УДК 004.896 Дата подачи статьи: 14.06.16
DOI: 10.15827/0236-235X.116.101-106 2016. Т. 29. № 4. С. 101–106

# ПРИМЕНЕНИЕ СИСТЕМЫ KLEE ДЛЯ АВТОМАТИЗАЦИИ ТЕСТИРОВАНИЯ ПРОГРАММ НА ЯЗЫКАХ C/C++

*А.Г. Зыков, к.т.н., zykov_a_g@mail.ru;*
*И.В. Кочетков, аспирант, melmacson@gmail.com;*
*В.И. Поляков, к.т.н., доцент, v_i_polyakov@mail.ru*
*(Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики (Университет ИТМО), Кронверкский просп., 49, г. Санкт-Петербург, 197101, Россия)*

В работе рассматривается верификация вычислительных процессов, в частности, автоматизация тестирования функциональных ошибок программных продуктов, которые крайне сложно исследовать и воспроизводить без непосредственного исполнения кодовых фрагментов. Основной идеей является применение программ или инструментов символьного исполнения. В частности, подробно рассматривается система KLEE, представляющая собой символьную виртуальную машину, эмулирующую окружение. В ней параллельно выполняются символические процессы, каждый из которых – один из путей в исследуемой программе. Система построена на анализе LLVM байт-кода программы с применением STP-решателя для предикатов. Рассматриваются ее архитектура, состав компонент, принципы работы, базовые возможности, способ моделирования окружения, пример работы на основе тестирования утилиты tr системы MINIX и др.

Целью исследования являются изучение общих возможностей систем символьного исполнения на примере разбора KLEE и их применение для решения задачи автоматизации тестирования. Актуальность данной проблемы высока в связи с постоянно растущей сложностью ПО, которая ведет к увеличению сложности, длительности и, главное, стоимости тестирования и верификации программных продуктов. Новизна данного исследования заключается в том, что на основании детального изучения принципов функционирования системы рассмотрены альтернативные способы применения программ символьного исполнения. К таким способам относятся тестирование с помощью эталона, поиск решения, восстановление схемы алгоритма программы. В качестве результата работы приведена статистика тестирования набора программ пакета COREUTILS 6.11.

***Ключевые слова:*** *автоматизация, тестирование, символьное исполнение, KLEE.*

В тестировании можно выделить классы потенциальных проблем, которые невозможно или крайне трудно исследовать без исполнения части кода [1]. К таким проблемам можно отнести ошибки функциональной корректности алгоритма и др. В подобных ситуациях применение ручного или случайного (на основе случайно сгенерированных наборов входных данных) тестирования крайне неэффективно. Многие работы по данной теме рассматривают применение систем символьного исполнения для анализа программы, поиска ошибок и автоматической генерации тестовых данных [2–4].

В данной работе рассмотрена система символьного исполнения KLEE [5], которая позиционируется как усовершенствование системы EXE, разработанной в 2006 году. Она позволяет автоматизировать тестирование широкого спектра приложений, в том числе и взаимодействующих с окружением [6].

## Принцип работы систем символьного исполнения

Основная идея систем символьного исполнения сводится к тому, что программа выполняется с использованием символьных переменных, которые могут принимать любое значение, а не на разработанных вручную или случайно сгенерированных входных данных. Каждая инструкция в программе, манипулирующая конкретными данными (например $x = y * 2$), заменяется на ту, которая использует символьные значения и переменные (например $x = \mu * 2$).

В ситуациях, когда программа достигает операторов ветвления, система отслеживает обе ветви, каждой из которых сопоставляет набор ограничений, запоминающийся на время исполнения этой ветви, называемый *условием пути*. Если ветвь исполнения завершается или генерирует исключение, на основе сохраненного набора условий пути можно сгенерировать тестовый набор, заменяя символьные переменные на конкретные значения.

Таким образом, если учитывать постоянство исходной программы, ее выполнение на сгенерированном тестовом наборе должно привести к появлению той же ошибки, найденной при символьном исполнении.

Системы символьного исполнения позволяют достигать высокого тестового покрытия на некоторых программах, однако для большинства программ [4] существуют проблемы, затрудняющие достижение подобного результата, а именно:

– экспоненциальный рост количества путей в коде программы;

– тестирование кода, взаимодействующего с окружением (ОС, сеть, пользователь);

– многопоточность и параллельные вычисления.

### Принцип работы системы KLEE

Принцип работы системы KLEE проще описать на примере тестирования конкретной программы. Возьмем утилиту tr операционной системы MINIX. Несмотря на малый размер исходного кода (169 строк кода / 83 исполняемых), она отражает две типичные проблемы при тестировании.

- Сложность. Код программы реализует задачу трансляции и манипуляции символами, поступающими на вход. Все это скрывается под непрозрачным кодом разбора входных аргументов, сложными условиями ветвления и запутанным потоком управления.
- Зависимость от окружения. Исполнение в основном зависит от входных значений аргументов, поступающих с системного потока чтения. Аргументы управляют тем, какую функцию будет выполнять программа и в какую сторону (по ветке IF или ELSE) будет происходить ветвление в ней. Также программа сильно зависит от самой возможности чтения из системного потока, по которому могут поступать как валидные, так и невалидные (и даже небезопасные) данные, которые должны быть обработаны корректно и без ошибок. Поэтому возникает большое количество тестовых сценариев, связанных с граничными условиями и ключевыми значениями.

KLEE как представитель класса систем символьного исполнения работает по следующему алгоритму:

– заменяет все операнды на символьные;

– проходит все ветви программы и все выражения с помощью символьных переменных;

– вычисляет условие пути (значения переменных, необходимые для прохождения от точки входа до текущей позиции), которое привело к окончанию выполнения ветви (программы) или к ошибке; на основе этого условия формируется тестовый набор.

После этого можно использовать сгенерированные тестовые наборы для повторного автоматизированного тестирования программы, то есть, если та или иная ошибка в программе будет исправлена, выполнение полученного ранее теста не должно приводить к ее генерации.

Рассмотрим подробнее пример теста, который приводит к генерации ошибки:

```
1:  void expand(char *arg, unsigned char *buffer) {          8
2:     int i. ac;                                            9
3:      while (*arg)    {                                   10*
4:        if (*arg = = ‘ \ \ ’)  {                          11*
5:          arg++;
6:          i = ac = 0;
7:          if (*arg >= ‘ 0 ‘ && *arg <= ‘ 7 ‘)  {
8:            do  {
9:              ac = (ac << 3) + *arg>++  - ‘ 0 ‘;
10:             i++;
11:           } while (i<4 &&>= ‘ 0 ‘ && *arg<=’ 7 ‘;
12:           *buffer++ = ac;
13:         } else if (*arg != ‘ 0 ‘)
14:           *buffer++  = *arg++;
15:       }  else if (*arg = = ‘ [ ‘) {                     12*
16:         arg++;                                           13
17:         i = *arg++;                                      14
18;         if (*arg++ !=  ‘ – ‘)  {                         15!
19:           *buffer++ = ‘ [ ‘;
20:           arg – = 2;
21:           continue;
22:         }
23:         ac = *arg++;
24:         while (i <=ac) *buffer++ = i++;
25:         arg++;       /*Skip ‘ ] ‘ */
26:       }  else
27:         * buffer++ = *arg++;
28:    }
29: }
30: . . .
31: int main(int argc, char* argv[ ]) {                       1
32:    int index = 1;                                         2
33:    if (argc > 1 && argv[index][0] = = ‘ – ‘)  {           3*
34:       . . .                                               4
35:    }                                                      5
36:    . . .                                                  6
37:   expand(argv[index++],  index);                          7
38:   . . .
39: }
```

Строка 18 генерирует ошибку переполнения буфера на конкретной входной строке (tr [ “” “”).

KLEE проходит исходный код утилиты следующим образом.

1. Сначала KLEE собирает символьные аргументы командной строки, которые не содержат никаких ограничений, кроме нуль-терминации строк, и вызывает метод main.

2. Достигнув ветвления на строке 33 (argc > 1), KLEE вызывает STP-решатель, чтобы узнать, выполнение каких ветвей оператора возможно. Для данного ветвления существуют оба пути, поэтому KLEE разветвляет исполнение и следует по обоим путям, добавляя к соответствующим условиям путей новое ограничение (argc > 1 для истинного, argc ≤ 1 для ложного).

3. Так как количество активных путей больше одного, KLEE выбирает один для исполнения с помощью стратегии отбора. Будем считать, что мы следуем по пути, который приводит к генерации ошибок. Таким образом, по ходу исполнения KLEE добавляет дальнейшие ограничения на содержимое массива arg и разветвляется пять раз (на помеченных “*” строках): дважды на строке 33 и по разу на строках 3, 4, 15.

4. На каждой опасной операции (например разыменование указателя) KLEE проверяет потенциальные причины для генерации исключения в условиях текущего пути. Ошибки не обнаруживаются до строки 18. На этой позиции происходит исключение из-за существования такого входного

значения, которое приводит к двойному инкременту указателя arg без проверки на окончание строки. Поэтому последний инкремент пропускает символ '\0' и начинает указывать на невалидную память.

5. KLEE генерирует конкретные значения для переменных argc и argv, позволяющие при воспроизведении получить идентичное поведение. Чтобы продолжить исполнение текущего пути, KLEE добавляет ограничение, которое не приводит к генерации ошибки.

## Архитектура

На высоком уровне KLEE работает как гибрид между операционной системой для символьных процессов и интерпретатором. Каждый символьный процесс имеет собственные регистровый файл, стек, кучу, программный счетчик и условия пути. Все тестируемые программы должны быть скомпилированы в LLVM-ассемблерный язык, RISC-подобный виртуальный набор инструкций. KLEE интерпретирует этот набор инструкций без каких-либо преобразований или упрощений [5].

KLEE оперирует множеством состояний процесса одновременно подобно диспетчеру процессов. В каждый момент времени выбирается одно из состояний, которое выполняет всего лишь одну инструкцию. Обработка продолжается до того момента, пока либо не останется возможных состояний, либо истечет таймаут, указанный пользователем.

В отличие от обычных процессов все хранилища данных для состояний (регистры, стек и т.п.) содержат не сырые данные, а выражения. Например, при вычислении операции

%dst = sub i32%src0, %src1

в регистр %dst записывается не результат %src0 – %src1, а выражение sub(%src0, %src1). Для достижения большей эффективности при вычислении результата проверяется, являются ли операнды константами; если да, то в регистр записывается не выражение, а его результат.

При достижении оператора ветвления система проверяет, какие ветви возможны. Для этого используется STP-решатель. В случаях, когда возможно выполнение обеих ветвей, текущее состояний клонируется, к каждой из ветвей добавляется соответствующее ограничение, после чего анализ продолжается в обоих направлениях.

Каждая потенциально опасная инструкция проверяет, возможны ли ситуации, при которых входные значения приводят к ошибке. Например, при делении происходит проверка деления на нуль, при сложении и вычитании – проверка на переполнение и т.п. Если KLEE детектирует потенциальную ошибку, генерируется тестовый сценарий и исполнение текущего состояния завершается.

Операторы чтения и записи также являются потенциально опасными, например, при выполнении может произойти ошибка выхода за границы памяти. Память в KLEE представляется в виде набора различных массивов под каждый объект в проверяемом коде. Это сделано потому, что из-за бесконечного количества вариаций невозможно решить задачу поиска ограничений в случае, если представлять память единой для всего символьного процесса.

Многие операции (например проверка границ) требуют наличия специальной информации об объекте. Если указатель может ссылаться на $N$ различных объектов, текущее состояние KLEE клонируется $N$ раз. В каждом состоянии разыменованный указатель ссылается лишь на один конкретный объект.

При работе KLEE активно взаимодействует с модулями STP и SAT.

STP (SMT) – решатель задачи выполнимости формул в теориях. STP оперирует выражениями ($a > 1$, $x != 2$ и т.п.) и взаимосвязью между ними. Результатом его работы является утверждение о выполнимости/невыполнимости входной теоремы (SAT/UNSAT). При работе все входные выражения в конечном итоге преобразуются в единую КНФ-форму, которая поступает на вход уже другому решателю, SAT [7].

SAT – решатель задачи о выполнимости булевой функции. Данный компонент оперирует набором переменных, скобок и операций И, ИЛИ и НЕ. Результатом его работы являются утверждение о выполнимости/невыполнимости входной булевой функции и, в случае выполнимости функции, битовый вектор, каждый из элементов которого соответствует значению одной из входных переменных, при котором функция принимает значение TRUE. Задача о выполнимости булевой функции является NP-полной [8].

## Моделирование окружения

Известно, что при любой автоматизации тестирования (приемочного, модульного и т.п.) всегда встает проблема работы с окружением. Так как сама программа может работать на различных операционных системах, то и тестироваться она должна на различных операционных системах (например, запускать тесты C на ОС Linux или ОС Windows). При этом взаимодействие с окружением – всегда случайный фактор, который невозможно контролировать и который может приводить к ложным срабатываниям (например, ветвление в зависимости от значения системных переменных), а значит, невозможно гарантировать идентичный результат воспроизведения одного и того же теста. Следовательно, требуется моделировать окружение либо пропускать зависящие от него участки кода.

В системе KLEE все вызовы (open, read, write, stat, ioctl и др.), оперирующие окружением, взаимодействуют не с реальной операционной системой, а с ее моделью. Таким образом, символьный процесс работает в «модели» окружения, с которой он может взаимодействовать как с обычной, без необходимости каким-либо образом изменять исходный код. Все модели написаны на языке C и предоставляют возможность модификации и расширения по желанию разработчика.

## Применение системы KLEE

Базовое предназначение системы KLEE – тестирование исходного кода программы на функциональные недостатки и генерация тестовых сценариев, покрывающих все пути активации программы. Однако возможности системы позволяют на ее основе решать и другие задачи, большинство из которых можно решать «из коробки», без внесения дополнительных модификаций. Некоторые из этих возможностей можно реализовать на большинстве инструментов символьного исполнения, так как в их основе лежат общие принципы работы. Перечислим возможные применения системы KLEE.

1. Функциональное тестирование исходного кода программы (описано выше).

2. Приемочное тестирование с помощью эталона.

- С помощью тестовых сценариев. Данный метод основан на том, что одни и те же тестовые сценарии можно воспроизводить на различных исходных файлах при соблюдении наименований входных переменных. При наличии одной эталонной программы (например образец решения лабораторной работы), которая решает известную задачу, сгенерированные по этой программе тесты можно считать набором приемочных тестов (test suite). Данный набор можно использовать для проверки программы-подобия (решения студентами лабораторной работы) на соответствие ожидаемым результатам. Таким образом, если программа-подобие генерирует исключения или дает неверный результат, можно считать ее некорректной.
- С помощью оператора Assert. Пример двух различных вариантов одной функции:

```
1:   unsigned mod_opt (unsigned  x,  unsigned  y)  {
2:   if  (y & ?)  = = y)
3:   return x & (y?1) ;
4:   else
5:   return x % y;
6:   }
7:   unsigned mod (unsigned  x,  unsigned  y)  {
8:   return x % y;
9:   }
10:  int main ()  {
11:  unsigned  x,  y;
12:  make symbolpc(&x,  sizeof(x));
13:  make symbolpc(&y,  sizeof(y));
14:  assert(mod(x, y) = = mod_opt (x, y));
15:  return 0;
16:  }
```

На данном примере можно убедиться в эквивалентности двух функций во всем диапазоне входных значений (y! = 0). Решая задачу на невыполнение условия в Assert, KLEE на основе перебора всех возможных путей приходит к выводу об эквивалентности двух функций на всем диапазоне значений.

3. Поиск решения. Данный метод основан на том, что задача системы KLEE – пройти все возможные пути активации программы. Поэтому, основываясь на этом принципе, для поиска решения математической или логической задачи можно написать простейшую тестовую программу, содержащую в себе данный предикат. При обработке системой KLEE будут сгенерированы тестовые сценарии, которые удовлетворят данный предикат, то есть будет найдено решение, удовлетворяющее поставленной задаче. Рассмотрим пример для простейшего уравнения:

```
1:   if (x^2 + 4x + 4 = = 0)  {
2:   return true;
3:   } else {
4:   return false;
5:   }
```

Система KLEE сгенерирует набор, в котором $x = 2$, что является одним из решений уравнения. К задачам, решение которых можно найти с помощью символьного исполнения, относятся поиск пути в лабиринте и другие. Однако для них требуются более сложные тестовые программы.

4. Восстановление схемы алгоритма готовой программы. Данный метод основан на том, что система KLEE проходит и «исполняет» каждую инструкцию в программе, следуя от точки входа до останова, если таковой имеется. Поэтому, если требуется проанализировать алгоритм, заложенный в некотором исходном коде, и существует ряд препятствий (нет знания языка, исходный код крайне сложен или обфусцирован), можно использовать KLEE в качестве анализатора исходного кода, трассируя и преобразовывая его в некоторый универсальный алгоритмический язык. Этот результат можно визуализировать в классическую схему алгоритма, понятную без специальных знаний. Однако данный метод требует доработки системы, так как подобная функциональность не предоставляется «из коробки».

## Результаты тестирования

Для получения реальных результатов тестирования авторы проанализировали весь набор программ пакета COREUTILS 6.11. Среднее покрытие

кода составило 94 %. Всего программа сгенерировала 3 321 набор входных данных, позволяющих покрыть весь указанный процент кода. Также было найдено 10 уникальных ошибок, признанных разработчиками пакета как реальные дефекты в программах, что является очень хорошим достижением, так как этот набор программ разрабатывается более 20 лет и большинство ошибок было устранено.

## Ограничения

Применение системы KLEE имеет следующие ограничения:

– нет поддержки многопоточных приложений, символьных данных с плавающей точкой (ограничение STP), ассемблерных вставок;

– тестируемое приложение должно быть скомпилировано в LLVM-байт код (как и все его зависимые библиотеки);

– невозможно тестирование программ под системами семейства Windows.

Сделаем выводы. Система KLEE, рассмотренная в данной статье, является одной из систем символьного исполнения, которые все чаще фигурируют в статьях, так или иначе касающихся процесса автоматизации тестирования. KLEE «из коробки» можно применять не только для автоматизации тестирования ПО, но и для решения нестандартных задач.

Из преимуществ данной системы можно выделить следующие:

– достижение тестового покрытия, близкого к полному (90 %+);

– возможность тестирования программ, работающих с окружением;

– возможность воспроизведения полученных тестовых наборов в автоматическом режиме;

– возможность анализа программ, написанных на любом языке программирования, имеющем LLVM-интерпретатор;

– является свободным ПО.

Однако система имеет ряд недостатков:

– может работать только на системах семейства Linux, так как использует компоненты (STP, SAT, LLVM), которые крайне затруднительно установить на Windows;

– требует инструментировать исходный код тестируемых программ.

Дальнейшие исследования будут направлены на усовершенствование системы KLEE для решения проблемы don't care в сложных предикатах [9], добавление поддержки чисел с плавающей запятой в текущий STP проекта либо интеграцию KLEE с другим STP-решателем, так как это необходимо для тестирования математических программ, реализацию возможности тестирования многопоточных программ, а также на изучение и рассмотрение ближайших аналогов системы KLEE (KLOVER [10], $S^2E$ [11, 12], janala2 [13] и др.), анализ их преимуществ, ключевых особенностей, применяемых техник и оптимизаций.

### *Литература*


1. Немолочнов О.Ф., Зыков А.Г., Поляков В.И. и др. Верификация в исследовательских, учебных и промышленных системах // Науч.-технич. вестн. СПбГУ ИТМО. Вып. 11. Актуальные проблемы анализа и синтеза сложных технических систем. 2003. С. 146–151.
2. Zamfir C., Candea G. Execution Synthesis: A technique for automated software debugging. ACM SI-GOPS/EuroSys European Conf. on Comp. Syst., 2010, pp. 321–334.
3. Boonstoppel P., Cadar C., Engler D. RWset: Attacking path explosion in constraint-based test generation. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2008, pp. 351–366.
4. Artzi S. Finding bugs in web applications using dynamic test generation and explicit-state model checking. Soft. Eng., IEEE Transactions on 36.4, 2010, pp. 474–494.
5. Документация KLEE. URL: http://klee.github.io/docs/ (дата обращения: 18.02.2016).
6. Cadar C., Dunbar D., and Engler D.R. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. Proc. 8th USENIX Symposium OSDI, 2008, pp. 209–224.
7. STP constraint solver. URL: http://stp.github.io/ (дата обращения: 20.02.2016).
8. Домашняя страница MiniSAT. URL: http://minisat.se/ (дата обращения: 15.02.2016).
9. Немолочнов О.Ф., Зыков А.Г., Поляков В.И. и др. Метод обнаружения недекларированных возможностей и значений DON'T CARE вычислительного процесса // Изв. вузов. Приборостроение. 2009. Т. 52. № 12. С. 32–39.
10. Guodong L., Indradeep G., Sreeranga R. KLOVER: A symbolic execution and automatic test generation tool for C++ programs. Proc. Intern. Conf. CAV 2011, Cliff Lodge, Snowbird, UT, USA, 2011, pp. 609–615.
11. Chipounov V., Kuznetsov V., Candea G. S2E: a platform for in vivo multi-path analysis of software systems. Proc. Intern. Conf. ASPLOS 2011, Newport Beach, CA., 2011, pp. 265–278.
12. Chipounov V., Kuznetsov V., Candea G. The S2E platform: design, implementation, and applications. Jour. ACM TOCS, 2012, vol. 30, no. 1, pp. 1–49.
13. A concolic testing engine for Java. URL: https://github.com/ksen007/janala2 (дата обращения: 20.02.2016).


Software & Systems Received 14.06.16
DOI: 10.15827/0236-235X.116.101-106 2016, vol. 29, no. 4, pp. 101–106

**KLEE FOR AUTOMATIC TESTING PROGRAMS IN C/C++**

***A.G. Zykov***[1], *Ph.D. (Engineering), zykov_a_g@mail.ru*
***I.V. Kochetkov***[1], *Postgraduate Student, melmacson@gmail.com*
***V.I. Polyakov***[1], *Ph.D. (Engineering), Associate Professor, v_i_polyakov@mail.ru*
[1] *The National Research University of Information Technologies, Mechanics and Optics, Kronverksky Ave. 49, St. Petersburg, 197101, Russian Federation*

**Abstract.** The article considers the problems of automation of software functional error testing. This kind of bugs is extremely difficult to find and reproduce without code execution. This problem is relevant due to ever-growing software complexity. It leads to increasing duration, complexity and cost of software testing and verification.

The purpose of the research is to analyze symbolic execution (evaluation) system basic features and their application in finding functional bugs and software testing in general. In particular, the article discusses the KLEE system in detail. This system is a symbolic virtual machine with environment emulation that executes parallel symbolic processes. It is based on analyzing a program LLVM byte-code. The article deals with KLEE architecture, components set, operation principles, features and some other aspects.

Novelty of this research is in the consideration of alternative methods of using symbolic execution tools. They are developed based on using their basic features. There is testing with reference program, problem solving (for example, finding a way in a labyrinth) and restoring an algorithm flowchart.

The article presents KLEE testing statistics (code coverage, bugs count, etc.) of COREUTILS 6.11 package, where there were found several serious bugs that had been missing for over ten years.

**Keywords:** automation, testing, symbolic execution, KLEE.

***References***

1. Nemolochnov O.F., Zykov A.G., Polyakov V.I. Verification in research, training and industrial systems. *Nauchno-tekhnich. vestnik SPbGU ITMO. Vyp. 11. Aktualnye problemy analiza i sinteza slozhnykh tekhnicheskikh sistem* [Scientific and Technical Journal of Information Technologies, Mechanics and Optics]. St. Petersburg, SPbGU ITMO Publ., 2003, pp. 146–151 (in Russ.).

2. Zamfir C., Candea G. Execution Synthesis: A Technique for Automated Software Debugging. *ACM SI-GOPS/ EuroSys European Conf. on Computer Systems*. Paris, France, 2010, pp. 321–334.

3. Boonstoppel P., Cadar C., Engler D. RWset: Attacking path explosion in constraint-based test generation. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2008, pp. 351–366.

4. Artzi S. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Transactions on Software Engineering*. 2010, vol. 36.4, pp. 474–494.

5. *KLEE Documentation*. Available at: http://klee.github.io/docs/ (accessed February 18, 2016).

6. Cadar C., Dunbar D., Engler D. R. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. *Proc. 8th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*. 2008, pp. 209–224.

7. *STP constraint solver*. Available at: http://stp.github.io/ (accessed February 20, 2016).

8. *MiniSAT home page*. Available at: http://minisat.se/ (accessed February 15, 2016).

9. Nemolochnov O.F., Zykov A.G., Polyakov V.I. Method of detecting undocumented features and DON'T CARE values of a computation process. *Izvestiya vysshikh uchebnykh zavedeny. Priborostroenie* [Journal of Instrument Engineering]. 2009, vol. 52, no. 12, pp. 32–39 (in Russ.).

10. Guodong L., Indradeep G., Sreeranga R. KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs. *Proc. Int. Conf. on Computer Aided Verification (CAV 2011)*. Cliff Lodge Publ., Snowbird, UT, USA, 2011, pp. 609–615.

11. Chipounov V., Kuznetsov V., Candea G. S2E: A Platform for In Vivo Multi-Path Analysis of Software Systems. *Proc. Int. Conf. on ASPLOS 2011*. Newport Beach, CA, 2011, pp. 265–278.

12. Chipounov V., Kuznetsov V., Candea G. The S2E Platform: Design, Implementation, and Applications. *Jour. ACM TOCS*. 2012, vol. 30, no. 1, pp. 1–49.

13. *A concolic testing engine for Java*. Available at: https://github.com/ksen007/janala2 (accessed February 20, 2016).

## Примеры библиографического описания статьи

1. Зыков А.Г., Кочетков И.В., Поляков В.И. Применение системы KLEE для автоматизации тестирования программ на языках C/C++ // Программные продукты и системы. 2016. Т. 29. № 4. С. 101–106; DOI: 10.15827/0236-235X.116.101-106.

2. Zykov A.G., Kochetkov I.V., Polyakov V.I. KLEE for automatic testing programs in C/C++. *Programmnye produkty i sistemy* [Software & Systems]. 2016, vol. 29, no. 4, pp. 101–106 (in Russ.); DOI: 10.15827/0236-235X.116.101-106.