Санкт-Петербургский государственный университет

Системное программирование

Группа 21.Б07-мм

Оптимизация модели символьной памяти в проекте V#

ПОЖАРСКИЙ Роман Николаевич

Отчёт по учебной практике
в форме «Производственное задание»

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

В настоящее время 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 применять инструменты генерации тестов на основе анализа исходного кода. Несмотря на то, невозможно полностью автоматизировать процесс тестирования, обойдясь только тестами, сгенерированными подобными инструментами, можно существенно сэкономить ресурсы при тестировании далеко не тривиальных сценариев работы системы. Одной из популярных техник анализа программ является символьное исполнение.

Работе программы при конкретных входных данных соответствует один определённый путь, поскольку предопределены выборы в условных операторах. При исследовании исходного кода для генерациии тестового покрытия одной из главных целей является достижение наибольшего покрытия исходного кода. Символьное исполнение позволяет исполнить код с символьными константами, подставленными вместо входных данных. Таким образом, для анализа программы не обязательно предоставлять конкретные входные данные. В символьном исполнении возникает проблема выполнимости формул логики первого порядка, в которые кодируются ограничения, возникающие из условных операторов в программе, и она ложится на SMT-решатели (Satisfiability Modulo Theories). Если при исследовании встречается условный оператор, можно исследовать все ветви, которые соответствуют выполнимым условиям, то есть условиям, для которых существуют конкретные данные, на которых они реализуются. Отсюда же вытекает одна из главных проблем символьного исполнения — экспоненциальный рост возможных сценариев работы программы (англ. path explosion), поэтому при символьном исполнении активно применяются различные эвристики и

оптимизации, которые на практике помогают гораздо быстрее анализировать программы.

Символьная виртуальная машина позволяет символьно интерпретировать инструкции программы. V# — это платформа для генерации тестового покрытия программ на платформе .NET, .NET Core, .NET Framework, которая содержит в себе символьную виртуальную машину, спроектированную для исполнения CIL (Common Intermediate Language) с учётом особенностей платформы .NET.

Как и любая другая виртуальная машина, символьная виртуальная машина обладает определённой моделью памяти, которая определяет, как выполняются операции чтения, записи и выделения. В данной работе будет описана серия оптимизаций, реализованных для модели памяти в проекте V#.

1. Постановка задачи

Целью работы является оптимизация модели памяти в проекте V#. Для её выполнения были поставлены следующие задачи:

1. изучить литературу по моделированию символьной памяти;
2. разработать алгоритмы оптимизации символьной памяти;
3. реализовать алгоритмы оптимизации символьной памяти;
4. протестировать влияние проведённых оптимизаций на крупных проектах на платформе .NET.

2. Обзор

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 и все операции производятся над ними. Для проверки выполнимости условий, в которых участвуют символьные константы, используются SMT-решатели.

Решатели поддерживают различные теории, например, теория линейной арифметики, теория массивов, теория битовых векторов, и в эти теории кодируются ограничения из условных операторов в программе. Стоит 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, удовлетворяющие ограничениям, или UNSAT, это означает, что не существует таких входных данных для исследуемой функции, при которых управление дойдёт до блока кода, защищённого этим условием.

Существуют разные подходы к тому, когда вызывать решатель для проверки выполнимости условия. Самым популярным 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 работы 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 этой стадии;

3. символьная память σ, которая определяет, какие значения хранятся в символьных константах

Листинг 2.1: функция модуля

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

Разберём изменение этих параметров при исполнении функции подсчёта модуля целого числа, код которой представлен на листинге 2.1.

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

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

3. теперь встречается ветвление, при котором состояние разбивается на два других, соответствующих веткам `if` и `then`, поскольку и условие в строке 2, и его отрицание оказываются выполнимыми после обращения к решателю

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

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 (стек вызова, стек вычислений, выделенные объекты), то бесконтрольное копирование состояний может привести к огромным затратам по памяти, поэтому в символьном исполнении применяют слияние состояний. Финальные состояния после исследования функции модуля можно представить следующим образом

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

где ITE обозначает `if-then-else` выражение, принимающее 3 аргумента: *condition, expr1, expr2* и возвращающее *expr1*, если выполняется *condition*, иначе возвращает *expr2*. У ITE выражения есть обобщённая конструкция $(g_1, v_1), \ldots, (g_n, v_n)$, которая возвращает выражение v_i при выполнении условия g_i. В проекте V# она носит название Union. 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 пременные для кодирвоания ограничений;
- можно выполнять операции с памятью без вызова решателя.

Более подробно преимущества описанного метода слияния состояния описаны в статье MultiSE[1].

2.2. Модель памяти с регионами

Основной задачей символьной памяти является «запоминание» всех изменений, произведённых над памятью. В V# используется модель памяти с регионами. Регионом считается множество объектов, которые могут иметь одинаковый адрес в памяти. 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 *aliases*. При этом имеется гарантия, что изменение объектов в одном регионе не модифицирует объекты в другиъ регионах. Данная модель позволяет кластеризовать объекты в памяти программы, осуществлять быстрый доступ к ним, а также легко учитывать случаи, когда две разные символьные константы, которые принимает на вход функция, являются одним и тем же объектом. На листинге 2.2 приведён пример такой программы, где элементы массивов a и b могут подменять друг друга. При символьном исполнии нужно учитывать подобные сценарии.

Данная модель памяти подробнее описана в работах Мандрыкина[2].

Листинг 2.2: Пример программы, где один и тот же объект передается в качестве нескольких параметров

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

Среди всех изменений можно выделять наиболее значимые, чтобы использовать во время символьного исполнения. Например, можно сократить количество обращений к решателю, отмечая недостижимые ветки только благодаря некоторым данным о состоянии символьной памяти.

2.3. Используемые инструменты

Основная логика фреймворка написана на функциональном языке программирования F# для платформы .NET. Функции, которые исследуются символьным интерпретатором, написаны на C#.

Влияние проведённых оптимизаций планируется измерять на крупных открытых проектах Unity, Powershell, JetBrains.Lifetimes.

3. Алгоритмы оптимизации

Основной задачей в работе является реализация алгоритма разделённой записи. Подготовительными этапами для неё являются реализация условия соответствия ключей и реализация алгоритма разделённого чтения. В разделе 3.1 будет более подробно описана модель памяти в проекте, что позволит показать пользу разделения результатов чтения и записи. Модифицированный алгоритм чтения и подготовительные этапы для его реализации описаны в разделе 3.2.

3.1. Модель памяти V#

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.

Все операции с памятью при символьном исполнении необходимо моделировать с учётом подобных особенностей платформы .NET. Данная работа нацелена на оптимизацию работы символьного исполнения с динамической памятью.

Состояние — объект, характеризующий стадию символьного исполнения, в котором хранится вся необходимая информация (символьная память, условие пути, стек вызовов и т.д.). Части составных объектов (элементы массивов, поля классов) также хранятся в состоянии в специальных словарях. Словари имеются для

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

Ключами словарей являются идентификаторы, например, для полей классов это имя, тип и класс, в котором объявлено поле, а значениями — *регионы памяти* (англ. memory region). В регионе памяти хранятся объекты, которые могут друг друга подменять (aliases). В рамках данной работы можно отождествить регион памяти с *деревом обновлений*, которое определяется так

$$\begin{aligned} \langle\text{update tree}\rangle &::= \texttt{Dictionary<}\langle\text{region}\rangle, (\langle\text{update tree key}\rangle\texttt{,}\langle\text{update tree}\rangle)\texttt{>} \\ &\quad \mid \texttt{NilTree} \\ \langle\text{update tree key}\rangle &::= \langle\text{key}\rangle\texttt{,value} \end{aligned}$$

Дерево обновлений индексируется регионами и хранит в узлах ключ, по которому делалась запись, и значение записи. Так, операции $a[5] = 5$ соответствует пара ключ-значение $a[5], 5$. Понятие ключа будет формализовано далее.

Дерево обновлений поддерживает 2 инварианта:

1. регионы, индексирующие поддеревья конкретного дерева, не пересекаются;
2. все регионы, индексирующие узлы в поддереве, являются подмножеством региона дерева.

При операции чтения по определённому ключу из региона памяти происходит обход соответствующего дерева обновлений, из которого выбирается поддерево, индексирующееся регионами, которые пересекаются с регионом ключа, по которому происходит чтение. То есть операция чтения локализуется и возвращает множество всех возможных значений, которые могли бы быть прочитаны.

Отношение наследования в дереве регионов означает, что запись предка перекрывает записи дочерних узлов. Записи дерева, которые лежат выше, имеют больший приоритет. Но в связи с особенностями символьного исполнения и реализации дерева обновлений в V#, запись, лежащая выше по иерархии, не всегда перезаписывает результаты записей ниже (см. 2 инвариант дерева обновлений), поэтому нужно поддерживать всю историю записей.

Осталось формализовать понятие *региона*. В символьном исполнении, как и при конкретном, создаются объекты, которые имеют некоторый адрес в памяти. В конкретном случае это реальный адрес в памяти компьютера, например, `0x0000`. При символьном исполнении же удобно идентифицировать объекты временем их создания. Если 1 считать временем начала анализа функции, то её параметры, символьные константы, очевидно, уже были созданы, и имеют адрес меньше 1. У конкретных объектов, которые аллоцируются в процессе исполнения, адреса больше или равны 1 (0 считается *null*-объектом, значением по умолчанию всех ссылочных типов).

Если в качестве аргумента анализируемой функции приходит ссылочный тип, то адрес этого объекта является символьным. В контексте символьного исполнения нужно учесть все возможные значения параметров, поэтому считается, что аргумент мог быть создан когда угодно, и его адрес представляет множество значений из интервала $(-\infty, 0)$. На практике хватает диапазона, которые предоставляют целые числа, занимающие в памяти 4 байта, то есть множество адресов аргумента это $[-2147483648, 0)$.

Для того, чтобы идентифицировать объекты внутри одного региона памяти, необходимы *ключи*, которые представляют из себя адрес

(актуально для всех объектов) и, возможно, индексы (актуально для массивов). Каждый ключ имеет свой *регион*, который определяет множество возможных значений его составных частей. Более формальное определение региона:

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

Регион *interval* необходим для представления возможных значений адресов объектов. Этот регион есть у всех ключей, выделенных на куче. Структура данных *points* представляет множество возможных значений индексов массива. Она представляет из себя либо конечное множество точек, либо дополнение к бесконечному (в этом случае считается, что индекс не может принимать хранящиеся значения). Декартово произведение n раз, где n — размерность массива, берётся в произведение с интервалом для представления возможных значений ключа многомерного массива. Под $\mathbb{Z}$ подразумевается множество возможных значений 4-байтного `int`. На регионах естественным образом определяюстя такие теоретико-множественные операции, как объединение, пересечение, вычитание.

Разберём поведение дерева обновлений при выполнения кода на листинге 3.1.

Листинг 3.1: Демонстрационная функция

```
public object Foo(object[] a, int i, int j) {
  a[i] = 5;
  a[1] = 1;
```

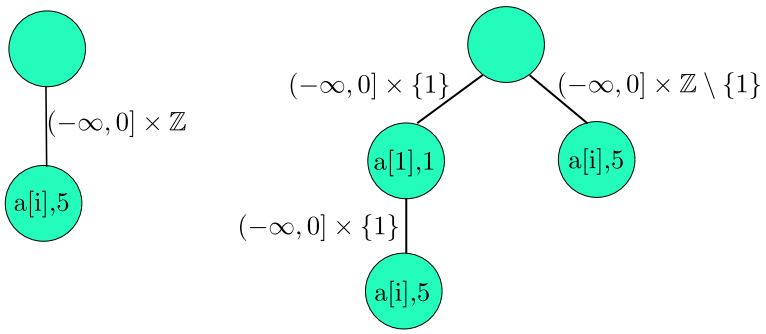

Рис. 1: Дерево обновлений после первой операции записи и после второй

```
return a[j];}
```

Первой записи $a[i] = 5$ соответствует пара $(key, value) = (a[i], 5)$. Регион ключа $a[i]$ это $(-\infty, 0) \times \mathbb{Z}$, так как массив a и индекс i, по которому делается запись, символьные. Поскольку дерево было пустое, добавляем в него новую запись по соответствующему региону. Далее делается запись $a[1] = 1$. Для этого из дерева обновлений выбирается поддерево, индекусирующееся регионами, пересекающимися с регионом ключа $a[1]$, а именно $r_1 = (-\infty, 0) \times \{1\}$. Уже имеющееся поддерево, которое индексирует регион $(-\infty, 0) \times \mathbb{Z}$, разбивается на два поддерева t_1 и t_2: t_1 индексируется пересекающимся с $a[1]$ регионом r_1 (в данном случае совпадающим), а t_2 — не пересекающимся с $a[1]$ регионом $r_2 = (-\infty, 0) \times \mathbb{Z} \setminus \{1\}$. Новая запись перекрывает записи из t_1, и не может перезаписывать записи из t_2. Обновлённое дерево обновлений будет выглядеть так:

$$\{r_1 \mapsto ((a[1], 1), t_1), r_2 \mapsto ((a[i], 5), \texttt{NilTree})\}$$

3.2. Модифицированный алгоритм чтения

Если после исполнения описанной функции совершить операцию чтения по $a[1]$, то мы гарантированно прочитаем значение 1. Но если читать по ключу, которого нет в дереве, например, $a[j]$, где j — тоже символьная константа, то операция чтения возвращает символьную константу, сохраняя имеющееся дерево обновлений внутри этой константы

для будущего запроса к решателю. Такой результат чтения не теряет информации о программе и является корректным. Однако такая абстракция не позволяет выполнять оптимизиации конкретных случаев чтения во время символьного исполнения. Если сразу после исполнения этой функции будет идти условный оператор $if(j == 1 \wedge a[j] \neq 1)$, то выяснение выполнимости этого условия ложится на SMT-решатель, хотя эту ветку легко можно отметить недостижимой. Отсюда возникает идея разделённого чтения и записи. На листинге 3.2 представлен псевдокод алгоритма разделения результата чтения.

Листинг 3.2: Псевдокод модифицированного алгоритма чтения

```
func splitRead(tree, readKey, predicate, exploringPath):
 symblicCaseTree = new()
 union = new()
 notMatchPathCond = Conjuction(exploringPath.map(writeKey => not MatchCondition(readKey, writeKey, writeRegion)))
 foreach (reg, (utKey, subtree)) in tree:
   matchCond = MatchCondition(readKey, utKey.key, reg)
   g = (Conjuction(notMatchPathCond, matchCond|)
   recUnion, recSymbolicCaseTree = splitRead(subtree, readKey, predicate, exploringPath + [subtree.key])
   union.add(recUnion)
   if (predicate(utKey) and g != false)
     v = utKey.value
     connectSubtree(symbolicCaseTree, recSymbolicCaseTree)
   union.add((g,v))
   else if (g != false)
     connectSubtreeByNode(symbolicCaseTree, reg, utKey, recSymbolicCaseTree)
   else
     connectSubtree(symbolicCaseTree, recSymbolicCaseTree)
 return union, symbolicCaseTree

func connectSubtree(tree, subtree):
 foreach (reg, (utKey, subtree)) in subtree:
   tree.Add(reg, (utKey, subtree))

func connectSubtreeByNode(tree, nodeReg, nodeUtKey, subtree):
 newUpdateTree = new()
 foreach (reg, (utKey, subtree)) in subtree:
   newUpdateTree.Add(reg, (utKey, subtree))
 tree.Add(nodeReg, (nodeUtKey, newUpdateTree))
```

Если ключ, по которому читают, отсутствует в дереве обновлений, можно выбрать из дерева некоторые записи по определённому предикату, сделать из них Union $(g_1, v_1), \ldots, (g_n, v_n)$, где g_i — условие, при котором результатом чтения должно быть значение v_i. Записи, не подошедшие предикату как и ранее будут оставаться символьной константой с сохранённым для решателя деревом обновлений. Возвращясь к

примеру с условным оператором $if(j == 1 \wedge a[j] \neq 1)$, если выбрать предикат «запись имеет конкретное значение», то в результате чтения $a[j]$ Union результата чтения упростится до единственного возможного значения «1», что является несовместным условием с $a[j] \neq 1$, поэтому эту ветку можно не исследовать, даже не выясняя выполнимость условия через SMT-решатель.

3.2.1. Условие соответствия ключей

Возникает необходимость составлять Union из записей, подошедших некоторому предикату. Для этого необходимо на уровне логических выражений с термами уметь представлять условия, по которому чтение по требуему ключу происходит из ключей, по которым ранее делалась запись, иными словами, условие соответствия ключей.

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

1. $k_1 = k_2$;
2. компоненты k_1 должны принадлежать r_2 (или компоненты k_2 должны принадлежать r_2)

Чтобы убедиться в том, что второе условие необходимо, достаточно вернуться к рассмотренному ранее примеру дерева обновлений и поставить вопрос: «Когда произвольный ключ $b[k]$ будет соответстовать ключу $a[i]$ с записанным значением 5?» Во-первых, должно выполняться равенство компонент ключей $b = a$ (b и a — один и тот же массив) и $k = i$ (равенство индексов). Но этого недостаточно, поскольку последующие записи в теории могли переписать значение по адресу $a[i]$. Если $k = 1$, то будет прочитан ключ $a[1]$, которому соответствует единица. Отсюда возникает второе требование $k \neq 1$.

Для всех ключей была реализована функция MatchCondition (в псевдокоде алгоритма MC) соответствия другому ключу, возвращающая логическое выражение. Для рассмотренного примера с $b[k]$ и $a[i]$ это условие будет выглядеть $a = b \wedge k = i \wedge a \in (-\infty, 0) \wedge k \neq 1$. Так как

решатель кодирует термы, в том числе и выражения, было несложно переиспользовать написанное условие соответствия и при кодировании условий в теории SMT-решателя.

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

$$g_i = MC(k, key_i, r_i) \wedge \bigwedge_{m=1}^{j_i} \neg MC(r, c_{im}.key, r_{im})$$

3.2.3. Использование модифицированного алгоритма чтения

Внедрение нового алгоритма чтения позволило обнаружить в системе места, где не ожидается Union, реализовать его поддержку и исправить некоторые другие ошибки, связанные, например, с неполным упрощением логических выражений.

Как и ожидалось, время работы символьного исполнения в некоторых ситуациях ощутимо возросло, поскольку при каждом чтении обходить дерево обновлений может быть довольно дорого. Отсюда возникает идея значения, удовлетворяющие предикату, который всегда будет фиксированным, перетаскивать в корень дерева, и поддерживать этот инвариант при каждой записи. Таким образом, при чтении не нужно будет обходить всё дерево.

Реализация алгоритма разделённой записи запланирована на следующий семестр.

4. Эксперимент

В следующем семестре планируется провести влияние проведённых оптимизаций на больших проектах, написанных при помощи платформы .NET (Unity, Powershell, JetBrains.Lifetimes).

5. Реализация

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 было реализовано и протестировано. Ожидается, что `pull request` поступит в основную ветку после реализации модифицированного алгоритма записи. Ознакомиться с кодом можно по ссылке https://github.com/VSharp-team/VSharp/pull/295

Заключение

По итогу первого семестра

- изучена литература по моделированию символьной памяти;
- частично разработаны алгоритмы по оптимизации символьной памяти в V#;
- частично реализованы алгоритмы по оптимизации символьной памяти в V#;

Код доступен по следующим ссылкам: https://github.com/VSharp-team/VSharp/pull/291 и https://github.com/VSharp-team/VSharp/pull/295.

Список литературы

[1] 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.).

[2] 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.).