Оптимизация читаемости тестов порождаемых при символьных вычислениях

И.А. Якимов <ivan.yakimov.research@yandex.ru>
А.С. Кузнецов <ASKuznetsov@sfu-kras.ru>
Институт Космических и Информационных Технологий, Сибирский Федеральный Университет,
660074, Россия, г. Красноярск, ул. Академика Киренского, д. 26

Аннотация. Занимая около половины времени разработки, тестирование остается наиболее распространенным методом контроля качества ПО, и его недостаток может приводить к финансовым потерям. При систематическом подходе тестовый набор считается полным, если он обеспечивает определенное покрытие кода. На данный момент существует большое количество систематических генераторов тестов, направленных на поиск стандартных ошибок. Подобные инструменты порождают огромное количество трудночитаемых тестов, обладающих высокой ценой проверки человеком. Представленный в данной работе метод позволяет улучшить читаемость тестов, автоматически сгенерированных при помощи символьных вычислений, обеспечивая качественное снижение данной цены. Экспериментальные исследования генератора тестов, включающего данный метод в качестве заключительной фазы работы, были проведены на 12-ти строковых функциях из репозитория Linux. Оценка степени читаемости строк, содержащихся в оптимизированных тестах, сопоставима со случаем использования слов натурального языка, что положительно сказывается на процессе верификации результатов тестирования человеком.

Ключевые слова: автоматическая генерация тестов; символьные вычисления; цена проверки тестов человеком; биграммная модель языка.

DOI: 10.15514/ISPRAS-2016-28(5)-14

Для цитирования: И.А. Якимов, А.С. Кузнецов. Оптимизация читаемости тестов порождаемых при символьных вычислениях. Труды ИСП РАН, том 28, вып. 5, 2016 г., стр. 227-238. DOI: 10.15514/ISPRAS-2016-28(5)-14

1. Введение

1.1 Обзор литературы

Занимая около половины времени разработки [1], тестирование остается наиболее распространенным методом контроля качества ПО, и его недостаток может приводить к финансовым потерям [2]. При систематическом подходе тестовый набор считается полным, если он обеспечивает определенное покрытие кода [3]. На данный момент существует большое количество систематических генераторов тестов, направленных на поиск стандартных ошибок [3, 4]. Подобные инструменты порождают огромное количество трудночитаемых тестов, обладающих высокой ценой проверки человеком [5]. В предлагаемой работе в целях качественного снижения данной цены использована концепция улучшения читаемости тестов при помощи биграммной модели естественного языка [6]. Данная концепция перенесена с генерации тестов на основе мета-эвристического поиска (SBST) на динамические символьные вычисления (DSE). На сколько известно авторам данной статьи, вышеуказанная концепция ранее не применялась к DSE и предложенный в данной работе метод является первым подобным алгоритмом оптимизации читаемости тестов при помощи модели натурального языка в контексте DSE.

Биграммная модель дает оценку вероятности P принадлежности строки корпусу языка. Согласно модели, степень читаемости строки растет по мере увеличения P. Для строки из n символов данная оценка выражается следующей формулой:

$$\hat{P}(c_1^n) \approx \prod_{i=1}^{n} P(c_i | c_{i-1}) \quad (1)$$

где $P(c_i|c_{i-1})$ — вероятность появления символа c_{i-1} после c_{i-1} [6]. Так как оценка вероятности появления строки в корпусе языка выражается в виде произведения вероятностей появления отдельных биграмм, составляющих слово, то данная оценка для слов большей длины всегда ниже оценки для слов меньшей длины. В целях снижения влияния данного эффекта на оценку читаемости производится ее нормализация, что позволяет сравнивать строки разной длины. Оценка читаемости строки определяется [6] как нормализованное значение N оценки P:

$$N = \hat{P}(c_1^n)^{1/n} \quad (2)$$

При использовании SBST [7] цель тестирования (например, покрытие кода) формулируется в виде функции приспособленности, отображающей входные данные на некоторый количественный показатель. Генерация тестов сводится к многократному запуску программы с подстройкой входных данных для оптимизации функции приспособленности до тех пор, пока не будет достигнута поставленная цель тестирования. Для улучшения читаемости значение N включается в функцию приспособленности [6].

По мере выполнения символьных вычислений [8, 9] на переменные программы накладываются ограничения, образуя ограничение пути (PC), характеризующее класс эквивалентности входных данных, проводящих программу по текущему пути. В каждой точке принятия решения вычисления разветвляются, обеспечивая покрытие кода. Тесты генерируются через решение PC. В отличие от метода оптимизации читаемости на основе SBST [6] в данной работе оптимизация производится над PC.

1.2 Постановка задачи

Задачей данной работы является разработка и экспериментальное исследование метода оптимизации читаемости строк, входящих в состав тестовых случаев, порождаемых во время DSE. Оптимизация читаемости строки сводится к максимизации степени читаемости данной строки, с сохранением случайной природы порождаемых при этом слов.

2. Методы

2.1 Краткое описание

Работа генератора с оптимизацией читаемости выполняется в две фазы. На первой фазе проводятся символьные вычисления, и для каждого рассмотренного пути формируется соответствующее PC. После завершения символьных вычислений, на второй фазе над полученным PC производится оптимизация читаемости.

Возможность оптимизации читаемости исходит из того, что PC представляет собой класс эквивалентности входных данных, проводящих программу по одному пути выполнения. Для каждого отдельного пути выполнения множество решений PC может включать входные векторы со строками, содержащими печатные символы и целые слова. Предложенный в данной статье метод последовательно конкретизирует значения входящих в строки символов, по мере возможности выстраивая их согласно биграммной модели.

2.2 Пример

Для начала рассмотрим работу предложенного метода оптимизации на примере функции strlen. В данном примере (Рис. 1а) имеется одна строка — $\{a_1,a_2,a_3,'\backslash 0'\}$, где a_i — символьная переменная типа char без ограничений.

На первом проходе удается сузить диапазон значений переменных a_1-a_3 до диапазона букв (рисунки 1б и 1в). Последний символ строки '\0' изменить невозможно, так его значение фиксировано. На втором проходе удается произвести конкретизацию пар переменных (a_1, a_2) и (a_2, a_3) согласно биграммной модели, при этом получаются биграммы ('k', 'e') и ('e', 'y') соответственно. Для пары (a3, '\0') вместо второго элемента нельзя подставить букву вследствие фиксированности '\0'. Итоговая строка имеет вид "key".

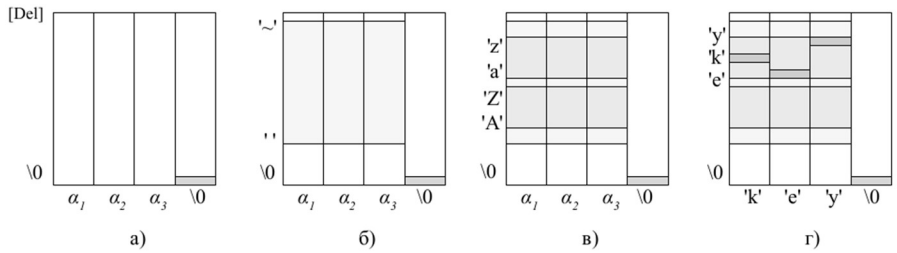

*Рис. 1. Визуализация работы алгоритма оптимизации читаемости для функции **strlen***

*Fig. 1. Example of readablility optimization for the **strlen** function*

2.3 Описание алгоритма

Псевдокод алгоритма оптимизации читаемости представлен на листинге 1.

Внутренним представлением тестовых данных является граф памяти M [10]. Узлами M могут быть скаляры (как символьные, так и конкретные), указатели на узлы, массивы узлов. Оптимизация читаемости осуществляется процедурами Сужение и Конкретизация, в каждой из которых происходит обход M в глубину, при этом рассматриваются только строки.

Процедура Сужение производит последовательные пробы ограничений для каждого отдельного элемента строки. Сначала производится попытка сужения диапазона значений элемента до множества печатных символов (ограничение вида $' ' \le a_i \le '\sim'$), при успешной попытке осуществляется проба ограничения до диапазона букв (ограничение вида $'A' \le a_i \le 'Z' \vee 'a' \le a_i \le 'z'$).

В процедуре Конкретизация используется биграммная модель. На подготовительном шаге производится попытка присвоить элементу a_1 значение произвольной буквы. Далее последовательно просматриваются биграммы (a_i, a_{i+1}). Для первого элемента a_i биграммы при вызове Знач берется значение, именуемое fst, при этом учитывается, является ли данный элемент изначально конкретным или символьным. Если fst — буква, и при этом следующий за a_i элемент a_{i+1} является символьным, то происходит попытка конкретизировать значение a_{i+1}, то есть приравнять его такой букве snd, которая с большой вероятностью может следовать за fst, образуя биграмму (fst, snd). Это осуществляется при помощи вызовов След и Проба.

Процедура Сужение(граф памяти **M**)

Для каждого узла **A** графа памяти **M**

Если **A** — **char[n]**, то для каждого **i** = **1...n:**

Пусть $\mathbf{a_i}$ **= i-**й элемент массива **A**, пусть **printable** = **Проба**($' ' \le \mathbf{a_i} \le '\sim'$)

Если **printable**, то **Проба**($'A' \le \mathbf{a_i} \le 'Z') \vee ('a' \le \mathbf{a_i} \le 'z')$)

Процедура Конкретизация(граф памяти **M**)
Для каждого узла **A** графа памяти **M**
  Если **A** — **char** [**n**], то
    Если a_1 — символьный, то
      Проба($\mathbf{a_1}$ = **alpha**), где **alpha** — произвольная буква
    Для каждого **i** = **1...n-1**:
      Пусть $\mathbf{a_i}$ $\mathbf{a_{i+1}}$ — пара соседних элементов массива, пусть **fst** = **Знач**(a_i)
        Если **fst** — буква и $\mathbf{a_{i+1}}$ — символьный, то
          Пусть **snd** = **След**(**fst**)
          Проба($\mathbf{a_{i+1}}$ = **snd**)

Процедура Знач(Узел **a** графа памяти такой, что **a** — скаляр)
  Если **a** — конкретное, то вернуть значение **a**
  Если **a** — символьное, то запросить у решателя значение **a** и если оно существует вернуть его

Процедура Проба(Ограничение **e**) : булево значение
  Протолкнуть пространство имен в стек решателя
    Пусть **PC' = PC** /\ **e**
    Пусть **sat** = **true**, если для **PC'** существует модель
  Вытолкнуть пространство имен из стека решателя
  Если **sat**, то **PC** ← **PC'**, вернуть **true**
  Иначе — вернуть **false**

Процедура След(ASCII символ **a**) : ASCII символ
Используя биграммную модель и алгоритм рулетки вернуть символ **b**, который может следовать за **a**

Листинг 1. Алгоритм оптимизации читаемости

Algorithm 1. Readability optimization

Для биграммы (fst, snd) процедура След(fst) : snd генерирует символ snd, который с большой вероятностью может следовать за символом fst. При этом используется метод рулетки, схожий со стратегией селекции, применяемой в генетических алгоритмах [11]. Колесо рулетки содержит 26 областей, по одной для каждого символа алфавита. Размер каждой области S_b определяется по следующей формуле:

$$S_b = P(b)/T \text{, где } T = \sum_1^n P(c_i|b) \quad (3)$$

Символ snd отбирается при помощи «запуска» рулетки, то есть генерации числа от нуля до T. Таким образом, символы с более высокой вероятностью

вхождения в биграмму вида (fst | _) имеют больше шансов быть возвращенными процедурой След.

2.4 Консервативность алгоритма

Консервативность данного алгоритма заключается в том, что каждый оптимизированный тест проводит программу по тому же пути, что и соответствующий неоптимизированный вариант. К PC добавляются только такие ограничения (образуя PC'), которые не нарушают его выполнимости. Данное свойство достигается за счет использования процедуры Проба. Таким образом, решение для PC' всегда является решением для PC. Отсюда следует, что для цепочки PC, PC_1, …, PC_n, где PC — исходное (неоптимизированное) ограничение пути, PC_n — конечное (оптимизированное), решение для PC_n всегда будет решением для PC. В то же самое время, обратное неверно.

Следствием консервативности алгоритма является следующее. Если для данного PC в качестве решения не существует входного вектора, включающего строки с печатными символами, то оптимизации не происходит. Тогда тест остается без изменений, и работа завершается на проходе Сужение. Если же для данного PC существуют входные векторы, включающие подобные строки, то: во-первых, все символы, которые возможно, данный алгоритм переведет в буквы или, по крайней мере, печатные символы (проход Сужение); во-вторых, для всех строк и подстрок, где это возможно, будет применена биграммная модель для дальнейшей оптимизации читаемости (проход Конкретизация).

3. Результаты

3.1 Описание генератора тестов

Для апробации метода разработан DSE-генератор на базе инструментов LLVM [12] и CVC4 [13]. При этом использована биграммная модель естественного языка, полученная при анализе ~183 млн. слов [14].

Для начала работы пользователю достаточно написать простой драйвер на языке Си. Тесты выводятся в виде строк f: a_1 a_2 … a_n => r, где f — имя функции, a_i — значение i-го аргумента, а r — результата. Скаляры кодируются целыми числами, указатели — символом '&', строки имеют формат «$a_1a_2...a_n$»{...}, где a_i — символы слева от терминатора, {…} — дамп памяти.

3.2 Описание эксперимента

Экспериментальные исследования генератора проведены на ряде библиотечных функций из репозитория Linux [15]. Результаты эксперимента представлены в таблице 1. В первом столбце указано имя функции, во втором — количество сгенерированных тестов, в третьем закодированы передаваемые аргументы. Строка символьных переменных представлена числом в скобках, например «[6]» обозначает строку из пяти символьных переменных и

терминатора (исключение составляет strpbrk, для нее второй аргумент это строковый литерал «aeiou»). Конкретные значения (размер буфера или длина строки) приведены как целочисленные литералы без скобок. Четвертый столбец содержит процент покрытия инструкций, полученный при помощи gcov. В последующих столбцах указаны результаты усредненной оценки читаемости для оптимизированных тестов.

Табл. 1. Результаты экспериментальных исследований

Table 1. Experimantal results

Функ	***#***	***Арг***	***Покр***	***Нет***	***Баз***	***Прос***	***СП***	***Рул***	***Случ.***
M			**95%**	**-**	**0,07**	**0,08**	**0,08**	**0,07**	**0,06**
σ			**8%**	**-**	**0,03**	**0,03**	**0,03**	**0,03**	**0,02**
strlen	5	[6][6]	100%	-	0,08	0,10	0,10	0,09	0,08
strnlen	6	[6][6]5	100%	-	0,08	0,10	0,10	0,09	0,08
strcmp	15	[6][6]	100%	-	0,04	0,05	0,05	0,04	0,04
strncmp	16	[6][6] 5	100%	-	0,04	0,05	0,05	0,04	0,04
sysfs_ streq	39	[6][6]	100%	-	0,05	0,07	0,06	0,06	0,05
strcpy	35	[6][6]	100%	-	0,08	0,10	0,10	0,09	0,07
strncpy	36	[6][6] 5	100%	-	0,08	0,10	0,10	0,09	0,07
strcat	40	[10][5]	100%	-	0,07	0,08	0,08	0,07	0,06
strncat	41	[10][5] 4	100%	-	0,07	0,08	0,08	0,07	0,06
strstr	19	[6][3]	90%	-	0,12	0,14	0,14	0,13	0,11
strnstr	4	[6][3]2	80%	-	0,09	0,10	0,10	0,09	0,07
strpbrk	10	[6][6]	80%	-	0,03	0,03	0,03	0,03	0,03

Столбец «Нет» содержит данные по запуску без оптимизации. Неоптимизированный вывод не содержит буквы и оценка читаемости для него не определена. В качестве базового метода «Баз.» взят проход Сужение без последующей конкретизации. Условный «простейший» метод применения биграммной модели отображен в столбце «Прост.», здесь для каждой строки первый символ берется без дополнительных ограничений, все последующие выстраиваются по биграммной модели. Модифицированной для эксперимента

процедурой След возвращается символ с *наибольшим* значением вероятности вхождения в биграмму. В эксперименте «СП» включена предусмотренная методом попытка случайной генерации первого символа строки. Столбец «Рул.» отображает результаты итоговой версии метода – здесь в процедуре След использован метод рулетки. Завершающий столбец «Случ.» содержит данные эксперимента, в котором процедура След возвращает случайную букву, т.е. биграммная модель не задействована. Так как в использованной биграммной модели значения приведены с точность 2 знака после запятой, тот же формат принят и в таблице.

Для экспериментов, результаты которых представлены в столбцах «СП», «Рул» и «Случ», где используется генерация случайных чисел, было проведено 5 запусков. Среднеквадратическое отклонение результатов данных запусков от среднего близко к нулю (при рассмотрении двух цифр после запятой это 0), поэтому данный показатель не был включен в таблицу, а указано лишь среднее значение по 5 запускам. Для столбцов, отображающих покрытие кода, а также столбцов с оценками читаемости по разным экспериментам, приведено усредненное значение M результатов по отдельным функциям и среднеквадратическое отклонение σ данных результатов от M.

3.3 Обсуждение

Наибольшая оценка читаемости получена для условно «простейшего» метода. Однако с его применением связаны некоторые неудобства со стороны пользователя. Для наглядности рассмотрим тесты для функции strcpy. Примером неоптимизированного вывода является «&"\x01\x01\x01\x01\x01"{\0} &"\x01\x01\x01"{\0,\0,\0} :=> &"\x01\x01\x01"{\0,\x01,\0}». По данному тесту при ручной проверке без дополнительной информации об исходном коде, затруднительно установить, произошло ли копирование в действительности. Так как strcpy производит копирование данных из одного буфера в другой, не накладывая дополнительных ограничений на символы в строке, то CVC4 в качестве решения для каждой символьной переменной выдает одни и те же значения. Далее, рассмотрим вывод базового метода: «&"aaaaa"{\0} &"aaa"{\0,p,\0} :=> &"aaa"{\0,a,\0}». Читаемость вывода улучшилась, однако проверить корректность также затруднительно. Также стоит отметить, что человеку легче воспринимать слова, нежели последовательности повторяющихся букв. В случае использования «простейшего» метода получаются тесты вида «&"athes"{\0} &"ath"{\0,p,\0} :=> &"ath"{\0,s,\0}». Читаемость тестов повышается, однако проблема проверки остается. Данная проблема решается случайной генерацией первого символа строки. В эксперименте «СП» примером вывода является «&"kesth"{\0} &"pre"{\0,p,\0} :=> &"pre"{\0,h,\0}». Видно, что содержимое буфера назначения действительно изменилось, как и ожидалось. Однако данный метод имеет тенденцию к зацикливанию, т. к. результат процедуры След в данном случае строго детерминирован. Например,

может получиться строка «thesthes...», при этом без дополнительных ограничений она зациклится. Для того чтобы избежать данной проблемы используется метод рулетки. Так как метод рулетки не является детерминированным, и в то же время опирается на вероятность следования символов, заложенную в биграммную модель, он улучшает читаемость по сравнению со случайной генерацией символов, одновременно ликвидируя зацикливание. Примером вывода по данному методу является: «&"fmf"{\0,\x01,\x01} &"emslo"{\0} 5 :=> &"emslo"{\0}». Наконец, для сравнения использована стратегия «Случ» случайного выбора символа, пример «&"jbg"{\0,\x01,\x01} &"ttabn"{\0} :=> &"ttabn"{\0}».

3.4 Достоверность

Для оценки читаемости тестов использованы усредненные значения всех тестов одной функции. Данная стратегия была выбрана вследствие того, что читаемость входных строк зависит от вида функции, и в то же самое время в тестах для одной функции сочетается множество комбинаций строк различной длины (т. к. длина отличается от размера буфера и задается символом '\0'). Различие в длине строк частично компенсируется нормализацией, однако это ведет к тому, что строки меньшей длины получают завышенные оценки. На наш взгляд, в первом приближении этого достаточно для оценки результатов, однако для их уточнения необходимы дальнейшие исследования.

Также стоит отметить, что целью работы не является исследование биграммной модели в изоляции, в отрыве от задачи автоматизации тестирования. Однако для валидации метода оценки читаемости были взяты 100 наиболее частотных слов английского языка [16], а также сто случайных строк длины 2-5 символов. В первом случае полученная оценка составляет 0,10, во втором — 0,07. Эксперименты показывают, что результаты работы оптимизатора сопоставимы со значениями, полученными изолированно.

Традиционно критерием полноты тестирования в символьных вычислениях является покрытие инструкций. В проведенных экспериментах оно в среднем составило 95% и для 9 из 12 функций — 100%. В функциях с неполным покрытием не был рассмотрен вариант, при котором возвращается NULL.

В заключение добавим, что так как символьные вычисления генерируют тесты, систематически исследуя развилки в коде программы, то количество тестов зависит от количества пройденных развилок. В случае функций strlen и strnlen покрытие кода для заданных аргументов максимально возможно — рассмотрены все возможные пути выполнения. В остальных случаях сказалась неполнота теории использованной для поиска решений РС.

4. Заключение

Представленный метод позволяет улучшить читаемость тестов, автоматически сгенерированных при помощи символьных вычислений. Экспериментальные

исследования системы, реализующей данный метод, были проведены на 12-ти функциях по работе со строками из репозитория Linux. Проведенные исследования демонстрируют преимущества данного метода при генерации тестов для функций, включающих строковые данные, в том случае, когда они должны проверяться человеком. Оценка степени читаемости строк, содержащихся в оптимизированных тестах, сопоставима со случаем использования слов натурального языка, что, как показывают исследования в смежной с DST области SBST [6], положительно сказывается на процессе верификации тестов человеком. В то же время неоптимизированные тесты включают трудночитаемые последовательности произвольных символов внутри широкого диапазона ASCII-кодировки. Данный метод может быть встроен в другие системы DSE-генераторы. Настройки метода, приведенные в экспериментах, могут быть перенесены в подобную систему в качестве параметров командной строки. В дальнейшем планируется работа по расширению возможностей генератора, а также экспериментальные исследования его эффективности для более широкого класса программ.

Список литературы

[1]. Hambling B. Realistic and Cost-effective Software Testing. Kelly, Management and Measurement of Software Quality, UNICOM SEMINARS, Middlesex, UK, 1993, pp. 95-112.

[2]. Tassey G. The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology, Final Report, May 2002.

[3]. Saswat A., Burke E.K., Chen T.Y., Clark J., Cohen M.B., Griskamp W., Harman M., Harrold M.J., McMinn P. An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software, 2013, vol. 86, issue 8, pp. 1978-2001.

[4]. Cadar C., Godefroid P., Khundish S., Păsăreanu C.S., Sen Koushik, Tillman N., Visser W. Symbolic Execution for Software Testing in Practice – Preliminary Assessment. ICSE'11 Proceedings of the 33rd International Conference on Software Engineering, 2011, pp. 1066-1071.

[5]. Barr E.T., Harman M., Shahbaz M., Yoo S. The Oracle Problem in Software Testing: A Survey. IEEE Transactions on Software Engineering, 2015, vol. 41, issue 5, pp. 507-525.

[6]. Afshan S., McMinn P., Stevenson M. Evolving readable string test inputs using a natural language model to reduce human oracle cost. International Conference on Software Testing, Verification and Validation, 2013.

[7]. Tracey N., Clark J., Mander K., McDermid J. An automated framework for structural test-data generation. Proceedings of the International Conference on Automated Software Engineering, 1998, pp. 285–288.

[8]. King J.C. Symbolic execution and program testing. Communication of The ACM, 1976, vol. 19, issue 17, pp. 385-394.

[9]. 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 Communication security, 2006, pp 322-335.

[10]. Sen K., Marinov D., Agha G.. CUTE: a concolic unit testing engine for C. ESEC/FSE-13 Proceedings of the 10th European software engineering conference held

jointly with 13th ACM SIGSOFT international symposium on Foundationsof software engineering, 2006, vol. 30, issue 5, pp. 263-272
[11]. J. H. Holland. Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor, 1975.
[12]. Компиляторная инфраструктура LLVM. Режим доступа: http://llvm.org/
[13]. SMT-решатель CVC4. Режим доступа: http://cvc4.cs.nyu.edu/web/
[14]. Jones M. N., Mewhort D.J.K. Case-sensitive letter and bigram frequency counts from large-scale English corpora, 2004, vol. 36, issue 3, pp. 388
[15]. Исходный код ядра Linux. Режим доступа: https://github.com/torvalds/linux/tree/master/kernel
[16]. 100 наиболее частотных английских слов. Режим доступа: https://en.oxforddictionaries.com/explore/what-can-corpus-tell-us-about-language

Test Readability Optimization in Context of Symbolic Execution

I.A. Yakimov <ivan.yakimov.research@yandex.ru>
A.S. Kuznetsov <ASKuznetsov@sfu-kras.ru>
Institute of space and informatic technologies, Siberian Federal University, Akademika Kirenskogo 26 st., Krasnoyarsk, 660074, Russia.

Abstract. Software testing is a time consuming process. In general, software companies spend about 50% of development time on testing. On the other hand, lack of testing implies financial and other risks. In the case of systematic testing a suitable test suite should provide an appropriate code coverage. A lot of code-based test generators have been developed in order to provide systematic code coverage. Such tools tend to produce lots of almost unreadable test suites which hard to verify manually. This problem is formulated as a Human Oracle Cost Problem. This work introduces a method for readability optimization of automatically generated test suites in context of Symbolic Execution. It uses natural language model in order to optimze each test case. This conception has been applied first in the Search Based Testing paradyghm. In contrast of existing search based tool proposed DSE-based tool uses SMT-solver in order to incrementally improve readability of a single test for a single program path. To validate proposed method a tool on top of LLVM and CVC4 frameworks is created. Experimental evaluation on 12 string processing routins from the Linux repository shows that this method can improve test inputs gracefully.

Keywords: code-based test generation; symbolic execution; human oracle cost; bigram language model;

DOI: 10.15514/ISPRAS-2016-28(5)-14

For citation: I.A. Yakimov, A.S. Kuznetsov. Test Readability Optimization in Context of Symbolic Execution. *Trudy ISP RAN/Proc. ISP RAS*, vol. 1, issue 2, 2016. pp. 227-238 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-14

References

[1]. Hambling B. Realistic and Cost-effective Software Testing. Kelly, Management and Measurement of Software Quality, UNICOM SEMINARS, Middlesex, UK, 1993, pp. 95-112.
[2]. Tassey G. The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology, Final Report, May 2002.
[3]. Saswat A., Burke E.K., Chen T.Y., Clark J., Cohen M.B., Griskamp W., Harman M., Harrold M.J., McMinn P. An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software, 2013, vol. 86, issue 8, pp. 1978-2001.
[4]. Cadar C., Godefroid P., Khundish S., Păsăreanu C.S., Sen Koushik, Tillman N., Visser W. Symbolic Execution for Software Testing in Practice – Preliminary Assessment. ICSE'11 Proceedings of the 33rd International Conference on Software Engineering, 2011, pp. 1066-1071.
[5]. Barr E.T., Harman M., Shahbaz M., Yoo S. The Oracle Problem in Software Testing: A Survey. IEEE Transactions on Software Engineering, 2015, vol. 41, issue 5, pp. 507-525.
[6]. Afshan S., McMinn P., Stevenson M. Evolving readable string test inputs using a natural language model to reduce human oracle cost. International Conference on Software Testing, Verification and Validation, 2013.
[7]. Tracey N., Clark J., Mander K., McDermid J. An automated framework for structural test-data generation. Proceedings of the International Conference on Automated Software Engineering, 1998, pp. 285–288.
[8]. King J.C. Symbolic execution and program testing. Communication of The ACM, 1976, vol. 19, issue 17, pp. 385-394.
[9]. 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 Communication security, 2006, pp 322-335.
[10]. Sen K., Marinov D., Agha G.. CUTE: a concolic unit testing engine for C. ESEC/FSE-13 Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundationsof software engineering, 2006, vol. 30, issue 5, pp. 263-272
[11]. J. H. Holland. Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor, 1975.
[12]. LLVM: http://llvm.org/
[13]. CVC4: http://cvc4.cs.nyu.edu/web/
[14]. Jones M. N., Mewhort D.J.K. Case-sensitive letter and bigram frequency counts from large-scale English corpora, 2004, vol. 36, issue 3, pp. 388
[15]. Linux: https://github.com/torvalds/linux/tree/master/kernel
[16]. The 100 commonest English words https://en.oxforddictionaries.com/explore/what-can-corpus-tell-us-about-language