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.

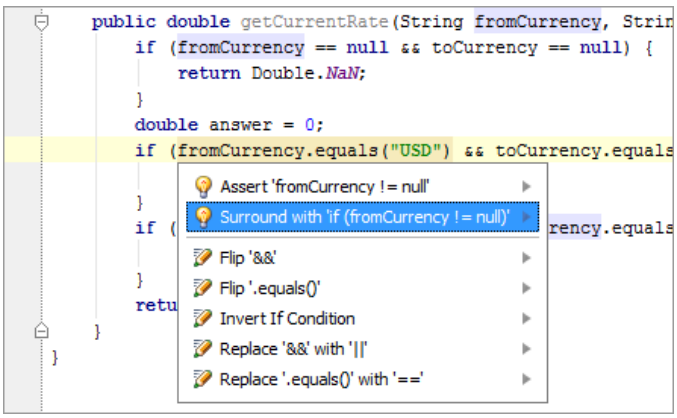

Рис. 1. Предупреждение от статического анализатора в IntelliJ IDEA и всплывающее меню Quick Fix

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.

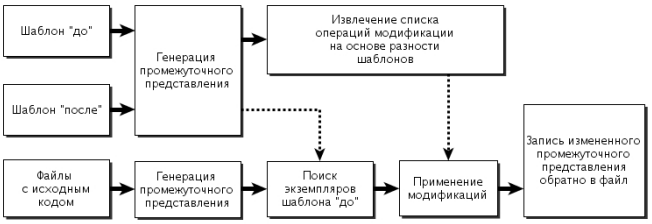

Рис. 2. Структура проекта TemplateRefactorings

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.

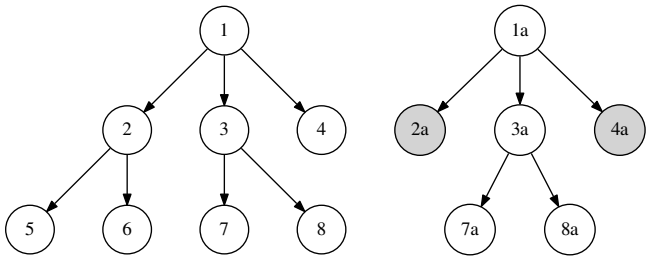

Рис. 3. Пример деревьев, для которых приведённый подход работает некорректно

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:

```
javaStart
:    compilationUnit EOF
|    block EOF
|    blockStatement EOF
|    memberDeclaration EOF
|    expression EOF
;
```

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 #:

```
selector
:    '#' Identifier
;
```

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

```
expression
:    primary
|    expression '.' Identifier
|    expression '.' 'this'
|    expression '.' 'new' nonWildcardTypeArguments
     ?
     innerCreator
|    expression '.' 'super' superSuffix
|    expression '.' explicitGenericInvocation
|    expression '[' expression ']'
|    expression '(' expressionList? ')'
|    'new' creator
|    '(' type ')' expression
|    expression ('++' | '--')
|    ('+'|'-'|'++'|'--') expression
|    ('~'|'!') expression
|    expression ('*'|'/'|'%') expression
|    expression ('+'|'-') expression
|    expression ('<' '<' | '>' '>' '>' | '>' '>')
     expression
|    expression ('<=' | '>=' | '>' | '<')
     expression
|    expression 'instanceof' type
|    expression ('==' | '!=') expression
|    expression '&' expression
|    expression '^' expression
|    expression '|' expression
|    expression '&&' expression
|    expression '||' expression
|    expression '?' expression ':' expression
|    selector
|    <assoc=right> expression
(    '='
|    '+='
|    '-='
|    '*='
|    '/='
|    '&='
|    '|='
|    '^='
|    '>>='
|    '>>>='
|    '<<='
|    '%='
)
expression
;
```

В правило для блока добавлена поддержку селекторов:

```
block
:    '{' (selectorStatement | blockStatement)* '}'
;

selectorStatement
:    selector ';'
;
```

Добавим возможность использования селекторов в объявлениях переменных:

```
localVariableDeclaration
:    (selector|variableModifier* type)
     variableDeclarators

variableDeclarator
:    (selector | variableDeclaratorId) ('='
     variableInitializer)?
;
```

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:

```
WS
:    [ \t]+ -> channel(HIDDEN)
;

BR
:    [\r\n\u000C]+ -> channel(HIDDEN)
;

COMMENT
:    '/*' .*? '*/' -> channel(HIDDEN)
;

LINE_COMMENT
:    '//' ~[\r\n]* -> channel(HIDDEN)
;
```

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

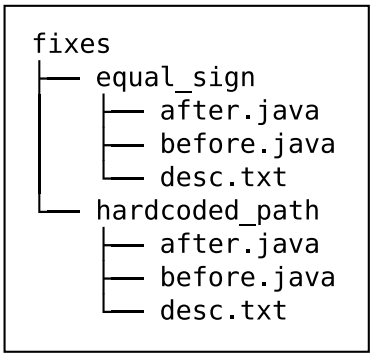

Рис. 4. Структура каталога с исправлениями

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 fool = "foo";
2 String foo2 = String.valueOf( new char [] { 'f',
      'o', 'o' } );
3 fool == foo2 // false!
4 fool.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:

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

Ниже приведена замена:

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

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

```
@Override public boolean equals(Object obj)
{
    LabelsEdge otherEdge = (LabelsEdge) obj;
    if ((this.source == otherEdge.source)
        && (this.target == otherEdge.target))
    {
        return true;
    } else {
        return false;
    }
}
```

Замена:

```
@Override public boolean equals(Object obj)
{
if (obj == null) {
    return false;
}
if (!obj.getClass().isInstance(this)) {
    return false;
}
    LabelsEdge otherEdge = (LabelsEdge) obj;
    if ((this.source == otherEdge.source)
        && (this.target == otherEdge.target))
    {
        return true;
    } else {
        return false;
    }
}
```

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/