Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.М07-мм

Паршин Максим Алексеевич

Автоматический синтез объектов для символьного исполнения

Отчёт по производственной практике
в форме «Решение»

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

Санкт-Петербург
2023

Оглавление

Введение

Практически каждая достаточно сложная, чтобы быть полезной, программа содержит ошибки. Если у программного продукта миллионы пользователей, ошибки и связанные с ними уязвимости могут стоить слишком дорого[1] (во всех смыслах). Поиск ошибок — нетривиальная задача: как правило, в некорректное состояние программа попадает только в определённых условиях, например, на входных данных определённого вида. Определить, а тем более случайным образом подобрать, такие данные тестировщику удаётся не всегда.

Неудивительно, что исследования в области автоматического тестирования программ ведутся достаточно давно и достаточно успешно. В основе многих инструментов для генерации тестов и поиска уязвимостей лежат различные техники статического анализа кода. Символьное исполнение — одна из таких техник. Отличительная (и наиболее заманчивая) особенность символьного исполнения — это теоретически полное покрытие кода. Исполнение программы не на конкретных, а на переменных значениях позволяет для каждой возможной ветви получить либо входные данные, на которых эта ветвь достигается, либо заключить, что ветвь не достигается вообще. Тем не менее, создать эффективный инструмент, основанный на символьном исполнении, непросто: вычислительная сложность задачи символьного исполнения слишком велика, чтобы её можно было решить без использования различных оптимизаций.

Вычислительная сложность — не единственная проблема, с которой приходится сталкиваться разработчикам символьной виртуальной машины. Для современного разработчика работа программы — это уже не просто операции над «нулями» и «единицами», а взаимодействие объектов. Важна семантика, все внутренние инварианты объектов обязательно должны выполняться. Для того, чтобы «убедить» в этом символьную машину, требуются дополнительные усилия.

Рассмотрим реальный пример кода на C# из библиотеки

[1] https://heartbleed.com/, дата обращения — 31.05.2023

JetBrains.Lifetimes[2] (листинг 1).

Листинг 1: Класс StaticsForType из библиотеки JetBrains.Lifetimes

```
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 }
```

Символьная виртуальная машина V#[3] находит две ошибки в методе *ForEachValue*.

- Во-первых, метод выбрасывает исключение *NullReferenceException* в 12-ой строке, если в качестве аргумента *action* передан *null*. Пусть данное поведение и сложно назвать серьезной и опасной ошибкой, явная валидация аргумента отсутствует, поэтому такой результат работы символьной машины можно назвать полезным.

- Во-вторых, метод выбрасывает *ArgumentNullException* в восьмой строке, если значение поля *myList* равно *null*. Это действительно так, но у реальных объектов класса *StaticsForType* инициализиро-

[2]https://github.com/JetBrains/rd, дата обращения — 31.05.2023
[3]https://github.com/VSharp-team/VSharp, дата обращения — 31.05.2023

ванное в третьей строке *readonly* поле никак не может быть равно *null*.

Невоспроизводимых ошибок, таких как вторая, может быть найдено гораздо больше, чем реальных. При этом некорректные результаты — это только часть проблемы. Символьная машина, никак не учитывающая внутренние инварианты объектов, может тратить время на исследование путей, в которых эти инварианты заведомо не выполняются. Таким образом, в условиях тестирования реального программного продукта, когда время на исследование может быть жёстко ограничено, поддержание внутренних инвариантов объектов становится критически важной задачей.

1. Постановка задачи

Целью данной работы является разработка алгоритма автоматического синтеза объектов на основе механизма символьного исполнения. Для достижения цели были сформулированы следующие 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 прямом символьном исполнении.

2. Обзор

Можно выделить два способа, с помощью которых генераторы тестов создают объекты с необходимыми свойствами [9].

- *Создание объектов напрямую.* Значения всех полей объекта, в том числе приватных (*private*), выставляются явным образом. Для этого может использоваться, например, рефлексия.

- *Генерация последовательности методов (method sequence).* Определяется последовательность методов из публичного интерфейса класса, которые необходимо вызвать, чтобы получить объект с заданными свойствами.

Следует сказать, что данные подходы к созданию объектов по своей сути соответствуют двум техникам, которые лежат в основе генераторов тестов — символьному исполнению и методам, основанным на эвристических алгоритмах оптимизации (Search-Based Software Testing [6]).

2.1. Символьное исполнение

Техника символьного исполнения [7] заключается в исполнении программы не на конкретных значениях аргументов, а на так называемых символьных переменных. При этом для каждой ветви программы поддерживается условие пути (*path condition*) — формула, содержащая символьные переменные. Чтобы данная ветвь программы исполнилась, конкретные значения аргументов должны удовлетворять этой формуле. При каждом ветвлении состояние исполнителя раздваивается, и условия пути обновляются.

Для того, чтобы получить набор конкретных значений символьных переменных, удовлетворяющих условию пути (*модель*), или сделать заключение о том, что такого набора нет, символьные виртуальные машины используют SMT-решатели [1].

Модели, возвращаемые SMT-решателями — это набор конкретных значений полей объектов. Могут ли объекты с такими значениями быть созданы через публичные интерфейсы? На данный вопрос SMT-решатели сами по себе ответить не способны, как и предоставить последовательность методов, с помощью которой объект можно создать. Создавать же объекты напрямую с помощью рефлексии при символьном исполнении более «естественно».

С другой стороны, символьное исполнение теоретически гарантирует, что даже труднодоступные пути исполнения будут исследованы.

2.2. Search-Based Software Testing

Техники, основанные на эвристических алгоритмах оптимизации, например, на генетических алгоритмах, напротив, в первую очередь генерируют различные последовательности методов. Затем среди получившихся тестов определённым образом выбираются те, которые обеспечивают покрытие редких путей исполнения.

Например, в случае генетического алгоритма формируется «начальная популяция» тестов, которая затем «эволюционирует», приспосабливаясь к заданным условиям. Условия определяются «функцией приспособленности» (*fitness function*). При генерации тестов наиболее приспособленными можно, например, считать те тесты, которые при исполнении посещают новые ветви.

В качестве примера инструмента генерации тестов, в основе которого лежит генетический алгоритм, можно привести *EvoSuite* [2]. Также существует расширение *EvoObj* [3], адаптированное для работы с объектами.

Очевидным плюсом SBST-подходов является то, что они изначально подразумевают генерацию последовательности методов из публичных интерфейсов. С другой стороны, в отличие от символьного исполнения, данные подходы используют эвристические алгоритмы, которые даже теоретически не могут гарантировать посещение труднодоступных локаций.

2.3. Комбинации символьного исполнения и SBST

Многие существующие подходы к генерации объектов в той или иной степени основываются на комбинации символьного исполнения и техник Search-Based Software Testing.

Авторы инструмента *Symstra*, описанного в [8], генерируют всевозможные последовательности вызовов методов класса, заменяя параметры методов примитивных методов символьными переменными. Затем последовательность методов исполняется символьно, чтобы определить возможные конкретные значения этих параметров. Следует отметить, что при данном подходе сам тестируемый метод не исполняется символьно (более того, чётко выделенного тестируемого метода нет) — по сути, это полный перебор всевозможных сценариев использования класса, использующий символьное исполнение для устранения «повторов».

Подход, используемый в *Evacon* [4], явным образом совмещает в себе символьное исполнение и генетический алгоритм. С одной стороны, так же как и в *Symstra*, последовательности методов, синтезированные генетическим алгоритмом, исполняются символьно, чтобы определить наиболее «интересные» конкретные значения параметров. С другой стороны, тесты, сгенерированные при помощи символьного исполнения, используются генетическим алгоритмом при выводе нового поколения.

Тем не менее, данные инструменты, как замечено в [5], используют в качестве основы SBST-подходы, а символьное исполнение лишь дополняет их. *SUSHI* [5], в отличие от них, в первую очередь опирается на символьное исполнение, чтобы вывести условия пути, которые затем конвертируются во входные данные для SBST-оптимизатора. Именно конвертер условий пути, позволяющий объединить символьный исполнитель и оптимизатор, является главным элементом *SUSHI*.

3. Описание решения

В данной главе определяются основные объекты, с которыми работает предложенные алгоритмы, и приводятся их описания. Также рассматриваются аспекты взаимодействия алгоритма с механизмом символьного исполнения.

3.1. Основные понятия

Определим *последовательность методов* как последовательность троек $(M_i, Ret_i, Args_i)$, где:

- M_i — метод;
- Ret_i — множество $\{res_{i_1}, ..., res_{i_{r_i}}\}$ переменных непримитивных типов, в которые M_i возвращает значения. В том случае, если M_i не возвращает значение, то это множество пусто. Если же помимо возвращаемого значения метод имеет out-параметры[4], то это множество содержит более одного элемента.
- $Args_i$ — множество $\{arg_{i_1}, ..., arg_{i_{a_i}}\}$ аргументов параметров, не являющимися out, метода. $Args_i \subset Ret_1 \cup ... \cup Ret_{i-1} \cup Prim \cup \{null\}$, где $Prim$ — множество значений примитивных типов. Таким образом, аргументом метода может быть либо примитивное значение, либо значение, который вернул один из предыдущих методов. $this$, экземпляр класса в случае нестатического метода, рассматривается как первый параметр.

Исполнить последовательность методов — последовательно для каждого элемента $(M_i, Ret_i, Args_i)$ вызвать метод M_i с аргументами $Args_i$ и записать значения в соответствующие переменные Ret_i.

Также введём понятие «скелета» последовательности методов. «Скелет» отличается от обычной последовательности методов тем, что

[4]Язык C# поддерживает out-параметры, которые являются ссылками на переменные (возможно, неинициализированные), в которые метод метод обязан записать значение.

для его элементов аргументы из множества $Args_i$ могут также принимать выделенное значение $hole$, означающее, что соответствующий конкретный объект пока что неизвестен. Значение $hole$ допустимо только для объектов непримитивных типов. Также «скелет» последовательности не может быть исполнен.

Пусть M_{target} — тестируемый метод с параметрами $p_1, ..., p_n$ ($this$ так же, как и ранее, рассматривается как первый параметр). s — состояние символьного исполнения метода M_{target} с условием пути $\pi(p_1, ..., p_n)$ (логической формулой, являющейся ограничением на параметры $p_1, ..., p_n$). Задача, решаемая алгоритмом, заключается в том, чтобы найти такую последовательность методов $\{(M_i, Ret_i, Args_i)\}_{1...l}$, что существуют значения $\{arg_1, ..., arg_n\} \subset Ret_1 \cup ... \cup Ret_l \cup Prim \cup \{null\}$, и после исполнения последовательности выполняется $\pi(arg_1, ..., arg_n)$. Иными словами, последовательность методов создаёт конкретные объекты, при передаче которых в метод в качестве параметров будет достигнута заданная ветвь исполнения.

Предполагается, что все возвращаемые значения и аргументы типизированы, и рассматриваются только те последовательности, где в качестве аргументов методам передаются значения подходящих типов. Также заметим, что для простоты элементы множеств Ret_i рассматриваются одновременно как переменные до исполнения элемента последовательности и как значения, записанные в них после исполнения.

3.2. Алгоритм, основанный на прямом символьном исполнении

Рассмотрим работу предлагаемого алгоритма на примере метода `Withdraw` (M_{target}), моделирующего операцию снятия денег со счёта в банке (листинг 2). Пусть после символьного исполнения метода одно из состояний s завершилось, посетив инструкцию `return` 34-ой строке.

Условие пути данного состояния имеет следующий вид:

$$\pi = (this! = null) \wedge (sum > 0) \wedge (account! = null) \wedge \\ (account.Currency == currency) \wedge (account.Sum >= sum) \quad (1)$$

Как можно заметить, чтобы такое условие выполнилось при конкретном исполнении метода, и он вернул *true*, необходимо:

- создать объекты классов `Bank` и `Account` с помощью их конструкторов;
- вызвать метод `set` для свойства `Sum` объекта `Account`, передав ему некоторое положительное значение суммы;
- вызвать метод `Withdraw` объекта `Bank`, передав ему в качестве параметров `Account`, значение `sum`, не превышающее значения, переданного методу `set` ранее, и значение `Currency`, совпадающее со значением, переданным конструктору `Account`.

Листинг 2: Примитивный код для работы с банковским счётом

```
 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  }
18
19  public class Bank
20  {
21      public bool Withdraw(Account account, int sum,
            Currency currency)
22      {
23          if (sum ≤0 || account.Currency != currency)
24          {
25              throw new InvalidOperationException();
26          }
27
28          if (account.Sum < sum)
29          {
30              return false;
31          }
32
33          account.Sum -= sum;
34          return true;
35      }
36  }
```

Состояние алгоритма — это тройка $(s_{seq}, Current, Upcoming)$, состоящая из следующих элементов.

- $Current$ — построенная на данный момент последовательность методов.
- $Upcoming$ — «скелет» последовательности методов, которая в будущем будет сконкатенирована с $Current$. Представляется в виде стека с операциями *push* и *pop*. В начальном состоянии на стеке лежит элемент $(M_{target}, \emptyset, Args_{target})$.

- s_{seq} — состояние символьного исполнителя после символьного исполнения последовательности $Current$. При символьном исполнении аргументы методов примитивных типов (из множества $Prim$) становятся символьными переменными. Условие пути данного состояния обозначим за π_{seq}.

Возможны следующие переходы между состояниями (Рис. 1).

- Если последним элементом $Current$ является $(M_i, Ret_i, Args_i)$, а s_{seq} в текущий момент находится в одном из методов, сделать шаг символьного исполнения внутри метода. При этом состояние символьного исполнителя может разветвиться, и в таком случае состояние алгоритма также разветвляется. Если же после шага произошёл выход из метода, и были возвращены значения, то они записываются в переменные Ret_i.

- Если s_{seq} в текущий момент не находится внутри метода, а на вершине стека $Upcoming$ лежит элемент $(M_i, Ret_i, Args_i)$, и $hole \notin Args_i$, снять данный элемент со стека, добавить его к $Current$, сделать шаг символьной машины внутрь M_i. Если Ret_i непусто, то выделить соответствующие переменные в состоянии s_{seq}.

- Если s_{seq} в текущий момент не находится внутри метода, а на вершине стека $Upcoming$ лежит элемент $(M_i, Ret_i, Args_i)$, то некоторым образом выбрать новый элемент «скелета» $(M_j, Ret_j, Args_j)$ и положить его на стек $Upcoming$. При этом среди Ret_j могут быть переменные, соответствующие по типам элементам $hole$ внутри $Args_i$ (если они там есть). В таком случае $(M_i, Ret_i, Args_i)$ может быть предварительно обновлён, и такие элементы заменены на переменные из Ret_j. Также элементы $hole$ могут быть заменены на значение $null$.

Шаги второго типа показаны на Рис. 1 сплошными стрелками, третьего — пунктирными. Шаги внутри методов на рисунке не показаны, поскольку исполнение конструкторов и методов Set в данном примере

не ветвится. В нижней части прямоугольников показан стек $Upcoming$, в верхней — последовательность $Current$.

После того, как в результате шага второго типа происходит вход символьного состояния s_{seq} в метод M_{target} (и стек $Upcoming$ становится пустым), с помощью SMT-решателя проверяется выполнимость условия $\pi_{seq} \wedge subst(\pi)$ ($subst$ — функция, которая производит подстановку соответствующих аргументов в условие пути π). Если данная формула невыполнима, то состояние отбрасывается. Иначе в $Current$ подставляются конкретные значения из модели (они могут быть только примитивными, поскольку лишь переменные примитивных типов были символьными), и $Current$ становится искомой последовательностью и результатом работы алгоритма.

3.3. Алгоритм, основанный на обратном символьном исполнении

Как выяснилось в ходе экспериментов, рассмотренный алгоритм, основанный на прямом символьном исполнении, обладает рядом недостатков, затрудняющих его использованию на реальном коде.

- Во-первых, поскольку проверка последовательности на соответствие целевому условию пути происходит только после того, как последовательность уже целиком создана, существует риск бесполезного перебора последовательностей с заведомо неподходящими префиксами.

- Во-вторых, до того, как какой-либо метод будет символьно исполнен, в ряде случаев нельзя сказать какое количество объектов необходимо создать, чтобы передать в качестве параметров. Такое бывает, например, тогда, когда метод принимает в качестве аргумента массив из объектов сложной структуры.

Тогда как первая проблема может быть решена путём оптимизации процесса выбора следующего состояния алгоритма для обработки,

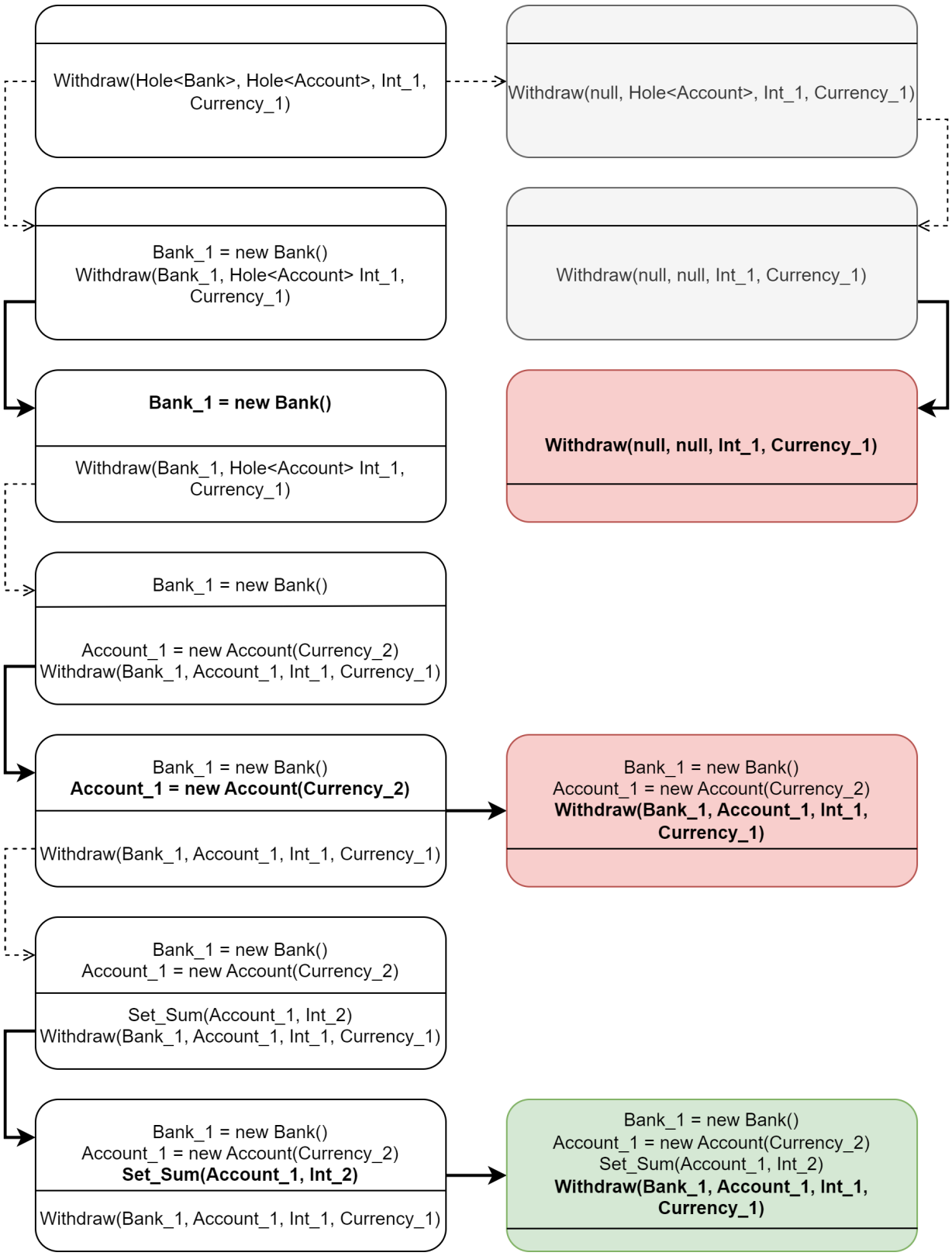

Рис. 1: Схема генерации последовательности вызовов для метода Withdraw

вторая проблема является более серьёзной. Наиболее ясным её реше-

нием является исполнение метода с символьными аргументами таких сложных типов, как массивы, которые не могут быть инициализированы простым вызовом метода. Из этого следует, что последовательность должна генерироваться не от начала к целевому методу, а от целевого метода к началу. Такое возможно при использовании концепции обратного символьного исполнения.

При обратном символьном исполнении методы, которые могут быть включены в последовательность (а в общем случае произвольные фрагменты кода) исполняются прямым символьным исполнением в изоляции, а затем результаты исполнения композируются в обратном порядке.

```
function GenerateSequence(state)
    sequences ←GetInitialSequence(state)
    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
```

Рис. 2: Псевдокод алгоритма, основанного на обратном символьном исполнении

Рассмотрим основные шаги предлагаемого алгоритма (Рис. 2).

- *GetInitialSequence.* Генерирует исходную последовательность методов, состоящую только из целевого метода. При этом последовательность хранит в себе условие пути заданного состояния в качестве предусловия. Затем данное предусловие будет обновляться.

- *GetElements.* По текущей последовательности возвращает состояния символьного исполнения методов, которые можно добавить к ней с начала. Например, если первый метод текущей последовательности принимает на вход аргумент какого-либо сложного типа, это может быть его конструктор.

- *WP (Weakest Precondition).* Вычисляет слабейшее предусловие по состоянию исполнения метода-кандидата и текущего предусловия последовательности (которое для состояния метода-кандидата является постусловием).

- *AnalyzeConflict.* В случае, если слабейшее предусловие невыполнимо, производит анализ информации о конфликте (включающей в себя unsat-ядро), который влияет на то, какие именно методы возвращаются *GetElement*.

3.4. Взаимодействие с символьным исполнением

Одно из ключевых наблюдений, используемых в настоящей работе, заключается в том, что процессы символьного исполнения и генерации последовательностей вызовов могут не только выполняться одновременно, но и эффективно взаимодействовать между собой.

Во-первых, в условиях ограниченного времени работы символьной машины следует отдавать предпочтение тем состояниям, для объектов которых последовательности найти проще, и наоборот, исследовать пути с трудно синтезируемыми объектами в последнюю очередь.

Во-вторых, вследствие инкрементальности символьного исполнения (условие пути лишь увеличивается с каждым ветвлением) справедливо

следующее утверждение: если решение задачи поиска последовательности методов в текущий момент не найдено для состояния s, то оно не найдено и ни для какого состояния s', являющегося его потомком.

4. Особенности реализации

В рамках данной работы был реализован алгоритм синтеза объектов, основанный на прямом символьном исполнении. Реализация алгоритма, основанного на обратном символьном исполнении, не входит в задачи данной работы вследствие нерелевантности и сложности.

При реализации в V# логика алгоритма была разделена между следующими сущностями (Рис. 3).

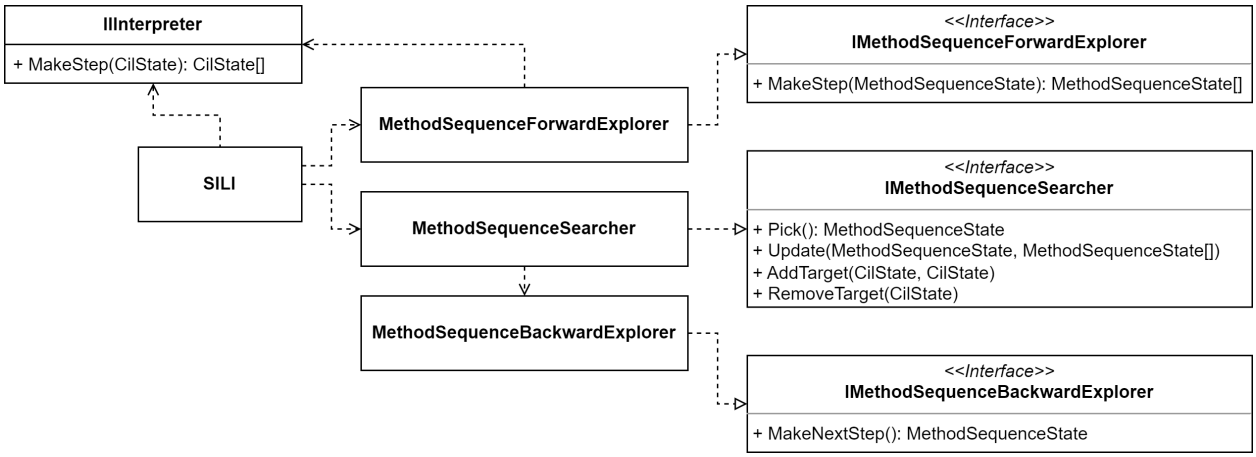

Рис. 3: Диаграмма реализованных классов

- `MethodSequenceSearcher`. Класс, осуществляющий хранение доступных состояний и выбор следующего состояния для исполнения.

- `MethodSequenceForwardExplorer`. Реализует логику шага вперед. Для состояний, находящихся внутри одного из методов последовательности, происходит шаг символьного исполнителя. Если же состояние еще не находится внутри метода, то информация о вызове следующего метода снимается со стека, и происходит соответствующий вызов.

- `MethodSequenceBackwardExplorer`. Для состояния, не находящегося внутри метода, выбирает метод, который следует положить на стек. Например, если для вызова текущего метода на вершине стека не хватает аргумента сложного типа, то произойдёт выбор

конструктора данного типа. Если же все необходимые объекты уже созданы, то происходит вызов какого-либо метода, модифицирующего уже существующий объект.

Одновременная работа алгоритма поиска последовательностей методов и символьной машины была реализована путём чередования их шагов. При этом процентная доля от общего времени работы, которая отдаётся на поиск последовательностей методов, регулируется параметром. Если его значение больше 50, то на один шаг символьной машины приходится несколько шагов алгоритма поиска последовательностей, если меньше — то наоборот. Также имеется возможность задать время на генерацию последовательностей, которое будет дополнительно выделено после символьного исполнения.

5. Эксперименты

На момент написания данной работы алгоритм, основанный на прямом символьном исполнении, был реализован на уровне прототипа, и поддерживал только генерацию последовательностей, состоящих из конструкторов и методов `Set` свойств классов. Также не были поддержаны массивы и интерфейсы. В связи с этим эксперименты проводились на синтетических тестовых данных, что, бесспорно, является основной угрозой нарушения их корректности.

В тестовый набор были включены методы, принимающие на вход объекты с вложенной структурой. Чтобы создать некоторые из них, требовалось вызвать несколько конструкторов и методов `Set`.

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 1: Результаты экспериментов

Символьная машина запускалась без лимита времени, 10 процентов от времени исполнения было выделено под генерацию последовательностей. Дополнительный лимит времени на генерацию последовательностей был задан равным 10 секундам.

Результаты экспериментов приведены в таблице 1. Для всех сгенерированных тестов, кроме одного, были найдены правильные последовательности методов. Один тест, последовательность для которого найдена не была, требует создания объекта класса `List` с нарушенным инвариантом.

Заключение

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;
- прототип разработанного алгоритма реализован в символьной виртуальной машине V#;
- проведены эксперименты на синтетических данных для определения эффективности реализованного алгоритма;
- разработан алгоритм синтеза объектов, основанный на обратном символьном исполнении.

Исходный код реализованного алгоритма открыт и может быть найден в GitHub-репозитории[5] (имя аккаунта — mxprshn).

[5]https://github.com/mxprshn/VSharp/tree/sequences Дата обращения: 28.12.2023

Список литературы

[1] English Lyn, Sriraman Bharath. Problem Solving for the 21st Century. — 2010. — 01.

[2] Fraser Gordon, Arcuri Andrea. Evolutionary Generation of Whole Test Suites // 2011 11th International Conference on Quality Software. — 2011. — P. 31–40.

[3] Graph-Based Seed Object Synthesis for Search-Based Unit Testing / Yun Lin, You Sheng Ong, Jun Sun et al. // Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. — ESEC/FSE 2021. — New York, NY, USA : Association for Computing Machinery, 2021. — P. 1068–1080. — URL: https://doi.org/10.1145/3468264.3468619.

[4] Inkumsah K., Xie Tao. Improving Structural Testing of Object-Oriented Programs via Integrating Evolutionary Testing and Symbolic Execution. — ASE '08. — USA : IEEE Computer Society, 2008. — P. 297–306. — URL: https://doi.org/10.1109/ASE.2008.40.

[5] Inkumsah K., Xie Tao. Improving Structural Testing of Object-Oriented Programs via Integrating Evolutionary Testing and Symbolic Execution // Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. — ASE '08. — USA : IEEE Computer Society, 2008. — P. 297–306. — URL: https://doi.org/10.1109/ASE.2008.40.

[6] McMinn Phil. Search-Based Software Testing: Past, Present and Future // 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops. — 2011. — P. 153–163.

[7] 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.

[8] Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution / Tao Xie, Darko Marinov, Wolfram Schulte, David Notkin. — TACAS'05. — Berlin, Heidelberg : Springer-Verlag, 2005. — P. 365–381. — URL: https://doi.org/10.1007/978-3-540-31980-1_24.

[9] Synthesizing Method Sequences for High-Coverage Testing / Suresh Thummalapenta, Tao Xie, Nikolai Tillmann et al. // Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications. — OOPSLA '11. — New York, NY, USA : Association for Computing Machinery, 2011. — P. 189–206. — URL: https://doi.org/10.1145/2048066.2048083.