Динамический анализ приложений с графическим пользовательским интерфейсом на основе символьного исполнения[1]

С.П. Вартанов <svartanov@ispras.ru>
А.Ю. Герасимов <agerasimov@ispras.ru>
М.К. Ермаков <mermakov@ispras.ru>
Д.О. Куц <kutz@ispras.ru>
А.А. Новиков <a.novikov@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. Данная статья посвящена исследованию возможностей применения современных методов динамического анализа программ на основе символьного исполнения к программному обеспечению, предоставляющему графический пользовательский интерфейс. Отличительными особенностями подобных программ является интерактивная обработка данных и применение параллельных потоков команд. Данные особенности значительно усложняют эффективность применения подходов автоматического обхода путей выполнения на основе символьного исполнения. В рамках статьи предлагается проводить анализ программ, предоставляющих графический интерфейс, с помощью гибридного метода, включающего символьное исполнение и стандартные подходы к извлечению модели графического интерфейса для построения тестовых сценариев. В статье представлен обзор существующих программных средств, предоставляющих возможности анализа и тестирования программ, и выделены средства GUITAR и Coffee Machine, совмещение которых позволяет эффективно анализировать Java-программы с графическим пользовательским интерфейсом. Рассматривается схема внедрения модулей инструментации байт-кода системы Coffee Machine в рабочий цикл инструмента GUITAR. Модель структуры графического интерфейса, извлекаемая инструментом GUITAR, расширяется фрагментами предикатов пути, построенных с помощью символьного исполнения. Представлен алгоритм составления предикатов в сложные трассы, обрабатываемые инструментами проверки выполнимости булевых ограничений, позволяющий автоматически генерировать тестовые сценарии для обхода различных путей выполнения по коду функций обработки событий взаимодействия с элементами графического интерфейса. Представлены практические результаты

[1] Работа проводится при финансовой поддержке Российского фонда фундаментальных исследований, номер проекта 14-07-00609

применения совмещенного метода, позволившего обнаружить необработанные исключения в ряде проектов с открытым исходным кодом, и дана оценка полученных результатов. В заключении статьи даётся оценка эффективности предложенного метода и рассмотрены основные ограничения, избавление от которых представляется актуальным направлением дальнейших исследований.

Ключевые слова: динамический анализ программ; анализ программ; тестирование GUI, тестовое покрытие.

DOI: 10.15514/ISPRAS-2017-29(1)-10

Для цитирования: Вартанов С.П., Герасимов А.Ю., Ермаков М.К., Куц Д.О., Новиков А.А. Динамический анализ приложений с графическим пользовательским интерфейсом. Труды ИСП РАН, том 29, вып. 1, 2017 г., стр. 149-166. DOI: 10.15514/ISPRAS-2017-29(1)-10

1. Введение

В настоящее время современные высокотехнологичные устройства становятся неотъемлемой частью повседневной жизни человека. Растущая популярность компьютерных технологий неразрывно связана с применением графического пользовательского интерфейса операционной системы и прикладных приложений, который позволяет предоставлять информацию и управлять устройствами в интуитивно понятном виде. Например, количество пользователей услуги мобильного банкинга для частных лиц в России за 2015 год увеличилось на 58 % [1]. Создание приложений с графическим пользовательским интерфейсом сопряжено с достаточно трудоёмким процессом тестирования разработанного приложения. Сложность процесса тестирования связана с количеством элементов графического пользовательского интерфейса приложения: чем больше элементов управления графическим пользовательским интерфейсом используется приложением, тем больше количество возможных сценариев работы, которые необходимо протестировать.

Современные средства разработки программ предоставляют возможности автоматизации создания приложений с графическим пользовательским интерфейсом. Данные инструментальные средства ориентированы на создание модели, описывающей структуру графического пользовательского интерфейса. На основе модели при участии эксперта создаются тестовые сценарии, описывающие последовательность действий над элементами графического пользовательского интерфейса. Воспроизведение тестовых наборов и отслеживание работы программы осуществляется автоматически. По результатам исполнения программы с использованием тестовых наборов контролируется качество программного обеспечения. В то время как такой подход позволяет приблизить набор тестовых сценариев к ожидаемым от пользователя действиям, он не позволяет подробно исследовать алгоритмы, лежащие в основе функций-обработчиков событий от элементов

пользовательского интерфейса. Если данные алгоритмы достаточно сложны, то вероятность обнаружения последовательности действий пользователя, приводящей к нарушению работы программы, является достаточно низкой. Подобная проблема актуальна, например, для программ, предоставляющих графический интерфейс пользователя для составления сложных выражений и запуска обработки данных выражений. Простой обход последовательности взаимодействия с элементами графического пользовательского интерфейса не позволит обнаружить экстремальные ситуации и краевые случаи для встроенного разборщика выражений и модуля подсчета значений. Можно отметить, что графический интерфейс в данном случае маскирует наиболее сложные фрагменты кода программы, а инструменты, составляющие тестовые сценарии по извлеченной модели структуры графического интерфейса, будут проводить недостаточно эффективный анализ.

В настоящее время развиваются методы итеративного динамического анализа программ, проводящие обход различных путей выполнения с целью выявления краевых значений входных данных для алгоритмов программы и обнаружения потенциальных дефектов и уязвимостей. Данные методы используют принципы символьного исполнения и анализа потока помеченных данных программы для сбора предикатов пути (трасс ограничений) и решатели булевых ограничений (солверы) для автоматического построения наборов входных данных.

К сожалению, их прямое применение к приложениям, предоставляющим графический пользовательский интерфейс (и в общем смысле – к интерактивным приложениям) часто не бывает достаточно эффективным. К основным факторам, обуславливающим данный эффект, можно отнести следующие:

- активные потоки данных проходят через библиотеки, осуществляющие отображение элементов графического интерфейса и обработку взаимодействия с данными элементами со стороны пользователя, что приводит к необоснованному увеличению объема трасс выполнения;
- многопоточное выполнение, в рамках которого происходит ожидание и отклик на действия пользователя, затрудняет выделение актуальных фрагментов трассы.

В связи с этим становится актуальной задача создания и развития методов анализа программ, объединяющих подходы автоматического тестирования приложений с графическим пользовательским интерфейсом и динамического символьного исполнения. Построение модели графического интерфейса будет предоставлять возможность создания корректных сценариев взаимодействия с программой, а применение символьного исполнения позволит исследовать логику работы программы в рамках данных сценариев.

В данной статье описан подход, совмещающий автоматическое построение модели графического интерфейса с анализом, основанным на символьном

исполнении кода программ. В рамках подхода проводится исследование программ на языке Java.

Статья имеет следующую структуру. В разд. 2 приводится краткое описание методик анализа, совмещение которых позволит получить преимущества для автоматического анализа программ с графическим пользовательским интерфейсом, и существующих инструментов, реализующих методики в рамках данных направлений. В разд. 3 описаны основные положения предлагаемого совмещенного метода анализа. В разд. 4 рассматриваются результаты применения совмещенного метода анализа к ряду проектов с открытым исходным кодом и дается оценка полученных результатов. В заключении представлена оценка проделанной работы и рассмотрены перспективные направления дальнейшего развития темы.

2. Обзор существующих решений

В данной главе производится обзор существующих методов и инструментов, используемых для тестирования программ с графическим пользовательским интерфейсом и применяемых для динамического символьного исполнения программ.

2.1 Автоматизация тестирования программ с графическим пользовательским интерфейсом.

Большинство средств тестирования программ с графическим интерфейсом в своей работе используют подходы, не зависящие от кода исследуемых программ. Стандартная схема данных подходов включает в себя работу с моделью графического пользовательского интерфейса программы, описывающей структурные элементы данного интерфейса и особенности взаимодействия между ними. С помощью языка описания тестовых сценариев на основе данной модели осуществляется создание тестового покрытия; для выполнения данных сценариев используются компоненты, программно эмулирующие действия пользователя по взаимодействию с графическим пользовательским интерфейсом программы.

Среди успешных и распространённых инструментов, следующих данной методике, можно выделить такие средства, как Ranorex [2], Abbot [3], Maveryx [4], Squish [5], Sikuli [6]. Основные отличия данных инструментальных средств сводятся к особенностям модели структуры графического интерфейса, механизмам идентификации и взаимодействия с элементами графического интерфейса, языкам описания тестовых сценариев.

Отдельно стоит выделить инструмент GUITAR [7], разработанный в Мэрилендском университете. В отличие от указанных выше инструментов GUITAR позволяет составлять сценарии тестирования в полностью автоматическом режиме без участия эксперта. За счёт данного свойства инструмент GUITAR наилучшим образом подходит для совмещения с другими

автоматическими методами динамического анализа программ и в частности, динамического анализа на основе символьного исполнения.

Инструмент GUITAR имеет четыре взаимосвязанных компоненты, схема работы которых представлена на рис. 1.

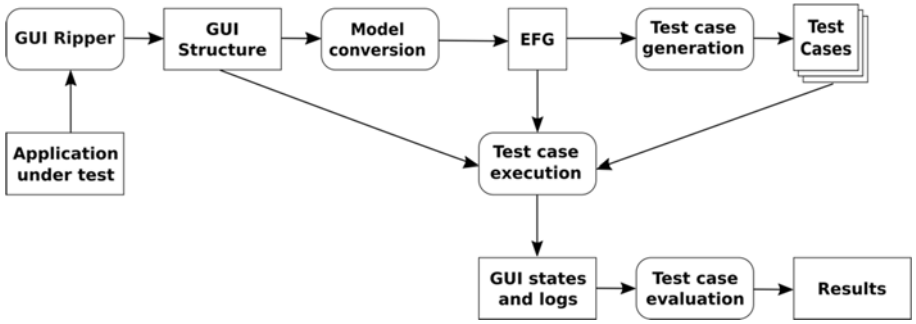

Рис. 1. Схема работы инструмента GUITAR

Fig. 1. GUITAR tool scheme

Практически каждое приложение с графическим пользовательским интерфейсом имеет несколько окон, которые открываются в результате определенных действий пользователя. Графический пользовательский интерфейс приложения представляет собой иерархическую структуру, в которой верхним элементом является главное окно, на средних уровнях располагаются контейнеры элементов графического пользовательского интерфейса и на нижних — контейнеры и отдельные элементы. Компонент инструмента GUITAR Ripper проводит анализ всех окон и иерархии контейнеров и элементов управления графического пользовательского интерфейса. При анализе окон, отображаемых на экране, компонент Ripper инициирует выполнение обработчиков событий взаимодействия с элементами интерфейса, находящихся в отображаемых окнах. Таким образом имитируются действия пользователя. Компонент Ripper сохраняет информацию об элементах графического интерфейса, об изменениях структуры графического пользовательского интерфейса, которые произошли после совершения действий, а также параметры отдельных объектов. Сохраняемая информация записывается в формате XML и составляет модель структуры графического интерфейса, с которой будет производиться дальнейшая работа инструмента.

После получения необходимой информации об элементах графического интерфейса, строится EFG (Event Flow Graph) – ориентированный граф потока событий. В нем описываются взаимосвязи между элементами графического интерфейса. Связи между элементами в графе описывают множество возможных последовательностей взаимодействия с приложением. Произвольный путь в графе от одного из элементов в окне приложения,

отображаемом при запуске, до другого элемента задает возможный тестовый сценарий. Генерация тестовых сценариев проводится компонентом Test case generator, осуществляющим выделение путей в графе по заданным алгоритмам (например, случайная выборка или исчерпывающая выборка всех путей заданной длины и т. д.). Воспроизведение тестового сценария осуществляется компонентом Replayer.

Описанный алгоритм предоставляет возможность инструменту автоматически формировать тестовое покрытие в виде воздействия на элементы графического пользовательского интерфейса и воспроизводить его.

2.2 Автоматизация обхода путей выполнения на основе символьного исполнения

Методы итеративного динамического анализа предоставляют возможность автоматически обходить возможные пути выполнения программ. Центральным механизмом в рамках итеративного динамического анализа является символьное исполнение. В рамках символьного исполнения для произвольного пути выполнения собирается трасса булевых ограничений (предикат пути), задающая связь между входными данными программы и внутренними данными программы, обрабатываемыми во время выполнения. В составляемых трассах ограничений можно выделять точки ветвления, соответствующие условным конструкциям. Модификация булевого ограничения, порожденного инструкциями в точке ветвления, позволяет описать предикат, соответствующий альтернативному пути выполнения.

Модифицированные предикаты пути передаются на обработку инструментам проверки выполнимости булевых формул, которые позволяют установить, является ли предикат выполнимым на каком-либо наборе значений входных данных. В случае выполнимости инструменты позволяют построить один из подобных наборов. Запуск программы на полученном наборе приведет к ее выполнению по альтернативному пути. Последовательный обход и обработка точек ветвления в программе позволяют осуществлять создание наборов входных данных для исследования различных путей выполнения, обеспечивая высокий уровень покрытия кода.

Существующие инструменты, использующие рассмотренные выше принципы, отличаются подходами к обработке кода программы с целью построения предикатов пути, дополнительными возможностями по оптимизации предикатов пути и метриками, определяющими последовательность обхода точек ветвления в программе.

Инструмент Java Pathfinder [8, 9], разработанный в NASA и представляющий из себя специализированную виртуальную машину Java, стал основой для создания различных инструментальных средств для анализа и тестирования Java-программ. В инструменте реализован подход к проведению динамического анализа, базирующийся на символьном исполнении. Код

исследуемой программы инструментируется с целью получения дерева состояний приложения. Данное дерево как пространство состояний анализируется с помощью проверки на модели (model-checking). В процессе динамического символьного исполнения строятся булевы ограничения, на основе которых генерируются наборы входных значений для тестирования программы. Java PathFinder позволяет генерировать тестовые входные данные и строить контрпримеры для отдельных свойств модели.

JDart [10] – инструмент динамического анализа программ, разработанный в Исследовательском центре Эймса NASA. Инструмент реализует символьное исполнение программ и состоит из двух основных модулей — *Executor* и *Explorer*. Модуль Executor отвечает за исполнение анализируемой программы и создание символьных ограничений на используемые данные. Данный модуль реализован как расширение инструмента Java PathFinder. Explorer позволяет находить неисследованные пути исполнения программ на основе символьных ограничений. Данный модуль использует библиотеку JConstraints в качестве дополнительного уровня абстракции для эффективного преобразования символьных выражений и предоставления общего интерфейса решателям булевых ограничений. Инструмент поддерживает такие решатели, как CORAL, SMTInterpol и Z3. Дополнительным модулем JDart является JUnit для генерации тестовых наборов.

Инструмент JavaFAN [11] предназначен для формального анализа многопоточных Java-программ, как на уровне исходного кода, так и на уровне байт-кода виртуальной машины Java. Инструмент поддерживает символьное исполнение программ, обход в ширину пространства состояний программы с целью нахождения нарушений свойств безопасности, проверку на модели свойств логики линейного времени (LTL) для программ с конечным пространством состояний. В ходе работы инструмента Java-программа транслируется в Maude-описание, что позволяет проводить более эффективное исполнение, поиск и проверку на модели LTL.

Инструмент Coffee Machine [12], разработанный в Институте системного программирования РАН, предоставляет возможности статической инструментации байт-кода программ Java с целью создания трасс булевых ограничений и трасс событий, использующихся для поиска дефектов синхронизации [13]. Управляющие модули инструмента позволяют организовывать обход возможных путей выполнения программы и интеграцию с решателями булевых ограничений для построения наборов путей.

Применение статической инструментации делает средство Coffee Machine независимым от конкретной реализации виртуальной машины Java, что расширяет область применимости инструмента (в частности, позволяет осуществлять анализ приложений операционной системы Android, использующей виртуальную машину Dalvik).

3. Особенности предлагаемого совмещенного метода анализа

По итогам анализа предметной области в качестве базовых инструментов для исследования и разработки методов совмещенного анализа приложений, предоставляющих графический пользовательский интерфейс, были выбраны средства GUITAR и Coffee Machine. Выбор инструментов обусловлен следующими обстоятельствами:

- Инструменты GUITAR и Coffee Machine распространяются свободно вместе с исходным кодом, что позволяет проводить модификацию и расширение его функциональности;
- Инструмент GUITAR предоставляет поддержку ряда библиотек работы с графическим интерфейсом (SWT, Swing для стандартной виртуальной машины Java HotSpot; библиотека организации интерфейса приложений операционной системы Android для виртуальной машины Dalvik);
- Инструмент Coffee Machine использует статическую инструментацию байт-кода для обработки программ, что позволяет использовать его для различных виртуальных машин, включая целевые Java HotSpot и Dalvik. Модули инструментации, проводящие модификацию кода с целью создания трасс булевых ограничений при выполнении программы, могут использоваться независимо от управляющего модуля Coffee Machine, что позволяет с легкостью встроить их в инструмент GUITAR.

В рамках совмещенного метода анализа используется схема инструмента GUITAR для определения модели структуры графического интерфейса исследуемой программы и выявления связей между элементами графического интерфейса для создания тестовых сценариев. Определение модели и исследование тестовых сценариев проводится для заранее проинструментированной программы, что позволяет строить фрагменты трасс ограничений для функций-обработчиков событий взаимодействия с элементами пользовательского интерфейса. Фрагменты трасс объединяются в более сложные конструкции для определения тестовых наборов, позволяющих исследовать перспективные точки ветвления в обработчиках событий взаимодействия с элементами графического интерфейса.

3.1 Символьное исполнение программ с графическим пользовательским интерфейсом

Существующие инструменты динамического анализа на основе символьного исполнения стандартно обрабатывают такие источники входных данных, как файлы, сетевые интерфейсы, переменные окружения и др. Исследование того, как программа производит доступ к содержимому данных источников, обычно

реализуется с помощью перехвата входных данных, полученных от функций и системных вызовов. Перехваченные данные используются программой непосредственно и явно участвуют в потоке данных. Составление предиката пути позволяет задать функциональную зависимость между символьными переменными, описывающими значения входных данных и значения операндов инструкций.

Для приложений с графическим интерфейсом понятие входных данных определяется более широко. Помимо рассмотренных выше источников (файлы, сетевые интерфейсы и др.) в качестве входных данных необходимо рассматривать взаимодействие пользователя с элементами интерфейса. Последовательность действий (тестовый сценарий) непосредственно определяет работу программы, однако в явном виде не может быть связана с потоком данных, обрабатываемым инструкциями кода (на уровне ячеек памяти, регистров и т. д.). Для обеспечения возможности манипуляции составляющими тестового сценария необходимо ввести в трассу ограничений новые символьные переменные, задающие идентификаторы элементов графического интерфейса, и новые ограничения, связывающие данные переменные с потоком данных программы. Подобные модификации трассы ограничений являются частичным решением известной проблемы символьного исполнения, относящейся к так называемым зависимостям по потоку управления.

Построение трасс булевых ограничений в рамках совмещенного подхода базируется на следующих принципах:

- В трассы ограничений попадают только инструкции, выполняющиеся в функциях обработки действий над элементами графического интерфейса (например, в функциях, выполняющихся как отклик на нажатие кнопки или изменение состояния переключателя) или до вызова первой функции обработки действий.
- При отслеживании потока данных в программе применяется модель перепомеченности – все долгоживущие объекты, которые могут использоваться одновременно в нескольких функциях обработки действий над элементами графического интерфейса, рассматриваются как символьные переменные. Для байт-кода Java к таким объектам относятся:
    - Содержимое статических полей примитивных типов загруженных виртуальной машиной классов.
    - Массивы примитивного типа и поля объектов, доступ к которым можно осуществить из статических полей загруженных виртуальной машиной классов.
- Промежуточные объекты, создаваемые внутри функций обработки действий над элементами графического интерфейса, рассматриваются как символьные переменные только в том случае, если они напрямую зависят от объектов, указанные выше.

- Фрагмент предиката пути, порождаемый инструкциями функции обработки действия над элементом графического интерфейса, обрамляется специальными маркерами начали и конца события.

3.2 Сбор предикатов пути и составление тестовых сценариев

Обработка программы при помощи инструмента GUITAR осуществляется для исполняемых файлов, инструментированных с целью построения трасс ограничений. Поэтому при составлении модели структуры графического интерфейса существует возможность сопоставить каждой паре {элемент графической интерфейса, действие над графическим интерфейсом} фрагмент подтрассы. В получаемых фрагментах подтрасс выделяются точки ветвления. По общей схеме проведения динамического анализа на основе символьного исполнения осуществляется модификация ограничения в точке ветвления для обхода альтернативного пути выполнения. Однако расчёт входных данных для модифицированной трассы ограничений необходимо осуществлять с учетом других фрагментов трасс, соответствующих обработке действий над элементами графического интерфейса, т. к. именно последовательность действий и является набором входных данных.

Для достижения подобного эффекта в рамках GUITAR используется модифицированный модуль генерации тестовых сценариев. Данный модуль составляет трассу ограничений особой структуры, учитывая граф потока событий и модель графического интерфейса. Общая структура трассы выглядит следующим образом:

$$\begin{gathered}((e_1 = i_{11}) \wedge (e_2 = i_{12}) \wedge \dots \wedge (e_N = i_{1N})) \vee \\ ((e_1 = i_{21}) \wedge (e_2 = i_{22}) \wedge \dots \wedge (e_N = i_{2N})) \vee \\ \dots \\ ((e_1 = i_{K1}) \wedge (e_2 = i_{K2}) \wedge \dots \wedge (e_n = i_{KN}))) \wedge \\ ((((e_1 = i_1) \wedge it_1) \vee ((e_1 = i_2) \wedge it_2) \vee \dots \vee ((e_1 = i_M) \wedge it_M)) \vee \\ (((e_2 = i_1) \wedge it_1) \vee ((e_2 = i_2) \wedge it_2) \vee \dots \vee ((e_2 = i_M) \wedge it_M)) \vee \\ \dots \\ (((e_{N-1} = i_1) \wedge it_1) \vee ((e_{N-1} = i_2) \wedge it_2) \vee \dots \vee ((e_{N-1} = i_M) \wedge it_M))) \wedge \\ ((e_N = i_X) \wedge it_X)\end{gathered}$$

Здесь:

e_i - специальная символьная переменная, определяющая i-ое событий в тестовом сценарии; данные переменные могут принимать значения $i_1, \dots, i_M$, соответствующие всем событиям (парам <элемент; графического интерфейса, действие над графическим интерфейсом>) в модели

it_i – фрагмент трассы, порождаемый функцией обработки события i_i;

i_X – событие, точка ветвления в подтрассе которого обрабатывается модулем генерации тестовых сценариев;

i_{i1}, …, i_{iN} – возможный путь в графе потока событий;

N – максимальная длина тестового набора;

M – количество событий в модели графического интерфейса;

K – количество возможных путей в графе потока событий.

Инструмент проверки выполнимости булевых формул осуществляет обработку данной трассы. В случае её выполнимости инструмент предоставляет набор значений e_1, ..., e_N, который:

- позволяет обойти нужный путь выполнения по обрабатываемой точке ветвления;
- задает корректный тестовый сценарий согласно модели графического интерфейса.

Тем самым в рамках совмещенного режима осуществляется построение тестовых сценариев, направленных на исследование точек ветвления внутри обработчиков событий, что позволяет обнаруживать не только дефекты проектирования самого интерфейса, но и раскрывать ошибки во внутренних алгоритмах обработки данных.

3.3 Подход к генерации опасных входных значений

Ряд уязвимостей в программном обеспечении связан с отсутствием проверок на разрешенные значения обрабатываемых данных. Так, достаточно распространенной ошибкой является считывание данных из полей ввода (EditText) и их дальнейшая обработка без предварительной проверки на соответствие определенному формату. Например, рассмотрим следующую конструкцию:

```
float f =
Float.parseFloat(editText.getText().toString());
```

Если введенные в текстовое поле данные содержат недопустимые символы, то средствами виртуальной машины будет создано исключение `java.lang.NumberFormatException`. Анализ программы с помощью символьного исполнения построенного на инструментации кода Java-программы не сможет обнаружить данную уязвимость, если точки ветвления, отвечающие за проверку формата входных данных, скрыты в реализации виртуальной машины. К тому же, инструмент не имеет возможности генерировать данные для ввода в текстовые поля напрямую, но в случае, когда содержимое таких полей формируется посредством взаимодействий с другими элементами графической структуры (нажатий на кнопки и т. д.), становится возможным создание механизма для обнаружения такого рода ошибок. Для решения данной проблемы добавляется искусственный условный переход перед соответствующей уязвимой операцией. Условие в таком переходе

должно соответствовать проверке опасных данных на корректность, тогда во время инвертирования этого перехода будет получена последовательность событий, формирующая данные, приводящие к ошибке. Подход может быть реализован с помощью внедрения перед уязвимой инструкцией метода-симулятора, генерирующего трассу условного перехода.

В рамках реализации совмещенного метода в инструмент Coffee Machine был добавлен ряд симуляторов, направленных на обнаружение подобных ошибок, включающий проверку функций трансформации и обработки примитивных типов языка Java.

4. Оценка предложенного решения

Для исследования эффективности разработанного метода анализа было проведено тестирование инструмента на наборе реальных проектов. Для полноценной проверки качества анализа приложения выбирались в соответствии со следующими критериями:

- большое количество элементов управления;
- наличие скрытой за интерфейсом вычислительной логики приложения.

Данные требования необходимы для создания нагрузки на решатель булевых ограничений, увеличении количества помеченных данных и числа итераций. Исходя из этих критериев были проанализированы следующие проекты:

- Calculator – примитивный калькулятор для платформы Android;
- ProgCalc – калькулятор для платформы Android, позволяющий проводить вычисления в различных системах счисления;
- Calculator – калькулятор для платформы Swing.

В качестве контрольного испытания был проведен анализ предложенных программ инструментом GUITAR без модификаций, результаты которого приведены в таблице 1. Анализ заключается в построении модели графического интерфейса тестируемого приложения, генерации на основе этой модели тестового покрытия и его последующего воспроизведения. Тестовое покрытие представляет собой набор отдельных тестов – фиксированное количество взаимодействий с приложением. Тесты представляют собой все возможные комбинации элементов интерфейса, поэтому размер тестового покрытия зависит от величины модели и от количества взаимодействий в каждом тесте. Для проведения анализа было сгенерировано тестовое покрытие, состоящее из двух взаимодействий, как оптимальное с точки зрения времени воспроизведения и достаточности для обнаружения дефектов. Из-за величины тестового покрытия такой анализ проводится достаточно долго, причем большее время анализа Android-приложений обусловлено задержками при взаимодействии фреймворка GUITAR и Android-устройства (эмулятора). В результате проведенного тестирования были обнаружены программные дефекты в приложениях и достаточно большое количество входных данных для

их воспроизведения. В приложении Calculator (Android) было найдено 3 дефекта, связанных с необработанной исключительной ситуацией типа `java.lang.NumberFormatException`, и 17 различных последовательностей взаимодействий с пользовательским интерфейсом для их воспроизведения. В приложении ProgCalc (Android) было обнаружено 2 уязвимости: ошибка деления на ноль (`java.lang.ArithmeticException`) и переполнения окна ввода, приводящего к необработанной исключительной ситуации неверного формата числа (`java.lang.NumberFormatException`), для каждой уязвимости по одной последовательности событий. В приложении Calculator (Swing) была найдена 1 уязвимость, связанная с обработкой недопустимых символов (`java.lang.NumberFormatException`), и 5 различных последовательностей событий, позволяющих воспроизвести дефект.

Табл. 1. Результаты инструмента GUITAR

Table 1. GUITAR results

Проект		Calculator (Android)	PrgCalc (Android)	Calculator (Swing)
Найдено дефектов		3	2	1
Покрытие		587	1507	608
Время (ч)		4	12,5	0,66
Размер модели	Окон	1	1	2
	Элементов	42	51	47
	Событий	2	30	30

Результаты применения предложенного метода представлены в «Табл. 2». Динамический анализ на основе символьного исполнения позволил существенно сократить время тестирования, однако в результате было найдено меньшее число уязвимостей, чем при воспроизведении тестового покрытия. Это связанно со слишком сильным ограничением длины цепочек событий графического пользовательского интерфейса, которое принудительно было выставлено в значение 2. Так, в приложении Calculator (Android) было обнаружено только 2 уязвимости и 2 набора уязвимых входных данных для их воспроизведения. В приложении ProgCalc (Android) был найден один дефект, связанный с делением на ноль, а в приложении Calculator (Swing) не было найдено ни одного дефекта. Прежде всего, это обусловлено тем, что некоторые уязвимости не находятся на отдельном пути исполнения программы, и воспроизведение только одного набора данных, позволяющего достичь такого пути, недостаточно. Часть уязвимостей была обнаружена с помощью

механизма построения условий, выполнимость которых означает возникновение ошибки на данном пути, однако ограничения этого метода не позволили найти остальные уязвимости. Но в связи с ограничением на длину цепочки событий графического пользовательского интерфейса в два события у инструмента не было возможности ввести слишком длинное число, чтобы спровоцировать исключительную ситуацию неправильного формата числа. Предложенная схема работы инструмента производит обновление модели на каждой итерации, однако в данном случае модели остались неизменными в силу того, что рассматриваемые приложения либо не имеют дополнительных элементов, либо они не становятся активными при исследовании путей исполнения.

Табл. 2. Результаты символьного исполнения

Table 2. Symbolic execution results

Проект		Calculator (Android)	PrgCalc (Android)	Calculator (Swing)
Найдено дефектов		2	1	0
Количество итераций		35	17	21
Время (ч)		0,42	0,33	0,06
Размер модели	Окон	1	1	2
	Элементов	42	51	47
	Событий	2	30	30

5. Заключение

В рамках работы, представленной в статье, разработан совмещённый метод динамического анализа программ, предоставляющих графический пользовательский интерфейс. Данный метод включает в себя автоматическое извлечение модели графического пользовательского интерфейса, описывающей возможные последовательности взаимодействия с элементами интерфейса (тестовые сценарии), и направленное построение тестовых сценариев, позволяющих обойти возможные пути в программе.

Для реализации метода были использованы свободно распространяемые инструменты GUITAR и Coffee Machine; инструмент GUITAR был расширен генератором тестовых сценариев, который вычисляет тестовый сценарий используя символьное исполнение.

Данный метод позволяет целенаправленно добиваться увеличения полноты покрытия кода при анализе программ и обхода точек ветвления в коде

обработчиков событий взаимодействия с элементами графического пользовательского интерфейса.

С помощью полученного инструмента был проанализирован набор свободно распространяемых приложений на языке Java (библиотека управления графическим интерфейсом в составе ОС Android; библиотека управления графическим интерфейсом Swing). С помощью направленного обхода внутренних точек ветвления в обработчиках событий были обнаружены дефекты, связанные с некорректной обработкой исключительных ситуаций.

В то же время, при проведении практических экспериментов были замечены ограничения представленного метода. В первую очередь, конструирование трасс с перебором действий над элементами интерфейса приводит к ещё большему усугублению проблемы экспоненциального роста количества путей для анализа, присущей методам анализа на основе символьного исполнения. В статьях, посвящённых инструменту GUITAR, указывались практические ограничения на перебор сценариев взаимодействия, включающие фиксацию максимальной длины создаваемых сценариев. При работе в контексте символьного исполнения данное ограничение приходится усиливать.

Во-вторых, усложнение структур трасс привело к увеличению затрат на проверку выполнимости данных трасс; в общем случае данное увеличение не ограничивается линейными зависимостями.

На основе указанных недостатков можно определить направления дальнейших исследований в области анализа программ с графическим пользовательским интерфейсом. Технические улучшения работы разработанного инструмента, направленные на отслеживание особенностей графических элементов и их свойств, позволят составлять более точную модель графического пользовательского интерфейса, что в свою очередь снизит количество сценариев выполнения программы, требующих анализа. Второе направлений исследований связано с оптимизациями трасс ограничений и использования возможностей современных решателей булевых ограничений (инкрементальное решение, выявление групп зависимых условий и пр.).

Список литературы

[1]. Сайт Mobile Banking Rank 2015 (http://markswebb.ru/e-finance/internet-banking-rank-2015/)
[2]. Страница проекта Ranorex. http://www.ranorex.com/ [HTML]. Обращение от 10.10.2016
[3]. Abbot framework for automated testing of Java GUI components and programs. https://abbot.sourceforge.net. Обращение от 10.10.2016
[4]. Страница проекта Maveryx. https://sourceforge.net/projects/maveryx/ [HTML]. Обращение от 10.10.2016
[5]. Страница проекта Squish. https://www.froglogic.com/squish/ [HTML]. Обращение от 10.10.2016

[6]. Sikuli: using GUI screenshots for search automation. UIST'09 Proceedings of the 22nd annual ACM symposium on User Interface software and technology. Victoria, BC, Canada, October 04-07, 2009. pp. 183-192
[7]. Bao N. Nguyen, Bryan Robbins, Ishan Banerjee, Atif Memon. GUITAR: an innovative tool for automated testing of GUI-driven software. Automated software engineering, 2013, vol. 21(1), pp. 65-105.
[8]. Willem Visser, Corina S. Pāsāreanu, Sarfranz Khurshid. Test input generation with java PathFinder. ISSTA '04 Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis. Boston, Massachusetts, USA, July 11-14, 2004, pp. 97-107.
[9]. Gary Lindstrom, Peter C. Mahlitz, Willem Visser. Model checking real time java using java pathfinder. Proceeding ATVA'05 Proceedings of the Third international conference on Automated Technology for Verification and Analysis, Taipei, Taiwan, October 04-07, 2005, pp. 444-456.
[10]. Howar, Falk, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamaric, and Vishwanath Raman. "JDART: A Dynamic Symbolic Analysis Framework." In Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings, vol. 9636, Springer, 2016. p. 442.
[11]. Farzan A, Chen F, Meseguer J, Roşu G. Formal analysis of Java programs in JavaFAN. International Conference on Computer Aided Verification 2004 Jul 13. Springer Berlin Heidelberg. pp. 501-505
[12]. М.К. Ермаков, С.П. Вартанов. Применение статической инструментации байт-кода языка Java для динамического анализа программ. Труды Института системного программирования РАН, том 27, вып. 1, 2015 г., стр. 25-38. DOI: 10.15514/ISPRAS-2015-27(1)-2

Dynamic analysis of programs with graphical user interface based on symbolic execution

S.P. Vartanov <svartanov@ispras.ru>
A.Y. Gerasimov <agerasimov@ispras.ru>
M.K. Ermakov <mermakov@ispras.ru>
D.O. Kutz <kutz@ispras.ru>
A.A. Novikov <a.novikov@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. In this paper, we explore the possibilities of applying dynamic symbolic execution (or concolic testing) methods to applications with graphical user interfaces. Such applications inherently feature interactive user input processing and multithreaded execution. These features typically decrease the effectiveness of dynamic symbolic execution by increasing the volume of processed code not related to actual application functionality. We present a hybrid approach that combines commonly used GUI test automation methods based on GUI model excavation

with dynamic symbolic execution methods to construct test cases for checking internal application logic. We have implemented this approach using two open-source tools – test automation framework GUITAR and Java byte-code static instrumentation framework Coffee Machine. GUI model extracted automatically by GUITAR tool is extended with symbolic traces relevant to application GUI event handlers. Our test generation module for GUITAR combines these symbolic traces into complex queries to be processed by SMT solver. The resulting test cases are valid within automatically extracted GUI structure model and allow to check different execution paths in GUI event handler code. We have checked our hybrid approach on a set of small open-source applications and identified several bugs caused by uncaught exceptions. The paper is concluded with an overview of current limitations and possible improvements of the hybrid approach.

Keywords: dynamic analysis; program analysis; GUI testing; test coverage.

DOI: 10.15514/ISPRAS-2017-29(1)-10

For citation: Vartanov S.P., Gerasimov A.Y., Ermakov M.K., Kutz D. O., Novikov A.A. Dynamic analysis of programs with graphical user interface. *Trudy ISP RAN/Proc. ISP RAS*, vol. 29, issue 1, 2017. pp. 149-166 (in Russian). DOI: 10.15514/ISPRAS-2017-29(1)-10

References

[1]. Mobile Banking Rank 2015 (http://markswebb.ru/e-finance/internet-banking-rank-2015/)
[2]. Ranorex. http://www.ranorex.com/ [HTML]. Accessed at 10.10.2016
[3]. Abbot framework for automated testing of Java GUI components and programs. https://abbot.sourceforge.net. Accessed at 10.10.2016
[4]. Maveryx. https://sourceforge.net/projects/maveryx/ [HTML]. Обращение от 10.10.2016
[5]. Squish: https://www.froglogic.com/squish/ [HTML]. Обращение от 10.10.2016
[6]. Sikuli: using GUI screenshots for search automation. UIST'09 Proceedings of the 22nd annual ACM symposium on User Interface software and technology. Victoria, BC, Canada. Octomer 04-07, 2009. pp. 183-192
[7]. Bao N. Nguyen, Bryan Robbins, Ishan Banerjee, Atif Memon. GUITAR: an innovative tool for automated testing of GUI-driven software. Automated software engineering. - 2013. - Vol. 21(1). pp. 65-105.
[8]. Willem Visser, Corina S. Pāsāreanu, Sarfranz Khurshid. Test input generation with java PathFinder. ISSTA '04 Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis. Boston, Massachusetts, USA. July 11-14, 2004. pp. 97-107.
[9]. Gary Lindstrom, Peter C. Mahlitz, Willem Visser. Model checking real time java using java pathfinder. Proceeding ATVA'05 Proceedings of the Third international conference on Automated Technology for Verification and Analysis. Taipei, Taiwan. October 04-07, 2005. pp. 444-456.
[10]. Howar, Falk, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamaric, and Vishwanath Raman. "JDART: A Dynamic Symbolic Analysis Framework." In Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software,

ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings, vol. 9636, Springer, 2016. p. 442.
[11]. Farzan A, Chen F, Meseguer J, Roşu G. Formal analysis of Java programs in JavaFAN. InInternational Conference on Computer Aided Verification 2004 Jul 13. Springer Berlin Heidelberg. pp. 501-505
[12]. M.K. Ermakov, S.P. Vartanov. Applying Java bytecode static instrumentation for software dynamic analysis. Trudy ISP RAN / Proc. ISP RAS, vol 27, issue 1, 2015, pp. 25-38 (in Russian). DOI: 10.15514/ISPRAS-2015-27(1)-2.