в аналогичных ситуациях пишется эмулятор реального устройства, у нас же эмулятор фактически является неотъемлемой частью системы и активно используется конечными пользователями. Предлагаемый подход к функциональному тестированию TRIK Studio таков:

[19] например, набор уроков И.Ю. Широколобова, URL: http://bit.ly/TRIK_presentations (дата обращения 17.08.2015)

[20] Домашняя страница ECRobot, URL: http://lejos-osek.sourceforge.net/html/index.html (дата обращения 17.08.2015)

[21] Домашняя страница nxtOSEK, URL: http://lejos-osek.sourceforge.net/ (дата обращения 17.08.2015)

1) На визуальном языке рисуется программа для исполнения на имитационной модели, решающая ту или иную практическую задачу, а также рисуется модель мира для имитационной модели, с которым эта программа будет работать.
2) Программа снабжается описанием ограничений на состояние системы на специальном XML-языке, описание которого приводится ниже. Наборов ограничений для одной программы может быть несколько, при этом они могут использовать разные модели мира.
3) Файл с сохранённой программой запускается на интерпретацию в двумерной модели, для этого используется специально созданный для нужд тестирования исполняемый файл, представляющий собой по сути TRIK Studio без пользовательского интерфейса. Тестирующая программа перед запуском тестируемой программы подменяет модель мира и набор ограничений в сохранении, что даёт возможность использовать несколько тестов для одной программы. Программа исполняется так, как она исполнялась бы, будучи запущенной пользователем в TRIK Studio, и в случае, если одно из ограничений не выполнено, программа работает некорректно — делается вывод, что поведение интерпретатора или среды имитационного моделирования отличается от эталонного, тест не пройден.
4) Тестирующая программа вызывается системой непрерывной интеграции Travis[22] каждый раз после сборки проекта, для всех файлов сохранений и моделей мира для них, имеющихся в репозитории.
5) Результаты тестов рассылаются по электронной почте разработчикам.

Архитектурно тестирующая система представляет собой отдельную программу, использующую в качестве разделяемых библиотек ядро и имитационную модель системы TRIK Studio. При этом она активно использует рефлексию, реализованную для C++ в библиотеке Qt для доступа к свойствам объектов имитационной модели. Таким образом, при расширении возможностей системы или добавлении нового вида симулированного оборудования новые объекты и свойства будут немедленно доступны из описаний ограничений, без дополнительной модификации кода. Тестовая система подписывается на события имитационной модели и вызывает проверку ограничений на каждом «тике» модельного времени. При этом проверяются только активные ограничения, которых в каждый конкретный момент времени немного, поэтому скорость работы оказывается вполне удовлетворительной.

[22]Система непрерывной интеграции Travis, URL: https://travis-ci.org/ (дата обращения 17.08.2015)

C. Описание языка задания ограничений

Описание ограничений на имитационную модель робота производится на событийном языке, основанном на XML, разработанном как дополнение к механизму имитационного моделирования в среде. Выбор в пользу XML-фронтэнда был произведен из чисто практических соображений удобства интеграции с описанием остальной имитационной модели.

Программа на таком языке представляет собой множество $\{e_1, e_2, ..., e_n\}$ событий, где каждое событие e_i — это тройка (id_i, c_i, T_i).

- id_i — идентификатор события, внутренняя метка, по которой к событию e_i смогут обращаться другие события.
- c_i — условие срабатывания события, формула логики первого порядка, о предикатных и функциональных символах которой будет рассказано чуть ниже.
- T_i — упорядоченный список элементарных триггеров $[t_{i1}, t_{i2}, ..., t_{in}]$. Элементарный триггер — это некоторое действие, выполняемое в момент истинности условия срабатывания события c_i.

Одному событию соответствует один элемент в XML-спецификации программы. Каждое событие в текстовом описании программы может быть представлено либо в виде ограничения, либо в каноническом виде.

Событие в канонической форме — уже описанная тройка (id_i, c_i, T_i).

Событие в форме ограничения — это событие вида $(id_i, !c_i, [fail(message_j)])$, где «!» означает логическое отрицание, а $fail(message_j)$ — триггер прекращения исполнения имитационной модели с выдачей сообщения об ошибке $message_j$. Другими словами, ограничение — это событие, которое срабатывает тогда, когда нарушается какое-то условие, и которое сообщает об этом нарушении. Особый случай такого ограничения — лимит времени на выполнение программы. Такое ограничение должно присутствовать в любой программе проверки ограничений имитационной модели TRIK Studio (что проверяется на уровне самого языка задания ограничений), так как проверка, очевидно, не может осуществляться бесконечное время.

Хоть в выделении ограничений в отдельную группу событий особой надобности нет, на практике удобно описывать утверждения вида «робот x не должен выезжать за пределы зоны z» или «к роботу x должен быть подключен набор датчиков s» именно в терминах ограничений, а не событий.

Коротко опишем множество доступных в языке предикатных и функциональных символов и элементарных триггеров. Предикатные символы можно поделить на несколько групп:

- Предикаты сравнения значений функциональных символов $>, <, <=, >=, =, ! =$.

- Пространственные предикаты. Имеют вид «тело x находится внутри области y».
- Предикаты состояния событий $settedUp(id_i)$ и $dropped(id_i)$, описывающие состояние события (взведено/опущено). Во взведенном состоянии событие может быть выполнено, в опущенном оно остается неактивным, не выполняя триггеры даже при выполнении условия срабатывания события.
- Предикат времени $timer(t)$, истинный в моменты времени t и позже.
- Прочие предикаты, являющиеся надстройками над уже описанными и поддержанные в целях удобства.

Функциональные символы бывают следующих видов:

- Константы различных типов (целочисленные, с плавающей точкой, строковые, логические, цветовые, геометрические и пр.).
- Символ $variableValue(id)$ для получения значения переменной с идентификатором id. Переменные могут быть полезны при сложной логике проверки, например, при подсчете числа проделанных роботом итераций.
- Арифметические и геометрические операции над значениями других символов, например, модуль числа, подсчет расстояния между двумя точками или выпуклая оболочка множества точек.
- Символы сравнения формы двух объектов с использованием расстояния Левенштейна, полезные при проверке схожести фигур, нарисованных роботом.
- Символ $objectState(path)$ — основное средство получения информации о состоянии устройств робота и свойствах предмета из внешнего мира. Аргумент $path$ — путь к желаемому свойству в иерархии объектов в модели мира. Подробный рассказ обо всех свойствах, предоставляемых окружением, довольно громоздкий и выходит за рамки данной статьи.

Наконец, элементарные триггеры делятся на следующие категории:

- $success$, $fail(message)$ — управление состоянием проверки. Первый помечает результат проверки как успешный, второй — как неудавшийся, отображая заданное сообщение об ошибке.
- Триггеры задания значения переменных и изменения значения свойств объектов из имитационной модели мира.
- Триггеры управления состоянием событий. Каждое событие может взвести или опустить другое или себя, с помощью этого механизма можно задавать сложную логику проверки.

Пример простейшей программы на языке задания ограничений имитационной модели мира в TRIK Studio приведён в листинге 1.

D. Использование для проверки заданий в онлайн-курсе

Та же система, что используется для тестирования среды, используется и для проверки задач, решаемых пользователями в рамках онлайн-курса по робототехнике, подготавливаемого в данный момент для платформы Stepic[23]. На сервере размещён набор задач в виде файлов сохранения TRIK Studio, в которых готова (и закрыта для модификации) модель мира для имитационной модели, некий набор ограничений, но не до конца нарисована программа (или отсутствует вовсе). Учащимся предлагается скачать файл с сохранением, открыть его в TRIK Studio, дорисовать программу, проверить, что программа работает у них в имитационной модели и удовлетворяет ограничениям, записанным в файле сохранения, и загрузить программу на сервер.

На сервере используется та же программа тестирования, которая подменяет модель мира и набор ограничений на тестовые и запускает на исполнение пользовательскую программу. Возможность подменить модель мира позволяет проверить программу на работоспособность в различных (возможно, не известных пользователю заранее) условиях, а различные ограничения позволяют управлять случайными факторами, влияющими на работу программы (например, датчиком случайных чисел, или симулировать ввод пользователя). По результатам пользователю отправляется сообщение с результатом проверки.

Результат работы программы может быть отображён в браузере пользователя с помощью JavaScript-программы, рисующей двумерную модель и поведение робота в ходе выполнения задачи. Для этого сервер также готовит и отправляет по запросу файл с трассой робота — набором пройденных им точек и состоянием подключенных к нему устройств. Некоторые задачи могут быть решены прямо в браузере, без открытия сохранения в TRIK Studio, для этого используется среда программирования, частично реализующая возможности TRIK Studio на JavaScript. Среда позволяет нарисовать диаграмму, после чего формирует файл сохранения TRIK Studio, который открывается системой проверки.

E. Результаты

На данный момент с помощью предлагаемой технологии реализовано 20 тестовых задач и тестовая утилита, запускающая все задачи и контролирующая корректность работы системы. Эти же тестовые задачи используются как упражнения для онлайн-курса[24] и покрывают функциональность от самой простой (проехать вперёд) до довольно сложной (движение по лаби-

[23] Платформа онлайн-обучения Stepic, URL: https://stepic.org/ (дата обращения 17.08.2015)

[24] пример такой задачи доступен по ссылке http://bit.ly/online_task_demo (дата обращения: 01.11.2015)

```
<!-- Корневой элемент, означающий начало программы проверки ограничений -->
<constraints>

    <!-- Обязательное в любой программе ограничение на время работы -->
    <timelimit value="2000"/>

    <!-- Ограничение на местоположение робота -->
    <constraint failMessage="Робот покинул допустимую зону!">
        <inside objectId="robot1" regionId="warzone"/>
    </constraint>

    <!-- Условие успешности программы: робот должен сказать "Привет" с помощью
        встроенного механизма синтезирования речи и нарисовать улыбку на дисплее -->
    <event settedUpInitially="true">
        <conditions glue="and">
            <equals>
                <objectState object="robot1.shell.lastPhrase"/>
                <string value="Привет"/>
            </equals>
            <equals>
                <objectState object="robot1.display.smiles"/>
                <bool value="true"/>
            </equals>
        </conditions>
        <trigger>
            <success/>
        </trigger>
    </event>

</constraints>
```

Листинг 1: Пример программы проверки ограничений имитационной модели мира в TRIK Studio.

ринту с использованием ультразвукового датчика расстояния). Все блоки, доступные пользователю, встречаются хотя бы один раз в наборе тестов, кроме того, тесты фактически строятся по материалам годового курса обучения школьников, что даёт уверенность в удовлетворительном покрытии типичных случаев использования имитационной модели. Прогон полного набора существующих на данный момент тестов занимает порядка минуты, что позволяет запускать тесты после каждого коммита в систему контроля версий.

На создание задач, ограничений на XML-языке и эталонных решений, которые, собственно, и используются для тестирования среды, ушло примерно две человеко-недели. В результате тестирования было выявлено и исправлено порядка 10 дефектов имитационной модели, в том числе и редко проявляющиеся при «обычном» использовании проблемы с зависимостью показаний, возвращаемых некоторыми датчиками, от случайных факторов. Таким образом, технология уже оказалась нам полезна.

V. Заключение

В статье было рассмотрено применение ряда методик тестирования в проекте TRIK Studio. Было показано, что некоторые технологии полезны не только для тестирования, но и для реализации пользовательской функциональности: средство для тестирования пользовательских интерфейсов переиспользовано для реализации режима обучения пользователей, а средство для тестирования работы интерпретатора программ и имитационной модели — для проверки заданий учащихся в онлайн-курсе.

Подходы, рассматриваемые в этой работе допускают интересные, на наш взгляд, обобщения и углубленное исследования. В частности, интересными представляются идеи использования формальной верификации по методу Model Checking в функциональном тестировании среды программирования. Данное направление исследований интересно еще и потому, что на момент написания статьи находится в разработке событийный визуальный язык для программирования сложных мультиагентных систем, где результаты такого исследования, как представляется, могут быть эффективно применены.

Список литературы

[1] Литвинов Ю.В., Кириленко Я.А. TRIK Studio: среда обучения программированию с применением роботов // V Всероссийская конференция «Современное технологическое обучение: от компьютера к роботу» (сборник тезисов). ЗАО «Полиграфическое предприятие № 3», 2015. С. 5–7.

[2] QReal DSM platform-An Environment for Creation of Specific Visual IDEs. / Anastasiia Kuzenkova, Anna Deripaska, Timofey Bryksin [и др.] // ENASE. 2013. С. 205–211.

[3] Средства быстрой разработки предметно-ориентированных решений в metaCASE-средстве QReal / А.С. Кузенкова, А.О. Дерипаска, К.С. Таран [и др.] // Научно-технические ведомости СПбГПУ, Информатика, телекоммуникации, управление. 2011. № 4 (128). С. 142–145.

[4] Архитектура среды визуального моделирования QReal / А.Н. Терехов, Т.А. Брыксин, Ю.В. Литвинов [и др.] // Системное программирование. 2009. № 4. С. 171–196.

[5] Терехов А.Н., Брыксин Т.А., Литвинов Ю.В. Среда визуального программирования роботов QReal:Robots // III Всероссийская конференция "Современное технологическое обучение: от компьютера к роботу"(сборник тезисов). 2013. С. 2–5.

[6] Литвинов Ю.В. Визуальные средства программирования роботов и их использование в школах // Современные информационные технологии и ИТ-образование, сборник избранных трудов VII Международной научно - практической конференции. ИНТУИТ.РУ, 2012. С. 858–868.

[7] Тихонова М.В. Среда программирования QReal:Robots // Список-2012: Материалы всероссийской научной конференции по проблемам информатики. 25-27 апр. 2012г, Санкт-Петербург. Изд-во ВВМ, 2012. С. 70–75.

[8] Брыксин Т.А., Литвинов Ю.В. Среда визуального программирования роботов QReal:Robots // Материалы международной конференции "Информационные технологии в образовании и науке". Самарский филиал МГПУ, МГПУ, 2011. С. 332–334.

[9] Portsmore Merredith. ROBOLAB: Intuitive Robotic Programming Software to Support Life Long Learning // APPLE Learning Technology Review. 1999.

[10] Graphical user interface (GUI) testing: Systematic mapping and repository / Ishan Banerjee, Bao Nguyen, Vahid Garousi [и др.] // Information and Software Technology. 2013. Т. 55, № 10. С. 1679–1694.

[11] Котляров В.П., Коликова Т.В. Основы тестирования программного обеспечения. Бином. С. 285.

[12] Бажутин М.С. Фреймворки юнит-тестирования для C++ // курс. работа. 2012.

[13] Yeh Tom, Chang Tsung-Hsiang, Miller Robert C. Sikuli: Using GUI Screenshots for Search and Automation // Proceedings of the 22Nd Annual ACM Symposium on User Interface Software and Technology. UIST '09. New York, NY, USA: ACM, 2009. С. 183–192. URL: http://doi.acm.org/10.1145/1622176.1622213.

[14] Mernik Marjan, Heering Jan, Sloane Anthony M. When and how to develop domain-specific languages // ACM computing surveys (CSUR). 2005. Т. 37, № 4. С. 316–344.

[15] О средствах разработки проблемно-ориентированных визуальных языков / А.А. Павлинов, Д.В. Кознов, А.Ф. Перегудов [и др.] // Системное программирование. 2006. Т. 2, № 1. С. 116–141.

[16] Kelly Steven, Tolvanen Juha-Pekka. Domain-specific modeling: enabling full code generation. Wiley-IEEE Computer Society Press, 2008. С. 446.

[17] Memon Atif M, Pollack Martha E, Soffa Mary Lou. Hierarchical GUI test case generation using automated planning // Software Engineering, IEEE Transactions on. 2001. Т. 27, № 2. С. 144–155.

[18] Sugiura Atsushi, Koseki Yoshiyuki. Simplifying macro definition in programming by demonstration // Proceedings of the 9th annual ACM symposium on User interface software and technology / ACM. 1996. С. 173–182.

[19] Филиппов С.А. Робототехника для детей и их родителей. НАУКА, 2013. С. 319.

[20] Osechkina Maria, Litvinov Yuri, Bryksin Timofey. Multistroke Mouse Gestures Recognition in QReal metaCASE Technology // SYRCoSE 2012: Proceedings of the 6th Spring/Summer Young Researchers' Colloquium on Software Engineering. ISPRAS, 2012. С. 194–200.

[21] Соковикова Н.А. Usability в проекте QReal:Robots // Список-2012: Материалы всероссийской научной конференции по проблемам информатики. 25-27 апр. 2012г., Санкт-Петербург. Изд-во ВВМ, 2012. С. 66–69.

[22] Кузенкова А.С. Анализ пользовательского интерфейса QReal:Robots // курс. работа. 2013. URL: http://se.math.spbu.ru/SE/YearlyProjects/2013/445/445-Kuzenkova-report.pdf.

[23] Terekhov Andrey, Litvinov Yurii, Bryksin Timofey. QReal: Robots an environment for teaching computer science and robotics in schools // Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia / ACM. 2013. С. 10.

[24] Kirsanov Alexander, Kirilenko Iakov, Melentyev Kirill. Robotics reactive programming with F#/Mono // Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia / ACM. 2014. С. 16.

Generation of Test Scenarios for Non Deterministic and Concurrent Telecommunication Applications

Kotlyarov Vsevolod, Drobintsev Pavel, Voinov Nikita
Peter the Great Saint-Petersburg Polytechnic University
Saint-Petersburg, Russia
vpk@spbstu.ru

***Abstract*— The paper presents an approach to tests generation for industrial software systems with non deterministic and concurrent behavior. A brief overview of modern model driven test technologies is presented; benefits and problems of these approaches are highlighted. Specifics of concurrent and non deterministic behavior are analyzed to identify issues with such behavior testing. As result of the issues analysis usage of non-linear symbolic test scenarios for reducing test suite size is suggested and presented in examples. Based on the suggestion an approach for construction of non linear tests from linear ones is described with an example of industrial project. Results with description of main benefits for suggested approach are presented.**

Keywords*— *Model Driven Testing; Automation design of test scenarios; Reducing of tests explosion.

I. Introduction

Creation of technologies for software quality assurance is one of the most actively growing areas of software developing process. A lot of developing technologies for support of quality guarantying process are based on formal approach. Such technologies are called model-oriented and lead to creation of application formal model with usage of graphical language, verification or some checking of the model and different types of generation. If formal representation of software system is used for generation of application target code then such approaches are called MDD (Model Driven Development) [1], MDSD (Model Driven Software Development) [2] or MDE (Model Driven Engineering)[3]. In case of formal model is used for testing automation purposes such as scripts or test generation based on behavior described in the model the approach can be called MDT (Model Driven Testing) [4] or MBT (Model Based Testing) [5].

The paper is devoted to description of some experience in usage of model-oriented technologies for testing of industrial telecommunication projects. The experience is based on usage of tests generated from formal models for checking of non deterministic and concurrent systems quality. Existing approach is based on linear test generation and execution, paper provides enhancement for such approach with non linear tests producing.

II. Model Oriented Aproach in Testing Automation

Usage of a model-oriented testing is based on creation of formal models which can be used on different testing phases. The main feature of such approaches is generation of test suite in accordance with user defined criteria of coverage.

Statistics collected in companies which use such approaches shows [6] that model-oriented techniques are usually used on system testing phase (up to 80% of projects) with the main goal - functional testing (up to 96%). The reason of such company's approach is complexity of system testing for big industrial projects, which is based on huge efforts, spent to quality guarantying [7]. To resolve this problem software developing companies try to reduce efforts for tests creation and simplify tests execution process. Usually reducing of testing efforts is linked to communication with customers because only customer of the software has deep knowledge about domain specifics and model oriented approach helps to simplify such communications.

Researchers also consider that more than 80% [6] of model-oriented approaches are using graphical notations, which allows simplifying of work with formal notations for developers. Requirements for knowledge of testers and customer representatives are reduced by this way and process of models developing is also simplified.

The following advantages of model-oriented approaches in comparison with manual test development methods can be found in research papers [6]:

- Reduction of software development cost due to usage of testing automation and verification techniques;
- Ability to use abstract models, which allow to simplify testing development process and involve customer in work with software quality checking;
- Tests generation and execution automation, which allow to reduce cost of testing process;
- Ability to have communication with customer on requirements level starting from earlier phases of development process, which allow to identify inconsistencies on stage of requirements gathering before system code implementation;

The following issues are usually highlighted in process of MDT implementation in software development process [6]:

- Different levels of abstraction in formal model and code of developing software. The issue leads to necessity of generated tests customization before execution on target software code;
- Necessity to customize MDT approaches in projects from different areas of industry. Usually practices and techniques from one target domain are not applicable in another one, this paper presents some unified approach that can be used in different domains because it is based on syntax of the model and not on semantics;
- A shortage of engineers with MDT expertise due to the lack of such specialists in the market of software developers and the need to train qualified personnel for the implementation of MDT approaches within the software development process of companies. Approach presented in the paper is based on very simple and intuitive formal language and this helps to resolve issue with strong MDT expertise;
- The problem of test suite explosion. A potential number of tests generated with MDT for checking of industrial software quality is too huge but the time for testing cycle is restricted. The main target for approach presented in the paper is reducing of test suite in case of non deterministic and concurrent behaviors presented in formal specification.

TABLE I. COMPARATIVE ANALYSIS OF MDT AND MANUAL APPROACHES

Testing phase	**MT**		**MDT**	
	Flexibility	***Speed***	***Flexibility***	***Speed***
Creation of test suite	high	low	middle	high
Maintenance of tests	middle	low	middle	high
Maintenance of test suite	low	low	high	high
Tests execution	high	low	high	high
Adding of tests	high	low	middle	high
Test analysis	middle	low	high	high
Coverage analysis	low	low	high	high

Table 1 contains comparative analysis of traditional manual approach and MDT approaches based on different criteria. Values “high”, “middle” and “low” show qualitative analysis for different criterion of flexibility and speed obtained for manual testing (MT) and model driven testing (MDT). A result of the analysis shows that MDT has a lot of benefits and its usage in industrial software testing is reasonable. The following criteria were used:

- Creation of test suite – developing of tests based on requirements;
- Maintenance of tests – correction of particular tests associated with the change of the functionality of the System Under Test (SUT);
- Maintenance of test suite – configuration activity with goal to make correction in all tests which belong to test suite;
- Tests execution – execution of tests on SUT;
- Adding of tests – creation of tests on a new SUT functionality;
- Tests analysis – analysis of tests suite execution results, debugging and bug fixing;
- Coverage analysis – analysis of test suite coverage based on user defined criterion.

Comparative analysis shows that the main problems with MDT are flexibility of creation and adding of test suite. The issue is mainly based on usage of different levels of abstraction which are used in software model and real implementation. The solution here is selection of appropriate abstraction level for SUT model with usage of effective graphical notation which not only allows system semantic description but also simplifies process of model creation.

III. TESTING AUTOMATION IN INDUSTRIAL SOFTWARE SYSTEMS

One of the main characteristics of industrial software systems testing is constraint on test suite size, which shall be used for quality checking. This is connected with automatic or semi automatic approaches to test creation and execution when formal model is used for generation of test suite as a set of system behavior scenarios. Usage of such approach for models of industrial systems leads to generation of huge test suite, which supports all scenarios of SUT behavior. But execution of such test suite is impossible because of constraints on time to testing and therefore only subset of tests shall be selected from all test suite for execution on SUT. This process is called test suite reduction [8].

The goal of test suite reduction is creation of minimal test suite, which will allow to find the same set of defects as initial test suite [9]. Usage of such reduction allows to reduce efforts for maintenance of tests and their execution and as a result efforts for all testing cycle. Traditional approach to reduction is based on static analysis and instrumentation of SUT code to determine a set of tests which satisfy to selected criterion [10]. The main difference in reduction methods is criterion of test selection. Usually the following criteria are used:

- “All-uses” coverage [11];
- Definition-use associations coverage [12];
- Control flow graph coverage [13, 14];
- Modified Condition/Decision Coverage [15];
- Branches coverage [16];

- Paths coverage [16].

Common part of presented coverage methods is necessity to SUT code instrumentation for test suite reduction and this leads to a set of usage limitations. First of all usage of methods is impossible before finishing of development phase because code of SUT is needed. Secondly process of code analysis for big software systems is very complicated task and therefore testing with initial test suite without reduction can be more effective than code analysis.

Usage of MDT for test suite reduction is more effective because code of application is not needed and all coverage analysis can be done based on system model. Therefore it is not needed to wait for the finish of code development and test suite creation can be started when only requirements exist. The only condition for such method is that semantics of the model shall not be changed during development, and if such changes happen then they shall be addressed in model checking procedure. At the same time, the existence of a formal model makes it possible to apply the following methods of software verification that can be used effectively to reduce the number of tests:

- Model checking;
- Symbolic verification;
- Searching for equivalence classes;
- Searching for cycle invariants;
- e.t.c.

Thus, the use of MDT test approaches in industrial software is justified not only on test generation step, but also during analysis and reduction of the test set to be executed.

IV. Model of SUT

In modern project documentation formulation of initial requirements is specified either constructively, when the checking procedure can be reconstructed from the requirement text in a natural language, or non constructively, when functionality specified in the requirement contains no hints on how to check it.

A procedure which checks the current requirement is an exact sequence of the causes and the results of some activities (coded with actions, signals, states), which analysis can prove or refute that this requirement is covered. Such checking procedure can be used as a criterion that a certain requirement is covered; i.e., it can be a so-called criteria procedure. In the text below the term “sequence” or “chain” of events will be used for a criteria procedure.

Different notations can be used for defining such sequences with to reduce manual efforts for tests development. One of them is high level notation TTCN [22]. But usage of formal notation for definition of test suite cannot give any guaranty of coverage or consistency of test specification. To achieve this verification procedure shall be used for proving of different tests properties.

In suggested approach an existing technology based on VRS/TAT toolset is used [23]. In VRS/TAT the Use Case Maps (UCM) notation [17] is used for a model high level description, while tools for automation of checking and test generation work with a model in the basic protocols language [18].

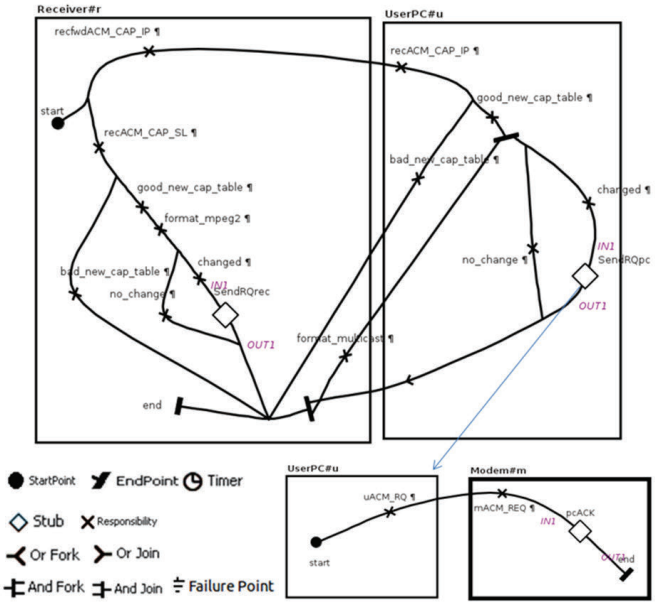


Fig.1 Example of UCM model

A UCM model (Fig. 1) contains a description of a model of two interacting instances. Each path in the graph from the event “start” to the event “end” represents some behavioral scenario. Each path contains a certain number of events (Responsibilities). Events in the diagram are marked with the symbol ×, while Stub elements which encode inserted diagrams are marked with the symbol ◇. As a result, each scenario contains a certain sequence of events. The set of possible scenarios is specified through a set of such sequences. In these terms a chain is determined as a subsequence of events enough to conclude that the requirement is satisfied. A

path in the UCM diagram, containing the sequence of events of some chain is called a trace covering the respective requirement. Tests for experimental evidence that the requirement is covered can be generated from such trace.

V. Formal Model in Basic Protocols

A basic protocol (BP) [18] is the main element of the formal model. It codes the minimal observable step of the system behavior and represents an analog of the Hoare triplet with a pre-condition, a post-condition, and a process (an observable action, a series of actions). The pre and post conditions are logical formulae with inequalities and arithmetic operations which describe a subset of system states before and after the process actions which consist in sending/receiving messages and/or changing the values of application variables. BPs may contain symbolic or concrete parameter values (i.e., for variables in pre and post-conditions, in expressions for signals, state descriptions, etc.), the respective tolerance ranges being specified for symbolic values. Specifying concrete values for BP parameters is called BP concretization.

A BP may refer to one or more requirements, as well as a number of BPs or a number of ordered BP sets (BP chains) may refer to one requirement.

A set of BPs composes a requirement model. One may construct scenarios or traces to visualize possible behaviors of the system under design by just combining consistent pre and post conditions of various BPs. Traces with symbolic parameters are called symbolic traces. The verifier [3, 4] proves correctness of a behavior case represented by a concrete or symbolic trace [19].

A trace set which covers all chains of all requirements forms the requirement model of the application under development. This model may be used for generation of a complete test suite which checks the functional coverage of all constructively specified requirements [8].

Let s_0 be some initial system state, which includes the state of the environment and the states of all agents inserted into it [20]. Then all possible traces (histories of system functioning) may be obtained as sequences of the form:

$$s_0 \xrightarrow{b_1(n_1,m_1)} s_1 \xrightarrow{b_2(n_2,m_2)} \ldots$$

$b_1(n_1,m_1), b_2(n_2,m_2),\ldots$ being the concretized BPs. Here $b_1, b_2,\ldots$ are the names of the respective BPs; $n_1, n_2,\ldots$ are the names of key agents (a key agent is a particular BP, whose state may change in this BP); and $m_1, m_2,\ldots$ are the values of the remaining parameters which satisfy the pre-conditions of the respective BPs. Post-conditions defining deterministic transformations, the states $s_1, s_2,\ldots$ are unambiguously determined by the initial system state and BP concretization. Let's describe a system S(P) which realizes a system P of basic protocols in form of an attributed (or labeled) transition system. Let's define the behavior of the system S(P) in its different states. The states of the system S(P) are equated to basic language formulae extended with intermediate states which correspond to execution of the respective BPs [20].

Let's denote the behavior of the system S in the state α as S_α. Letichevsky proved [18] that the equation for $\mathbf{S}_\alpha^\infty$ ($\mathbf{S}_\alpha^\infty$ means that all non-deadlock traces are infinite) has the form:

$$\mathbf{S}_\alpha^\infty = \sum_{p \in P(\alpha)} \mathbf{proc}(p) * (\mathbf{T}(\alpha, p) : \Delta) * \mathbf{S}_{\mathbf{T}(\alpha,p)}^\infty$$

Here * is a composition of transitions (proc) and states of the behavior graph, $P(\alpha) = \{p \in P_{inst} \mid \alpha \to \mathbf{pre}(p)\}$, P_{inst} is the set of initiated concretized protocols, proc (p) - is the BP process, β is a constant of successful termination (shows that the protocol was applied successfully),

$\mathbf{T}(\alpha, p) = \mathbf{Tr}(\alpha, \mathbf{post}(p))$. $\mathbf{Tr}(\alpha, \beta)$- is a predicate transformer. Two formulas α and β are provided to its input, and a new formula γ is generated as its output which strengthens the post-condition β and considers no changed variables in pre-condition β. In other words, the formula γ is such that $\gamma \to \beta$. This strengthening is necessary to apply next BPs whose post-condition is an indirect corollary of the post-condition β and the pre-condition β.

The next definition includes finite traces with a successful termination. For generality, let's identify a set P^0 of terminating protocols and let $P^1 = P \setminus P^0$. Terminating protocols are supposed to terminate the system work and do not expect its continuation. The equation for a complete system has the form:

$$\mathbf{S}_\alpha = \sum_{p \in P^1(\alpha)} \mathbf{proc}(p) * (\mathbf{T}(\alpha, p) : \Delta) * \mathbf{S}_{\mathbf{T}(\alpha,p)} + \sum_{p \in P^0(\alpha)} \mathbf{proc}(p) * (\mathbf{T}(\alpha, p) : \Delta) * (\mathbf{S}_{\mathbf{T}(\alpha,p)} + \Delta)$$

VI. Methods of Tests creation for non Deterministic and Concurrent Behavior

Guarantying of SUT quality with usage of MDT approach is based on execution of generated test suite. For example generated test can represent a counterexample of some property violation obtained from verification system. Usage of verification counterexamples for testing leads to issue with coverage of system behavior because it covers only particular system behavior, but for testing purposes it is more interesting to cover some particular part of all behaviors of the SUT. One of the possible solutions to avoid the issue with coverage is traversing of all behavior tree of the system with the goal to obtain test suite, which will satisfy some coverage criterion. In

this case the user will have test suite, which covers for example all branches of behavior or all instructions of the program. Very important that such coverage will be produced by verification system so we can say that system behavior was proved in accordance with user defined properties and criteria. The tests will be linear paths with stimulus sent to the SUT and responses for controlling of behavior, where linear means absence of alternative behaviors in the scope of one test. But usage of such linear tests is not correct in some cases for testing of complex software systems. The main problems will appear for testing of passive alternatives and concurrent behavior. These two cases will be considered in details.

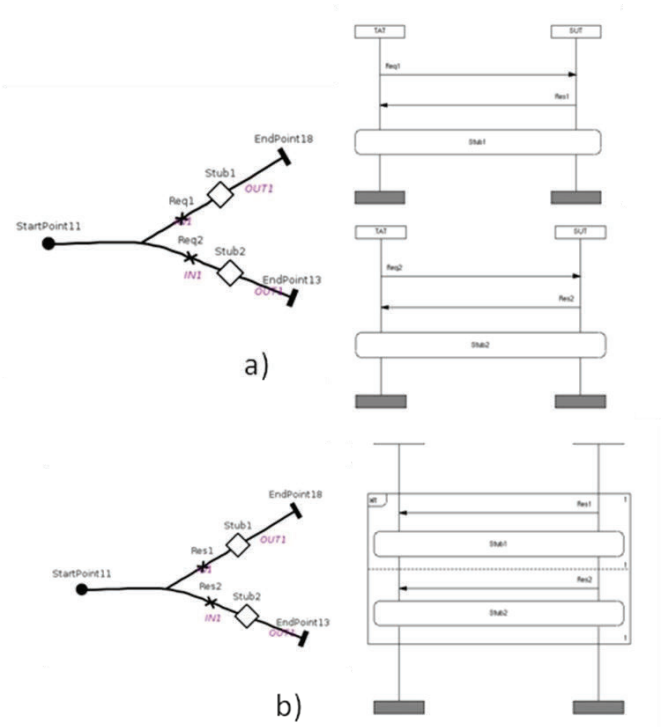


Fig 2. Example of active alternative behavior

A. Testing of Passive Alternatives

Alternative behavior is describing behavior of the SUT, which is associated with usage of deterministic or non deterministic selection of future path. If this selection is deterministic then it can managed by test with sending of some stimulus or signal to the SUT and controlling of SUT response. In this case alternative is called “active”. Fig. 2 a) shows example of active alternative in UCM notation and test diagram in MSC [21] language, which will cover such behavior.

The example illustrates the main feature defining that this situation belongs to active alternative template – this is direction of the signals Req1 and Req2 in corresponding responsibilities. These signals are sent into SUT and its behavior depends on the signals. In other words behavior of the SUT is managed by test. Testing of such behavior can be done based on a set of two linear tests, which are presented on the figure.

“Passive” alternative is a situation when a signal which determines future system behavior comes from SUT. More over this signal cannot be determined based on analysis of behavior history presented in the test. Such behavior is non deterministic and cannot be tested with usage of linear test, because test will check only one of possible behavior and its correctness will be determined only on execution phase, so some mechanism for definition of a set of tests shall be used. One of the possible solutions for such situation is usage of non-linear test with “alt” construction. Fig. 2 b) shows a passive alternative example and non-linear test for checking of its correctness.

Decision about correctness of signal coming from SUT is made in process of the test execution, but not on test generation phase. So the test will work in both cases with incoming signal Res1 and Res2 and will fail only if some different signal comes. Usage of non-linear constructions allows to decompose tests and use them for testing of passive alternatives

However usage of non-linear tests is not usual practice for MDT approach because generated tests are based on counterexamples, which are still linear. So generation of the non-linear tests shall be defined by additional algorithms, which can be based on UCM structure analysis with future gluing of a set of traces into non-linear test.

B. Testing of Concurrent Behavior

As it was described earlier one of the main problems for testing of concurrent systems is state explosion which is caused by interleaving between interacting concurrent threads. A number of states which shall be tested is huge and grows dramatically in case of adding states to each particular thread. Testing of such systems becomes very complex due to the following reasons:

- First of all, in the process of test suite running all possible combination of states which are traversed while execution of the system shall be checked.
- Secondly, initialization of the system and its transition into some particular state is very complex because execution of one thread can affect another threads and it leads to necessity of threads interaction analysis.

In the scope of described approach usage of partial order reduction method is suggested for generation of test suite with coverage of independent concurrent threads behavior. Also in this approach a special analysis of threads dependencies based on UCM language is provided to define additional synchronization points which helps to enlarge area of state reduction method applicability. Below detailed description of the approach is presented and illustrated in examples.

If independency of threads set is proved by some way then task of test generation is pretty simple. On the first step the state of the model before AndFork element shall be stored and concurrent threads shall be defined (threads start from AndFork and finish in AndJoin elements). After threads were determined they shall be glued into linear test scenario. As a result test scenario with coverage of all branches is obtained without any interleaving.

If concurrent threads are dependent to each other then static analysis of such dependency shall be performed. As a result of analysis additional synchronization points shall be added into the system description and in this case the task will be reduced to the previous one.

It should be noted that even if independency of threads was proved, usage of one linear trace is not enough due to the fact that in concurrent systems in most cases it is impossible to control execution of each thread. This leads to usage of non-linear traces for testing.

Let's consider an example of simple concurrent system described in Fig. 3.

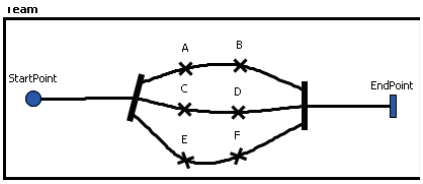
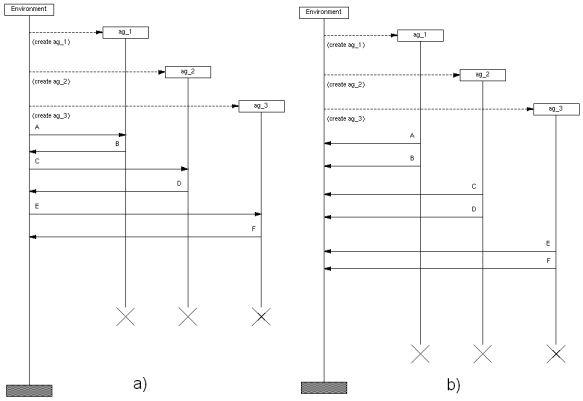


Fig.3 Test cases for concurrent threads

Suppose that responsibility elements in this diagram describe receiving and sending of signals pairs (A;B), (B;C) and (C;D) by system under test. Then the test which is based on assumption about threads independency can be presented like diagram in Fig. 4 a).

For description of the test MSC language is used. Fig 4 a) shows that only one sequence of system behavior is A,B,C,D,E,F and in this case it will be correct because the test manages signals sending into system under test. Now let's suppose that responsibilities describe receiving pairs of signals (A;B), (B;C) and (C;D) from system under test. The test for such behavior can be described by diagram in Fig. 4 b). This test will be incorrect because it will fail in cases when the system behavior is correct. The problem in this case is that in testing of real software it is impossible without special control of the test to ensure that the signal pair (A; B) will come first



(the same statement applies to the remaining pairs of signals). For example sequence C,D,A,B,E,F will be correct from system point of view, but it will be failed by test case.

Fig.4 Test cases for concurrent threads

To solve this problem, the operator "par" can be used [21], which is the syntactic structure of language MSC and allows to describe the interaction of parallel processes. Fig. 5 shows a valid test, obtained by adding the operator par.

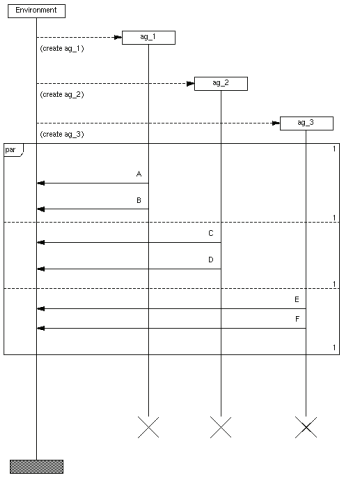


Fig.5 Test scenario with “par” construction

The figure shows that the operator consists of three blocks, each of which describes the interaction of the environment with one of the concurrent threads. Such record determines the sequence of arrival of signals in one particular stream and allows interleaving (without explicitly specifying all possible options, which significantly reduces the entry test) between the concurrent processes. It should be noted that the addition of the operator "par" is a simple procedure and can be fully automated on the basis of information on the generation and synchronization of threads.

Thus, the presented approach allows to automatically generate test suite for testing of concurrent systems based on information extracted from UCM diagram. The main advantage of the approach is automatic obtaining of tests without interleaving which is dramatically reduces test description. Another benefit is automatic definition of reduction methods applicability area based on threads independency analysis.

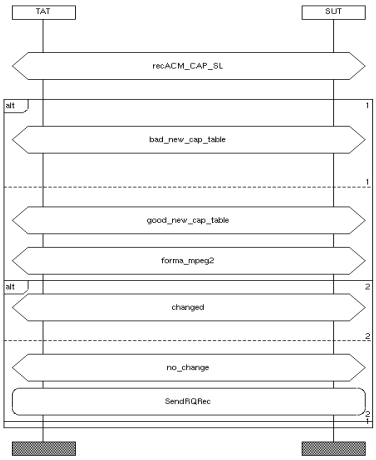
VII. An Example of Presented Approach

Let’s consider an example of work with passive alternative and concurrent behavior on small part of industrial project. The project describes a communication protocol between user terminal and receiver of satellite signal. Only small part devoted to configuration activities is considered. The behavior is presented on Fig 1. in UCM notation.

Behavior of the system starts from ability to receive two signals which are recfwdACM_CAP_IP and recACM_CAP_SL. These signals are coming from environment (in the process of test execution they will be sent by the test), such situation can be determined as active alternative behavior.

In case when signal recACM_CAP_SL was received receiver shall analyze address table and based on analysis results move to one of the possible alternative: “good_new_cap_table” or “bad_new_cap_table”. Condition which determines future path is hidden in variables of SUT and can not be managed by signals on test level so this situation is identifying passive alternative behavior and shall be tested with usage of non-linear construction “alt”. Branches “changed” and “no_change” have the same semantics.

In case when signal recfwdACM_CAP_IP was received analysis of address table shall be done by user application. Passive alternative here is presented in pair of branches “good_new_cap_table” and “bad_new_cap_table”. In case of “good_new_cap_table” concurrent behavior with branches “format_multicast” and “changed/no_change” shall be tested. As described earlier operator “par” can be used for such testing. Fig. 6 shows two tests which can be used for testing of described SUT behavior.



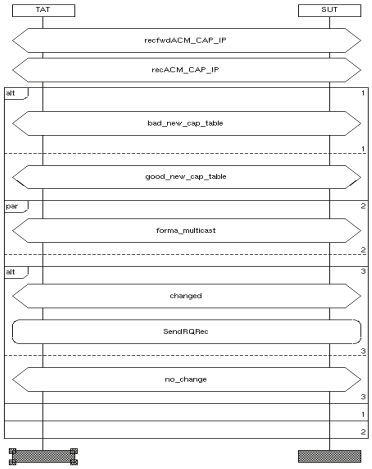


Fig.6 Example of non linear tests

On the figure 6 “conditions” elements of MSC notation are used to show responsibilities of UCM map with appropriate signal and actions, which are described in metadata of these responsibilities. Also MSC element “reference” is used to describe stubs of UCM map. The first test describes checking of case with receiving of “recACM_CAP_SL” signal. Here two “alt” constructions are used and one of them with “changed” and “no_change” branches is nested. The second test describes receiving of recfwd_ACM_CAP_IP signal and consists of two “alt” and one nested “par” constructions. These two tests fully cover all possible behaviors of the system and will be converted into linear representation in process of tests execution.

VIII. Problems of Suggested Approach

Let’s consider possible problems of suggested approach implementation. For passive alternative case the problem with coverage analysis still exists. The issue is based on test execution, which leads to coverage of only one behavior of the system in one execution cycle. As described earlier one non-linear test contains a set of behaviors but in process of its execution the only one behavior will be used. To avoid such issue additional coverage analysis shall be made after test execution procedure. This analysis will allow to identify which particular behavior were covered and which shall be covered by additional tests executions.

Another problem is provided by concurrent behavior testing. As described earlier such tests can be created based on linear tests analysis. But a number of tests can be huge because of interleaving in concurrent systems. Therefore complexity of non-linear tests creation will grow with adding of concurrent processes into SUT specification. So additional methods to reduce a number of tests shall be used to simplify process of non-linear tests creation.

IX. Results of the Approach Piloting

The approach was piloted on three different industrial telecommunication projects with different number of concurrent parts. SMTP project devoted to description of communication protocol has small size (50-100 BPs). Projects CDMA and Satellite terminal which describe modules of telecommunication systems have middle size (100-500 BPs). The Table 2 presents results of the piloting.

TABLE II. Results of piloting in telecommunication projects.

Project	BP number	Non linear BP %	Test scenario number (Linear)	Test scenario number (Linear+ non linear)	Reduce of test suite %
SMTP	30	10	10	8	20
CDMA	205	43	1171	615	48
Satellite terminal	191	15	396	291	27

The results show that number of non-linear tests is less than number of linear for all of the projects. The reducing of test suite strictly depends on a number of protocols which present non linear behavior. Average reducing of test suite with usage of suggested approach is near 30%.

X. Conclusions

Usage of presented approach is effective for testing of industrial telecommunication systems. It allows to reduce a number of tests to be executed due to hiding of interleaving with non linear tests in case of concurrent behavior checking. Also passive alternative cases can be checked correctly with non linear tests. Reducing of tests amount depends on structure of the SUT but in average for piloted project is was near 30 per cent. This work was supported by FCP grant.

References

[1] Oscar Pastor, Sergio Espana, Jose Ignacio Panach, Nathalie Aquino. Model-Driven Development. Springer-Verlag Informatik-Spektrum October 2008, Volume 31, Issue 5, pp 394-407.

[2] Sami Beydeda, Matthias Book, Volker Gruhn. Model-Driven Software Development. Springer-Verlag Berlin Heidelberg 2005.

[3] Paul Baker, Shiou Loh, FrankWeil, Model-Driven Engineering in a Large Industrial Context — Motorola Case Study.

[4] Baillargeon, R. and Flores, R., "Model Driven Testing," SAE Technical Paper 2008-01-0743, 2008, doi:10.4271/2008-01-0743.

[5] Dias-Neto, A.C.; Horta Travassos, G. "Supporting the Combined Selection of Model-Based Testing Techniques", Software Engineering, IEEE Transactions on, On page(s): 1025 - 1041 Volume: 40, Issue: 10, Oct. 2014.

[6] Robert V. Binder, Anne Kramer, Bruno Legeard, 2014 Model-based Testing User Survey: Results, 2014 http://model-based-

testing.info/wordpress/wp-content/uploads/2014_MBT_User_Survey_Results.pdf

[7] Fenton, N.E., Ohlsson, N. Quantitative analysis of faults and failures in a complex software system. Software Engineering, IEEE Transactions on. Issue 8 Aug 2000

[8] P. Drobintsev, V. Kotlyarov, A. Letichevsky, A Formal Approach to Test Scenarios Generation Based on Guides. Automatic Control and Computer Sciences, 2014, Vol. 48, No. 7, pp. 415–423.

[9] McMaster, S. and Memon, A. M. 2005. Call Stack Coverage for Test Suite Reduction. In Proceedings of the 21st IEEE international Conference on Software Maintenance (ICSM'05) - Volume 00 (September 25 - 30, 2005). ICSM. IEEE Computer Society, Washington, DC, 539-548.

[10] Y. Kichigin, An approch to test suite reduction Proceedings of System Programming Institute http://citforum.ru/SE/testing/kichigin/#1

[11] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. ACM Transactions on Software Engineering and Methodology, 2(3):270-285, July 1993.

[12] H. Zhu, P. A. V. Hall, and J. H. R. May. Software Unit Test Coverage and Adequacy. ACM Computing Surveys, Vol.29, No.4, pp.366-427, December 1997

[13] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong. Empirical studies of test-suite reduction. Journal of Software Testing, Verification, and Reliability, V. 12, no. 4, December, 2002.

[14] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In Proceedings of the 16th International Conference on Software Engineering, pages 191-200, May 1994.

[15] J. A. Jones, M. J. Harrold. Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage. IEEE Transactions on Software Engineering, Vol. 29, No. 3, March, 2003, pp. 195-209.

[16] B. Beizer, “Software Testing Techniques,” Second Edition, Van Nostrand Reinhold Company Limited, 1990.

[17] Z.151 : User requirements notation (URN) - Language definition http://www.itu.int/rec/T-REC-Z.151-200811-I/en

[18] Letichevsky A.A., Kapitonova J.V., Kotlyarov V.P., Letichevsky O.O., Volkov V.V., Baranov S.N., Weigert T.: Basic Protocols, Message Sequence Charts, and the Verification of Requirements Specifications. In: Proc of ISSRE04 Workshop on Integrated Reliability Engineering (ISSRE04:WITUL), IRISA, Rennes France (2004)

[19] Kolchin, A., Letichevsky, A., Peschanenko, V., Drobintsev, P., Kotlyarov, V. An approach to creating concretized test scenarios within test automation technology for industrial software projects. Automatic Control and Computer Sciences Volume 47, Issue 7, December 2013, Pages 433-442

[20] A.A. Letichevsky, J.V. Kapitonova, V.P. Kotlyarov, A.A. Letichevsky Jr., N.S.Nikitchenko, V.A. Volkov, and T.Weigert. Insertion modeling in distributed system design // Проблеми програмування. – 2008. – С. 13–38.

[21] Recommendation ITU_T Z. 120. Message Sequence Chart (MSC), 11/2000.

[22] Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language; ETSI ES 201 873-1 V4.7.1 (2015-06)

[23] Baranov, S., Kotlyarov, V., Letichevsky, A., and Drobintsev, P., The technology of automation verification and testing in industrial projects, Proc. St. Petersburg IEEE Chapter, Int. Conf., St. Petersburg, 2005, pp. 81–86.

Система информационной поддержки разработки макропрограмм автономного управления космическим аппаратом

А.А. Тюгашев
кафедра КОТ
Университет ИТМО
Санкт-Петербург, Россия
e-mail: tau797@mail.ru

А.А. Насекин
магистрант
СГАУ имени С.П. Королева
Самара, Россия
e-mail: zzzzlzzzz@yandex.ru

***Аннотация* — В докладе описывается ход работ по созданию инструментальных программных средств информационной поддержки процессов жизненного цикла разновидности бортового программного обеспечения космических аппаратов – «макропрограмм интегрального управления». Работы выполняются для конкретного предприятия-заказчика российской космической отрасли.**

Ключевые слова* — *макропрограмма интегрального управления; космический аппарат; бортовое программное обеспечение; визуализация; проектирование ПО; верификация программ;

I. Введение

Управление современными изделиями ракетно-космической техники (включая нано- и даже пикоспутники) осуществляется с применением бортовых цифровых вычислительных машин (БЦВМ). При этом реализация логики управления бортовыми системами как в штатных, так и в нештатных ситуациях осуществляется программным обеспечением [1]. Ошибка в бортовом программном обеспечении (БПО) может привести к катастрофическим последствиям, включая потерю дорогостоящих космических аппаратов (КА) – зачастую результатов труда многотысячных коллективов предприятий-участников разработки на протяжении ряда лет. Это касается и международных проектов, аварии и катастрофы при реализации которых вызывают значительный общественный резонанс [2, 3].

Неудивительно, что проектированию и отработке БПО уделяется большое внимание. БПО проходит многостадийную отладку с использованием имитационных моделей бортовой аппаратуры на специальных стендах [2]. Эти процессы весьма трудоемки и занимают продолжительное время. При этом по оценке специалистов ЦНИИМАШ [3], стоимость создания комплекса программного обеспечения системы управления изделием РКТ на порядок превосходит стоимость создания аппаратных средств БЦВМ. По времени разработка бортового программного обеспечения может являться критическим путем на сетевом графике работ по созданию авиационного или космического комплекса в целом [1].

Под руководством А.А. Тюгашева с выполнением существенной части работ студентами, магистрантами (в том числе А.А. Насекиным) и аспирантами, в настоящее время по заказу АО «ИСС», г. Железногорск Красноярского края, реализуется проект создания инструментального программного комплекса СИПР МП (может быть расшифровано как «система информационной поддержки разработки макропрограмм автономного управления КА), описываемый в докладе.

Это дает возможность преподавания основ программной инженерии систем реального времени на реальном нетривиальном примере со всеми его особенностями, активного привлечения обучающихся к решению практически значимых, сложных и интересных задач.

II. Описание предмета исследования

Основным объектом (предметом) проектирования, верификации и документирования в СИПР МП являются так называемые макропрограммы интегрального управления КА производства АО «Информационные спутниковые системы». АО «ИСС» является крупнейшим производителем спутниковых систем в нашей стране – 3/4 активно действующей на орбите отечественной группировки произведено в Железногорске. БПО КА производства АО «ИСС» представляет собой хорошо структурированный комплекс с выделением разновидностей программ как по назначению, так и по «уровню» [4]. Термин «макропрограммы интегрального автономного управления» принадлежит Заказчику и подразумевает особый слой БПО, реализующий функции координатора («дирижера»), задающего, какие модули ПО должны запускаться на выполнение, в какое время и в зависимости от каких логических условий [5]. Помимо управления вызовами модулей ПО функционального назначения, макропрограммы реализуют непосредственное управление бортовой аппаратурой путем выдачи на нее команд управления. В связи со

сказанным, в наименовании фигурирует слово «интегральное управление». Автономность управления подразумевает, что основные решения, призванные в том числе сохранить работоспособность КА в случае развития нештатных ситуаций, принимаются на борту без вмешательства специалистов с Земли на основе логики, заложенной в макропрограмму.

Нетрудно понять важность макропрограмм в общей структуре БПО – это классический пример «критически важного» программного обеспечения.

При этом важно отметить следующее. АО «ИСС» в настоящее время уже располагает достаточно развитой технологией производства и отработки БПО, обеспечивающей хорошие показатели качества получаемой программной продукции.

В рамках технологии Заказчика макропрограммы не формируются как «программы» в классическом понимании этого слова, записываемые в виде текстов на языке низкого, высокого уровня (или даже проблемно-ориентированном языке, что сейчас называют Domain Specific Language и что было известно и применялось в СССР еще в 1970-е). Фактически, «макропрограммы» интегрального автономного управления представляют собой массивы исходных данных (в табличной форме) для специальной разновидности БПО – бортовых интерпретаторов. Подобный подход обеспечивает значительную гибкость – при необходимости внесения изменений в логику управлении КА нет необходимости каждый раз удаленно «перепрошивать» ПЗУ бортовой цифровой вычислительной машины. В то же время, подобное обстоятельство в значительной мере обуславливает ограниченность возможности применения известных методов проектирования и верификации ПО. Формат таблиц – закрытый, принадлежит Заказчику, более того, в этой части присутствует специфика отрасли с набором режимных ограничений.

В настоящее время в рамках технологии производства БПО АО «ИСС» подготовка (проектирование и документирование) исходных данных макропрограмм осуществляется самостоятельно разработанными специалистами предприятия инструментальными программными средствами на основе таблиц. При этом существующие формы представления не обеспечивают наглядности. Тестирование хотя и является тщательным и многоэтапным, проходит, в том числе, с применением специальных автоматизированных стендов, все равно в рамках существующей технологии предполагает, что подготовка и запуск тестов возлагаются на человека.

В теории и практике программной инженерии известно достаточно большое число работ, связанных с автоматизацией жизненного цикла программ реального времени [6-14]. Однако, они либо представляют больше теоретически, нежели конкретно практически значимые для предприятий космической отрасли разработки [8,10-14], либо на создание программ на языках Ада, Си, С++, и пр. Специфика макропрограмм Заказчика и их представление не в виде исходных текстов и объектного кода, а в виде фактически данных закрытого формата, интерпретируемых бортовым интерпретатором, исключает прямое использование известных средств в данном случае. Еще одной важнейшей особенностью служит то, что элементарными «акторами» макропрограммы являются не привычные операторы присваивания, а вызовы «команд управления спутником», которые, в свою очередь, интерпретируются другим бортовым интерпретатором. Команды управления имеют достаточно сложную внутреннюю структуру, в частности – могут в ходе своего выполнения инициировать занесение запроса на включение (постановку в очередь диспетчера бортовой операционной системы) той или иной макропрограммы.

Особенностью разрабатываемой СИПР МП является также комплексный подход – ставится задача автоматизировать проектирование (используются визуальные модели), тестирование, создание программной документации и управление версиями. Написание программы как таковое предполагаемой технологией исключается.

III. Назначение и составные части СИПР МП

Назначением системы является снижение трудоемкости разработки одной из ключевых частей БПО – макропрограмм автономного управления КА, повышение надежности и качества программ за счет достижения лучшего взаимопонимания в коллективе разработчиков, автоматизации процессов тестирования на наземном отладочном комплексе и автоматизации генерации программной документации.

СИПР МП состоит из следующих основных частей:

- Интегрирующей оболочки.
- Средств визуализации и графического конструирования макропрограмм.
- Средств генерации тестов.
- Средств исполнения тестов.
- Средств документирования.
- Системы контроля версий программной документации и ее соответствия версиям макропрограмм.

Средства визуализации и графического конструирования обеспечивают представление создаваемых макропрограмм в наглядной эргономичной форме, включая внутреннюю структуру программы и межпрограммные связи.

Средства генерации тестов обеспечивают с использованием подхода «белого ящика» автоматическое построение набора тестов с заданной степенью покрытия (в настоящее время реализованы покрытие всех ветвей, покрытие всех путей и покрытие сложных условий).

Средства документирования на основе гибко настраиваемых шаблонов документов позволяют

автоматически генерировать актуальную программную документацию с поддержкой контроля версий и автоматической проверкой актуальности документации на маропрограмму (соответствия версии документа версии программы).

IV. Средства визуализации и графического конструирования

Одной из возможных причин возникновения ошибок в критическом ПО можно считать сложность самого процесса разработки бортового программного обеспечения. На начальном этапе логику программы разрабатывают проектировщики (алгоритмисты, системные аналитики), которые затем передают эту информацию программистам. Недопонимание или неточность при взаимодействии может привести (и, увы, приводит) к ошибкам. Для минимизации ошибок в программах, создаваемых для исполнения на бортовых вычислительных системах, был предложен метод визуального контроля ранее созданных макропрограмм и графического конструирования новых, описываемый ниже. Таким образом возможно уменьшить степень недопонимания и предоставить процесс разработки макропрограмм проектантам системной логики управления изделием, исключив программистов (раз человеку-программисту свойственно ошибаться, наиболее радикальным решением с позиций обеспечения надлежащего качества и надежности является устранение человека из процесса!).

A. «Группы логических последовательностей»

Макропрограмма интегрального автономного управления состоит из так называемых «групп логических последовательностей» (термин Заказчика). Каждая логическая последовательность имеет главное («спусковое, охраняющее» - *guard*) условие, в случае истинности которого будут выполнена заданная последовательность команд управления спутником, возможно – с указанием временного интервала между командами (блок команд с привязкой к таймеру).

Суть модели легко понять по визуальному представлению, показанному на рисунке 1.

Разрабатываемое инструментальное программное обеспечение должно «без швов» интегрироваться с существующим действующим программным обеспечением Заказчика и взаимодействовать с уже существующей базой данных с ее форматом представления макропрограмм. Необходимо обеспечить возможность графического редактирования и сохранения всех изменений в базе данных макропрограмм.

B. Состояние разработки

На данный момент был создан инструмент, который позволяет визуализировать и конструировать группы логических последовательностей. При этом для каждого элемента группы существует отдельный редактор, с помощью которого можно настраивать его свойства, и тем самым влиять на функционирование макропрограммы (рисунок 1). В связи с тем, что в структуре макропрограмм существуют команды и для перехода из одной группы логических последовательностей в другую, был также разработан визуализатор связей между группами. Макропрограмма представляется графом, в котором множество вершин есть множество групп логических последовательностей, а множество ребер – отражает связи, когда в логической последовательности присутствует отдается команда на заявку в очередь исполнения другой группы.

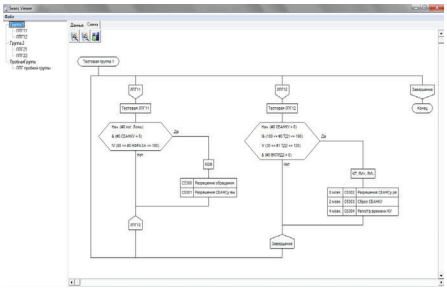


Рис. 1. Экран средств визуализации и графического конструирования

Внешний вид программы визуализации связей групп представлен на рисунке 2.

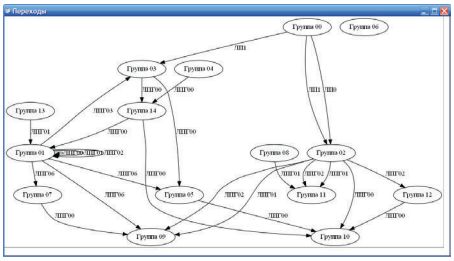


Рис. 2. Визуализация связей между группами ЛП

V. Средства генерации тестов

В СИПР МП применяются следующие базовые подходы к автоматизации тестирования.

A. Степень покрытия

В силу критической важности макропрограмм, весьма важно убедиться, что тесты максимально полно покрывают исходную программу. Критерий покрытия операторов (всех действий, возможных в макропрограмме) в данном случае недостаточен. В СИПР МП реализуется по выбору пользователя либо покрытие ветвей, либо – маршрутов на управляющем графе программы. Задача покрытия маршрутов в данном случае

вполне разрешима на практике вследствие природы макропрограмм интегрального управления, в которых циклы в классическом виде отсутствуют.

В действительности применяется модифицированный критерий покрытия ветвей/условий [15], в соответствии с которым обеспечивается достаточный уровень покрытия, поскольку он проверяет влияние каждой составляющей сложных условий и проходит по всем ветвям управляющего графа. При этом используется допущение о взаимной независимости логических условий (до некоторой степени обоснованное тем, что макропрограммой интегрального управления проверяются в основном значения телеметрические параметры, множество которых проектируется так, чтобы они в максимальной степени были независимыми).

B. Дополнительные требования

По желанию Заказчика, каждый тест сопровождается графическим представлением «трассы исполнения», на которой наглядно отображается последовательность действий, предусматриваемых макропрограммой для данной ситуации.

C. Корректные и некорректные исходные данные

Тесты генерируются не только для случая корректных, но и некорректных исходных данных. Используются различные виды проверяемых в условиях параметров. Типы условий: "значение имеет допустимые границы", "логическая переменная (ДА/НЕТ)", и "вызов функции". Для каждого типа параметров применяется свой подход к генерации корректных и некорректных входных значений.

D. Тестирование и макропрограммы в целом, и частей

Необходимо проверить и каждую «группу логических последовательностей» в отдельности, и всю сложную взаимосвязанную структуру, определяемую командами передачи управления между группами логических последовательностей.

E. Реализация на данный момент

Для генерации тестов используется алгоритм обхода графа связей между группами логических последовательностей с указанием начальной группы для каждой макропрограммы (при входе в группу затем происходит обход дерева группы слева направо, при прохождении разрешающего условия строится набор значений параметров, позволяющий пройти по ветви «да» или «нет», с учетом всех компонент сложного условия).

На настоящий момент построен прототип, позволяющий:

- Подключаться к существующей базе макропрограмм, разработанной у Заказчика ранее.
- Находить все возможные корректные и некорректные комбинации проверяемых параметров.
- Генерировать набор покрывающих тестов по группам и для всей макропрограммы.
- Визуализировать исполнение выбранного варианта.

Копия экрана средств генерации тестов представлена на рисунке 3.

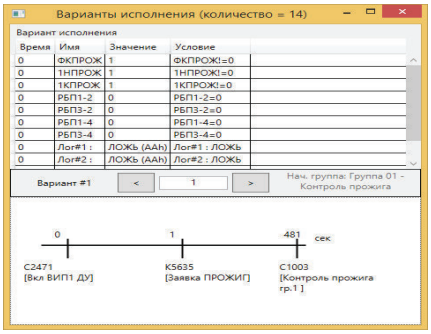


Рис. 3. Экран прототипа средств генерации тестов

VI. Заключение

В докладе представлен ход разработки набора инструментальных программных средств специального назначения, разрабатываемый для конкретного Заказчика из российской космической отрасли.

С целью улучшения взаимопонимания в коллективе разработчиков, снижения числа ошибок предлагается подход, основанный на использовании визуального языка для представления макропрограммы (фактически – визуального программирования).

Средства комплекса СИПР МП должны позволить повысить качество и надежность критически важной составляющей бортового программного обеспечения КА – «макропрограмм интегрального автономного управления» за счет реализации следующих возможностей:

- Наглядного просмотра структуры ранее созданных макропрограмм.
- Средств графического конструирования вновь создаваемых макропрограмм.
- Средств автоматической генерации тестов с заданным уровнем покрытия.
- Средств автоматизированного исполнения тестов.
- Средств документирования, обеспечивающих автоматизированное формирование заданной документации на макропрограмму с гарантией соответствия версий документов версиям программ.

На настоящий момент успешно сдан заказчику первый этап - прототип создаваемого комплекса СИПР МП,

завершен опытный образец в соответствии с техническим заданием и дополнениями к нему.

Список литературы

[1] Авиастроение. Том 6 (Итоги науки и техники, ВИНИТИ АН СССР). М., 1978

[2] Есюнин В.В. Основные принципы построения испытательного стенда бортового комплекса управления для перспективных изделий ОАО «ИСС». Вестник СибГАУ, 2010, Т. 2, С. 93-96.

[3] А.А. Тюгашев, И.А. Ильин, И.Е. Ермаков. Пути повышения надежности и качества программного обеспечения в космической отрасли // Управление большими системами. Сборник трудов. Инс т проблем управления им. В.А.Трапезникова РАН, Вып.39 2012, с. 288-299.

[4] Антамошкин А.Н., Колташев А.А. Технологические аспекты создания бортового программного обеспечения спутников связи. Вестник Сибирского государственного аэрокосмического университета им. академика М.Ф. Решетнева. 2005. № 3. С. 93-95.

[5] Кочура Е.В. Разработка макропрограмм интегрального управления космическими аппаратами. Вестник СибГАУ, 2011, Т.1, С 105-107.

[6] Бахмуров А.Г., Смелянский Р.Л. Проблемы инструментальной поддержки разработки распределенных встроенных систем реального времени // Программирование. 2013. Т. 39. № 5. С. 5-21.

[7] Смелянский Р.Л., Бахмуров А.Г., Волканов Д.Ю., Чемерицкий Е.В. Интегрированная среда для анализа и разработки распределенных встроенных вычислительных систем реального времени // Программирование. 2013. Т. 39. № 5. С. 35-52.

[8] Шаров О.Г., Афанасьев А.Н. Методы и средства трансляции графических диаграмм // Программирование Т.37 № 3 2011 С. 65-75

[9] Буздалов Д.В., Зеленов С.В., Корныхин Е.В., Петренко А.К., Страх А.В., Угненко А.А., Хорошилов А.В. Инструментальные средства проектирования систем интегрированной модульной авионики // Труды Института системного программирования РАН. 2014. Т. 26. № 1. С. 201-230.

[10] Swarup Mohalik, Ambar A. Gadkari, Anand Yeolekar, K.C. Shashidhar, S. Ramesh. Automatic test case generation from Simulink/Stateflow models using model checking. Software Testing, Verification and Reliability, Vol. 24, Iss. 2, pp. 155–180, March 2014

[11] Maximiliano Cristiá, Pablo Albertengo, Claudia Frydman, Brian Plüss, Pablo Rodríguez Monetti. Tool support for the Test Template Framework. Software Testing, Verification and Reliability, Vol. 24, Iss. 1, pp. 3–37, January 2014

[12] Adilson Luiz Bonifácio, Arnaldo Vieira Moura. A new method for testing timed systems. Software Testing, Verification and Reliability. Vol. 23, Iss. 2, pp. 91–117, March 2013

[13] Abdeslam En-Nouaary, A test purpose-based approach for testing timed input output automata Software Testing, Verification and Reliability, Vol. 23, Iss. 1, pp. 53–76, January 2013

[14] Mercedes G. Merayo, Manuel Núñez and Ismael Rodríguez. A formal framework to test soft and hard deadlines in timed systems. Software Testing, Verification and Reliability, Vol. 22, Iss. 8, pp. 583–608, December 2012

[15] Покрытие программного кода. ИНТУИТ – Интернет-университет информационныхтехнологий. http://www.intuit.ru/studies/courses/1040/209/lecture/5410.

обозначена соответствующая
страница доклада в сборнике

Программа ТМРА-2015

САНКТ-ПЕТЕРБУРГ, 12-14 НОЯБРЯ 2015
САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО

12 НОЯБРЯ

9.00	*Регистрация*	
9.30	*Открытие конференции*	
10.00	Николя Гюльфи, *Университет Люксембурга* **Software engineering education: the messir approach**	**Приглашенный доклад**

СТАТИЧЕСКИЙ И ДИНАМИЧЕСКИЙ АНАЛИЗ ПРОГРАММ

11.00	Михаил Ермаков, *Институт системного программирования РАН* **Динамический анализ исполняемого кода в формате ELF на основе статической бинарной инструментации**	14
11.30	Яков Роскошный, Дмитрий Цителов, Виталий Трифанов, Роман Елизаров, *Университет ИТМО, ООО «Эксперт-Система»* **Применение статического анализа для оптимизации динамического поиска гонок**	21
12.00	*Кофе-Брейк*	
12.30	Марина Полубелова, Семён Григорьев, *Санкт-Петербургский государственный университет* **Лексический анализ динамически формируемых строковых выражений**	27
13.00	Артем Алексюк, Владимир Ицыксон, *Санкт-Петербургский государственный политехнический университет Петра Великого* **Применение технологии иерархических параметризируемых шаблонов для автоматизированного исправления ошибок в программном коде**	38
13.30	Антон Евдокимов, Дмитрий Цителов, Роман Елизаров, Виталий Трифанов, *Университет ИТМО, ООО «Эксперт-Система»* **Автоматическое тестирование линеаризуемости реализаций многопоточных структур данных**	48

12 НОЯБРЯ

14.00 *Обед*

15.15 Михаил Глухих, *JetBrains*
Язык Котлин: от разадресации нуля до умных преобразований типов
Приглашенный доклад

ТРЕЙДИНГОВЫЕ СИСТЕМЫ

16.00 Валерий Флоров, Павел Гарин, Павел Смирнов, Максим Метельков, *Exactpro Systems*
Спонсорский доступ с малой задержкой посредством FPGA 58

16.30 Виктория Леончик, Алексей Сухов, Евгений Ушаков, Иосиф Иткин, Анна-Мария Лукина, *Exactpro Systems*
Reference test harness for algorithmic trading platforms 65

17.00 Алёна Булда, Мария Орлова, *Exactpro Systems*
Dynamic verification of input and output data streams for market data aggregation and quote dissemination systems (Ticker Plant) 95

17.15 *Кофе-Брейк*

17.45 Анна Торопова, Екатерина Димова, Иосиф Иткин, *Exactpro Systems*
Инструмент для автоматизированного тестирования систем проведения расчетов и клиринга ClearTH 72

18.15 Анна Торопова, Сергей Павлов, Андрей Соловьев, Александр Бормотин, Иосиф Иткин, *Exactpro Systems*
Автоматизированное создание сценариев тестирования финансовых протоколов и подключений 82

18.30 *Завершение дня*

13 НОЯБРЯ

9.00	Бертран Мейер, *Eifel Software* **A Theory of Programs**	**Приглашенный доклад**

ВЕРИФИКАЦИЯ

10.10	Алексей Промский, Дмитрий Кондратьев, *Институт систем информатики им. А.П. Ершова СО РАН* **Implementing the metaVCG approach in the C-light system**	101
10.40	Дмитрий Кондратьев, *Институт систем информатики им. А.П. Ершова СО РАН* **Расширение метагенерации условий корректности концепцией семантической разметки**	107
11.10	*Кофе-Брейк*	
11.30	Николай Шилов, *Институт систем информатики им. А.П. Ершова СО РАН* **A need to specify and verify standard functions**	119
12.00	Дмитрий Мордвинов, Юрий Литвинов, *Санкт-Петербургский государственный университет* **Формальные методы в робототехнике**	123
12.30	Андрей Миронов, *Институт проблем информатики РАН* **Верификация функциональных программ методом построения диаграмм состояний**	135
12.45	Владимир Коваленко, Галина Алперович, *JetBrains* **Towards a usable defect prediction tool: crossbreeding machine learning and heuristics**	146
13.00	Николай Пакулин, *Институт системного программирования РАН* **Стандарты и стандартизация в программной инженерии. Какое отношение это имеет к вам?**	**Приглашенный доклад**
13.45	*Обед*	
15.00	**СЕМИНАР OS DAY**	
17.15	*Кофе-Брейк*	
17.45	**СЕМИНАР OS DAY**	
20.00	*Завершение дня*	

14 НОЯБРЯ

9.30 Антон Семенченко, *DPI Solutions*
Автоматизированное тестирование вчера, сегодня, завтра – векторы развития

10.00 Матей Михалко, *DECENT*
Blockchain technologies: the future of publishing will have no middleman

ИНСТРУМЕНТЫ ТЕСТИРОВАНИЯ И ОТЛАДКИ

10.30	Павел Довгалюк, Мария Климушенкова, Денис Дмитриев, Владимир Макаров, *Новгородский государственный университет им. Ярослава Мудрого* **Мультиплатформенный метод обратной отладки виртуальных машин**	161
11.00	*Кофе-Брейк*	
11.30	Ростислав Ефремов, *Санкт-Петербургский государственный университет* **Трассировка многомодульного приложения в среде операционной системы z/OS**	169
12.00	Дмитрий Мордвинов, Юрий Литвинов, *Санкт-Петербургский государственный университет* **Тестирование среды программирования роботов**	176
12.15	Павел Дробинцев, Всеволод Котляров, Никита Воинов, *Санкт-Петербургский государственный политехнический университет Петра Великого* **Generation of test scenarios for non deterministic and concurrent telecommunication applications**	186
12.30	Андрей Тюгашев, Антон Насекин, *Университет ИТМО, СГАУ* **Система информационной поддержки разработки макропрограмм автономного управления космическим аппаратом**	195
12.45	*Закрытие конференции*	

Партнеры

ПОЛИТЕХ
Санкт-Петербургский
Политехнический Университет
Петра Великого

Санкт-Петербургский политехнический университет Петра Великого — Политех — крупнейший научно-образовательный центр Северо-Запада России.

В 2010 году Политех получил статус научно-исследовательского университета, что явилось признанием его научно-исследовательского потенциала и достигнутых результатов в науке, образовании и индустриальных разработках.

В 2013 году Политех принял участие в амбициозном конкурсе по программе повышения конкурентоспособности среди ведущих научно-образовательных центров России и стал одним из 15 его победителей. Цель программы – активная модернизация университетов с целью повышения их конкурентоспособности в мировом научно-образовательном пространстве.

С 2013 года университет активно принимает участие в ведущих международных рейтингах высших учебных заведений мира. В 2015 году в рейтинге QS World University Rankings Политех занял позицию в группе 471-480 в мире и 7 место среди российских университетов. В другом престижном рейтинге The Times Higher Education Supplement (THE) Политех занял позицию 201- 250 в мире и второе место в России после МГУ.

В 2015 году университет был переименован, вернув себе имя Петра Великого.

GATE*lab*

GATElab, founded by Ferdinando La Posta & Antonio Caroselli in October 1989 has been part of the LSE Group since December 2012 (LSEG as majority stake holder).

For over 25 years, GATElab has been creating front office trading solutions for the international financial community.

Its products and services cover:

- Multi-asset class electronic trading platforms for price taking, market making, brokering

Ultra low latency pre-trade risk market adapters:

- Order routing via FIX standard protocol
- Smart order routing for different asset classes
- Co-located and distributable algorithmic trading platform for price taking, market making, brokering
- Matching engine for internalization and MTFs
- Post trading deal capture & cross asset margining

Quality Assurance & Software Development for Innovative Trading Systems

General Information

Exactpro Systems is a fully owned subsidiary of London Stock Exchange Group (LSEG). Being part of LSEG's Technology Services division since 29 May 2015, Exactpro specialises in automated Quality Assurance testing solutions for the wholesale financial services industry.

To achieve the most effective automated testing, we develop and supply our own software testing tools suite based on open source solutions.

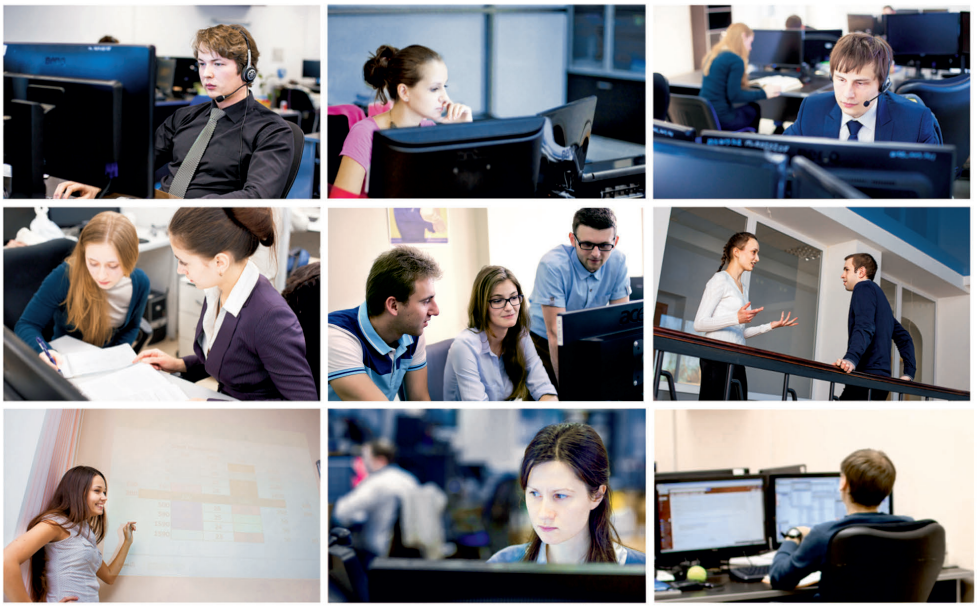
Our team has in-depth understanding of the technologies and business logic behind matching engines, smart order routing, algorithmic trading, order management, market data dissemination, straight-through processing, clearing and settlement for equities, fixed income, FX, listed and OTC derivatives.

Our existing customers include investment banks, brokers, buy-side, technology vendors, and execution venues. In addition to functional and non-functional testing, we also verify that our customers' systems are compliant with regulations.

We have considerable experience working in ultra-low latency systems and FIX-based solutions.

Exactpro is a young and dynamic company. Thanks to our university relationship program we have been able to attract and retain a superb technical team of talented engineers.

All Exactpro customers are delighted with QA tools and services delivered by us at a highly competitive cost.
Are you considering improving quality, time to market, regulatory compliance, reducing cost or latency?
If so, how may we assist you?
Visit www.exactpro.com



London Stock Exchange Group

exactpro
EXITUS ACTA PROBAT

What Makes Exactpro Different?

Build Software To Test Software	• Automated testing (and analysis of results) wherever possible • We develop our own tool suite we use & create new tools as needed* • Test Automation is in our DNA, our tool set is a key part of our added value!
Highest Calibre	• Core of highly experienced industry consultant practitioners • Deepest domain expertise of instruments/contracts, exchanges, workflow, functionality, risk models, industry standards, etc. • Attract the best graduates through close links with Russian technical universities • Extensive internal training program to grow our talent pool
Stability	• Extremely low levels of staff turnover • High levels of customer retention
Work Ethos/Culture	• Proactive and flexible approach to engaging with the project team • High energy, determination and discipline • Attention to detail • Work on customers chosen time zone
Cost/Value	• Very attractive offshore rates • Majority of work carried out offshore (via remote access) • The added value we bring to projects has been proven to maximize the potential for delivery on schedule
Related Services	• Data cleaning, data migration & other • Development of additional components/sub-systems

* We also use open source tools where applicable, and we are well versed in the QA industry standard tools & hence can use those where it is customer policy to do so.

QA and Dev for Innovative Trading Systems

Domain Capabilities - We are 100% focused on systems that process financial products, with particular focus on Electronic Trading

- Matching Engines
- Algorithmic Trading
- Smart Order Routing
- Clearing and Settlement
- Market Venue Connectivity
- Deal Capture & Position Keeping
- Order and Execution Management
- Market Data Distribution
- Risk Management
- Reference Data
- Middle Office
- Messaging

We are highly effective in all of these aspects of Quality Assurance

- Test automation
- Test data management
- Creating and productizing state-of-the-art test harnesses
- Intelligent functional and exploratory testing
- Protocol level testing using FIX/FAST, SOAP, HTTP, ITCH, SWIFT, MQ, SQL, proprietary binary and text based data formats, etc.
- Gathering requirements and test scenario creation (human, message & reporting interfaces)
- Latency measurements and capacity planning
- Behavioral testing for SOR and algorithmic trading systems
- Process audit and test coverage analysis
- Automated regression testing

Related Services - Added Value for our main Customers in other complementary areas (in addition to Quality Assurance) through which we gain additional depth

- Unit testing
- Tuning and optimization
- Supporting legacy applications
- Business analysis from historic data
- Developing trading and post-trade systems and their components
- Customization and integration of trading and post-trade platforms
- Data cleanup & migration
- Prototyping new systems
- Code review

Our Bespoke Test Automation Tools Suite

Sailfish

- Can test Order Entry, Market Data and Post Trade connections in one test scenario
- Each test scenario is independent
- Allows running test scripts in any sequence
- Simulation of multiple user connections
- Server simulators
- All messages are stored into a data base
- Generates test reports

ClearTH

- Clearing & Settlement testing tool
- Verifies each stage of the DLC
- Integrated schedule
- Automated matrices
- Can create multiple days test scenarios
- Concurrent multiple tests
- Integrated simulators
- SWIFT ISO 15022 / 20022, FIX protocols support

Dolphin

- Model-based testing of market surveillance systems
- Production-scale capacity and throughput
- Interactive real-time alerts and reports

Shsha

- Post-transactional tool
- Analyzes clients' activity and forecasts system response
- Parses and displays logs in a user-friendly way
- Parses messages and then puts each to a data base table where each column corresponds to each message field
- Allows making summarized reports, etc.
- Easy to understand GUI

Load Injector

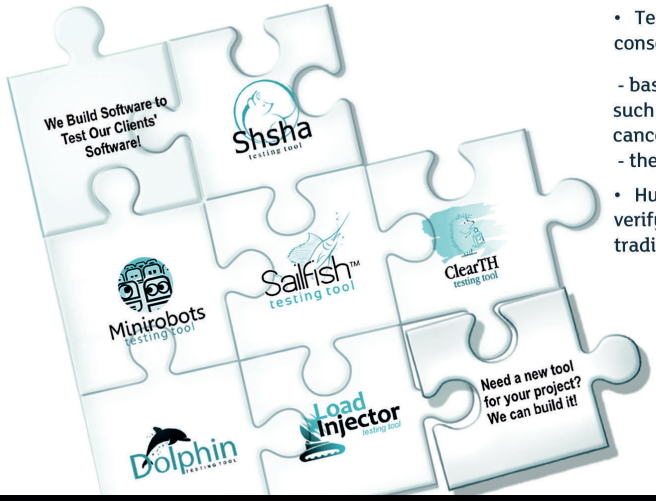
- Simulates multiple client connections with a specified load shape for each connection or a group of connections
- Up to 75K messages / second from a single CPU core
- Measures latencies in microsecond range
- Performance test reports

Minirobots

- A multi-threaded Trading Engine for simulating trading algorithms behavior and market conditions
- Using an extensive and easy to use API, testers can develop trading robots in Java and define custom parameters to be displayed in real-time
- Testers can control the robots via a Web-based console. Metrics are available in real-time:

  - basic parameters of robots' activity, such as the number of new orders, cancels, amends, trades
  - the number of issues observed

- Hundreds of internal event flow validations verify the consistency of responses from the trading system in real-time



Need a new tool for your project? We can build it!

ВСЕ ВОЗМОЖНОСТИ ФИНАНСОВОГО РЫНКА

МОСКОВСКАЯ БИРЖА

moex.com

ОАО МОСКОВСКАЯ БИРЖА РЕКЛАМА 14+

Московская Биржа – *крупнейший в России и Восточной Европе биржевой холдинг, образованный 19 декабря 2011 года в результате слияния биржевых групп ММВБ (основана в 1992) и РТС (основана в 1995). Московская Биржа входит в двадцатку ведущих мировых площадок по объему торгов ценными бумагами, суммарной капитализации торгуемых акций и в десятку крупнейших бирж производных финансовых инструментов.*

На Московской Бирже участникам доступна современная и высокотехнологичная инфраструктура по торговле акциями, облигациями, валютой, инвестиционными паями, биржевыми фондами ETF, товарами (зерном, золотом, серебром), производными финансовыми инструментами, в том числе и внебиржевыми деривативами.

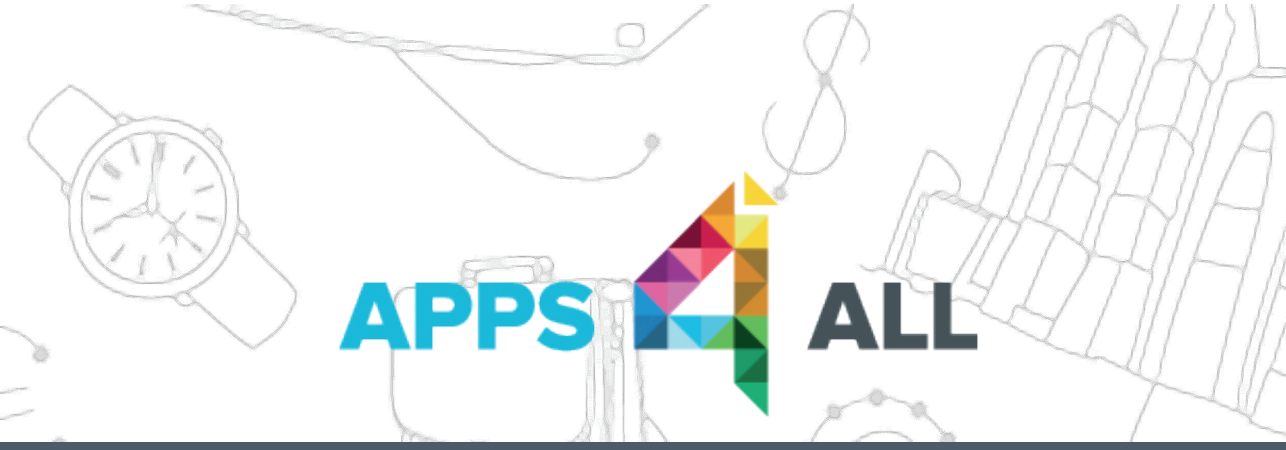
Торги ценными бумагами проводятся на Фондовом рынке Московской Биржи по технологической схеме Т+2: частичное предварительное депонирование средств и расчеты через два дня после заключения сделки. Производные финансовые инструменты, такие как фьючерсы и опционы на акции, индексы, валютные пары, торгуются на Срочном рынке. Спотовые сделки и сделки своп с иностранной валютой заключаются на Валютном рынке: участникам доступны пары доллар/рубль, евро/доллар, евро/рубль, китайский юань/рубль, а также белорусский рубль, украинская гривна и казахский тенге. Денежный рынок предлагает участникам удобные сервисы управления ликвидностью. В секциях Денежного рынка заключаются сделки РЕПО с государственными ценными бумагами, инструментами денежного рынка, а с 2013 года доступно заключение сделок РЕПО с Центральным контрагентом (акции и облигации). На базе торговой платформы Валютного рынка проводятся торги золотом и серебром. Все биржевые рынки Московской Биржи доступны как юридическим, так и физическим лицам, как резидентам Российской Федерации, так и резидентам иностранных государств по схеме прямого доступа на рынок - DMA (Direct Market Access).

Профессиональным участникам, помимо фондового, срочного, денежного, валютного рынков и рынка драгоценных металлов, доступен рынок внебиржевых стандартизированных ПФИ с центральным контрагентом - сервис для заключения сделок процентный, валютный, валютно-процентный своп.

Кроме того, через ЗАО «Национальная товарная биржа», Московская Биржа реализует проекты по развитию торгов на рынке зерна.

В рамках Группы «Московская Биржа» трейдинговые услуги предоставляют ОАО Московская Биржа и ЗАО «Фондовая биржа ММВБ», пост-трейдинговые - НКО ЗАО «Национальный расчетный депозитарий» и ЗАО «Национальный клиринговый центр». Национальный расчетный депозитарий - центральный депозитарий Российской Федерации – проводит расчеты и осуществляет депозитарное обслуживание. Национальный клиринговый центр, которому Банк России присвоил статус квалифицированного контрагента, выполняет функции Центрального контрагента на всех рынках Московской Биржи, оказывая участникам клиринговые услуги. Статус квалифицированного центрального контрагента позволяет клиентам НКЦ (участникам торгов на рынках Биржи) эффективнее использовать свои активы.

Московская Биржа активно содействует развитию российского финансового рынка, его инфраструктуры, совершенствует технологии и повышает привлекательность своих торговых площадок и сервисов для отечественных и зарубежных инвесторов, и эмитентов.



Рады представить вам нашего партнера компанию Apps4All!

Это профессиональная социальная сеть для разработчиков приложений и участников экосистемы мобильных технологий.

Платформа Apps4All объединяет всех участников рынка: разработчиков, крупнейшие IT-корпорации, стартап-проекты, венчурных инвесторов, представителей бизнеса и осударственных органов для создания и развития качественных и успешных мобильных продуктов.

Ежедневно на сайте http://apps4all.ru/ публикуется несколько десятков новостей, посвященных всем аспектам индустрии мобильных приложений и игр:

- последние разработки;
- статьи о монетизации и продвижении приложений;
- интервью с лидерами рынка;
- анализ хитов продаж и советы по тестированию и дизайну;
- обзоры последних новинок ведущих магазинов приложений;
- аналитика российского и мирового рынка мобильных приложений.

Участники сообщества получают бесплатный доступ к уникальным возможностям экосистемы мобильных технологий, устройств и приложений.

Welcome to the club!

• INTERNET • E-COMMERCE • MOBILE • SOFTWARE INNOVATION •

• STARTUPS & INCUBATORS • INVESTMENT ACTIVITY • LEGAL ENVIRONMENT • EVENTS •

Digital Russia in English

East-West Digital News

Daily news • Analysis
International consulting
www.ewdn.com

East-West Digital News is the first international information and consulting company dedicated to Russian digital industries. Get in touch with us at
contact@ewdn.com

• STARTUPS & INCUBATORS • INVESTMENT ACTIVITY • LEGAL ENVIRONMENT • EVENTS •

• INTERNET • E-COMMERCE • MOBILE • SOFTWARE INNOVATION •

PFSOFT

PFSOFT — is a team of highly qualified specialists who implement cutting-edge technologies, developing trading software for a wide range of financial markets. It was started in 2003 from the foundation idea — to bring financial software market on the qualitatively new level. The company's main office is located in Gdansk, Poland while the development team is based in Dnipropetrovsk, Ukraine.

PFSOFT is a product company that provides development and establishment of their main product — Protrader Multi-Asset trading platform — to their clients. Currently it consists of 80+ workers in .Net, Java, PHP development, Quality Assurance, Marketing and Business development as well as Project management and Support spheres.

The company currently serves different market participants from the USA, Canada, Australia, Japan, India, New Zealand, China, Turkey, Russia, Great Britain, Poland and South Africa.

PROTRADER
MULTI-ASSET TRADING PLATFORM

Protrader – is a comprehensive and advanced software for brokerage, created to operate on all major markets including Forex, Stocks, Futures, Options and CFDs.

The platform consists of three functional parts: server, back office and powerful front-end suite: desktop, web, mobile. Protrader can be integrated with various functional APIs as well as with the unlimited number of liquidity providers and data feeds. Protrader trading environment contains a wide range of tools suitable for everyone. Traders of all kinds from beginners to professionals, scalpers, algo traders, and even asset managers can find their place in Protrader. It's got a lot of features including visual trading (trading from charts) and one click trading, Fund Management and MAM (for the asset management) etc. For those who are more interested in the technical side of trading, Protrader for desktop includes AlgoStudio for full development cycle of algorithmic strategies.

In order to know Protrader better, feel free to visit http://protrader.com, or contact our Sales representative via 'Send Request' form on the provided website.

ОРГАНИЗАТОРЫ КОНФЕРЕНЦИИ:

• Санкт-Петербургский политехнический университет Петра Великого (СПбПУ), кафедра компьютерных систем и программных технологий института компьютерных наук и технологий СПбПУ (КСПТ ИКНТ СПбПУ)

• Федеральный исследовательский центр «Информатика и управление» Российской академии наук (ФИЦ ИУ РАН)

• Компания Exactpro

ОРГАНИЗАЦИОННЫЙ КОМИТЕТ КОНФЕРЕНЦИИ:

- Заборовский В.С., профессор, д.т.н., директор ИКНТ СПбПУ, председатель
- Ицыксон В. М., доцент, к.т.н., зам. председателя
- Пышкин Е. В., доцент, к.т.н., КСПТ ИКНТ СПбПУ,
- Тамм Н. А., ИКНТ СПбПУ, секретарь конференции
- Ненилина Л. И., ИКНТ СПбПУ
- Эдеш А. С., Exactpro
- Лукина А.-М. А., Exactpro
- Журавлева Д. А., Exactpro
- Аввакумова Е. С., Exactpro

ГЕНЕРАЛЬНЫЕ ПАРТНЕРЫ

London Stock Exchange Group

London Stock Exchange Group (LSEG) is a diversified international exchange Group that sits at the heart of the world's financial community.

exactpro
EXITUS ACTA PROBAT

As a fully owned subsidiary of London Stock Exchange Group (LSEG), Exactpro is part of LSEG's Technology Services division, which also incorporates MillenniumIT and GATElab. A specialist firm focused on functional and non functional testing of securities data distribution, trading systems, risk management, market surveillance and post-trade infrastructures

МОСКОВСКАЯ БИРЖА

OJSC Moscow Exchange is the largest Russian exchange both in terms of turnover and the extensive client base. The main goal of the exchange is to provide Russian and foreign market participants and investors with a safe and effective access to the booming Russian market.

ПАРТНЕРЫ

PFSOFT PROTRADER MULTI-ASSET TRADING PLATFORM POLYTECH Peter the Great St.Petersburg Polytechnic University DC DECENT КСПТ QA CLUB KOSTROMA ИПИ РАН ИНСТИТУТ ПРОБЛЕМ ИНФОРМАТИКИ

HR ПАРТНЕРЫ

SuperJob iChar trudbox TRUD.com

ИНФОРМАЦИОННЫЕ ПАРТНЕРЫ

RUS®SOFT РАЭК CMS magazine East-West Digital News APPS ALL IT TRAINING ПрИн ITSZ.ru Системный администратор GENERAL EXPO ВЫСТАВОЧНЫЙ ПОРТАЛ APPTRACTOR Expomap ГидСобытий БИТ интернет infuture.ru СОДЕЙСТВИЕ Numeralis Fastestpc.ru www.COMPRICE.ru KV:1 EIN NEWS COMAQA.BY DPI ZILLION -RUNET-ID-

Научное издание

ИНСТРУМЕНТЫ И МЕТОДЫ АНАЛИЗА ПРОГРАММ – 2015
TOOLS & METHODS OF PROGRAM ANALYSIS (TMPA – 2015)

Материалы
Международной научно-практической конференции
12–14 ноября 2015 года

Налоговая льгота – Общероссийский классификатор продукции
ОК 005-93, т. 2; 95 3004 – научная и производственная литература

Подписано в печать 30.12.2015. Формат 70×100/16. Печать цифровая.
Усл. печ. л. 17,75. Тираж 120. Заказ 0088.

Отпечатано с готового оригинал-макета, предоставленного
редакционной коллегией,
в Типографии Политехнического университета.
195251, Санкт-Петербург, Политехническая ул., 29.
Тел.: (812) 552-77-17; 550-40-14.