

Санкт-Петербургский государственный университет

Системное программирование

Группа 21.Б07-мм

Реализация разделённого чтения и записи в символьной памяти в проекте V#

ПОЖАРСКИЙ Роман Николаевич

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
доцент кафедры СП, к. ф.-м. н., Д. А. Мордвинов

Консультант:
Магистр СПбГУ по направлению «Программная инженерия» М. П. Костицын

Санкт-Петербург
2024

Оглавление

Введение	3
1. Постановка задачи	5
2. Обзор	6
2.1. Символьное исполнение	6
2.2. Модель памяти с регионами	9
2.3. Существующие решения	11
2.4. Проект $V\#$	12
3. Алгоритм	14
3.1. Модель памяти $V\#$	14
3.2. Постановка проблемы	19
3.3. Разделённое чтение	20
3.4. Разделённая запись	22
3.5. Внедрение терма Ite	25
4. Реализация	26
4.1. Особенности	26
4.2. Исходный код	28
5. Эксперимент	29
Заключение	31
Список литературы	32

Введение

Тестирование является неотъемлемым этапом разработки программного обеспечения, поскольку оно позволяет проверить корректность работы системы на потенциальных входных данных и облегчает сопровождение продукта при внесении изменений и добавлении функциональности. Суть тестирования заключается в том, что фактический результат работы программы сравнивается с ожидаемым значением. Для тестирования могут применяться инструменты генерации тестов на основе анализа исходного кода. Несмотря на то, невозможно полностью автоматизировать процесс тестирования, обойдясь только кодом, сгенерированными подобными инструментами, можно существенно сэкономить ресурсы при тестировании далеко не тривиальных сценариев работы системы. В области формальной верификации и генерации тестового покрытия набирает популярность такая техника статического анализа программ, как символьное исполнение [5].

Символьное исполнение позволяет исполнить код с символьными константами, подставленными вместо входных данных: для анализа программы не обязательно предоставлять конкретные входные данные. Работе программы при конкретных входных данных соответствует один определённый путь, поскольку предопределены выборы в условных операторах, зависящих от входных параметров. При исследовании исходного кода для генерации тестового покрытия при помощи символьного исполнения главной целью является формирование наборов входных данных, исполнение на которых покрывает как можно больше ветвей исполнения программы.

Символьное исполнение невозможно без использования SMT-решателей (Satisfiability Modulo Theories). В теории решателей, представляющие формулы логики первого порядка, кодируются ограничения, возникающие из условных операторов и другой неопределённости в программе. Если при исследовании встречается условный оператор, выполнение которых нельзя проверить непосредственно, можно исследовать только те ветви, защитное условие которых является выполнимым, то есть для

которого существуют конкретные данные, на котором рассматриваемое условие является истинным.

Одной из главных проблем символьного исполнения является экспоненциальный рост возможных сценариев работы программы (англ. path explosion), который возникает из-за ветвлений на все достижимые ветви. Поэтому при символьном исполнении активно применяются различные эвристики и оптимизации, которые на практике помогают быстрее анализировать программы [3].

Как и любая другая виртуальная машина, символьная виртуальная машина обладает определённой моделью памяти, которая определяет, как выполняются операции чтения, записи и выделения. Символьная память должна хранить информацию о всех взаимодействиях с памятью в виде как удобном для выполнения типичных операций с памятью, так и для эффективного кодирования запросов к решателю. В данной работе будет описан алгоритм модификации чтения и записи для символьной памяти в проекте V#, а также некоторые другие оптимизации, которые стали возможны благодаря улучшенной модели памяти.

Символьная виртуальная машина позволяет символьно интерпретировать инструкции программы. V# — это платформа для генерации тестового покрытия программ на платформе .NET, .NET CORE, .NET FRAMEWORK, в рамках которой проводилась работы. Платформа содержит в себе символьную виртуальную машину, спроектированную для исполнения CIL (Common Intermediate Language). V# символьно исполняет CIL, что позволяет исследовать все языки платформы .NET, ограничения накладываются лишь поддержкой инструкций CLR (Common Language Runtime), виртуальной машины .NET.

1. Постановка задачи

Целью работы является реализация разделённого чтения и записи для модели символьной памяти в проекте $V\#$. Для её выполнения были поставлены следующие задачи:

1. изучить литературу по моделированию символьной памяти;
2. разработать алгоритмы разделённого чтения и записи для символьной памяти;
3. реализовать разработанные алгоритмы в проекте $V\#$;
4. сравнить полученное решение с существующей моделью символьной памяти $V\#$.

2. Обзор

В разделе 2.1 будет более подробно описано символьное исполнение, включая некоторые эвристики по объединению состояний и термов, позволяющие смягчить эффект взрыва путей. Некоторые модели символьной памяти, в частности та, которая применяется в $V\#$, описаны в разделе 2.2. Существующие решения представлены в разделе 2.3. В разделе 2.4 описаны некоторые высокоуровневые компоненты проекта $V\#$ и связи между ними, а также модули, в которые были внесены изменения в ходе работы.

2.1. Символьное исполнение

Символьное исполнение — техника статического анализа программного обеспечения, позволяющая исполнить код функции в условиях неопределённости входных данных. Вместо параметров функции подставляются символьные константы, представляющие из себя абстракцию над всеми возможными входными значениями, и программа символьно интерпретируется с этими константами. Символьные константы также могут возникать и при выполнении операций, которые задействуют символьные данные. Для проверки выполнимости условий, в которых участвуют символьные константы, используются SMT-решатели.

Решатели поддерживают различные теории, например, теорию линейной арифметики, хорошо подходящие для кодирований арифметических операций с целыми числами; теорию массивов, на которую хорошо ложатся условия, задействующие объекты из кучи. Важно отметить, что для кодирования тех или иных условий одни теории могут быть удобными, а другие — нет. Некоторые условия, например, касающиеся подтипирования объектов, проблематично пытаться кодировать в решатель, и для проверки их выполнимости могут создаваться отдельные системы, как, например, решатель для типов в $V\#$.

Результатом обращения к решателю для проверки выполнимости условия может быть SAT и модель, то есть конкретные данные, на которых условие реализуется, или UNSAT, что означает, что условие

является несовместным.

Существуют разные подходы к тому, когда вызывать решатель для проверки выполнимости условия. Самым популярным вариантом является проверка выполнимости условия сразу при встрече условного оператора (англ. *eager evaluation*). Этот подход позволяет не тратить ресурсы на заведомо недостижимые ветви работы программы, но закономерно имеет такой недостаток, как очень частые обращения к решателю. Именно такой подход и применяется в $V\#$. Но существует еще и подход с отложенной проверкой (англ. *lazy evaluation*).

Стадию символьного исполнения функции характеризуют следующие параметры

1. счётчик программы pc (англ. *program counter*), он определяет следующую к выполнению инструкцию;
2. условие пути π (англ. *path condition*), представляет собой логическое выражение, при котором поток управления дойдёт до текущего состояния;
3. символьная память σ , которая хранит информацию о взаимодействиях с памятью.

```
public int Abs(int a) {  
1   var result = a;  
2   if (result < 0) {  
3       result = -a; }  
4   return result; }  
}
```

Листинг 2.1: функция модуля

Рассмотрим изменение этих параметров при исполнении функции вычисления абсолютного значения целого числа, код которой представлен на листинге 2.1.

1. $\{pc = 1; \pi = True; \sigma = \{a \mapsto \alpha_a\}\}$
2. $\{pc = 2; \pi = True; \sigma = \{a \mapsto \alpha_a, result \mapsto a\}\}$

3. теперь встречается ветвление, необходимо сделать запрос к решателю, проверив как выполнимость самого условия, так и его отрицания. Очевидно, оба варианта выполнимы, поэтому исходное состояние разбивается на два других, соответствующих веткам **if** и **then**

- $\{pc = 3; \pi = \text{result} < 0; \sigma = \{a \mapsto \alpha_a, \text{result} \mapsto a\}\}$
 - $\{pc = 4; \pi = \neg(\text{result} < 0); \sigma = \{a \mapsto \alpha_a, \text{result} \mapsto a\}\}$
4. • $\{pc = 4; \pi = \text{result} < 0; \sigma = \{a \mapsto \alpha_a, \text{result} \mapsto -a\}\}$
- $\{pc = -1; \pi = \neg(\text{result} < 0); \sigma = \{a \mapsto \alpha_a, \text{result} \mapsto a\}\}$
5. • $\{pc = -1; \pi = \text{result} < 0; \sigma = \{a \mapsto \alpha_a, \text{result} \mapsto -a\}\}$
- $\{pc = -1; \pi = \neg(\text{result} < 0); \sigma = \{a \mapsto \alpha_a, \text{result} \mapsto a\}\}$

Таким образом, символьное исполнение функции завершилось двумя состояниями, из которых, можно, например, сгенерировать 2 теста, если сохранять модели, которые выдаёт SMT-решатель.

Поскольку в состоянии обычно приходится хранить много другой информации, в частности, всю символьную память, то неконтролируемое копирование состояний может привести к огромным затратам по памяти, поэтому в символьном исполнении по возможности стараются не разветвлять состояния, если, например неопределённость остаётся в рамках одной ветки исполнения и глобально на дальнейшее исполнение она не влияет. Но иногда может быть удобно объединять состояния. Финальные состояния после исследования кода на листинге 2.1 можно представить следующим образом

$$\begin{aligned}
 &\{pc = -1; \pi = \text{result} < 0 \vee \neg(\text{result} < 0); \\
 &\quad \sigma = \{a \mapsto \alpha_a, \text{result} \mapsto \text{ITE}(a < 0, -a, a)\}\} = \\
 &= \{pc = -1; \pi = \text{True}; \sigma = \{a \mapsto \alpha_a, \text{result} \mapsto \text{ITE}(a < 0, -a, a)\}\}
 \end{aligned}$$

где **ite** обозначает **if-then-else** выражение, принимающее 3 аргумента: **condition**, **expr1**, **expr2** и возвращающее **expr1**, если выполняется

`condition`, иначе — `expr2`. Подобный подход к слиянию состояний по сравнению с общепринятым имеет следующие преимущества:

- возможность продолжить исследование, если решатель не поддерживает кодирование определённых ограничений;
- избежание дополнительных обращений к решателю, поскольку в этом подходе не вводятся дополнительные символьные переменные для кодирования ограничений;
- можно выполнять операции с памятью без вызова решателя.

Более подробно преимущества описанного метода слияния состояний, а также сравнение по эффективности с вариантом символьного интерпретатора, не использующим слияние состояний, описаны в статье MultiSE[4].

2.2. Модель памяти с регионами

Основной задачей символьной памяти является сохранение всех изменений, произведённых над памятью. Данные должны быть представлены в корректном и достаточно эффективном виде для постоянных операций чтения и записи, а также кодирования в решатель. Под корректностью подразумевается соответствие реального исполнения программы на данных из модели финального состояния и тем, как символьная виртуальная машина проинтерпретирует код исследуемой программы, исходя из данных состояния, в частности, символьной памяти.

Наиболее сложным аспектом, который нужно моделировать в символьной памяти, является ситуация, когда разные переменные могут ссылаться на один и тот же объект, и изменения объекта через доступ к одной переменной влияют на последующие результаты чтения из других переменных. На листинге 2.2 приведён пример такой программы, где массивы *a* и *b* могут подменять друг друга. При моделировании символьной памяти необходимо учитывать подобные сценарии.

```
public void Foo(int[] a, int[] b) {
```

```
a[0] = 1;
Console.WriteLine(b[1]);
}
int[] a = new int[] {1; 2; 3};
Foo(a, a);
```

Листинг 2.2: Подмена объектов

Рассмотрим **линейную** и **ветвящуюся** модель памяти. Эти модели диаметрально противоположными способами разрешают проблему доступа к объектам по символьным ссылкам.

В линейной модели вся память представляется как один большой массив, внутри которого хранятся и все ссылки, поэтому доступ к любому объекту в памяти разрешается однозначно. Такая модель хорошо ложится на теорию массивов SMT-решателей, но имеет ряд недостатков, делающих такую модель неприменимой на практике. Главным недостатком является то, символьная ссылка может указывать на произвольный элемент массива, в котором будут храниться как ссылочные типы, не наследующиеся друг от друга, так и значимые типы, и решателю нужно будет проверять заведомо неподходящие варианты. Также такой подход плохо масштабируется при наличии 32-битного адресного пространства.

Ветвящаяся модель предполагает, что при чтении по символьному адресу p , мы рассматриваем все имеющиеся в памяти объекты и разветвляем текущую ветвь исполнения на каждый объект, на который действительно может указывать адрес p , и в каждой ветви адрес p уже определён однозначно. У этого подхода есть 2 очевидных недостатка: во-первых, необходимо проверить адреса всех объектов в памяти на соответствие адресу p ; во-вторых, количество состояний будет экспоненциально быстро расти. Тем не менее, этот подход находит применение на практике в таких символьных исполнителях, как KLEE и SPF.

В V# используется гибридный подход — разделение памяти на **регионы**. Регионом считается множество объектов, которые могут иметь одинаковый адрес в памяти. Для кластеризации объектов используется анализ указателей. Если система анализа указателей слишком ча-

сто решает, что два адреса могут указывать на один объект, память программы становится одним большим регионом, что является по сути линейной моделью памяти. Если же система часто решает, что адреса не могут указывать на один объект, то состояния будут часто разветвляться, и в этом случае память программы будет близка к ветвящейся. Практические результаты [2, 6] показывают, что память, разделённая на регионы, в зависимости от качества анализа указателей, предоставляет возможность и не слишком часто разветвлять состояния, и достаточно быстро совершать операции чтения и записи.

Объекты, хранящиеся в одном регионе, могут подменять друг друга, и изменение одного из них может сказываться на других объектах в этом же регионе. Такие объекты мы будем называть **псевдонимами**. При этом имеется гарантия, что изменение объектов из одного региона не модифицирует объекты в других регионах.

Итак, результатом операций чтения в большинстве случаев будет являться множество возможных значений, которые реализуются при тех или иных условиях. На данный момент из-за некоторых условностей символьного исполнения, на одном из этапов при чтении происходит некорректное поведение, возникающее из-за недостаточно точного обмена информацией между символьной памятью и конкретной. В работе будет описан способ решения этой проблемы через разделение результата чтения на две составляющие: конкретные ссылочные объекты, защищённые точными условиями, при которых результатом чтения должны быть соответствующие объекты; в оставшуюся часть, то есть в символьную константу, пойдут символьные данные в качестве контекста. Символьные данные не приводят к некорректному поведению.

2.3. Существующие решения

uSVM [1] — символьная виртуальная машина, позволяющая исполнять программы на нескольких языках программирования: сейчас поддерживаются JAVA, Go, PYTHON, благодаря абстрагированию рефлексии до отдельной библиотеки. В uSVM используется модель памяти с

регионами, аналогичная той, что используется в $V\#$: в рамках одного состояния объекты хранятся в специальных регионах памяти, имеющих древовидную структуру, позволяющую легко анализировать программы, где возникает проблема анализа псевдонимов.

Алгоритм разделённого чтения реализован в универсальной символьной виртуальной машине $uSVM$. Однако модифицированный алгоритм записи, позволяющий совершать операции чтения быстрее за счёт усложнения алгоритма записи, в $uSVM$ не реализован.

2.4. Проект $V\#$

На рисунке 1 представлена упрощённая архитектура проекта с основными модулями, с которыми необходимо было взаимодействовать в ходе работы.

Символьное исполнение всегда начинается в модуле *Explorer*, где определяется, например, стратегия исследования и выбора состояний для исследования и определены такие параметры, как максимальная глубина развёртывания циклов. Этот модуль на максимально высоком уровне управляет символьным исполнением.

Интерпретатор в модуле *SILI* (Symbolic Intermediate Language Interpreter) символьно интерпретирует инструкции *CIL*.

Финальные состояния (вызывающее необработанное исключение или дошедшие до конца исполнения) в модуле *TestGenerator* преобразуются в *Unit-тест* на языке $C\#$.

Модуль *SILI.Core* представляет из себя ядро платформы: там определяются термы, операции над ними, упрощения, решатель для типов, структуры данных, представляющие символьную память а также состояния и способы ветвлений по ним. Красным отмечены модули, в которые было внесено больше всего изменений.

В модуле **MemoryRegion** определены регионы памяти, в которых хранятся объекты, являющиеся псевдонимами, и операции над ними.

Memory является более высокоуровневым модулем, который работает с конкретной и символьной памятью, представленной регионами

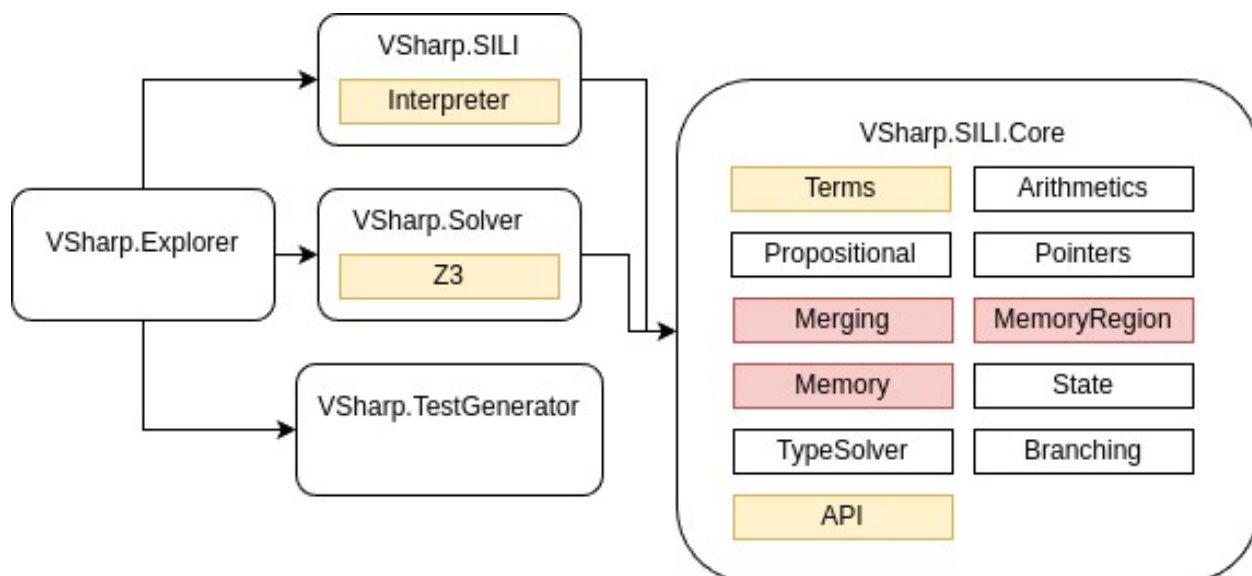


Рис. 1: Архитектура проекта

памяти. Здесь реализованы, например, функции чтения и записи из массивов, копирования из них, запись в поле класса и т.д.

В модуле **Merging** реализованы операция упрощения для специального терма ITE, реализация которого понадобилась при внедрении алгоритмов разделённого чтения и записи.

3. Алгоритм

Основной задачей работы является реализация алгоритма разделённого чтения и записи. Подготовительными этапами для неё являются реализация условия символьного соответствия ссылок. В разделе 3.1 будет более подробно описана модель памяти в проекте. В разделе 3.2 будет строго поставлена проблема, решениями которой являются разрабатываемые алгоритмы. Модифицированный алгоритм чтения и записи описаны в разделах 3.3 и 3.4 соответственно. Для обновлённой модели памяти понадобилось реализовать терм ITE: он формально определён в разделе 3.5.

3.1. Модель памяти $V\#$

Поскольку символьный интерпретатор исполняет байт-код платформы .NET, в символьной памяти необходимо моделировать и те области, которые использует CLR.

Память, используемая .NET программой, разделяется на 4 категории

1. *стек*, используется для хранения примитивных типов и структур, локальных переменных и параметров функций;
2. динамическая память, используемая для хранения ссылочных типов, например, пользовательских классов, массивов и строк;
3. статическая память, в ней хранятся статические поля классов;
4. пул итернирования строк, эта область необходима для специального механизма платформы .NET для оптимизации памяти при работе со строками.

Данная работа нацелена на повышение корректности символьного исполнения при работе с динамической памятью.

Состояние — объект, характеризующий стадию символьного исполнения, в котором хранится вся необходимая информация (символьная

память, условие пути, стек вызовов и т.д.) Конкретные объекты (выделенные явно в процессе исполнения), которые не подвергались изменениям, связанным с символьными константами, хранятся в специальной структуре данных, которая фактически является словарём, ключами которого выступают адреса, а значения — объекты. Символьные ссылочные типы разбиваются на части: классы — на поля, массивы — на элементы, и хранятся в специальной структуре данных, являющейся значением некоторого словаря. Там же хранятся и конкретные объекты, над которыми производились символьные изменения. Словари имеются для

1. буферов, выделенных на стеке при помощи *stackalloc*;
2. полей классов;
3. элементов массивов;
4. длин массивов;
5. нижних границ массивов;
6. статических полей;
7. аллоцированных в процессе исследования объектов.

Ключами этих словарей являются специальные идентификаторы, например, для полей классов это имя, тип и класс, в котором объявлено поле, а значениями — **регионы памяти** (англ. memory region), упомянутые в разделе 2. Регионы памяти с одинаковыми идентификаторами соответствуют разным состояниям. В рамках данной работы можно отождествить регион памяти с **деревом обновлений** — структурой данных, которая используется для представления записей в регион памяти в иерархическом порядке, позволяющем сохранить корректность

и полноту. Формально дерево обновлений определяется так

$$\begin{aligned}\langle \text{UpdateTree} \rangle &::= \text{Dictionary} \langle \text{Region} \rangle, \\ &\quad (\langle \text{UpdateTreeKey} \rangle, \langle \text{UpdateTree} \rangle) > \\ &\quad | \text{NilTree} \\ \langle \text{UpdateTreeKey} \rangle &::= \langle \text{Key} \rangle, \text{Value}\end{aligned}$$

Пока регионы ключей не определены формально, можно считать, что это некоторые множества.

Дерево обновлений поддерживает 2 инварианта:

1. регионы поддеревьев не пересекаются;
2. регионы поддеревьев являются подмножествами региона дерева.

Update tree key хранит ключ записи (адрес модифицируемого объекта), а также записываемое значение.

В разделе 2 ставилась проблема разрешения результата чтения из кучи по символьному адресу. При записи по ключам возникает необходимость отразить в символьной памяти информацию о том, на каких переменных может сказаться текущая запись. Для этого нужны **регионы ключей**, которые представляют из себя множество возможных значений компонент ключа. Грамматика региона ключа выглядит следующим образом

$$\begin{aligned}\langle \text{region} \rangle &::= \langle \text{interval} \rangle \langle \text{indices} \rangle \\ \langle \text{interval} \rangle &::= [\langle \text{int} \rangle, \langle \text{int} \rangle] \mid (\langle \text{int} \rangle, \langle \text{int} \rangle] \mid [\langle \text{int} \rangle, \langle \text{int} \rangle) \mid (\langle \text{int} \rangle, \langle \text{int} \rangle) \\ \langle \text{indices} \rangle &::= \times \langle \text{points} \rangle \langle \text{indices} \rangle \mid \epsilon \\ \langle \text{points} \rangle &::= \langle \text{complement} \rangle \{ \langle \text{int} \rangle \langle \text{ints} \rangle \} \mid \mathbb{Z} \\ \langle \text{complement} \rangle &::= \mathbb{Z} \setminus \mid \epsilon \\ \langle \text{ints} \rangle &::= , \langle \text{int} \rangle \langle \text{ints} \rangle \mid \epsilon \\ \langle \text{int} \rangle &::= -2 \ 147 \ 483 \ 648 \dots 2 \ 147 \ 483 \ 647\end{aligned}$$

Компонента **interval** необходима для представления возможных значений адресов объектов. В символьном исполнении адресом объекта удобно считать время его создания, что позволяет достаточно просто учитывать ситуации подмены объектов. Временем началом исследования считается 1, 0 служит null-объектом. Все конкретные объекты имеют положительные адреса. Адресом символьных констант удобно считать отрицательные числа, поэтому регион адреса символьной константы всегда выглядит одинаково: $(-\infty; 0]$. Этот регион есть у всех ключей, выделенных на куче. Структура данных *points* представляет множество возможных значений индексов массива. Она является либо конечным множеством точек, либо дополнением к бесконечному: в этом случае считается, что индекс не равен хранящимся значениям. Под $\mathbb{Z}$ подразумевается множество возможных значений 4-байтного `int`. Далее запись $-\infty$ будет обозначать `int.MinValue`. На регионах естественным образом определяются такие теоретико-множественные операции, как объединение, пересечение, вычитание.

Отношение предок-ребёнок в дереве обновлений означает, что запись предка теоретически может перекрывать записи по ключу, хранящемуся в детях. Поэтому для полноты полагается, что родитель перекрывает ребёнка. Условие, при котором предок не перекрывает ребёнка формируется как неравенство ключей записи предка и ребёнка.

Для чтения из региона памяти ключу k нужно сделать следующее:

1. определить регион ключа, пусть $r := k.Region$;
2. выбрать из имеющегося дерева обновлений поддереву, индексирующееся регионами, которые пересекаются с регионом r . Значения в ключах записей этого поддерева являются потенциальными результатами чтения.

Для того, чтобы сделать запись в регион памяти по ключу k , необходимо совершить чтение из дерева обновлений по ключу k , результатом которого будет дерево t_i . Также нужно сохранить поддерево, индексирующееся регионами, которые с регионом k не пересекаются, обозначим его за t_d . Обновлённый регион памяти должен выглядеть следующим

образом: к дереву t_d в корень подвешивается запись с ключом k , поддеревом которой является t_i .

Разберём изменения дерева обновлений для региона памяти записей по элементам одномерных массивов с типом `object` при выполнении кода на листинге 3.1.

```
public void Foo(object[] a, int i, int j) {
    a[i] = 5;
    a[1] = 1;
    var b = new int[5];
    b[i] = i;
}
```

Листинг 3.1: Разделение дерева обновлений

1. записи $a[i] = 5$ соответствует пара $(\text{key}, \text{value}) = (a[i], 5)$. Регион ключа $a[i]$ выглядит так: $(-\infty, 0] \times \mathbb{Z}$, так как массив a и индекс i являются символьными объектами. Поскольку дерево было пустое, добавляем в него новую запись по соответствующему региону.
2. далее делается запись $a[1] = 1$, которой соответствует пара $(\text{key}, \text{value}) = (a[1], 5)$. Регион ключа $a[1]$ это $r_1 = (-\infty, 0] \times \{1\}$. Из дерева обновлений выбирается поддерево, пересекающееся с регионом r_1 и не пересекающееся с r_1 , t_i и t_d соответственно. Деревья t_i и t_d индексируются регионами $r_1 = (-\infty, 0] \times \{1\}$ и $r_2 = (-\infty, 0] \times \mathbb{Z} \setminus \{1\}$. Новая запись перекрывает записи из t_i , и не может влиять на записи по ключам из t_d .

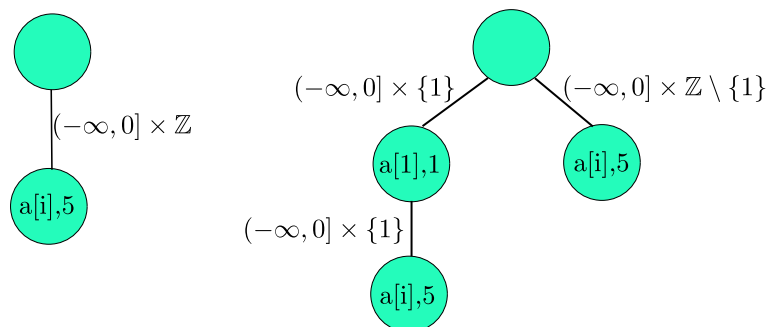


Рис. 2: Дерево обновлений после исполнения кода на Листинге 4.1

Дерево обновлений после всех операций будет выглядеть так:

$$\{r_1 \mapsto ((a[1], 1), t_i), t_d = r_2 \mapsto ((a[i], 5), \text{NilTree})\}$$

3.2. Постановка проблемы

После того, как было дано представление об устройстве символьной памяти в $V\#$, имеется все необходимая информация, чтобы детально поставить проблему, на решении которой направлена работа. Рассмотрим код на листинге

```
public void Foo(int i) {  
    var a = new string[] { "a", "b", "c", "d" };  
    if (a[i].Length != 1) {  
        return -1; } // unreachable  
    return 1; }
```

Листинг 3.2: Пример функции, вызывающий некорректное поведение в $V\#$

В функции на листинге 3.2 напрямую создаётся массив a , он является конкретным объектом, поэтому у него будет какой-то положительный адрес. Затем происходит чтение по символьному индексу i из массива, результатом которого должна быть какая-то конкретная строка, также имеющая положительный адрес. Но поскольку результат неизвестен из-за неопределённости в виде i , результатом чтения будет символьная константа, внутри которой будет сохранено содержимое массива. Но у символьных констант, откуда бы они не появились — как начальные параметры, или как неопределённые чтение каких-то конкретных объектов, адрес будет отрицательным. Поскольку массив a символьным изменением не подвергался, он вместе со всеми строками находится в конкретной памяти, а не в регионах памяти.

Далее читается длина строки, считанной по $a[i]$, и поскольку это символьная константа, длину строки необходимо искать в регионе памяти для поля `Length` класса `String`. Но этот регион является пустым (массив и его содержимое находятся в конкретной памяти), поэтому при подстановке данных из модели на уровне, ответственном за определение

результата операции чтения исходя из значений модели, длина какой-то конкретной строки считана не будет, и символьный интерпретатор считает блок кода, защищённого условия, достижимым, хотя защищающее его условие, очевидно, невыполнимо.

3.3. Разделённое чтение

Для решения этой проблемы предлагается при чтении по ключу k вытаскивать из дерева обновлений t для рассматриваемого региона памяти значения конкретных объектов, и делать из них $\text{Ite } (g_1, v_1), \dots, (g_n, v_n)$, где n — количество конкретных объектов в дереве t_l , локализованном по региону ключа k , а g_i — условие, при котором результатом чтения гарантировано является v_i . В t_l также могут содержаться записи с символьными значениями, поэтому поддерево t , узлы которого содержат запись с символьным значением, будет сохранено как контекст символьной константы, которая добавится как последний элемент Ite .

При такой реализации чтения символьное исполнение в рассмотренном примере не будет искать длину в регионе памяти для длинн строк, а прочтёт значение длинны непосредственно по объекту, находящемуся в конкретной памяти. Псевдокод алгоритма представлен на 3.3

3.3.1. Условие соответствия ключей

Для того, что составить корректное условие на чтение значения, необходимо на уровне логических выражений с термами, которыми оперирует символьный интерпретатор, уметь составлять условия, при котором один ключ будет читать из другого. Ранее в проекте эта логика нужна была только в модуле, который делает запросы к решателю, поэтому эта функциональность была перенесена в интерфейс ключей памяти. Благодаря тому, что в модуле решателя была функция, которая преобразует произвольный булевый терм в объекты решателя, получилось заметно упростить содержимое модуля, взаимодействующего с решателем.

Data: updateTree, readKey, predicate

Result: term

begin

```
    ite ← []
    path ← []
    symbolicTree ← updateTree.Copy()
    iterator ← BottomRightIterator(updateTree, path)
    while iterator.HasNext() do
        reg, node, subtree ← iterator.Next()
        if predicate(node.Value) then
            matchCondition ← readKey.MatchCondition(reg,
                node.Key)
            skipParentCondition ← Conjunction(path.Map(key → not
                key.Equals(readKey))
            guard ← matchCondition ∧ skipParentCondition
            ite ← ite + [(guard, node.Value)]
        else
            symbolicTree ← symbolicTree.Remove(node)
        end
    end
    constantGuard ← Conjunction(ite.Map((g,v) → not g))
    constant ← MakeConstant(symbolicTree)
    ite ← ite + [(constantGuard, constant)]
```

end

Algorithm 1: Псевдокод алгоритма разделённого чтения

Для того, чтобы ключи k_1 , k_2 с регионами r_1 , r_2 соответствовали друг другу, должны выполняться следующие условия

1. k_1 равен k_2 покомпонентно;
2. компоненты k_1 должны принадлежать r_2 (или компоненты k_2 должны принадлежать r_1).

Чтобы убедиться в том, что второе условие необходимо, рассмотрим последовательность записей (1) $a[i] = i$, (2) $a[1] = 1$. Чтобы ключ $a[i]$ гарантировано читал значение i , необходимо, чтобы запись (2) не перекрывала запись (1), именно эту информацию отражает условие 2.

3.3.2. Соответствие ключей в дереве обновлений

В условиях перекрытия записей в дереве обновлений для получения условия соответствия ключей недостаточно только ключ, по которому происходит чтение, проверить на соответствие с ключами имеющихся записей при помощи MatchCondition: необходимы дополнительные условия, отражающие информацию о том, что предки рассматриваемой записи не перекрыли её саму.

Пусть выполняется произвольная операция чтения по ключу k с регионом r и $k_1, \dots, k_n$ — все ключи дерева обновлений, удовлетворяющие некоторому предикату $P(k)$, $r_1, \dots, r_n$ — индексирующие их регионы. Пусть $c_{i1}, \dots, c_{ij_i}$, $1 \leq i \leq n$ — путь из корня дерева обновлений до узла, соответствующего записи k_i , $k_i = (\text{key}_i, \text{value}_i)$, а $r_{i1}, \dots, r_{ij_i}$ — регионы, индексирующие соответствующие узлы пути. Ite будет иметь вид $(g_1, v_1), \dots, (g_n, v_n)$, где g_i — условие, по которому ключ r будет читать из ключа k_i

$$g_i = k.\text{MatchCondition}(r_i, \text{key}_i) \wedge \bigwedge_{m=1}^{j_i} \neg k.\text{Equals}(c_{im}.\text{key}) \quad (1)$$

Как и ожидалось, время работы символьного исполнения в некоторых ситуациях возросло, поскольку при каждом чтении обходить дерево обновлений может быть довольно дорого. Отсюда возникает идея не давать записям, не удовлетворяющим предикату, перекрывать записи, удовлетворяющие ему, тем самым нарушая иерархию дерева, но сохраняя корректность за счёт добавления к узлам дерева дополнительного поля **guard**. Таким образом, при чтении нужно будет обходить каждую ветвь до тех пор, пока узлы на пути удовлетворяют предикату, а оставшийся подграф можно будет объединить в одно дерево для символьной константы, вообще не обходя его.

3.4. Разделённая запись

Как и раньше, для записи нужно локализовать поддерево t_i по региону ключа, по которому мы пишем, при этом сохранив оставшееся

поддерево t_d . С этого момента начинаются отличия: ранее всегда новая запись добавлялась поверх t_i и добавлялась в корень дерева t_d . Обновлённый алгоритм предполагает, что в дереве поддерживается следующий инвариант: не существует записи с конкретным значением, один из предков которой имеет не символьное значение. Поэтому если запись, которая добавляется, имеет конкретное значение, алгоритм обрабатывает так же, как и ранее. В противном случае необходимо обойти дерево t_i , и между конкретными ветвями (возможно, отсутствуют) и их символьными «остатками» (возможно, отсутствуют), поместить текущую запись. К конкретным записям по пути добавляется условие, что текущая запись их не перекрывает (поскольку иерархически она перекрывает всё дерево t_i).

С точки зрения чтения, результат чтения из дерева, иерархия в котором не нарушалась, эквивалентна чтению из дерева, иерархия которого нарушена, но корректность сохранена благодаря дополнительному полю `guard`. Для того, чтобы показать это, рассмотрим чтение по ключу k с регионом r из деревьев t_r и t_w , которые соответствуют моделям памяти с реализованными алгоритмами разделённого чтения и разделённой записи.

Пусть в локализованном для чтения по ключу `readKey` дереве по ключу `writtenKey` имеется запись с конкретным значением v , индексируемая регионом r , иначе результат чтения — одна лишь символьная константа с локализованным деревом. Тогда условие, при котором результатом чтения гарантировано будет v , определяется согласно формуле 1. Если же чтение производится из дерева t_w , где иерархия нарушается для записей с символьными значениями, то 1 даёт только часть условия для чтения v , поскольку в пути отсутствуют записи по символьным значениям, которые могли бы перекрывать `writtenKey`. Однако если такие записи существуют, то информация о том, что они не перекрывают `writtenKey` содержит в поле `guard` для записи (`writtenKey`, `value`), и чтобы получить полноценное условие для чтения, достаточно взять его в конъюнкцию к формуле 1.

За счёт усложнения логики записи, увеличивается скорость при чте-

Data: updateTree, record, predicate

Result: updateTree

begin

 region $\leftarrow$ record.Key.Region

 included, disjoint $\leftarrow$ updateTree.Split(region)

if *predicate(record)* **then**

 includedHanged $\leftarrow$ (record, included)

return disjoint.Add(region, includedHanged)

else

 iterator $\leftarrow$ TopLeftIterator(included)

while *iterator.HasNext()* **do**

 reg, node, subtree $\leftarrow$ iterator.Next()

if *predicate(node)* **then**

 nonEquality $\leftarrow$ not current.Key.Equals(record.Key)

 current.Guard $\leftarrow$ current.Guard $\wedge$ nonEquality

else

помещаем запись снизу конкретной ветви

 subtree.InsertTop(reg, record)

пропускаем оставшуюся символьную ветвь

 iterator.SkipWhilePredicateIsFalse()

end

end

end

included модифицированный циклом **return**

 disjoint.Add(region, included)

end

Algorithm 2: Псевдокод алгоритма разделённой записи

нии, а так как статистически чтения происходят чаще записей, то в общем скорость исследования программ должна вырасти относительно модели памяти, где реализовано только разделённое чтение.

3.5. Внедрение терма **Ite**

Результатом разделённого чтения стал терм, представляющий множество возможных значений из региона памяти, и каждому значению соответствует точное логическое условие, при котором всё выражение упрощается до соответствующего значения. Множеству значений размера n соответствует терм, ELSE значением которого тоже является ITE глубины $n - 1$, а ITE порядка 1 можно считать значение, защищённое условием TRUE. **Ite** имеет следующую структуру:

$$\langle \text{Ite} \rangle ::= (\langle \text{BoolExpr} \rangle, \langle \text{term} \rangle, \langle \text{Ite} \rangle) \mid \langle \text{term} \rangle$$

Во всём проекте была реализовано обработка этого терма, а также реализован ряд упрощений над ним в модуле **Merging** и некоторый набор полезных функций для работы с **Ite**.

4. Реализация

В разделе 4.1 будут описаны особенности реализации описанных алгоритмов и некоторые проведённые оптимизации оптимизации. Ссылки на исходный код представлены в разделе 4.2.

4.1. Особенности

4.1.1. Персистентные структуры данных

Поскольку в символьном исполнении часто копируются и разветвляются состояния, дерево обновлений реализовано при помощи персистентного словаря, ключами которого являются регионы ключей, а значениями — пары из узлов `UpdateTreeKey` с информацией о записи (ключ, значение, защитное условие), и поддерева.

4.1.2. Сохранение иерархии

Так как алгоритм разделённой записи стал нарушать иерархию записей, к `UpdateTreeKey` было добавлена временная метка, показывающая номер записи в регионе. Это поле позволяет сохранить корректность при чтении по ключам, которые не были потенциально перекрыты другими записями, но тем не менее находятся не наверху дерева для сохранения его инварианта. Последняя запись в ветке дерева, в котором иерархия нарушается, при наличии записей с символьными и конкретными значениями, определяется по наибольшему значению метки среди верхней записи (для конкретных записей иерархия сохраняется) и следующей после самой глубокой записи в конкретной подветви (для символьных записей иерархия сохраняется).

4.1.3. Дизъюнктность `Ite`

Для того, чтобы упростить запросы к решателю, условия в `Ite` из чтения не являются дизъюнктными, а сам `Ite` составляется снизу-вверх при обходе дерева обновлений. Если решателю поступит запрос на вы-

полнимость какого-либо условия на этом **Ite**, этот терм будет закодирован в **Z3 Ite** решателя. Поэтому, например, в самом глубоком **Z3 Ite** уже подразумевается, что условия из **Ite** более высоких уровней не выполнены. Однако если добавлять дополнительные условия на дизъюнктность условий, решатель тратит больше времени на выполнение запроса: он не может распознать, что условия на дизъюнктность условия на нижнем уровне со всеми предыдущими уже выполнены.

Однако полностью избавиться от дизъюнктности условий при работе с **Ite** не получилось. Если мы делаем запись по **Ite** (например, `a[i].Length = ...`), результатом чтения по `a[i]` будет **Ite** из конкретных записей в соответствующем регионе, и, возможно, символьной константы. Необходимо сделать запись в поле **Length** по всем термам из **Ite**, но регионы конкретных ссылочных объектов не пересекаются, и при переносе всех записей их иерархия будет потеряна. Поэтому чтобы отразить информацию о том, что k -ая запись из **Ite** могла происходить только при условии, что не были проведены все предыдущие, к её защитному условию нужно добавить отрицание защитных условий предшествующих компонент **Ite**.

4.1.4. Упрощения Ite

Для Ite-термов был реализован ряд упрощений:

1. если в i -ой компоненте защитным условием является **True**, все последующие компоненты можно отбросить, поскольку они недостижимы;
2. если в i -ой компоненте защитным условием является **False**, эту компоненту можно отбросить;
3. сглаживание Ite, значениями которого могут являться другие Ite, без нарушения порядка.

4.1.5. Копирования массивов

Операции копирования из одного массива в другой неопределённого количества элементов соответствует запись по специальному ключу `RangeArrayIndex` со значением в виде символьной константы. Тем не менее, поскольку в массиве, из которого производится копирование, могут храниться конкретные ссылочные объекты, удобно считать, что такие записи предикату удовлетворяют, и записи символьных значений не должны их перекрывать.

4.1.6. Оптимизации

Добавление поля `guard` к записи в регион памяти позволило провести важную оптимизацию, направленную на сокращение расходов памяти. Запись по ключу, компоненты которого являлись `Ite`, что стало довольно частым явлением после модификации алгоритмов чтения и записи, стало возможно проводить в одно состояние, поскольку записи в дереве обновлений теперь существуют только при выполнении условия в поле `guard`. Это позволило избежать копирования состояний на все компоненты `Ite`.

4.2. Исходный код

Условие соответствия ключей было реализовано, протестировано и поступило в `master`-ветку проекта. Исходный код доступен по ссылке <https://github.com/VSharp-team/VSharp/pull/291>.

Разделение результатов чтения и записи было реализовано и проверено на дополнительных тестах. `Pull request` поступил в основную ветку, с кодом ознакомиться можно по ссылке <https://github.com/VSharp-team/VSharp/pull/295>

Laptop	Victus by HP Laptop 16-e012ur
CPU	5 5600H
OS	Ubuntu 22.04 LTS
RAM	32 GB DDR4
.NET version	7
Configuration	Release

Рис. 3: Конфигурационный стенд

5. Эксперимент

Характеристики окружения, в котором проводилось тестирование, представлено на рисунке 3.

Для тестирования был создан регион памяти для массивов, в элементах которого хранятся ссылки на строки. В контексте символьного исполнения довольно проблематично исследовать изменения в производительности низкоуровневых операций с памятью из-за следующих проблем:

1. помимо операций с памятью, символьный интерпретатор делает много других операций, на фоне которых может быть непросто выделить значимость модификаций;
2. в течение нескольких запусков одного теста решатель может выдавать разные модели, которые будут влиять на выбор состояния для исследования;
3. в один момент времени обрабатывается только одно состояние, в котором находится регион, а также состояния соответствуют разным ветвям исполнения программы с разной длительностью, и эти ветви нужно соотносить между собой.

Были созданы 100 символьных адресов, 100 символьных индексов, 100 конкретных индексов. В течение 3 минут в регион по случайному адресу и случайному индексу производились записи конкретных и символьных строк, между записями производились чтения по случайным ключам. Как можно видеть на рисунке 4, где сравнивается количество операций с моделями памяти, где реализовано только разделённое чтение и

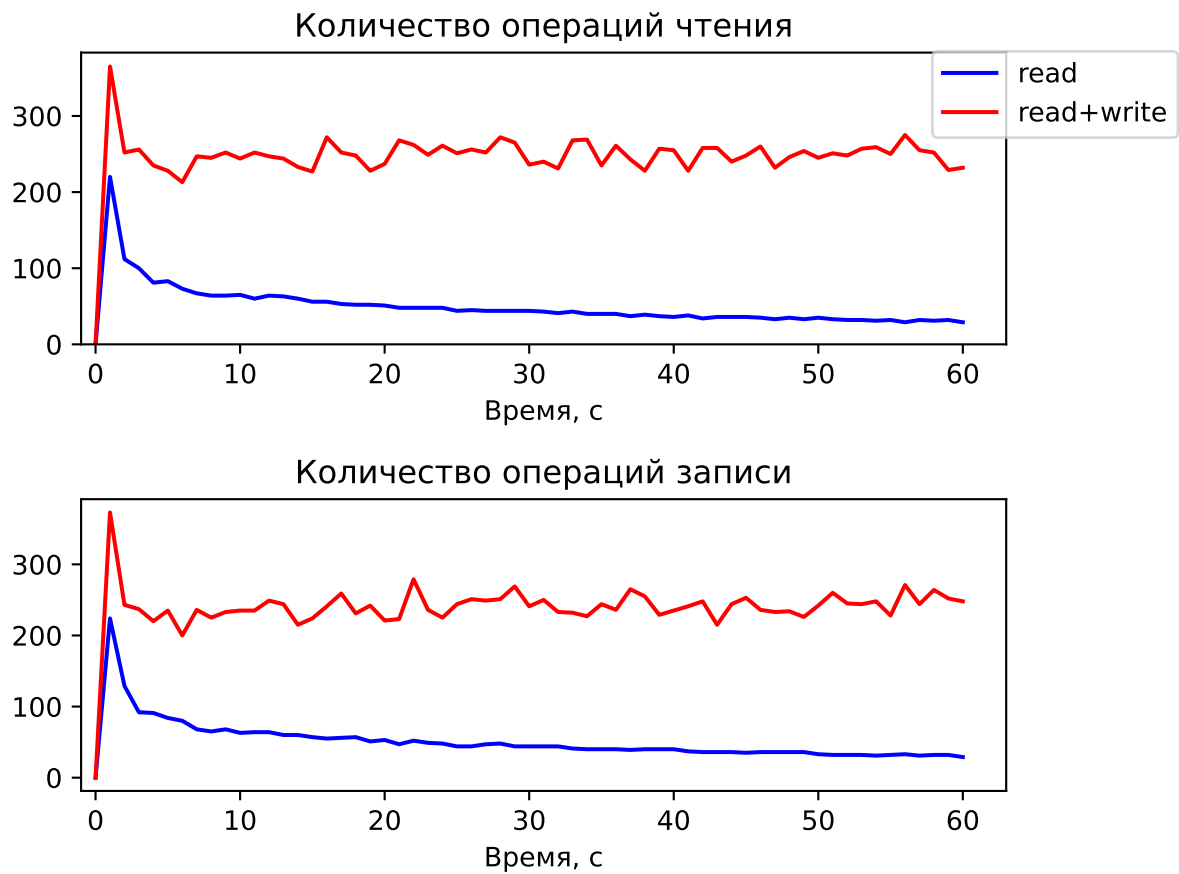


Рис. 4: Операции с регионом памяти

разделённое чтение с разделённой записью, скорость чтения и записи слабо зависит от размера региона.

Заключение

По итогу проведённой работы

- изучена литература по моделированию символьной памяти;
- разработаны алгоритмы разделённого чтения и записи для символьной памяти;
- реализованы алгоритмы разделённого чтения и записи для символьной памяти;
- проведено сравнение полученного решения с существующей моделью символьной памяти $V\#$.

Реализованный алгоритм разделённого чтения позволил улучшить механизм обмена информацией между конкретной памятью и символьной, тем самым повысив корректность символьного исполнения интерпретатором $V\#$. Ошибки, связанные с чтением конкретных объектов из пустых регионов в памяти (см листинг 3.2) больше не воспроизводятся: набор тестов на взаимодействие между символьной и конкретной памятью был успешно пройден.

Более того, корректность была повышена и при вызове виртуальных методов (`GetHashCode`, `Equals` и т.д.) на `Ite`-термах с конкретными ссылочными объектами, которые стали часто встречаться при исследовании. Теперь нужный метод вызывается 1 раз у определённого объекта в конкретной памяти, вместо того чтобы вызывать метод на всех кандидатах типов, подходящих символьной константе. На данный момент нельзя привести пример кода с виртуальными методами, который теперь корректно исследуется, из-за ошибки пропэгирования защитного условия компоненты `Ite`-терма в условие пути состояния, однако ошибка уже локализован и является гораздо более простой, чем та, которой решалась алгоритмами разделённого чтения и записи.

Изменения представлены двумя `Pull request`-ами и доступны по ссылкам: <https://github.com/VSharp-team/VSharp/pull/291> и <https://github.com/VSharp-team/VSharp/pull/295>.

Список литературы

- [1] URL: <https://github.com/UnitTestBot/usvm>.
- [2] Kapus Timotej, Cadar Cristian. [A segmented memory model for symbolic execution](#) // Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. — ESEC/FSE 2019. — New York, NY, USA : Association for Computing Machinery, 2019. — P. 774–784. — URL: <https://doi.org/10.1145/3338906.3338936>.
- [3] Krishnamoorthy Saparya, Hsiao Michael S., Lingappan Loganathan. [Tackling the Path Explosion Problem in Symbolic Execution-Driven Test Generation for Programs](#) // 2010 19th IEEE Asian Test Symposium. — 2010. — P. 59–64.
- [4] MultiSE: Multi-path Symbolic Execution using Value Summaries. — 2015. — URL: <https://people.eecs.berkeley.edu/~ksen/papers/multise.pdf> (дата обращения: 7 декабря 2023 г.).
- [5] A Survey of Symbolic Execution Techniques / Roberto Baldoni, Emilio Coppa, Daniele Cono D’elia et al. // [ACM Comput. Surv.](#) — 2018. — may. — Vol. 51, no. 3. — 39 p. — URL: <https://doi.org/10.1145/3182657>.
- [6] Volkov A. Mandrykin M. Predicate Abstractions Memory Modeling Method with Separation into Disjoint Regions. — URL: <https://ispranproceedings.elpub.ru/jour/article/view/322/168> (дата обращения: 7 декабря 2023 г.).