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

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

УДК 004.416.3+004.4'242

Семантически-ориентированная миграция Java-программ: опыт практического применения

Алексюк А. О., Ицыксон В. М.

получена 3 сентября 2017

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

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

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

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

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

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

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

Введение

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

Целями этого исследования являются:

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

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

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

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

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

1. Анализ существующих подходов к миграции

Рассмотрим существующие подходы к автоматизации миграции программных проектов.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2. Предлагаемый подход

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

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

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

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

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

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

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

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

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

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

Предлагаемая процедура миграции состоит из пяти шагов:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3. Разработка прототипа

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

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

Были рассмотрены следующие подходы для извлечения трассы программы:

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

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

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

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

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

4. Экспериментальная проверка инструмента миграции

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

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

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

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

4.1. Эксперименты с библиотеками, реализующими протокол HTTP

Были созданы модели для следующих библиотек:

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

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

Листинг 1. Пример использования класса HttpURLConnection из Java Class Library

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

Листинг 2. Пример использования библиотеки Apache HttpCliennt

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.2. Эксперименты с библиотеками протоколирования

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

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

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

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

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

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

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

Заключение

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

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

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

Список литературы / References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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