Санкт-Петербургский государственный университет
Кафедра системного программирования

Автоматический синтез объектов для символьного исполнения

Максим Алексеевич Паршин

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

Санкт-Петербург
2023

Введение

```
1  void Foo(int x, int y, int z)
2  {
3          int a = 0;
4
5          if (x < 42)
6          {
7                  a = y + z;
8          }
9
10         if (a > 73)
11         {
12                 throw new Exception();
13         }
14 }
```

- Символьное исполнение — исполнение кода не на конкретных значениях входных данных, а на символьных переменных
- Для каждого пути исполнения — условие на символьные переменные, при выполнении которого он достигается
- Например, функция `Foo` выбрасывает исключение в 12 строке, если $\pi = x < 42 \ \wedge \ y + z > 73$
- Используется для автоматической генерации тестов и поиска ошибок
- V# — символьная машина для .NET[a]

[a] https://github.com/VSharp-team/VSharp

Введение

```
1  public class StaticsForType<T> where T:class
2  {
3      private readonly List<T> myList = new
           List<T>();
4      private event Action? Changed;
5      ...
6      public void ForEachValue(Action action)
7      {
8        lock (myList)
9        {
10         Changed += action;
11       }
12       action();
13     }
14     ...
15 }
```

- Другой пример, из библиотеки JetBrains.Lifetimes[a]
- V# находит ошибку: при *myList* == *null* метод `ForEachValue` выбрасывает `ArgumentNullException` в строке 8
- Такой объект `StaticsForType` успешно создаётся рефлексией
- Однако *myList* не может быть равен *null* при реальном использовании данного класса
- Необходимо создавать объекты с заданными свойствами так же, как это делал бы пользователь

[a] https://github.com/JetBrains/rd

Постановка задачи

Целью данной работы является разработка алгоритма автоматического синтеза объектов на основе механизма символьного исполнения

Задачи:

- Провести обзор методов синтеза объектов, используемых в различных инструментах генерации тестов
- Разработать алгоритм синтеза объектов, основанный на прямом символьном исполнении
- Реализовать разработанный алгоритм в символьной виртуальной машине V#
- Провести эксперименты для определения эффективности реализованного алгоритма
- Разработать алгоритм синтеза объектов, основанный на обратном символьном исполнении

Существующие подходы к синтезу объектов

- *Automated Testing of Classes, 2000*
  - Генерация тестов, реализующих определённые def-use пары
  - Построение последовательности вызовов методов в обратном порядке
  - Методы последовательности исполняются символьно
  - Отсутствуют эксперименты, используются устаревшие инструменты
- *Symstra: A Framework for Generating Object-Oriented Unit Tests using Symbolic Execution, 2004*
  - Перебираются всевозможные последовательности методов и исполняются символьно с переменными примитивных типов
  - Состояния с изоморфными графами объектов объединяются
  - Отсутствует возможность создания вложенных объектов
- *Synthesizing Method Sequences for High-Coverage Testing, 2011*
  - Основа — символьная машина *Pex* для .NET
  - Последовательность методов генерируется для определённого пути исполнения
  - Комбинация статического анализа и динамического символьного исполнения
  - Тестирование на крупных проектах

Существующие подходы к синтезу объектов

- *Efficient Synthesis of Method Call Sequences for Test Generation and Bounded Verification, 2022*
  - Подход, схожий с *Symstra*: построение полного графа переходов между различными состояниями объектов
  - В процессе строится последовательность вызовов, методы исполняются символьно
  - Одно состояние — один тест
- Подходы, основанные на генетических алгоритмах
  - *Improving Structural Testing of Object-Oriented Programs via Integrating Evolutionary Testing and Symbolic Execution, 2008*
  - *Graph-Based Seed Object Synthesis for Search-Based Unit Testing, 2021*
- Инструменты генерации тестов, основанные на больших языковых моделях
  - *An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation (TestPilot), 2023*

Синтез объектов с помощью прямого символьного исполнения

- Задача — для заданного состояния символьного исполнения s_{target} с условием пути pc сгенерировать последовательность вызовов методов из публичного API, которые при исполнении создадут объекты, удовлетворяющие pc
- Последовательность вызовов инициализирующих методов исполняется символьно, получаем состояние s_{seq_n}
- Значения аргументов примитивных типов остаются символьными
- По s_{seq_n} и $s_{target}.pc$ (которое является постусловием) вычисляется *слабейшее предусловие*
- Если слабейшее предусловие выполнимо, то задача решена, при этом конкретные значения из модели (все примитивные) можно подставить в качестве аргументов

Синтез объектов с помощью прямого символьного исполнения

```
function GenerateSequence(state)
    sequences ←GetInitialSequence(state)
    while !sequences.isEmpty do
        currSeq ← sequences.Pick()
        for all newSeq ∈ MakeStep(currSeq) do
            if newSeq.isFinished then
                wp ←WP(newSeq.state, state.pc)
                if CheckSat(wp) then
                    return currSeq
            else
                sequences.Add(newSeq)
    return none
```

Синтез объектов с помощью прямого символьного исполнения

```
function MakeStep(sequence)
    newSeqs ← []
    if sequence.isInMethod then
        for all newState ∈ SVMStep(sequence.state) do
            newSeq ← sequence.Copy()
            newSeq.state ← newState
            newSeqs.Add(newSeq)
    else
        upcoming ← sequence.upcoming.Peek()
        if !upcoming.holes.isEmpty then
            for all hole ∈ upcoming.holes do
                for all newSeq ∈ FillHole(sequence, hole) do
                    newSeqs.Add(newSeq)
        else
            sequence.upcoming.Pop()
            newSeq ← sequence.Copy()
            newSeq.elements.Append(upcoming)
            newSeq.state ← SVMCall(sequence.state, upcoming)
            if upcoming.method = sequence.targetMethod then
                newSeq.finished ← true
            newSeqs.Add(newSeq)
    return newSeqs
```

Синтез объектов с помощью обратного символьного исполнения

- У рассмотренного алгоритма есть недостатки
  - Можно долго рассматривать последовательности с заведомо неподходящими префиксами
  - Например, для массивов заранее неизвестно, сколько объектов-элементов нужно создать
- Решение — строить последовательность с конца
- Методы, из которых может быть составлена последовательность, символьно исполняется изолированно, получаются состояния s_{m_n}
- По s_{m_n} и предусловию текущего суффикса вычисляется слабейшее предусловие
- Если слабейшее предусловие выполнимо, то метод можно добавить в начало суффикса последовательности
- Если слабейшее предусловие невыполнимо, то можно проанализировать конфликт (рассмотрев unsat-ядро)

Синтез объектов с помощью обратного символьного исполнения

```
function GenerateSequence(state)
    sequences ←GetInitialSequence(state)                    ▷ precondition = state.pc
    while !sequences.isEmpty do
        currSeq ← sequences.Pick()
        for all element ∈ GetElements(currSeq) do
            wp ←WP(element.state, curSeq.preconditon)
            checkResult ← CheckSat(wp)
            if checkResult = SAT then
                newSeq ← sequence.Copy()
                newSeq.precondition ← wp
                newSeq.elements.Prepend(element)
                if !wp.hasNonPrimitiveTerms then
                    return currSeq
                sequences.Add(newSeq)
            else
                AnalyzeConflict(checkResult, currSeq)
    return none
```

Эксперименты (прямое символьное исполнение)

- Пространство поиска ограничено конструкторами и методами `Set` свойств
- Синтетические тесты: методы, принимающие на вход объекты с вложенной структурой

Количество методов	20
Общее количество сгенерированных тестов	71
Количество тестов, для которых последовательность может быть сгенерирована	70
Количество тестов, для которых последовательность была сгенерирована	70
Максимальная длина последовательности	3

Таблица: Результаты экспериментов

Результаты

В ходе данной работы были получены следующие результаты

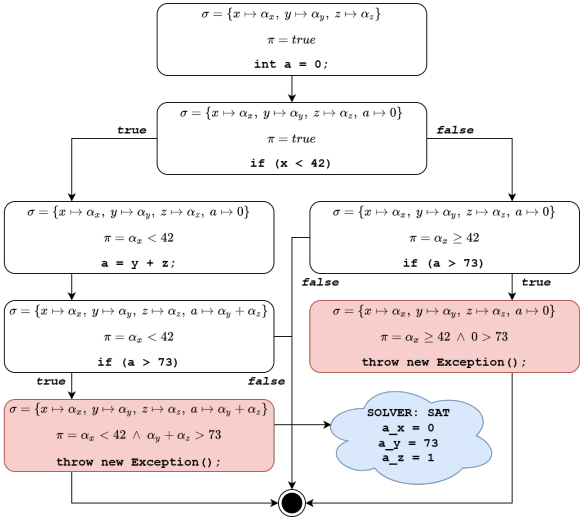
- Проведён обзор методов синтеза объектов, используемых в различных инструментах генерации тестов
- Разработан алгоритм синтеза объектов, основанный на прямом символьном исполнении
- Разработанный алгоритм реализован в символьной виртуальной машине V#
- Проведены эксперименты для определения эффективности реализованного алгоритма
- Разработан алгоритм синтеза объектов, основанный на обратном символьном исполнении

В ходе дальнейшей работы планируется

- Реализовать прототип алгоритма синтеза объектов, совмещающего технологии больших языковых моделей и обратного символьного исполнения в V#
- Провести эксперименты для определения эффективности реализованного алгоритма

Пример символьного исполнения

```
1  void Foo(int x, int y, int z)
2  {
3          int a = 0;
4
5          if (x < 42)
6          {
7                  a = y + z;
8          }
9
10         if (a > 73)
11         {
12                 throw new Exception();
13         }
14 }
```



Особенности реализации

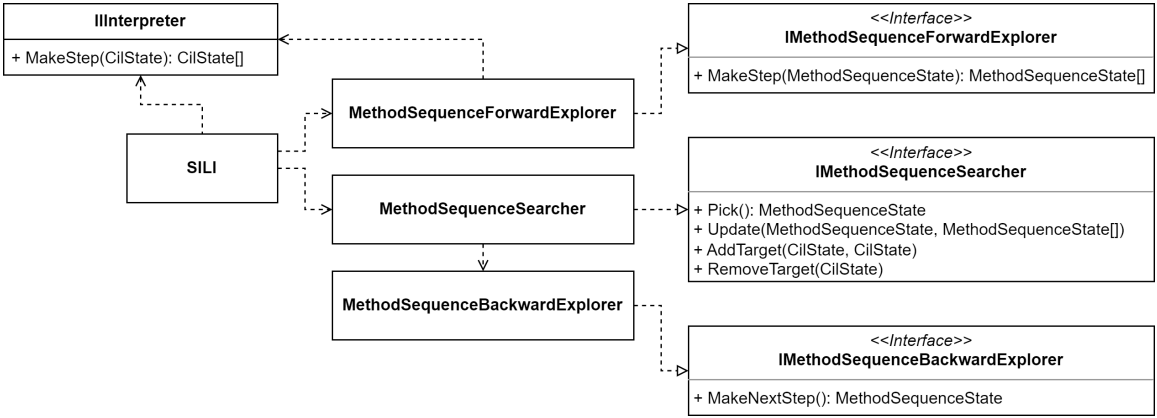


Рис.: Диаграмма реализованных классов

Пример работы алгоритма

```
1  public class Bank
2  {
3      public void Withdraw(Account account,
           int sum, Currency currency)
4      {
5          if (sum <= 0 || account.Currency !=
               currency)
6          {
7              throw new Exception();
8          }
9
10         if (account.Sum < sum) return;
11
12         account.Sum -= sum;
13     }
14 }
```

```
1  public enum Currency
2  {
3      RUB,
4      USD
5  }
6
7  public class Account
8  {
9      public int Sum { get; set; }
10
11     public Currency Currency { get; }
12
13     public Account(Currency currency)
14     {
15         Currency = currency;
16     }
17 }
```

Пример работы алгоритма (продолжение)