Применение динамического анализа для поиска дефектов в программах на языке Java

Вартанов С. П., Герасимов А. Ю.[1]
svartanov@ispras.ru, agerasimov@ispras.ru

Аннотация. В статье рассматриваются методы анализа программ и описывается практика применения данных методов для автоматизации задачи поиска ошибок в программном обеспечении. Подробно рассмотрен метод динамического анализа программ на основе инструментации, отслеживания потока помеченных данных и генерации наборов условий для автоматического построения входных данных. Рассмотрены особенности проведения подобного анализа для приложений, написанных на языке Java. Приведено описание прототипа инструмента, реализующего описанные подходы, представлены результаты его применения для анализа набора программ на языке Java и оценка полученных результатов.

Ключевые слова: итеративный динамический анализ программ; автоматический поиск ошибок; анализ Java программ.

1. Введение

На сегодняшний день в области разработки программного обеспечения одной из важных задач является гарантия качества конечных продуктов. Под качеством понимается совокупность характеристик программного обеспечения, а именно: надёжность, сопровождаемость, практичность, эффективность, мобильность и функциональность. В любой достаточно сложной программной системе присутствуют разного рода ошибки. Поскольку программное обеспечение используется и в критических сферах человеческой деятельности, наличие блокирующих или критических ошибок в программах может приводить к катастрофическим последствиям.

1 Работа проводится при финансовой поддержке Российского фонда фундаментальных исследований, номер проекта 11-07-00466-а

Поэтому неотъемлемой частью процесса разработки программ становится этап обнаружения ошибок. Среди различных методов поиска в программах дефектов можно выделить ручное и автоматическое обнаружение ошибок.

Общая структура статьи имеет следующий вид. Во введении даются общие сведения о предпосылках задачи проведения динамического анализа, даётся краткая аргументация выбранных подходов. Во второй части более подробно рассматриваются выбранные методы, исходя из особенностей языка Java. В третьей части описывается созданный прототип и использованные при его создании решения. Четвёртая, и заключительная, часть подводит итог исследования и демонстрирует основные результаты, полученные при помощи прототипа.

1.1 Автоматический поиск дефектов

Остановимся на рассмотрении случая, при котором использование понимания логики работы приложения невозможно или нежелательно из-за значительных временных затрат. При этом также будем считать, что известен исходный, исполняемый или интерпретируемый код приложения.

Подходы к решению задачи автоматического поиска дефектов традиционно разделяют на две группы — методы статического и методы динамического анализа. В случае статического анализа поиск возможных ошибок осуществляется без запуска исследуемого приложения, например, по исходному коду.

При этом следует подчеркнуть следующие характерные особенности статического анализа:

- Потенциально возможен полный анализ всего приложения. При этом возможно, что на любых входных данных реально выполняется лишь небольшая часть кода приложения.
- Возможны ложные срабатывания.

При обнаружении дефекта возникает, во-первых, проблема проверки истинности обнаруженного дефекта, и, во-вторых, проблема воспроизведения найденного дефекта при запуске программы на определённых входных данных.

1.2 Динамический анализ

Динамический анализ, подразумевающий запуск приложения на исполнение, в свою очередь, характеризуется следующими принципами:

- Для запуска программы требуются некоторые входные данные.
- Ложные срабатывания практически отсутствуют (исключением в данном случае может быть, например, утверждение о зацикливании

программы после истечения определённого временного интервала, в течение которого программа не завершается).

Основным достоинством динамического анализа можно считать отсутствие проблемы воспроизведения дефектов. Обширный опыт использования методов обеих групп показывает, что проведение динамического анализа, обеспечивающего как можно более полное покрытие исследуемой программы, требует значительных затрат времени.

Для методов статического анализа эта проблема выражается в меньшей степени, что во многом способствует более активному развитию новых подходов в этой области. Тем не менее, особенности динамического анализа, в частности, отсутствие проблемы воспроизведения дефектов и отсутствие ложных срабатываний, являются достаточными основаниями для предоставления времени и ресурсов для использования этого способа выявления ошибок.

1.3 Инструментация кода

Одним из возможных методов проведения динамического анализа является инструментация анализируемой программы. Под инструментацией программы понимается частичное изменение, при котором она (или её часть, которую необходимо исследовать) сохраняет свою функциональность, но также производит дополнительные действия, целью которых является извлечение информации о состоянии программы в ходе её выполнения, проверка отдельных значений и т. д.

Например, в инструментированную программу могут быть добавлены утверждения, которые должны быть верны всегда, когда исследуемая программа находится в заданной точке выполнения. Однако такой подход требует знания логики программы и понимания того, какие состояния программы являются допустимыми. Это означает, что его нельзя использовать для автоматического анализа.

Основным недостатком применения инструментации в процессе анализа программы являются временные потери при работе программы, возникающие из-за внедренных дополнительных инструментирующих инструкций.

1.4 Входные данные

Поскольку при динамическом анализе программы определяющим являются результаты её запуска, важно понимать, от чего зависят эти результаты. В самом простом случае поведение программы зависит от заранее фиксированных входных данных, например, файлов или аргументов командной строки.

Сложнее дело обстоит с интерактивными приложениями, поскольку в таком случае задача описания последовательности событий, которая бы полностью определяла поведение системы, становится нетривиальной. Это верно, например, для приложений с графическим интерфейсом, веб-приложений и т. д.

Для простоты изложения далее будем исходить из того, что анализируемая программа не является интерактивной и её выполнение полностью определяется конечным набором входных данных. В этом случае возникает возможность влиять на выполнение программы путём изменения входной информации.

В свете описанных ограничений рассмотрим промежуточную задачу: определить набор входных данных, на которых выполнение программы пойдёт по заранее определённому пути. Поскольку входная информация является дискретной, она может быть представлена в виде некоторой булевой формулы. В описанных условиях, любая используемая в ходе выполнения программы переменная либо имеет константное значение, либо значение, целиком зависящее от входных данных и пути выполнения программы, предшествующего использованию этой переменной.

Можно показать, что любой путь выполнения программы определяется набором условных переходов и выбором веток, следующих за ними. Таким образом, путь зависит от некоторого набора логических ограничений на значения используемых в этих условных переходах переменных, а значит полностью определяется некоторым булевым ограничением на входные данные.

Задача поиска входных данных, необходимых для того, чтобы программа прошла по заданному пути, может быть сведена к задаче проверки выполнимости булевых формул [1] с помощью решателей (solver).

1.5 Инвертирование условий и обнаружение дефектов

Рассмотрим более конкретно, как методы решения булевых формул вместе с вычислением входных данных для инвертирования условий можно применять для обнаружения ошибок в программах (удовлетворяющих указанным ограничениям).

Предполагается, что существует средство, способное после запуска программы на конкретных входных данных определить, произошла ли ошибка, обнаружена ли уязвимость или зафиксирована утечка памяти. Указанным средством может быть, например, инструмент Valgrind [5] для программ, представленных в виде набора процессорных инструкций, либо Java-машина для интерпретируемого ею байт-кода, которая в случае обнаружения ошибки выбрасывает исключение в поток ошибок.

Тогда сперва программа запускается на начальных входных данных, которые могут быть корректными (то есть соответствовать ожидаемому формату входного файла программы или протоколу) так и не корректными. Затем происходит анализ пути, по которому прошло вычисление. Для каждой вершины пути, соответствующей условному переходу, производится попытка изменить входные данные таким образом, чтобы при запуске на них вычисление программы в данной вершине пошло по альтернативному пути (имеется в виду, что в новом пути вычисления, исходящее из данной вершины ребро будет отлично от того, что было в предыдущем пути) [4].

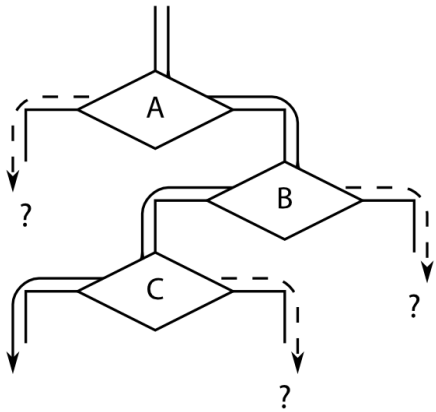

Рис. 1: Инвертирование условий

Очевидно, что такие входные данные удастся подобрать не всегда. Если, входные данные удалось обнаружить для нескольких условных переходов, на следующем шаге происходит выбор такого набора входных данных, который позволит наиболее эффективно продолжить анализ программы. В качестве примера может быть использована метрика, которая позволит проанализировать как можно большее количество ещё не проанализированных рёбер графа потока управления или тех рёбер, на которых вероятность возникновения ошибки наиболее высока.

С помощью описанного метода можно

- производить целенаправленный обход дерева путей выполнения,
- проверять наличие ошибок на заранее определённом пути выполнения.

2. Анализ программ на языке Java

Описанный выше метод проведения динамического анализа может быть применён для анализа широкого класса императивных языков: от анализируемого кода требуется строгая детерминированность выполнения последовательности действий, включая условные переходы.

В этой статье мы рассмотрим применение описанного подхода к программам на языке Java. Java — императивный объектно-ориентированный язык высокого уровня. Поскольку Java обладает средствами работы с многопоточностью, средствами генерации псевдослучайных последовательностей и прочими элементами, вносящими индетерминизм в работу программы, остановимся на подмножестве программ, не использующих данные средства. Таким образом, будем считать, что анализируемая программа имеет в качестве входных данных ограниченный набор информации и результат работы программы зависит исключительно от входного набора. Стоит отметить, что в силу широкой распространённости многопоточных приложений, таким ограничениям удовлетворяет лишь небольшой класс программ, однако зачастую описанным свойством обладают значительные части программ, требующих анализа, либо существуют простые преобразования, не нарушающие функциональности, которые приводят алгоритм в соответствие с требованиями детерминизма.

2.1 Инструментация

Для выполнения на виртуальной машине программа на языке Java транслируется в байт-код — промежуточное интерпретируемое представление программы в виде набора инструкций. Как видно из названия, любой код операции обязан умещаться в один байт, и, следовательно, число возможных операций не превышает 256.

В соответствии с описанным выше подходом, для извлечения информации о ходе выполнения программы будет применяться инструментация, т. е. изменение анализируемой программы, с целью сбора информации о работе программы.

Существует два основных способа проведения инструментации: статический и динамический. Динамическая инструментация заключается в проведении действий по изменению программы в ходе её выполнения, непосредственно перед интерпретацией. Основным достоинством динамической инструментации является то, что изменяются только действительно выполняемые части программы, т. е. не происходит избыточного изменения кода. Однако, в свете поставленной задачи, запуск программы должен происходить неоднократно, что должно приводить к повторной инструментации частей программы каждый раз, когда они будут выполняться.

Применительно к программам на языке Java это означает внесение изменений в класс-файлы (файлы, содержащие байт-код) анализируемой программы, включая класс-файлы используемых библиотек. В большинстве случаев в программе используется только некоторое подмножество классов и методов для каждого класса из подключаемых библиотек. Поэтому, прежде чем приступить к инструментации, необходимо определить это подмножество.

2.1.1 Определение списка методов для инструментации

Существуют два способа определения списка методов классов, для которых необходимо проводить инструментацию:

- Определять статически по исходному байт-коду и до запуска основного цикла программы.
- Динамически инструментировать методы, которые встретились в трассе, на каждой итерации основного цикла программы.

Какой из двух методов работает быстрее, зависит от особенностей анализируемого приложения. Влияние динамического способа на производительность программы и анализа будет выше в том случае, если в приложении в большинстве случаев используется только небольшая часть определённых в нём методов. Например, если в приложении реализованы несколько не связанных функциональностей, выбор одной из которых происходит в начале выполнения.

Однако этот метод не будет достаточно эффективным для использования в инструменте, поскольку после изменения входных данных методы, которые не вызывались при начальных входных данных, могут попасть в трассу. В этом случае придётся строить список классов и методов заново на каждой новой итерации проведения анализа и, в случае обнаружения новых используемых методов, проводить их инструментацию.

Для реализации в инструменте был выбран второй способ — статическое определение списка методов по коду программы. Этот метод имеет меньшее влияние на производительность программы и анализа в случае, если достаточно большая часть методов приложения выполняется при любых входных данных. Кроме того, уменьшается влияние на производительность самого процесса инструментации кода программы.

Построение списка методов, которые могут быть использованы, происходит итеративно. Изначально в список добавляются все методы из основного класса и считаются непомеченными. На каждом шаге итеративного анализа просматривается байт-код непомеченных методов из списка и в список непомеченных добавляются все методы, вызовы которых встретились. Просмотренные методы помечаются как инструментированные. Анализ завершается либо когда в списке не останется непомеченных методов, либо когда будет достигнуто ограничение числа шагов итеративного анализа.

Отдельно требуется добавить в список все методы инициализации классов, потому что их вызовы могут отсутствовать в байт-коде других классов, но они могут вызываться не явно самой средой выполнения программы.

2.2 Генерация условий на входные данные

Способ инструментации байт-кода программы полностью зависит от того набора информации, которую необходимо получить о ходе выполнения программы. С точки зрения обработки информации, можно выделить два основных способа инструментации: обработку по ходу выполнения программы и обработку после выполнения программы. Первый подход означает обработку информации в момент её получения, не дожидаясь окончания работы анализируемого приложения. Во втором случае в процессе выполнения программы происходит лишь сбор информации в некоторую трассу, которая будет проанализирована после завершения выполнения. Первый подход является более гибким, поскольку в ходе сбора информации можно оперировать уже извлечёнными данными и производить на их основе оптимизацию сбора оставшихся данных, однако может дополнительно увеличивать время выполнения программы, что может быть нежелательно.

Для применения описанного подхода будем использовать первый способ — обработку данных в процессе выполнения программы. Необходимо произвести инструментацию программы таким образом, чтобы в ходе выполнения программы производилось отслеживание помеченных данных (то есть таких данных, значение которых зависит от входных данных программы), определялись условные переходы, зависящие от помеченных данных, и производилось построение наборов условий для их инвертирования.

Условия, добавляемые в текущий набор, генерируются для всех операций с памятью (перемещение, копирование, арифметические операции и т. д.), если в них участвуют помеченные данные.

2.2.1 Условные переходы

Особые действия выполняются, если в трассе встретилась операция условного перехода. В терминах байт-кода это операции, в которых переход происходит в зависимости от значений операндов — это могут быть значения регистров, вершины стека, или передаваемых аргументов, в зависимости от виртуальной машины, на которой байт-код будет исполняться.

Если эта операция зависит от помеченных данных, сгенерированный на данный момент набор условий дублируется. В первую копию добавляется условие перехода, во вторую — его отрицание. Если условие данного перехода было истинным на текущих входных данных, вторая копия набора условий сохраняется отдельно, как набор, инвертирующий этот переход. Первая же копия используется для дальнейшего построения набора.

После этого, либо после завершения выполнения программы, построенный набор ограничений должен быть проверен на выполнимость, что равносильно проверке существования входных данных, удовлетворяющих этим ограничениям.

2.3 Итерационный механизм

Первый запуск анализа происходит на начальных входных данных. Это могут быть корректные, частично-корректные или совершенно произвольные входные данные. В ходе первого запуска фиксируются все встретившиеся условные переходы и значения условных выражений в этих переходах. Последовательность таких значений представляет собой двоичный вектор. Для краткости будем использовать 0 в значении «ложь» и 1 в значении «истина».

Допустим, после запуска вектор переходов выглядит следующим образом: (1, 0, 1, 0) — первый условный переход выполнился, второй не выполнился и т. д.

По числу переходов строятся векторы, инвертирующие каждый из них:

- (0);
- (1, 1);
- (1, 0, 0) и
- (1, 0, 1, 1).

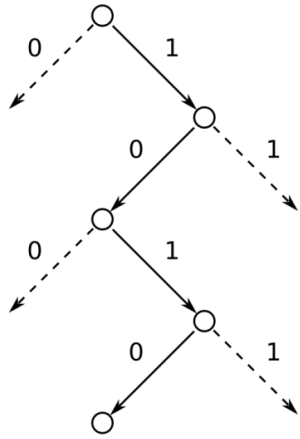

Рис. 2: Инвертирование переходов. Сплошной линией изображён ход выполнения приложения, пунктирными — возможные ветвления.

Каждый вектор соответствует очередному шагу работы анализа и набору утверждений, которые в случае выполнимости обеспечивают такие входные данные, на которых выполнимость первых n условных переходов (где n — длина вектора) соответствует значениям вектора.

Если рассматривать начальный вектор инвертированных переходов как вершину некоторого графа, а построенные векторы, как её дочерние вершины, то имеем упорядоченное корневое дерево. Потомки упорядочены по длине их векторов.

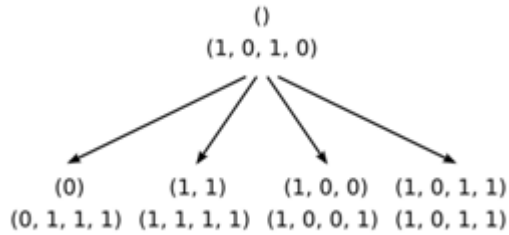

Рис. 3: Дерево переходов после первого шага. В вершинах изображены инвертирующий вектор и вектор запуска.

Для каждого из построенных векторов (т. е. для каждой построенной дочерней вершины) будет предпринята попытка генерации входных данных таким образом, чтобы в трассе запуска программы на этих данных первые n переходов соответствовали указанным в векторе инвертированных переходов.

После запуска вновь получаем вектор переходов. Если входные данные были построены верно, первые n значений в нём совпадают со значениями инвертирующего вектора.

Если длина нового вектора больше длины инвертирующего, все значения, начиная с $(n + 1)$ должны быть инвертированы. Если же длины векторов равны, вершина считается листовой и разбор данной ветки завершается.

Допустим, следующей будет раскрыта вершина с вектором (0).

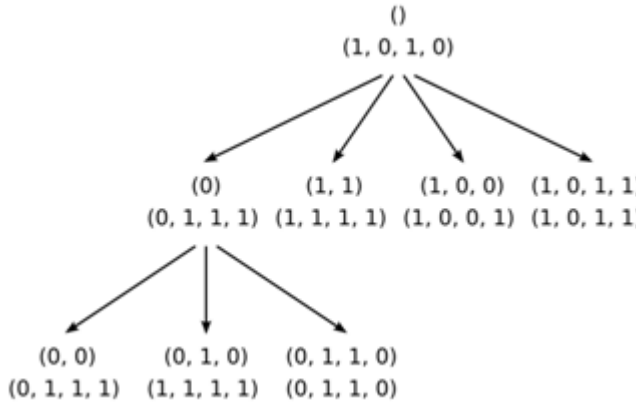

Рис. 4: Дерево переходов после второго шага.

На рисунке видно, что вершины с векторами (1, 0, 1, 1) и (0, 1, 1, 0) — листовые, а вершины (1, 1), (1, 0, 0), (0, 0) и (0, 1, 0) должны быть раскрыты.

Получаем дерево разбора всевозможных путей выполнения программы. Если в программе присутствуют m условных переходов, число вершин в дереве разбора может достигать 2^m. Соответственно, анализ всех путей выполнения программы может потребовать слишком большого времени. В связи с этим необходимо найти эвристики, которые позволят проводить целенаправленный поиск дефектов на наиболее вероятных или интересных с точки анализа путях исполнения программы.

2.3.1 Эвристики

После каждого очередного шага имеется набор векторов и соответствующие им утверждения. Необходимо иметь стратегию определения следующего вектора для анализа — наиболее перспективного с точки зрения обнаружения ошибок. Такая стратегия может быть выбрана, исходя из особенностей анализируемого приложения.

Наиболее простые стратегии — стратегии, не основывающиеся на какой-либо информации:

- обход дерева переходов в ширину,
- обход вглубь (с ограничением глубины или без).

Также можно предложить простые эвристические оценки для определения наиболее перспективной ветки — метрики трассы, полученной на предыдущем шаге:

- число инструкций,

- число потенциально опасных операций (например, целочисленное деление),
- число встретившихся условных переходов.

3. Прототип

Для проверки применимости предложенных методов к анализу программ на языке Java был разработан прототип инструмента динамического анализа. В этой части будут рассмотрены основные используемые средства и подходы.

3.1 Общая схема инструмента

В общих чертах схема инструмента может быть представлена следующим образом: шаг статической инструментации и далее вход в основной цикл работы, в ходе которого производится выполнение исследуемой программы, интерпретация и генерация входных данных.

Рис. 5: Общая схема инструмента

3.2 Инструментация

Описываемый прототип работает со стандартным байт-кодом, транслируемым, например, утилитой JDK javac и интерпретируемым, например, JVM (Java Virtual Machine).

Ограничив инструментацию определённым стандартом байт-кода, стоит отметить основное преимущество статической инструментации перед динамической. Динамическая инструментация происходит во время выполнения программы и может проводиться только при запуске программы на виртуальной машине, интерпретирующей соответствующий стандарт байт-кода. При этом, в случае статической инструментации, если существуют средства перевода между различными стандартами байт-кода,

инструментированный код может быть преобразован в необходимый формат без потери необходимой функциональности и анализ может быть произведён на виртуальной машине, не поддерживающей исходный формат байт-кода.

Примером использования этого преимущества может быть анализ программ на платформе Android, приложения на которой исполняются на виртуальной машине Dalvik и, таким образом, обязаны иметь иной формат байт-кода. Инструментированный байт-код может быть преобразован в формат DEX для анализа на целевой платформе.

Для проведения статической инструментации в прототипе используется библиотека BCEL — Byte Code Engineering Library [6]. Эта библиотека позволяет извлекать из класс-файлов информацию о содержащихся в них методах:

- сигнатуру метода,
- список констант, использующихся в методе,
- список инструкций и их параметры.

Также BCEL предоставляет возможности по изменению байт-кода: добавление, удаление, изменение инструкций; изменение списка констант; генерация новых методов.

На первом этапе работы инструмента происходит

- построение списка используемых в байт-коде методов,
- инструментация байт-кода исходного класс-файла и всех методов из списка.

В связи с тем, что обработка информации происходит в процессе выполнения программы, в прототип включены классы, предоставляющие необходимую функциональность по обработке информации о ходе выполнения программы. Инструментация заключается в добавлении в байт-код для каждой инструкции, если это требуется, предварительного и/или заключительного набора инструкций, добавляемых до и после этой инструкции соответственно. Каждый из этих наборов состоит из инструкций, производящих сохранение, копирование и минимальное преобразование различного рода аргументов, а также инструкций-вызовов методов классов, производящих обработку скопированной информации.

3.3 Генерация ограничений

Запуск анализируемой программы происходит с использованием инструментированных библиотек класс-файлов.

Для отслеживания помеченных данных и генерации ограничений на входные данные в процессе выполнения программы каждому значению ячейки памяти

(элемент стека или регистр) ставится в соответствие некоторая метапеременная. Изначально метапеременные сопоставляются байтам входных файлов. Если в ходе выполнения программы встречается операция, возвращающая значение, для каждого операнда проверяется, существует ли для него соответствующая метапеременная. Отсутствие метапеременной для всех аргументов означает, что в операции не участвуют помеченные переменные и она должна быть проигнорирована. Если же такие аргументы существуют, то создаётся новая метапеременная для результата операции и сопоставляется с ячейкой памяти, в которую результат записывается. В набор условий добавляется логическое утверждение — тождественное равенство, в левой части которого располагается созданная метапеременная, в правой части — метапеременные, соответствующие аргументам, или их извлечённые значения переменных, если метапеременные для них отсутствуют, связанные соответствующими операциями.

Для каждой операции условного перехода, как было сказано ранее, набор построенных ограничений дублируется, если в условие перехода входят переменные, которым сопоставлены метапеременные. Как и в случае с операциями, генерирующими значение, строится логическое выражение над метапеременными и извлечёнными значениями, соответствующее условию перехода в первой копии набора и соответствующее отрицанию этого условия во второй.

3.3.1 Проверка выполнимости ограничений

Для проверки выполнимости построенных ограничений используется инструмент STP — универсальный решатель логических ограничений [2].

На вход STP принимает последовательность логических утверждений — булевых операций над булевыми векторами — и как результат выдаёт сообщение о том, что данная последовательность является непротиворечивой, либо его ответом является контрпример — набор значений булевых векторов, при которых цепочка утверждений ложна.

Принцип работы инструмента STP заключается в переводе цепочки утверждений в конъюнктивную нормальную форму, после чего задача сводится к определению выполнимости булевой формулы, работа по проверки которой передаётся другому, более общему решателю [2].

В язык запросов STP специально разработан для поддержки основных логических операций, таких как логическое умножение и сложение, а также ряд арифметических операций, конструкция if — then — else и предикаты сравнения чисел, что делает его подходящим для решения поставленной задачи.

4. Результаты

4.1 Расход памяти при инструментации

В этом разделе приводится оценка увеличения размера класс-файлов после инструментации на примере набора пакетов.

Табл. 1: Изменение размеров файлов после инструментации

Пакет	Размер после инструментации	Размер до инструментации	Коэффициент
applet	20,2 КБ	7,3 КБ	2,7
awt	13,1 МБ	1,9 МБ	6,9
beans	1,9 МБ	359 КБ	5,3
io	2,3 МБ	352,2 КБ	6,5
lang	3,2 МБ	583 КБ	5,5
math	1,1 МБ	101,3 КБ	10,9
net	2,1 МБ	346,0 КБ	6,1
nio	2,1 МБ	376,5 КБ	5,6
rml	411,7 КБ	111,3 КБ	3,7
security	2,2 МБ	456,5 КБ	4,8
sql	283,7 КБ	102,4 КБ	2,8
text	2 МБ	279,6 КБ	7,2
util	11,1 МБ	1,9 МБ	5,8

В целом пакет увеличился в 6.16 раз. Похожий коэффициент можно ожидать от практически произвольного класс-файла.

4.2 Скорость работы

4.2.1 Инструментация

Скорость инструментации проверялась на классах из пакета rt.jar (реализация стандартной библиотеки классов Java для виртуальной машины HotSpot). Этот

архив содержит в себе 17 073 класс-файлов. Ниже приведены временные отметки инструментации этих классов:

Табл. 2: Время инструментации

Архив	Число классов	Время (секунды)	Скорость (классов в секунду)
rt.jar	16 796	96,220	174,55
tools.jar	3 679	21,623	170,14
ServeceabilityAgent	1 909	7,6	251,18
BCEL	383	2,568	149,14
JConsole	220	1,807	121,74
HTML Converter	52	1,353	38,43
dt.jar	47	0,878	53,53

Таким образом, за секунду инструментируется порядка сотни класс-файлов.

Такая скорость является приемлемой, поскольку инструментация производится один раз в ходе подготовки к итерации анализа и инструментируются не все подключённые класс-файлы, а те, чьи методы могут быть вызваны в ходе выполнения приложения.

Также на основе особенностей метода и приведённых результатов можно сделать вывод о том, что скорость инструментации линейно зависит от размера класс-файлов.

4.2.2 Выполнение

Проверка эффективности работы инструмента осуществлялась на приложениях с искусственно внесёнными ошибками (результаты приведены в таблице 6.3).

Тестовый пример. Демонстрирует основные возможности инструмента по нахождению ошибок, воспроизведение которых требует истинности большого числа условий в программе. Для такого приложения самым эффективным будет обход дерева разбора в глубину.

CSVReader. Представляет собой средство разбора простого формата данных CSV — набора слов и разделителей. Ошибка заключается в указании размера набора слов при помощи константного значения в коде программы. Инструмент обнаруживает входные данные, число слов в которых превышает

данную константу и вызывает исключение, связанное с выходом за границы массива. В этом случае наиболее эффективной оказывается эвристика, основывающаяся на подсчёте числа условных переходов.

Поиск пути в графе. Во входном файле задаётся граф в виде матрицы смежности и номера двух его вершин. Программа возвращает кратчайший путь из первой вершины во вторую. Ошибка заключается в том, что неверно оценивается размер буфера для хранения рёбер найденного пути.

Средство разбора арифметических выражений. Ошибка заключается в ограниченности размеров дерева разбора. Здесь также подсчёт числа условных переходов даёт наилучший результат.

Табл. 3: Время инструментации

Программа	Эвристика	Тип ошибки	Номер итерации и время обнаружения первой ошибки (сек.)
Тестовый пример	В глубину	Деление на ноль	4 (0,32 с)
	Число инструкций		7 (0,54 с)
CSVReader	В глубину	Выход за границы массива	43 (5,06 с)
	Число инструкций		39 (4,78 с)
	Число условных переходов		39 (4,78 с)
Поиск пути в графе	В глубину	Выход за границы массива	30 (6,51 с)
	В ширину		30 (6,5 с)
	Число условных переходов		32 (7,1 с)
Разбор арифметических выражений	В ширину	Выход за границы массива	29 (4,8 с)
	Число условных переходов		27 (4,6 с)

На основе приведённых результатов можно сделать вывод о том, что прототип инструмента демонстрирует эффективную работу по обнаружению ошибок, воспроизведение которых требует большого числа одновременно выполняющихся условий. При этом прототип инструмента не использует информацию о логике работы программы.

5. Заключение

В результате проведённого исследования был реализован прототип средства, осуществляющего динамический анализ приложений на языке Java.

В инструменте применена статическая инструментация байт-кода с целью получения трассы выполнения программы. В ходе анализа происходит извлечение трассы, частичная интерпретация и отслеживание потока помеченных данных, создание набора логических утверждений для инвертирования условных переходов и генерация новых входных данных для покрытия новых путей выполнения программы.

После проведения ряда тестов была подтверждена эффективность статической инструментации, проводимой один раз перед выполнением анализа.

В результате запуска анализа тестовых приложений инструмент продемонстрировал способность эффективно обнаруживать ошибки и создавать наборы входных данных для их воспроизведения без наличия исходного кода и информации о логике работы приложения. Генерация входных данных исключает возможность ложных сообщений об обнаруженных ошибках.

Для ускорения проведения анализа реализованы эвристические методы выбора наборов входных данных.

В качестве направлений для дальнейших исследований можно указать следующие:

- Разработка полноценного инструмента, позволяющего анализировать входные данные программ из различных источников: переменные окружения среды выполнения, сокеты и т. п.
- Разработка эвристик и алгоритмов для быстрого целенаправленного анализа интересующих частей приложения, например отдельно взятой функции или подсистемы в рамках приложения

Список литературы

[1]. Новикова Н. М. Основы оптимизации. М.: МГУ, 1998. 17–22 с.
[2]. Eén N., Sörensson N. MiniSat solver [HTML] (http://minisat.se/)
[3]. Ganesh V., Dill D. L. A Decision Procedure for Bit-Vectors and Arrays // In Proceedings of Computer Aided Verification. 2007. P. 524–536.
[4]. Исаев И. К., Сидоров Д. В. Применение динамического анализа для генерации входных данных, демонстрирующих критические ошибки и уязвимости в программах // Программирование. 2010. № 4. С. 1-16.
[5]. Valgrind. Instrumentation Framework for Building Dynamic Analysis Tools [HTML] (http://valgrind.org/)
[6]. Apache Commons Byte Code Engineering Library [HTML] (http://commons.apache.org/bcel)

Applying dynamic analysis for defect detection in Java-applications

S. Vartanov, A. Gerasimov
svartanov@ispras.ru, agerasimov@ispras.ru
ISP RAS, Moscow, Russia

Abstract. This paper provides an overview of program analysis techniques, and highlights details of practical implementation of static and dynamic approaches for automatic software defect detection, their pros and cons. The paper focuses on dynamic program analysis technique. The major advantage of this technique is the absence of defect reproduction problem. This approach is based on tainted data flow tracing, instrumentation and constraint set construction for automatic input generation.

An overview of practical considerations for developing a dynamic analysis tool for Java applications is given. The paper describes distinctive feature of Java bytecode static instrumentation approach and related static dependencies detection problems. It provides details of path conditions generation (set of Boolean formulas), paths coverage-based iterative mechanism. Input generating problem is solved using solver for checking Boolean constraints satisfiability. Heuristics are used for exponential growth problem solving.

It is complemented by a detailed description of actual prototype implementation created within the scope of this project. The prototype uses BCEL for static instrumentation and STP solver for path condition solving. Finally, the paper features an overview of practical results obtained on a number of Java applications and provides an evaluation of these results.

Keywords: software iterative dynamic analysis; automatic defect detection; Java bytecode instrumentation

References

[1]. Novikova N. M. Osnovy optimizatsii [The Basics of Optimization]. M.: MGU [MSU], 1998. 17–22 p. (in Russian)
[2]. Eén N., Sörensson N. MiniSat solver [HTML] (http://minisat.se/)
[3]. Ganesh V., Dill D. L. A Decision Procedure for Bit-Vectors and Arrays. In Proceedings of Computer Aided Verification. 2007. P. 524–536.
[4]. Isaev I. K., Sidorov D. V. Primenenie dinamicheskogo analiza dlya generatsii vkhodnykh dannykh, demonstriruyushhikh kriticheskie oshibki i uyazvimosti v programmakh [The Use of Dynamic Analysis for Generation of Input Data that Demonstrates Critical Bugs and Vulnerabilities in Programs]. Programmirovanie [Programming and Computer Software]. 2010. # 4. P. 1-16. (in Russian)
[5]. Valgrind. Instrumentation Framework for Building Dynamic Analysis Tools [HTML] (http://valgrind.org/)
[6]. Apache Commons Byte Code Engineering Library [HTML] (http://commons.apache.org/bcel)