Санкт-Петербургский государственный университет

Кафедра системного программирования

Костицын Михаил Павлович

Поддержка строк в символьной виртуальной машине .NET

Курсовая работа

Научный руководитель:
ст. преп. Кириленко Я. А.

Санкт-Петербург
2018

Оглавление

Введение

С каждым годом сложность программных продуктов возрастает, поэтому трудоёмкость их проверки растёт и количество ошибок в них увеличиваются. На данный момент наиболее популярным методом нахождения ошибок является тестирование. При тестировании ПО на вход программе подаётся подготовленный тестировщиком набор данных, после чего проверяется, соответствует ли результат работы программы ожидаемому. Однако данный метод не всегда даёт необходимые результаты.

В исследовании Рудакова [21] была выявлена основная проблема тестирования — невозможность его полной автоматизации. Для человека же полностью покрыть код тестами очень сложно, вследствие чего некоторые куски кода могут быть не протестированы. В таком случае нельзя гарантировать их корректность.

Имея автоматическую систему проверки корректности программы, можно было бы обеспечить гарантию полного тестового покрытия кода, рассмотреть те входные данные, на которых работа программы не соответствует ожиданиям, и многое другое. В качестве такой системы верификации рассмотрим символьный интерпретатор.

В процессе анализа программного кода многие данные могут быть неизвестны. К примеру, таковыми являются аргументы входных точек библиотек, результаты обращений к API операционной системы и т.д. Символьная интерпретация — техника анализа программ, которая позволяет исполнять код в условии такой неопределенности данных. В последнее время данная техника пользуется особой популярностью и является объектом исследования множества статей [11, 10, 17, 5, 16, 2, 7, 18, 3]. Наличие символьной виртуальной машины позволяет свести задачу анализа кода к задаче автоматического доказательства теорем в логике.

Помимо генерации тестового покрытия символьная виртуальная машина может быть применена в большом количестве задач. Среди них задачи верификации, синтеза программ, символьной отладки и сим-

вольного профилирования.

Проблема, решаемая в данной работе — поддержка строк в символьной виртуальной машине .NET. Задача анализа строк неразрешима, доказательство чего приводится в исследовании [19]. Несмотря на неразрешимость, данная задача является актуальной и активно изучается в академии, являясь предметом многих исследований [12, 13, 14, 9]. В конечном итоге хочется получить систему комплексного анализа строковых выражений в .NET, не уступающую современным системам.

В случае .NET задача осложняется множеством деталей, символьная интерпретация которых является совсем не тривиальной. Например, в .NET существует механизм интернирования строковых литералов, который может осложнить работу в случае недетерминированных строк, а также задачу композициональности. Практически каждое содержательное действие со строками использует хэш-функции, которые должны "экстраполировать" свое поведение на случай частично известной информации о строке. Более того, реализации многих базовых операций над строками в библиотеке mscorlib содержат т.н. unsafe-код, позволяющий работать с указателями, что также осложняет дело в случае наличия недетерминизма.

1. Постановка задачи

Целью данной работы является поддержка строк в символьной виртуальной машине .NET. Для её достижения были поставлены следующие задачи:

- исследовать область анализа программ со строками и выбрать наилучшие методы и подходы;
- изучить особенности работы со строками в платформе .NET;
- разработать представление строк;
- поддержать механизм интернирования;
- осуществить поддержку некоторых базовых методов для работы со строками (хэш-код, взятие длины, конструктор из массива char'ов);
- экспериментально проверить работу полученного решения;
- проанализировать полученные результаты.

```
    int x, y;
1:  if (x > y) {
2:    x = x + y;
3:    y = x - y;
4:    x = x - y;
5:    if (x - y > 0)
6:      assert(false);
    }
```

Рис. 1: Пример: может ли упасть assert?

2. Обзор предметной области

2.1. Символьное исполнение

Символьное исполнение — техника, которая позволяет исполнять программный код в условиях неопределённости входных данных. При символьном исполнении аргументы функции заменяются на ”символы”. Такая абстракция позволяет узнавать что-то о поведении функции в общем. На рис. 1, например, такими ”символами” будут ”x” и ”y”. Ограничения на такие ”символы” входят в path condition. Path condition (PC) — формула, которая определяет условие, что обеспечивает попадание в текущую ветку исполнения. PC является одним из элементов состояния интерпретатора, в которое входит и т.н. символьная память. В начале исследования функции PC = true, далее встречая ветвление исполнение разбивается на ветки, с соответствующими PC. Если символьно исполнить программу из примера на рис. 1, то получится дерево символьного исполнения, представленное на рис. 2, из которого можно заключить, что в рассмотренном примере assert никогда не упадёт.

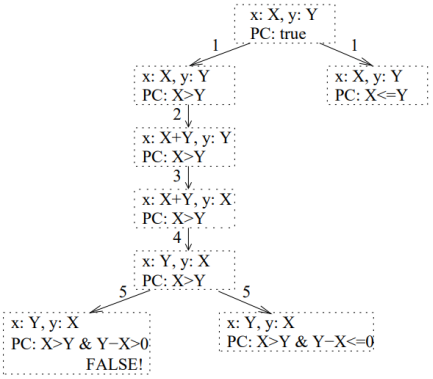

Рис. 2: Дерево символьного исполнения

Существуют два основных подхода символьного исполнения:

- динамический — подход, при котором состояния интерпретатора существуют независимо, никак друг с другом не взаимодействуя и не пересекаясь;
- статический — подход, который подразумевает слияние состояний, посредством ввода новых синтаксических конструкций.

Некоторые современные символьные интерпретаторы взаимодействуют с решателями логики первого порядка, которые могут быть применены, например, для проверки недостижимости ветки исполнения (невыполнимость PC).

Отличный обзор области символьного исполнения можно найти в исследованиях [4, 20].

2.2. SMT/Horn Solvers

Horn / SMT решатели — решатели логики первого порядка. Включают в себя некоторые теории, среди которых теория массивов, теория неинтерпретированных функций и другие. Решатели принимают на вход формулу логики первого порядка с символами из заранее заданных теорий. Если формула выполнима, решатель вернёт SAT и модель с конкретизированными свободными переменными. Иначе решатель возвращает UNSAT. Так как некоторые теории или их комбинации неразрешимы, решатель может иногда возвращать UNKNOWN или зависать.

2.3. Строки в .NET

В данном разделе буду рассмотрены особенности строк в .NET.

Во-первых, строки в .NET являются неизменяемым типом данных, то есть каждый метод, изменяющий строку, фактически возвращает новую структуру строки. Во-вторых, строки являются ссылочным типом, т.е. последовательно располагаются в куче. В-третьих, класс String

переопределяет оператор сравнения так, что сравнение происходит не по ссылке, а по значению, что для строк означает посимвольное сравнение.

2.3.1. Представление в памяти

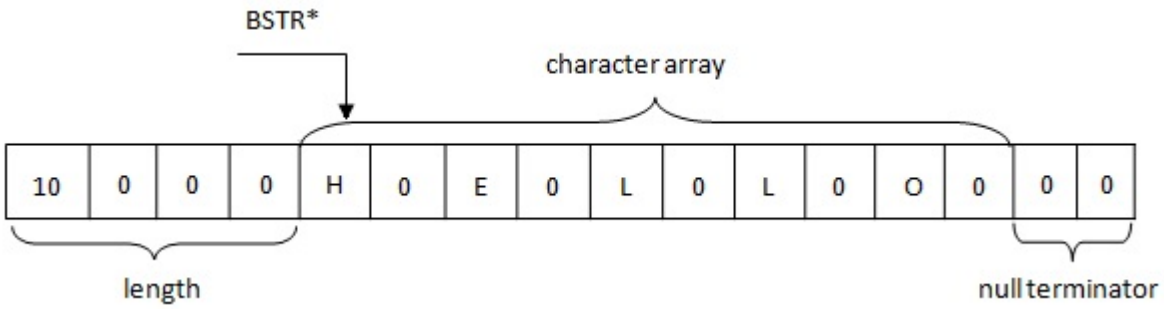

Рис. 3: Формат BSTR

В памяти строки представляются в формате BSTR (Рис. 3). Из рисунка видно, что первые четыре байта строки занимает её длина, после чего располагаются двухбайтовые символы строки в формате UTF-16. Самые последние два байта являются нулевыми и называются nullterminator. Также особенностью BSTR представления является указатель на строку (BSTR*), который всегда указывает на первый её символ. Такой особенности нет, например, в формате PWSZ, который используется в C++.

Так как менеджер памяти в CLR выделяет память кратной 4 байтам, то общий размер строки составляет 4 * ((14 + 2 * length + 3) / 4), где length - это количество символов в строке.

2.3.2. Интернирование

Интернирование — механизм, при котором одинаковые литералы представляют собой один объект в памяти.

В рамках домена существует хэш-таблица, называемая пулом интернирования. Ключами этой хэш-таблицы является хэши строк, а значениями – ссылки на эти строки. Во время JIT-компиляции литеральные

строки последовательно заносятся в таблицу. Каждая строка в таблице встречается только один раз. На этапе выполнения ссылки на литеральные строки присваиваются из этой таблицы. По умолчанию интернируются лишь строковые литералы. Для интернирования нелитеральных строк существуют два механизма взаимодействия с пулом интернирования:

- String.Intern(s : string) : string – метод класса String, который в случае отсутствия строки s в пуле интернирования помещает s в пул и возвращает ссылку на неё, иначе возвращает ссылку на строку, содержащуюся в пуле;
- String.IsInterned(s : string) : string – метод класса String, который в случае наличия строки s возвращает ссылку на интернированную строку, в противном случае – возвращает null.

3. Обзор существующих решений

3.1. Анализ программ

На текущий момент существует несколько проектов виртуальных машин, выполняющих анализ программ с использованием символьной интерпретации, Pex является одним из них.

3.1.1. Pex

Pex (IntelliTest) [15] — часть Visual Studio 2015 Enterprise предназначенная для генерации тестового покрытия .NET программ. Данный инструмент является ярким примером конкретно-символьного интерпретатора, т.е выполняет лишь одну конкретную трассу программы из целого набора веток исполнения, объединённых одним PC. Данный подход отлично справляется с задачей генерации полного тестового покрытия, однако в задачах верификации такой подход проигрывает полностью символьному исполнению. Так происходит, потому что исполнение конкретных веток, даёт лишь недоаппроксимацию поведения программы, а значит необходимая информация могла быть утеряна.

3.1.2. Система V#

V# — Это виртуальная машина .NET, использующая композициональное статическое символьное исполнение, с композициональным доказательством теорем над дизъюнктами Хорна.

Среди главных особенностей проекта можно выделить следующие:

- композициональность [1], где под этим термином подразумевается вначале символьное исполнения функций в изоляции с последующей композицией состояний;
- слияние состояний интерпретатора;
- символьная типизация;
- символьная память, использующая вложенные символьные кучи

- символьная интерпретация неограниченной рекурсии [8].

Данный проект разрабатывается в рамках JetBrains Research.

4. Архитектура

Схема работы проекта V# представлена на рисунке 3.

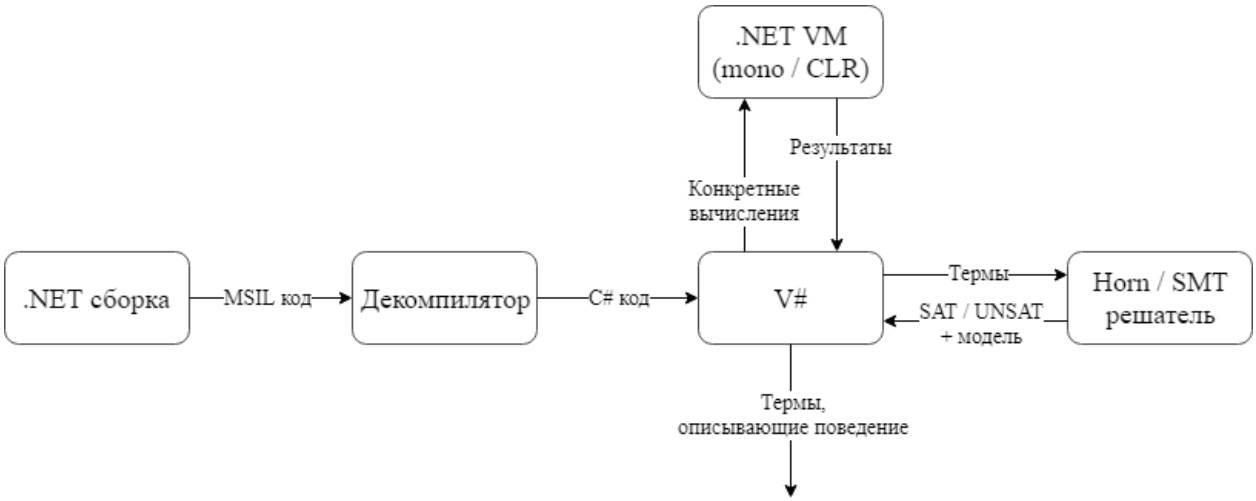

Рис. 4: Схема работы V#

В начале работы имеется .NET сборка, которую необходимо исследовать. При помощи декомпилятора dotPeek от компании JetBrains из неё мы получаем подмножество C# кода в виде AST. С этого дерева кода начинается символьное исполнение: внутри системы V# AST преобразовывается в символьные термы, которые вместе с ограничениями на пути исполнения кодируются в логику и передаются в решатель Spacer (модификация Microsoft Z3); благодаря результату решателя V# может судить о достижимости ветки исполнения. Исследование недостижимых веток прекращается. Данный процесс продолжается итеративно, пока не будут получены итоговые термы, описывающие поведение функции. Помимо термов результатом исследования функции является состояние символьной памяти после исполнения. Структура проекта изображена на рисунке 3.

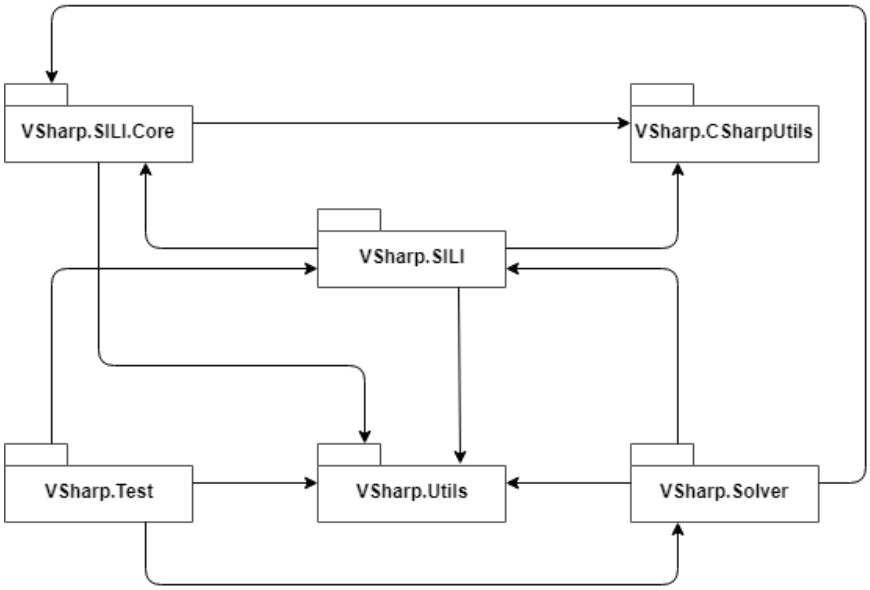

Рис. 5: Структура проекта V#

Мной были затронуты следующие части проекта:

- VSharp.SILI.Core — ядро системы, совершающее всю основную работу символьной интерпретации;
- VSharp.SILI — модуль, который преобразовывает C# AST в символьные термы и передаёт их в ядро VSharp.SILI.Core для дальнейших вычислений;
- VSharp.CSharpUtils — модуль, в котором находятся тесты для проверки корректности работы проекта.

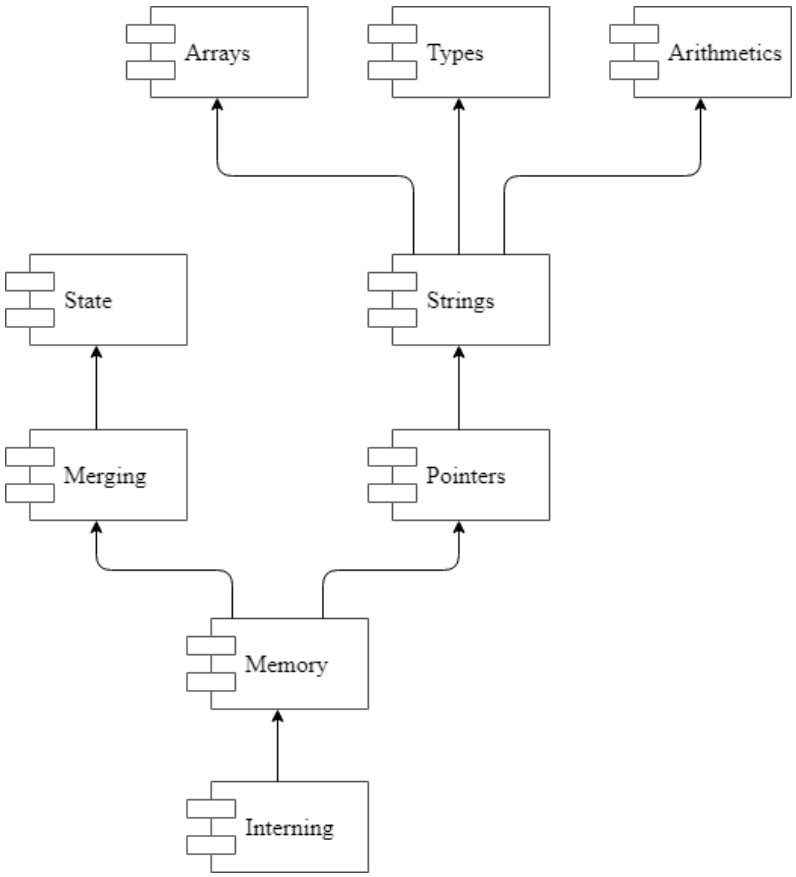

Рис. 6: Модули строк в иерархии модулей ядра V#

На рисунке 4 представлена часть модулей ядра, необходимая для поддержки строк в символьной виртуальной машине V#.

Среди использованных модулей ядра мной были модифицированы следующие:

- Interning — модуль для поддержки механизма интернирования;
- Memory — модуль, отвечающий за работу с символьной памятью;
- Pointers — модуль для работы с указателями;
- Strings — модуль, отвечающий за работу со строками;
- Types — модуль, отвечающий за построение символьных типов;
- Merging — модуль, отвечающий за слияние состояний и термов интерпретатора;
- State — модуль для работы с состоянием интерпретатора.

4.1. Подход к поддержке строк

Основная идея моего решения — кодирование строк в линейную арифметику с использованием теории массивов и неинтерпретированных функций [6]. В случае наличия неограниченной рекурсии предполагается использование вышеупомянутой эвристики Мордвинова, которая может сильно ослабить форму индуктивного инварианта и упростить задачу для решателей дизъюнктов Хорна, не прибегая к ограниченному развертыванию рекурсии.

Поддержка строк в символьной виртуальной машине, в основном, сводится к стандартным механизмам работы символьной памяти. Имея данный механизм, остается реализовать общее представление строк в символьной памяти, инициализацию строк в данной модели памяти, поддержку пула интернирования, а также операций для взаимодействия с ним, вычисление символьного хэш-кода и реализации примитивных строковых операций CLR (которые помечены модификатором extern в mscorlib).

4.1.1. Представления строк

В качестве представления строк было принято выбрать ссылку на структуру в памяти, полями которой являются длина строки и массив символов — содержимое строки. Данное представление строки совпадает с представлением класса String в библиотеке ядра .NET mscorlib.

4.1.2. Интернирование

Основная сложность поддержки механизма интернирования строк заключалась в экстраполяции его поведения на случай символьных строк, композициональности, а также интернировании литеральных строк на этапе JIT компиляции.

Данная задача была решена путём добавления символьной таблицы, ключи которой — содержимое строк, а значения — ссылки на строки. Для эмуляции JIT компиляции было принято решение использовать модуль SILI во время преобразования C# AST в символьные термы,

соответственно строковые литералы интернируются именно на этом этапе. Композициональность пула интернирования реализована через вызов механизмов символьной памяти, а также через механизм недетерминированного добавления строкового литерала в кучу (пул).

5. Реализация

Реализация поддержки строк в символьной виртуальной машине .NET была разбита на этапы, которые приведены ниже.

5.1. Базовая реализация

Для поддержки строк в символьной виртуальной машине V# было сделано следующее:

- в модуле Types добавлен новый тип-алиас String;
- добавлено преобразование строки в структуру с двумя полями: длина строки (String.m_StringLength) и массив символов (String.m_First
- реализован конструктор строки из массива char'ов;
- добавлено сравнение структур строк, использующее сравнение массивов (модуль Arrays) и линейную арифметику (модуль Arithmetics);

5.2. Реализация символьной хэш-функции для строк

Внутри ядра системы V# в модуль Strings была добавлена функция, которая разбивает случай строки на три варианта:

- конкретная строка
- частично-символьная строка
- полностью символьная строка

В конкретном случае строки функция вычисляет конкретное значение хэша с помощью .NET VM. В сивольных же случаях функция возвращает "символ", который сохраняет знание о ссылке на хэшируемую строку.

В дальнейшем планируется расширить поведение данной функции на случай произвольного объекта, а также усовершенствовать сам алгоритм введением специальных синтаксических конструкций, которые в

символьных случаях позволят сохранять знание и о самом содержимом объекта. Данная модификация позволила бы сильно улучшить работу функции благодаря композициональности.

5.3. Реализация механизма интернирования

- в модуле State ядра системы состояние интепретатора было расширено символьной кучей (пулом интернирования), ключами которой являются вышеупомянутые структуры строк, а значениями — ссылки на строки;
- создан модуль Interning, в котором находятся реализованные функции взаимодействия с пулом интернирования: intern и isIterned, использующие стандартные механизмы символьной памяти;
- в модуле SILI, в процесс преобразования C# AST в символьные термы встроен вызов вышеупомянутой функции intern в случае строковых литеролов;
- композициональность реализована путём разветвления состояния интерпретатора во время интернирования строкового литерала по условию наличия этого литерала в пуле;
- в модуль Merging добавлено слияние пула интернирования, которое вызывает слияние символьных куч;
- в модуль Pointers добавлен механизм сравнения ключей пула интернирования, который вызывает сравнение структур строк.

Заключение

В ходе работы были получены следующие результаты:

- исследована область анализа программ со строками, выбраны наилучшие методы и подходы;
- изучены особенности работы со строками в платформе .NET;
- разработано представление строк;
- поддержан механизм интернирования;
- поддержаны некоторые базовые методы для работы со строками (хэш-код, взятие длины, конструктор из массива char'ов);
- проведено экспериментальное тестирование решения;
- проанализированы полученные результаты;
- результаты работы представлены на конференции «Современные технологии в теории и практике программирования 2018».

Дальнейшее направление работы

В дальнейшем планируется изучить теорию строк в решателях, произвести полноценный анализ строк с кодирование их в решатели и использованием теории строк, а также дополнить список поддержанных методов работы со строками.

Данные задачи планируется выполнить в рамках дипломной работы.

Список литературы

[1] Anand Saswat, Godefroid Patrice, Tillmann Nikolai. Demand-driven compositional symbolic execution // International Conference on Tools and Algorithms for the Construction and Analysis of Systems / Springer. — 2008. — P. 367–381.

[2] Combining Symbolic Execution and Model Checking to Verify MPI Programs / Hengbiao Yu, Zhenbang Chen, Xianjin Fu et al. // arXiv preprint arXiv:1803.06300. — 2018.

[3] Dependence guided symbolic execution / Haijun Wang, Ting Liu, Xiaohong Guan et al. // IEEE Transactions on Software Engineering. — 2017. — Vol. 43, no. 3. — P. 252–271.

[4] D'silva Vijay, Kroening Daniel, Weissenbacher Georg. A survey of automated techniques for formal software verification // IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. — 2008. — Vol. 27, no. 7. — P. 1165–1178.

[5] Eliminating Path Redundancy via Postconditioned Symbolic Execution / Qiuping Yi, Zijiang Yang, Shengjian Guo et al. // IEEE Transactions on Software Engineering. — 2018. — Vol. 44, no. 1. — P. 25–43.

[6] Embedded software verification using symbolic execution and uninterpreted functions / David Currie, Xiushan Feng, Masahiro Fujita et al. // International Journal of Parallel Programming. — 2006. — Vol. 34, no. 1. — P. 61–91.

[7] Improving the Cooperation of Fuzzing and Symbolic Execution by Test-cases Prioritizing / Jiaxi Ye, Bin Zhang, Ziqing Ye et al. // Computational Intelligence and Security (CIS), 2017 13th International Conference on / IEEE. — 2017. — P. 543–547.

[8] Mordvinov Dmitry, Fedyukovich Grigory. Synchronizing constrained Horn clauses // LPAR, EPiC Series in Computing. EasyChair. — 2017.

[9] Reference Abstract Domains and Applications to String Analysis / Roberto Amadini, Graeme Gange, François Gauthier et al. // Fundamenta Informaticae. — 2018. — Vol. 158, no. 4. — P. 297–326.

[10] Selfie: Towards Minimal Symbolic Execution / Alireza S Abyaneh, Christoph M Kirsch, Sara Seidl Poncelet et al. — 2018.

[11] Sethu Ramesh. Systems and methods to reverse engineer code to models using program analysis and symbolic execution. — 2018. — Mar. 22. — US Patent App. 15/268,011.

[12] Bultan Tevfik, Yu Fang, Alkhalaf Muath, Aydin Abdulbaki. String Analysis for Software Verification and Security. — 2018.

[13] String Constraints with Concatenation and Transducers Solved Efficiently (Technical Report) / LUKAS HOLIK, PETR JANKU, ANTHONY W LIN et al. — 2018.

[14] String Manipulating Programs and Difficulty of Their Analysis / Tevfik Bultan, Fang Yu, Muath Alkhalaf, Abdulbaki Aydin // String Analysis for Software Verification and Security. — Springer, 2017. — P. 15–22.

[15] Tillmann Nikolai, De Halleux Jonathan. Pex–white box test generation for. net // International conference on tests and proofs / Springer. — 2008. — P. 134–153.

[16] Tuning parallel symbolic execution engine for better performance / Anil Kumar Karna, Jinbo Du, Haihao Shen et al. // Frontiers of Computer Science. — 2018. — Vol. 12, no. 1. — P. 86–100.

[17] Using Test Ranges to Improve Symbolic Execution / Rui Qiu, Sarfraz Khurshid, Corina S Păsăreanu et al. // NASA Formal Methods Symposium / Springer. — 2018. — P. 416–434.

[18] Veritesting Challenges in Symbolic Execution of Java / Vaibhav Sharma, Michael W Whalen, Stephen McCamant,

Willem Visser // ACM SIGSOFT Software Engineering Notes. — 2018. — Vol. 42, no. 4. — P. 1–5.

[19] Word equations with length constraints: what's decidable? / Vijay Ganesh, Mia Minnes, Armando Solar-Lezama, Martin Rinard // Haifa Verification Conference / Springer. — 2012. — P. 209–226.

[20] A survey of symbolic execution techniques / Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia et al. // ACM Computing Surveys (CSUR). — 2018. — Vol. 51, no. 3. — P. 50.

[21] Рудаков ИВ, Гурин РЕ. Разработка и исследование синтетического метода верификации программы с помощью SMT-решателей // Вестник Московского государственного технического университета им. НЭ Баумана. Серия «Приборостроение». — 2016. — no. 4 (109).