А.П. Стасенко

ТЕСТИРОВАНИЕ ИЗМЕНЕНИЙ В ПРОГРАММНОЙ СИСТЕМЕ НА ОСНОВЕ ПОКРЫТИЯ ИСХОДНОГО КОДА

ВВЕДЕНИЕ

Регулярное тестирование сборок программной системы является основой современных практик разработки ПО [1]. Широкое распространение получила технология непрерывной интеграции, которая предполагает тестирование каждой правки исходного кода в дополнение к традиционному тестированию ночных сборок. Развитием данной идеи является использование набора тестов, запускаемого перед отправкой изменений в основную ветвь разработки; при этом 100% прохождение данных тестов является необходимым условием для попадания правок разработчика в хранилище исходного кода. Такой подход позволяет избегать ситуаций, когда серьезная ошибка, допущенная одним из разработчиков, блокирует работу целой команды.

Для сложных программных систем, состоящих из множества взаимодействующих компонент, характерно использование объемного предварительного тестирования, в связи с необходимостью проверки работоспособности каждой из составляющих частей системы. Ручное определение состава тестирования не всегда надежно, так как правки в одной из компонент могут отразиться на работе системы совершенно неожиданным для разработчика образом. На практике это приводит к существенному замедлению процесса разработки, так длительность предварительного тестирования является одним из факторов, ограничивающих скорость работы над проектом.

Подходы к сокращению объемов тестирования, основанные на сопоставлении изменений в исходном коде с данными о тестовом покрытии, описаны в литературе и хорошо изучены с теоретической стороны [2–3]. Их основная идея состоит в исключении из тестового прогона тестов, не покрывающих изменившиеся части программы. Внедрение подобных методик в процесс разработки больших программных систем сталкивается с рядом трудностей, которые во многом определены объемами данных о тестовом покрытии. Наибольшую проблему представляет поддержание этих данных в актуальном состоянии, так как тестирование инструментированных сборок идет в несколько раз (как правило, в 3–5 раз) дольше обычных тестирований [4]. Таким образом, возрастающая нагрузка на тестирующую систему может свести на нет выигрыш, достигнутый сокращением объемов

тестирования. Передача, хранение и оперативный анализ данных о тестовом покрытии также вызывает затруднения, так как их объем может достигать сотен гигабайт [5].

В данной статье будет рассказано об опыте внедрения этой технологии в процесс разработки существующего программного продукта, написанного на языках Си и Си++. Описанные выше проблемы были решены при помощи снижения детализации данных о тестовом покрытии. Вместо традиционного покрытия базовых блоков используется покрытие процедур (модулей, классов либо других крупных частей программы). Данный подход позволяет инструментировать только точки входа в процедуры, что существенно (до 80%) сокращает стоимость обновления данных о покрытии. Вместо отдельных тестов рассматриваются логические группы тестов (исходя из предположения, что тестовая база имеет некоторую структуру), что сокращает объемы данных и повышает прозрачность работы алгоритма. Общее снижение детализации данных о тестовом покрытии влечет их меньшую изменчивость, что позволяет обновлять эти данные не для каждой правки в исходном коде, а на периодической основе. В статье будут освещены и другие вопросы, связанные с практическим внедрением данной методики, такие как разработка алгоритма анализа изменений в исходном коде.

ОБЩАЯ СХЕМА СИСТЕМЫ

На рис. 1 приведена общая схема генерации оптимизированного набора тестов.

Предполагается, что при разработке программной системы используется система контроля версий (svn, cvs, git). Под ветвью разработки системы подразумевается английский термин branch. Под локальной копией системы – термин checkout. Под изменением кода системы – термин checkin.

Как видно из схемы, в качестве большой подготовительной работы должна быть собрана информация о покрытии кода программной системы на всём наборе имеющихся тестов или как минимум на наборе тестов, использующихся для обычного предварительного тестирования изменений. В результате регулярного сбора информации о покрытии кода генерируется отображения имён функций исходного кода программной системы во множество тестов, исполнение которых приводит к вызову этих функций.

Далее разработчик формирует локальную копию системы и вносит изменение, которое требуется протестировать. На основании этого изменения

и информации о процессе построения системы находится список измененных функций. Информация о процессе построения системы требуется, чтобы получить точный набор опций компиляции (в частности, препроцессора) и сгенерированные заголовочные файлы, от которых могут зависеть измененные единицы трансляции, без чего невозможно получить исходный текст единицы трансляции без директив препроцессора.

Автоматическое определение имён изменившихся функций необходимо для исключения возможных ошибок, которые может допустить программист при ручном формировании. Также в случае изменений объявлений или определения модульных переменных список зависящих от них функций не всегда очевиден.

На финальном этапе происходит генерация оптимизированного набора тестов как объединения множеств тестов, полученных на этапе сбора информации о покрытии кода, для каждой измененной функции.

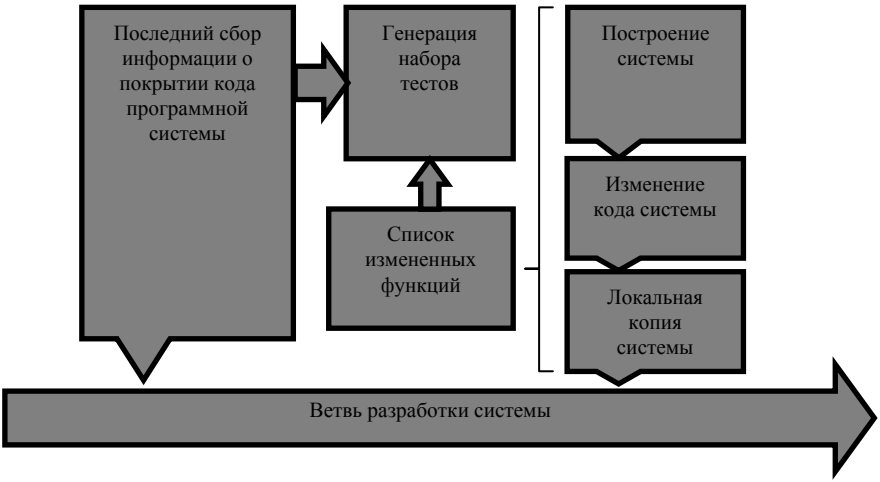

Рис. 1. Общая схема использования информации о покрытии кода для оптимизации набора тестов при тестировании изменений программной системы

РЕГУЛЯРНЫЙ СБОР ИНФОРМАЦИИ О ПОКРЫТИИ КОДА

Сбор информации о покрытии кода основывается на функциональности компилятора по сбору профилировочной информации (опция -prof-gen для

компилятора Intel, опция /profile для компилятора Microsoft, опция -p для компилятора gcc). Таким образом, требуется построить специальную версию программной системы с использованием этих ключей компилятора. Дальнейшее использование этой специальной версии системы при прогоне тестового набора будет вызывать создание специальных файлов с профилировочной информацией.

Как правило, из профилировочной информации можно извлечь подробные данные о покрытии линейных участков, однако для упрощения алгоритма поиска изменений в исходном коде и для сокращения объемов хранимой информации считаются только изменённые функции. Однако такое огрубление анализа сокращает количество тестов, которое можно удалить из регрессионного тестирования. Тест может покрывать функцию, затронутую изменением в исходном коде, но не покрывать ту часть её управляющего графа, где это изменение случилось.

Также для целей сокращения объемов хранимой информации было принято решение объединять информацию о покрытии кода для отдельных тестов в существующих тестовых сюитах (наборах тестов, имеющих общее назначение). Как уже было отмечено, регулярный сбор информации о покрытии кода требуется для генерации отображения имён функций в исходном коде программной системы во множество тестовых сюит, исполнение которых приводит к вызову этих функций.

Для целей сбора информации о покрытии кода нет необходимости выполнять все действия тестов, отличные от вызовов тестируемой программной системы, что позволяет существенно ускорить процесс сбора в некоторых случаях. Например, в случае тестирования компилятора можно не исполнять скомпилированные и слинкованные им тесты. Несмотря на эти возможности по ускорению прогона тестов, использование инструментированной программной системы для сбора информации о покрытии кода может замедлять исполнение тестов в несколько раз.

Как правило, сбор информации о покрытии кода требует незначительных модификаций тестовой системы для внедрения этапа объединения профилировочной информации после каждого теста тестовой сюиты, иначе в случае тестовых сюит с большим количеством тестов есть опасность переполнения дискового пространства машины профилировочной информацией индивидуальных тестов, что для каждого теста может занимать десятки мегабайт. Также в случае больших тестов, состоящих из тысяч вызовов программной системы, возникает потребность более раннего объединения профилировочной информации при достижении её определенного количе-

ства, что обычно реализуется с помощью оберток (wrapper) для вызовов программной системы.

Сбор информации о покрытии кода проводится на регулярной основе для ветви разработки системы, и его оптимальная частота зависит от типичной скорости добавления нового кода (функций) и времени, необходимого для сбора. Чем выше частота добавления новых функций или удаления существующих функций, тем чаще надо проводить новый сбор информации о покрытии кода. Однако, если время работы сбора занимает существенное время, то его частое исполнение может занять доступные тестовые ресурсы и свести на нет экономию от последующей оптимизации тестирования. Следует отметить, что регулярный сбор информации о покрытии кода может быть ценен не только для целей оптимизации тестирования изменений, но и, например, для планирования разработки новых тестов для слабо покрытых участков кода, поэтому в этом смысле для целей оптимизации тестирования регулярный сбор информации о покрытии кода может быть «бесплатным».

Временной разрыв между последним регулярным сбором информации о покрытии кода и изменением, для которого выполняется оптимизация регрессионного тестирования, строго говоря, нарушает свойство безопасности такой оптимизации. Безопасные (англ. *safe*) подходы либо не сокращают объемы тестирования, либо гарантируют, что исключение тестов не приведет к сокрытию дефектов. Как правило, происходит исключение тестов, результат исполнения которых не может быть затронут конкретными изменениями в исходном коде.

Для того чтобы явным образом нарушилось свойство безопасности, т.е. произошло сокрытие дефекта из-за сокращения объема регрессионного тестирования, необходимо выполнение следующих условий [4]:

1) агрегированные данные о тестовом покрытии должны были измениться с момента их последнего обновления, причем в «б**о**льшую» сторону;
2) рассматриваемая модификация исходного кода должна затрагивать часть программы, данные о покрытии которой устарели;
3) рассматриваемая модификация не должна затрагивать части программного кода, для которых данные о покрытии еще актуальны;
4) модификация должна привести к падению теста из группы тестов, которая была ошибочно исключена из тестирования;
5) дефект, вызвавший данное падение, должен быть специфическим для исключенной группы тестов и не проявиться на других тестах.

Сочетание данных событий возможно на практике, причем его вероятность тем выше, чем уже область программы, в которой в данный момент ведется активная разработка. В ходе практических экспериментов воспроизвести такую ситуацию не удалось. Следующие меры помогут снизить вероятность возникновения такой проблемы и опасность ее последствий: выбор удачного разбиения тестовой базы на группы, применение метода на подходящих стадиях жизненного цикла программного продукта, регулярное обновление данных о тестовом покрытии, периодическое тестирование на полной тестовой базе.

ПОИСК СПИСКА ИЗМЕНЁННЫХ ФУНКЦИЙ

После внесения изменения в исходный код программной системы для получения списка измененных функций требуется:

1) получить список измененных исходных текстов, которые для языков Си/Си++ можно разделить на два класса: единицы трансляции (translation unit) и заголовочные (header) файлы;
2) для каждого измененного заголовочного файла, используя данные о зависимостях между целями построения, определить множество зависимых от него единиц трансляции и объединить его с общим множеством измененных единиц трансляции;
3) препроцессировать текст единицы трансляции, для чего используются данные об опциях препроцессора и сгенерированных заголовочных файлах, полученные во время построения программной системы;
4) получить абстрактное синтаксическое дерево (AST) [6] для оригинальной и измененной версии единиц трансляции.

Использование сравнения абстрактных синтаксических деревьев позволяет игнорировать несущественные изменения в тексте программы, такие как изменения форматирования. В то же время препроцессирование встроенных макросов, таких как __LINE__ и __DATE__ добавляет неоднозначность, зависящую от форматирования программы. Для их устранения было принято решение предварительно обрабатывать все заголовочные файлы программной системы, текстуально заменяя эти макросы константными значениями, так как их переопределение через опции компилятора не всегда представляется возможным.

Изначально для построения абстрактного синтаксического дерева для поиска списка измененных функций был использован имеющийся синтакси-

ческий распознаватель (parser) языка Си/Си++ на языке Python, сгенерированный системой построения компиляторов ANTLR3. Однако высокое потребление памяти и низкая скорость работы такого Python распознавателя в реальных исходных текстах привела к поиску альтернативных решений, основанных на генерации внутреннего представления существующими компиляторами промышленного уровня.

Компилятор gcc предоставляет возможность получать абстрактные синтаксические деревья (AST) через опцию -fdump-syntax-tree, а специальная версия компилятора Intel позволяет генерировать дампы внутреннего представления и таблиц символов. Работающая версия системы основывается на дампах внутреннего представления компилятора Intel. Основное отличие данных дампов от абстрактного синтаксического дерева заключается в наличии отдельной модульной и локальной для каждой функции таблицы символов и заменой всех вхождений переменных на ссылки в эти таблицы.

С одной стороны, дампы компилятора позволяют дополнительно игнорировать некоторые несущественные изменения, такие как, например, переопределения типов, лишние скобки и иные несущественные для семантики элементы программы. С другой стороны, к именам переменных в дампах компилятора добавляется счётчик, который используется компилятором для разрешения неоднозначностей одинаковых имён в разных областях видимости. Таким образом, требуется исключать влияние случайного изменения счётчика для оригинальной и измененной версии программы. Также требуется устранять различия в декорировании имён Си++ (по крайней мере, в Windows), которые включают в себя дату изменения файла единицы трансляции.

Например, рассмотрим следующую простую Си-программу:

```
int i1 = 0;
int main() {
  int i2 = 0;
  return i1+i2;
}
```

Значительно усеченный дамп компилятора функции main и её символьная таблица будет выглядеть примерно следующим образом (жирным шрифтом выделены имена переменных с суффиксами добавленными компилятором для разрешения неоднозначности):

```
PACK| (i2.1) align: 0 MOD 4, size: 4, …
    VAR| (i2.1_V$1) type: SCALAR, size: 4,
       | offset: 0, esize: 4, …, edtype: SI32,  …
…
3       0           entry extern SI32  main
```

```
                    {
4       1               i2.1_V$1 = 0(SI32);
5       2               return ( (SI32) i1_V$0 + i2.1_V$1 );
6       3               return ;
                    }
Root Context C0.1 {
} C0.1
```

Модульная таблица символов выглядит примерно так:

```
PACK| (i1) align: 0 MOD 4, size: 4,
    | …, offset: 0, …
    VAR| (i1_V$0) type: SCALAR, size: 4,
           | offset: 0, esize: 4, …,
           | edtype: SI32,  …
    INIT| offset: 0, repeat: 1, ...,
           |data: 0(SI32)
```

Алгоритм нахождения изменённых функций между оригинальной и измененной версиями единицы трансляции работает следующим образом:

1) заменяет все вхождения имени переменной в функциях на её тип, её инициализацию и другие существенные семантические свойства в оригинальной и измененной версии единицы трансляции;
2) сличает все видимые извне единицы трансляции переменные с совпадающими именами между оригинальной и измененной версиями единицы трансляции и в случае нахождения различия в их семантических свойствах, сигнализирует, что уменьшение тестового набора не может быть осуществлено за приемлемое время, так как в этом случае требуется анализ всех единиц трансляции в программной системе;
3) текстуально сличает функции с совпадающими именами между оригинальной и измененной версиями единицы трансляции и сообщает функции с различиями в качестве искомых.

РЕЗУЛЬТАТЫ ОПТИМИЗАЦИИ ТЕСТИРОВАНИЯ

На рис. 2 приведены результаты оптимизации тестового набора, полученные для изменений, взятых за несколько месяцев в большой программной системе. Серый график показывает сокращение относительно полного набора тестов, а черный график – относительно набора тестов, обычно использующегося при тестировании изменений. График показывает процентное сокращение в наборе тестов и не учитывает их временные затраты, что, конечно, не совсем верно.

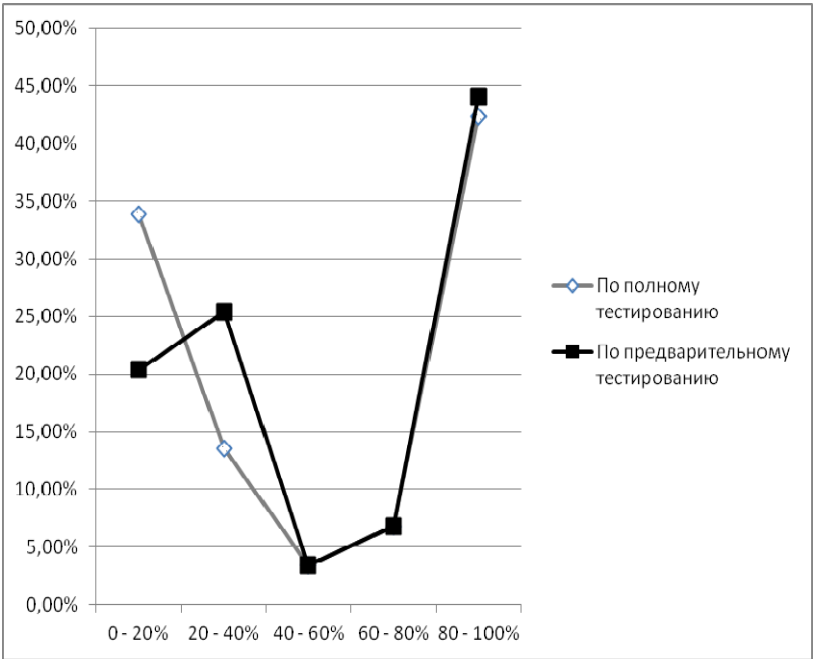

Рис. 2. Эффективность оптимизации тестирования изменений.
По оси X – процент тестов, оставшихся после оптимизации.
По оси Y – процент от общего количества изменений

Видно, что примерно в 45% случаев не удалось достичь существенного сокращения набора тестов (группа 80–100%). Однако для примерно 50% случаев удалось сократить объем тестов более чем в два раза, что является весьма неплохим результатом. Эксперименты также показали, что ожидаемо хорошо сокращалось тестирование у небольших изменений, тогда как тестирование больших изменений практически не сокращалось.

ЗАКЛЮЧЕНИЕ

Был налажен регулярный сбор информации о покрытии кода существующими тестами сложной программной системы, являющейся компилятором. Был разработан инструмент для определения измененных функций в

измененном исходном тексте программной системы, который реагирует только на фактические изменения синтаксической (AST дерева) или семантической (изменения определенных извне переменных) структуры функций. Для реализации инструмента использовались возможности по генерации внутреннего представления существующими компиляторами промышленного уровня. В результате применения данной системы в половине случаев удалось сократить объем требуемого тестирования изменений в два раза.

СПИСОК ЛИТЕРАТУРЫ

1. Leung H.K., White L. Insights into regression testing // IEEE Conf. on Software Maintenance. — IEEE Computer Society Press, 1989. – P. 60–69.
2. Rothermel G., Harrold M.J. A Safe, Efficient Regression Test Selection Technique // ACM Transactions on Software Engineering and Methodology. – 1997. –Vol. 6. – P. 173–210.
3. Rothermel G., Harrold M.J. Empirical Studies of a Safe Regression Test Selection Technique // IEEE Transactions on Software Engineering. – 1998. – Vol. 24.
4. A Framework for Reducing the Cost of Instrumented Code. Arnold M., Ryder B.G. б.м. // Proc. of the ACM SIGPLAN 2001 Confe. on Programming language design and implementation, 2001.
5. Салмин А.И. Исследование качества и эффективности функционального тестирования компилятора.– Новосибирск: НГУ, 2011.
6. Ахо А. Компиляторы: принципы, технологии, инструменты / Ахо А., Сети Р., Ульман Дж. — М: Издательский дом «Вильямс», 2003.