УДК 004.054

ПРОБЛЕМА ЦЕНЫ ПРОВЕРКИ ТЕСТОВ ЧЕЛОВЕКОМ В КОНТЕКСТЕ СИМВОЛЬНЫХ ВЫЧИСЛЕНИЙ (ОБЗОР ИНОСТРАННОЙ ЛИТЕРАТУРЫ)

Якимов И.А., Кузнецов А.С.

ФГАОУ ВПО «Сибирский федеральный университет», Красноярск,
e-mail: ivan.yakimov.research@yandex.ru, ASKuznetsov@sfu-kras.ru

Тестирование является важной составляющей жизненного цикла разработки ПО. Исследования в данной области сосредоточены на поиске новых способов улучшения данного процесса. В течение последнего десятилетия в научном сообществе наблюдается усиленный интерес к динамическим символьным вычислениям. Символьные вычисления представляют собой технику автоматизированной генерации тестов по исходному коду, обеспечивающей высокое покрытие кода. В данной области получены значительные результаты в применении к тестированию реальных программ, таких как открытое ПО и индустриальные системы. Однако исследователи уделяют недостаточно внимания проблеме проверки тестов вручную. Имеющиеся инструменты в основном предназначены для целенаправленного фаззинга. Данный обзор сосредоточен на поиске методов снижения цены проверки тестов человеком в контексте динамических символьных вычислений. Его целью является выработка стратегии по созданию новых инструментов, ориентированных на проверку тестов человеком.

Ключевые слова: генерация тестовых случаев, динамические символьные вычисления

HUMAN ORACLE COST PROBLEM IN CONTEXT OF SYMBOLIC EXECUTION (A SURVEY)

Yakimov I.A., Kuznetsov A.S.

Siberian Federal University, Krasnoyarsk,
e-mail: ivan.yakimov.research@yandex.ru; ASKuznetsov@sfu-kras.ru

Software testing plays an important role in the software development life cycle. Works in this field are focused on finding new ways to improve this process. Particularly, the research community demonstrates increasing interest on dynamic symbolic execution during the recent decade. Dynamic symbolic execution is a method for automatic code-based generation of tests with high code coverage. This method shows reliable results in different applications including testing of real open source and industry systems. However, researchers do not pay much attention to the problem of the manual checking of such tests. Most tools are implemented for directed automated fuzzing. This survey is focused on finding ways of reducing human oracle cost in context of dynamic symbolic execution. The goal is to propose a strategy for creating new tools oriented on solving of the human oracle cost problem.

Keywords: test suites generation, dynamic symbolic execution

Исследования в области тестирования сосредоточены на поиске путей увеличения полноты покрытия кода тестами и снижения затрат на его проведение. Задачу тестирования можно условно разделить на этапы создания тестов, их запуска и проверки полученных результатов. Автоматизация каждого из данных шагов может оказать существенное положительное влияние на улучшение процесса тестирования в целом.

Цена тестирования

Для автоматизации этапа создания тестов разработано множество средств. В данной статье внимание сосредоточено на генераторах тестов по исходному коду, работа которых основана на динамических символьных вычислениях [3] – DSE (dynamic symbolic execution). DSE позволяют систематически генерировать тесты, обеспечивающие желаемое покрытие код. Во время DSE исследуются тысячи и десятки тысяч отдельных путей и генерируя соответствующее число тестовых случаев. Основной целью существующих генераторов на основе DSE является поиск стандартных ошибок (деление на ноль, разыменовывание нулевого указателя, нарушение условия утверждения, утечки памяти [11]). Также данный метод нашел приложение к динамическому анализу (анализ производительности [13], обнаружение состояний гонок [14]). Далее будет использоваться понятие цены генерации тестов, которая возрастает с увеличением затрат на генерацию тестов при помощи DSE.

Другим важным этапом тестирования является проверка результатов тестирования, при котором задействован тестовый оракул [1]. Тестовый оракул – это субъект (отдельная процедура, программа или человек), который для каждого тестового сценария определяет соответствие поведения программы ожидаемому.

В случае, когда в качестве тестового оракула выступает человек, можно говорить о цене проверки тестов человеком (human oracle cost). Цена проверки тестов челове-

ком тем выше, чем больше усилий нужно затратить человеку для проверки соответствия реального поведения программы ожидаемому. При наличии имеющегося тестового набора и соответствующих результатов выполнения тестов можно рассмотреть количественный и качественный аспекты данной цены [1]:

Количественное снижение достигается за счет сокращения размера отдельных тестовых случаев и целых тестовых наборов при сохранении требуемого тестового покрытия, которое может быть сформулировано в терминах систематического покрытия инструкций *или* иных формальных требованиях.

Качественное снижение цены предполагает уменьшение усилий, необходимых человеку для понимания и проверки каждого *отдельного* тестового случая. Автоматизированные генераторы имеют тенденцию порождать тестовые наборы, выходящие за рамки реалистичных сценариев использования приложения. Росту сложности сценариев сопутствует увеличение времени проверки тестового набора вовлеченным в данный процесс инженером.

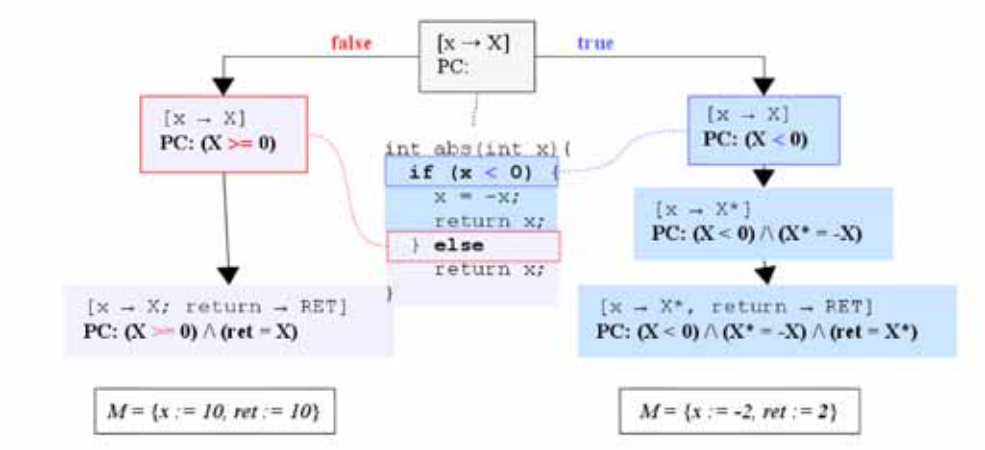
При объединении процессов автоматизации создания, запуска и проверки тестов может быть получена (полностью или частично) автоматизированная система тестирования программного обеспечения. *Затраты на тестирование программ человеком* в данном случае складываются из затрат на *создание* и *проверку* тестов. Целью данной работы является обзор методов автоматизированной генерации тестов с точки зрения снижения *затрат на тестирование программы человеком* в контексте использования генераторов на основе *динамических символьных вычислений*. Полученные выводы могут быть использованы в последующих исследованиях по автоматизации процессов создания и проверки тестов по исходному коду.

Символьные вычисления

Далее в этом параграфе будет в общем виде описан процесс символьных вычислений в чистом виде, предложенном более четырех десятилетий назад. Во время символьных вычислений (SE) запуск программы вместо конкретных и фиксированных, осуществляется на неограниченных, символьных входах [2]. Данная трансформация процесса вычислений предполагает поддержку символьного состояния программы S (symbolic state). Данное состояние S помимо стека, кучи и регистрового файла содержит ограничение пути PC (path constraint). PC является набором ограничений, описывающих класс эквивалентности входных данных, направляющих выполнение программы по текущему пути [3]. Изменение семантики выполнения программы предполагает модификацию семантики отдельных инструкций, выполнение которых приводит к созданию символьных выражений, и в отдельных случаях добавлению новых ограничений к PC. В каждой точке принятия решения (например операторе if-then-else) символьные вычисления концептуально разветвляются и продолжают выполняться независимо. В случае, когда новый путь является недостижимым выполнение прекращается, иначе – PC обновляется так, чтобы входящий в него набор ограничений соответствовал текущему пути. Решение PC, получаемое при достижении конца программы, является искомым вектором входных тестовых данных и может быть использовано для генерации тестов с их последующим запуском.

На рисунке ниже изображен пример выполнения символьных в одной из возможных реализаций данного метода генерации тестов. В начале выполнения программы переменная (соответствующий ей адрес памяти) x ссылается на символьную переменную X (имена символьных переменных и переменных программы никогда не совпадают), PC не содержит ограничений. По достижении конструкции if ($x < 0$) происходит развилка. На ветке false к PC добавляется ограничение $X>=0$, после чего выполняется инструкция return x и на специальную символьную переменную ret накладывается ограничение ret=X, а ссылка на эту переменную сохраняется в регистре RET. По мере выполнения ветви true на первом шаге добавляется ограничение $X<0$, затем после выполнения return –x выделяется свежая переменная X*, на которую накладывается ограничение $X^* = -X$ (теперь x ссылается на X*), и добавляется ограничение ret = X*, после чего устанавливается ссылка RET: ret. Генерация тестов заключается в поиске решений для PC, которые для данного примера могут иметь вид $\{x = -3\}$ для ветки true, и $\{x = 42\}$ для ветки false соответственно.

Символьные вычисления предлагают достаточно простой в реализации и одновременно мощный инструмент автоматизированной генерации тестов по исходному коду приложения, однако с данным методом связан ряд трудностей [15, 16]. Далее будут перечислены проблемы связанные с символьными вычислениями, выделенные в данном обзоре в контексте затрат на тестирование программ человеком.



Процесс символьных вычислений

1. Проблема взаимодействия с окружением – зависимость от окружения предполагает описание логики работы внешних систем, с которыми взаимодействует целевая программа. Любая нетривиальная программа взаимодействует с окружением (может ожидать пользовательский ввод, сетевой пакет и так далее). Таким образом, применение символьных вычислений на практике требует дополнительных усилий по работе с внешним окружением, что приводит к возрастанию цены создания тестов.

2. Выход ограничений за рамки используемой теории – символьные ограничения, накапливаемые по мере прохождения инструкций программы должны быть выражены на языке определенной теории T, с которой работает внешний решатель. Таким образом, возможность поиска решений для ограничений с последующей генерацией тестов или определением достижимости рассматриваемых путей ограничена полнотой T. Наибольшие трудности возникают при работе с указателями и числами с плавающей точкой. Данная проблема также будет условно отнесена к цене создания тестов.

3. Длительное время ожидания – символьные вычисления являются затратной по ресурсам техникой и требуют больших вычислительных мощностей. Генерация тестов программ промышленного уровня может занимать минуты, часы и даже сутки непрерывной работы. Данное обстоятельство может воспрепятствовать использованию символьных вычислений в контексте методологии гибкой разработки, предполагающей непрерывное внесение изменений в исходный код программы. Данная проблема будет отнесена к цене создания тестов с точки зрения затрат вычислительных ресурсов и времени ожидания.

4. Стремительный рост размера тестового набора – источником данной проблемы является комбинаторный взрыв числа путей выполнения в случае классических символьных вычислений. В общем случае, число возможных путей выполнения программы практически неограниченно. Помимо роста нагрузки на вычислительные ресурсы и увеличения времени ожидания, рассмотрение всех путей выполнения увеличивает число тестовых случаев в наборе. В данном случае цена проверки тестов возрастает количественно.

5. Выход тестового набора за пределы предметной области – значения входных данных, автоматически сгенерированные DSE-инструментами, как правило, лежат в более широком диапазоне значений, нежели значения данных, характерные для стандартных случаев работы приложения. Иными словами, генераторы имеют тенденцию порождать сложные сценарии тестирования. Человеку легче проверять характерные, поддающиеся более простому и скорому осмыслению сценарии. В данном случае можно говорить о качественной цене проверки тестов.

Существующие DSE-генераторы можно разделить на две группы – консольные генераторы и генераторы, порождающие тесты во время выполнения. В каждом из подходов используется одна основная идея – совмещение конкретных и символьных вычислений. Таким образом, рассматриваемые далее методы с некоторыми изменениями могут быть использованы в любой DSE-системе.

Консольное тестирование

Направленное случайное тестирование

При направленном случайном тестировании символьные вычисления и гене-

рация случайных тестов скомбинированы, образуя единую систему. Впервые данный метод был реализован в системе DART [3]. Программа многократно перезапускается (тысячи и десятки тысяч раз) на случайных входных данных. Символьные вычисления служат для поиска альтернативных путей выполнения, при этом рассматриваются только скалярные значения, входящие в изначальный входной вектор. В случае попытки обращения к указателю, находящемуся вне начального вектора данных, а также в случае провала решателем поиска решения для ограничения пути, программа перезапускается на новых случайных значениях. Вызовы внешних функций также заменяются генерацией случайных значений, таким образом отбрасывается необходимость моделирования окружения. Однако данная стратегия применима к ограниченному классу программ – тестируемая система должна быть готова принять любое хорошо типизированное значение как при запуске, так и при вызове внешних функций. При тестировании oSIP не удалось установить, действительно ли некоторые тесты выявляют ошибки, или же приводят к ложным срабатываниям.

Символьные указатели и предусловия

Идеи целенаправленного случайного поиска развиваются в системе CUTE [4]. Основным отличием является поддержка символьных ограничений вида «p=NULL, p≠NULL, p=q, p≠q» над указателями. Генерация входных данных осуществляется более систематическим путем, через решение накладываемых на указатели ограничения (каждый указатель изначально инициализируется значением NULL). Исходя из полученных ограничений, CUTE может не только менять значения скалярных переменных, но также добавлять и удалять объекты, адресуемые указателями, и отслеживать использование псевдонимов. для фильтрации некорректных данных, подаваемых на вход целевой функции, используются задаваемые пользователем предусловия. Использование контрактов снижает вероятность ложных срабатываний, так как сужает диапазон входных данных.

Ленивые символьные вычисления

В Latest [6] используется подход постепенной конкретизации абстракции целевой функции, позаимствованный из методологии проверки моделей. Работа системы проходит в два этапа: на первом этапе система при помощи DSE исследует целевую функцию получая при этом абстракции различных (возможно недостижимых) путей ее выполнения, подменяя результаты вызовов функций неограниченными входами. Во время второго прохода происходит рекурсивное раскрытие данных вызовов и поиск (если это возможно) конкретных значений, проводящих программу по рассматриваемому пути. Экспериментальные исследования показали, что в определенных условия данный метод работает значительно лучше DSE в чистом виде. Во-первых, затраты времени могут быть снижены на порядок. Во-вторых, сгенерированные входные данные имеют тенденцию быть более осмысленными. Так, для транслятора PL/0 было получено несколько сотен синтаксически корректных программ, тогда как CUTE не удалось сгенерировать ни одной.

Генерация контрактов функций

Система SMART [5] представляет альтернативный способ уменьшения числа путей, рассматриваемых при DSE. Концептуально, отдельные функции сначала тестируются изолированно, при этом результаты тестирования преобразуются в контракты, выраженные через пред и пост-условия над входами и выходами. Далее, вызовы обработанных функций подменяются использованием их контрактов. В проведенном авторами исследования для парсера oSIP удалось получить линейную зависимость количества перезапусков программы от размера подаваемых на вход пакетов, что отличает ее от DART, в которой наблюдался экспоненциальный рост. Однако у данного подхода возникают трудности при работе с рекурсивными вызовами. Также, для достаточно сложных функций контракты могут быть слишком сложными для внешнего решателя.

Генерация контрактов циклов

Идея генерации контрактов функций нашла продолжение в концентрации усилий вокруг циклов. Так, был предложен метод динамической генерации частичных контрактов циклов [7], в которых символьные ограничения, входящие в условия, линейно зависит от переменных индукции и аргументов функции. Анализ проводится динамически, при этом используется сопоставление с образцом для выявления условий, переменных индукции и зависимостей между ними. В проведенном эксперименте около 2,5% циклов поддалось такой обработке. При этом общее количество ограничений за все время вычислений в целом осталось практически без изменений, и затраты по времени напротив, увеличились.

Развитием идеи генерации контрактов для циклов стал метод обработки циклов,

осуществляющих обход Си-строк [8]. Данный метод работает для циклов со многими путями выполнения, которые производят обход Си-строк. Под рассмотрение попадают циклы, где в при каждой проверке условий сравниваются свойства строк, такие как их содержимое или длина. для выявления шаблонов также используется сопоставление с образцом.

Отбор приоритетных путей выполнения

Проект CREST [9] посвящен изучению стратегий выбора путей для исследования при динамических символьных вычислениях. Можно выделить три основные стратегии – обход в глубину, случайный выбор исследуемых путей, а также использование расстояния между инструкциями, выраженного в терминах графа потока. для сравнения данных стратегий были протестированы программы replace, grep и vim. В целях обеспечения чистоты эксперимента также использовалось случайное тестирование. для ряда случаев экспериментальные результаты показали проигрыш DSE с поиском в глубину по сравнению со случайным тестированием. Наиболее эффективными оказались стратегии на основе учета расстояния между инструкциями в графе потока, а также стратегии на основе случайного выбора исследуемых путей.

Другим подходом к направлению символьных вычислений является использование представленных пользователем аннотаций. Ключевой идеей является то, что разработчику известно как устроена программа, и на каких участках кода следует сосредоточить усилия по поиску ошибок. Примером использования данного подхода является система GuideSE [10], в которой пользователь может задать аннотации для направления DSE по нужным путям.

Тесты, порождаемые во время выполнения

Во всех рассмотренных ранее системах используется тактика итеративного исследования различных путей выполнения программы, при которой на каждом запуске исследуется ровно один путь выполнения, и параллельно поддерживается одно конкретное и несколько символьных состояний. Альтернативный подход к комбинации символьных и конкретных вычислений впервые был представлен в системе EXE [11]. Программа не перезапускается многократно, а буквально разветвляется (обращаясь к системному вызову fork) в каждой точке принятия решений, всякий раз, когда в условие входит символьное выражение. Перед запуском тестируемой программы пользователь помечает отдельные участки памяти как символьные или конкретные. Операция, включающая в качестве параметров лишь конкретные значения выполняется конкретно, но если хоть одно значение – символьное, конкретные значения переводятся в символьные и вычисления происходят символьно. В EXE реализована теория указателей с использованиям символьных массивов, что позволяет описывать все операции над указателями в виде символьных выражений и отказаться от генерации случайных выходных данных там, где в PC должны входить ограничения над указателями. для эмуляции взаимодействия с окружением не применяется тактика генерации случайных значений – в EXE смоделированы системные вызовы Linux, что позволяет тестировать программы, поддерживая относительно небольшое количество моделей внешних систем.

Логическим продолжением системы EXE является ее прямой наследник, KLEE [12] – гибрид операционной системы для символьных вычислений и символьного интерпретатора. Так же как в EXE, взаимодействие с окружением сводится к моделированию системных вызовов ядра операционной системы Linux. Стоит отметить, что для эмуляции процесса работы с файловой системой могут быть использованы как символьные, так и конкретные файлы. Однако при использовании конкретных файлов возможны наложения записей различных символьных процессов. Также важным нововведением KLEE является компактное представление состояний символьных процессов, для реализации которого задействована стратегия copy-on-write на уровне отдельных объектов.

Дальнейшим развитием подхода KLEE является распространение символьной операционной системы до размеров целой виртуальной машины. Система s2e [13] реализована на базе виртуальной машины QEMU, а также KLEE, которая в модифицированном виде задействована для выполнения символьных вычислений. Вычисления производятся на уровне машинного кода, система позволяет тестировать программы как в пространстве пользователя, так и в пространстве ядра. Переключение между символьными и конкретными вычислениями делает процесс расширяемым и позволяет тестировать целые программные стеки, такие как стек Microsoft Windows, при этом операции с файлами между отдельными символьными путями не пересекаются между собой.

Заключение

В работах по динамическим символьным вычислениям пройден большой путь

по развитию символьных вычислений. На основе обзора работ по данной тематике можно сделать определенные выводы. для решения проблемы снижения количественной цены проверки тестов человеком полезно применить стратегии выбора путей на основе эвристик или пользовательских аннотаций, так как по сравнению с обходом в глубину они дают большее покрытие за меньшее время. для решения проблемы качественного снижения цены проверки тестов человеком полезно применить ленивые символьные вычисления, так как эксперименты показывают, что они склонны генерировать тесты покрывающие логику работы, а не крайние случаи, приводящие к краху целевой программы. Отдельным вопросом является взаимодействие с окружением – стратегия зависит от архитектуры системы (консольной, либо порождающей тесты во время выполнения). Можно сделать вывод о том, что комбинация и подстройка методов, освещенных в данной работе может продвинуть исследования по качественном и количественному снижению цены проверки автоматически сгенерированных тестов человеком.

Список литературы

1. Barr E.T., Harman M., McMinn P., Shahbaz M., Yoo S. The Oracle Problem in Software Testing: A Survey. // IEEE Transactions on Software Engineering, 2015; 41(5): 507–525.

2. King J.C. Symbolic execution and program testing // Communications of the ACM, 1976; 19(7): 385–394.

3. Godefroid P., Klarlund N., Sen K. DART: directed automated random testing. // Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, 2005; 40(60): 213–223.

4. Sen K/, Marinov D., Agha G. CUTE: a concolic unit testing engine for C // Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, 2005; 30(5): 263–272.

5. Godefroid P. Compositional dynamic test generation // Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 2007; 42(1): 47–54.

6. Majumdar R., Sen K. LATEST: Lazy Dynamic Test Input Generation // ECS Department. University of California, Berkeley. Technical Report No. UCB/EECS-2007–36, March 20, 2007.

7. Godefroid P. Automatic partial loop summarization in dynamic test generation // Proceedings of the 2011 International Symposium on Software Testing and Analysis, 2011; 23–33.

8. Xie X., Liu Y., Le W., Li X., Chen H. S-looper: automatic summarization for multipath string loops. // Proceedings of the 2015 International Symposium on Software Testing and Analysis, 2015; 188–198.

9. Burnim J., Sen K. Heuristics for Scalable Dynamic Test Generation // Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, 2008; 443–446.

10. Sen K., Tanno H., Zhang X., Hoshino T. GuideSE: annotations for guiding concolic testing // Proceedings of the 10th International Workshop on Automation of Software Test, 2015; 23–27.

11. Cadar C., Ganesh V., Pawlowski P.M., Dill D.L., Engler D.R. EXE: automatically generating inputs of death // Proceedings of the 13th ACM conference on Computer and communications security, 2006; 322–335.

12. Cadar C., Dunbar D., Engler D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs // Proceedings of the 8th USENIX conference on Operating systems design and implementation, 2008; 209–224.

13. Chipounov V., Kuznetsov V., Candea G. S2E: a platform for in-vivo multi-path analysis of software systems // Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems, 2013; 265–278

14. Sen K., Agha G. Automated systematic testing of open distributed programs. // Proceedings of the 9th international conference on Fundamental Approaches to Software Engineering, 2006; 339–356.

15. Cadar C., Sen K. Symbolic execution for software testing: three decades later // Communications of the ACM, 2013; 56(2): 82–90.

16. Anand S., Burke E.K., Chen T.Y., Clark J., Cohen M.B., Grieskamp W., Harman M., Harrold M.J., Mcminn P. // An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software archive, 2013; 86(80): 1978–2001.