Синтез кода инициализации объектов для символьного исполнения

Паршин М.А., СПбГУ, Санкт-Петербург mxprshn@gmail.com

Аннотация

Многие инструменты автоматической генерации тестов основаны на технике символьного исполнения. Символьное исполнение позволяет для каждого пути исполнения программы получить конкретные данные, на которых он достигается, и обеспечить полное тестовое покрытие кода.

В случае исследования объектно-ориентированных программ синтез кода тестов по состояниям символьного исполнения является нетривиальной задачей. Наивный подход к её решению порождает трудночитаемый код, использующий рефлексию.

В настоящей работе предложен алгоритм синтеза кода тестов по состояниям символьного исполнения, основанный на вычислении слабейших предусловий путей в методах. Прототип предложенного алгоритма реализован в символьной машине V#[1]. В результате экспериментов для 43% состояний символьного исполнения были получены человекочитаемые тесты. Медианное время поиска последовательностей для метода составило около 5% от времени символьного исполнения.

Введение

Тестирование — неотъемлемый этап цикла разработки программного обеспечения, который часто подразумевает рутинную работу. Чтобы облегчить тестирование, разрабатываются инструменты для автоматической генерации тестов. Одна из технологий, лежащих в основе данных инструментов — символьное исполнение. Символьное исполнение [1] — техника статического анализа кода, заключающаяся в исполнении программы на *символьной памяти*, в которой данные представляются как термы формального языка. При этом для каждого пути исполнения выводится условие на термы, при выполнении которого он достигается — *условие пути*. С помощью SMT-решателя[2][2] можно получить конкретные данные, удовлетворяющие усло-

[1]https://github.com/VSharp-team/VSharp, дата обращения — 14.05.2024

[2]Satisfiability Modulo Theory, выполнимость в теориях

вию пути (*модель*), что позволит сгенерировать тест, покрывающий соответствующий путь.

Для объектно-ориентированных языков генерация тестов при помощи символьного исполнения затруднена тем, что объекты обязаны удовлетворять внутренним контрактам класса. Вследствие этого не по всем состояниям символьного исполнения можно получить тест. Например, если класс, представляющий односвязный список, содержит поле, отвечающее за его размер, оно никогда не принимает отрицательное значение при использовании публичных методов списка. При этом в результате символьного исполнения могут быть получены состояния с отрицательными значениями данного поля. Если же условие пути не противоречит внутренним контрактам класса, то возникает задача поиска *инициализирующей последовательности*. Инициализирующая последовательность — это последовательность вызовов публичных методов, в результате исполнения которой будут получены объекты, удовлетворяющие условию пути.

В данной работе предложен алгоритм поиска инициализирующих последовательностей. Основная идея предложенного алгоритма заключается в том, что инициализирующая последовательность строится от конца к началу. Производится перебор методов, позволяют построить объект данного типа. Чтобы определить, может ли вызов очередного метода быть присоединен к текущей последовательности, вычисляется *слабейшее предусловие* пути. В случае, если оно выполнимо, метод добавляется к последовательности.

Предложенный алгоритм был реализован на уровне прототипа в символьной виртуальной машине V#. В результате проведённых экспериментов для 43% состояний символьного исполнения были найдены инициализирующие последовательности. Медианное время, затраченное на их поиск для одного метода, составило около 5% от медианного времени его символьного исполнения.

Слабейшие предусловия

Пусть π — путь исполнения программы. *Постусловием пути* называется условие R на состояние программы, которое должно выполняться после исполнения инструкций из пути π. *Предусловием пути* $P(\pi, R)$ называется условие на состояние программы, из выполнимости которого следует, что после исполнения инструкций из пути π постусловие R будет выполняться. Соответственно, слабейшим предусловием пути $wp(\pi, R)$ называется такое предусловие пути, что любое другое следует из него.

В работе [8] предложен алгоритм вычисления слабейших предусловий

при помощи символьного исполнения.

Существующие решения

Расширение *Seeker* для символьной виртуальной машины *Pex* [5] решает задачу поиска инициализирующих последовательностей по заданному пути исполнения. «Скелеты»[3] последовательностей исполняются символьно целиком, при этом отсутствует способ проверять их на корректность ещё на этапе их построения. Это приводит к расширению пространства поиска и, в результате, к увеличению времени, требуемого на исследование. С другой стороны, в работе *Seeker* также предлагается алгоритм статического анализа, позволяющий добавлять в «скелет» наиболее релевантные методы.

В работе [3] задача генерации инициализирующих последовательностей для символьного исполнения решается использованием инструментов *эволюционного тестирования*. Для этого условие пути конвертируется в функцию приспособленности для генетического алгоритма, что не применимо к сложным условиям пути, учитывающим, например, взаимодействие с «моковыми»[4] объектами. Метод, предложенный в настоящей работе, может быть расширен для поддержки таких случаев.

В работах [4; 6; 7] также описан поиск инициализирующих последовательностей, но этот процесс рассматривается как самостоятельная технология тестирования, а не как вспомогательная задача для символьного исполнения.

Описание алгоритма

В данном разделе описан алгоритм, решающий задачу поиска инициализирующей последовательности по данному состоянию символьного исполнения. Состояние символьного исполнения представляет собой пару (π, μ), где π — условие пути, а μ — символьная память.

Входными данными алгоритма является состояние (π_0, μ_0) символьного исполнения метода M, для которого требуется найти инициализирующую последовательность.

---

[3] англ. skeletons

[4] англ. mock

Состояние алгоритма

Состояние алгоритма — это тройка (sx, A, p), где sx — *суффикс последовательности*, A — множество аргументов, а p — *предусловие суффикса*.

Суффикс последовательности представляет собой кортеж $(m_n, ..., m_2, m_1, M)$, где $m_1, m_2, ..., m_n$ — методы последовательности (например, конструкторы), а M — тестируемый метод. Начальное значение — суффикс (M), состоящий только из тестируемого метода.

Множество A содержит аргументы методов из суффикса, которые еще не были проинициализированы в нём. Изначально множество A содержит все аргументы метода M.

Предусловие p является логической формулой от аргументов из множества A. Все состояния исполнения программы, удовлетворяющие p, удовлетворяют π_0 после исполнения методов $m_n, ..., m_2, m_1$. Начальное значение — π_0.

Композиция

Пусть $s_0 = ((m_n, ..., m_2, m_1, M), A_0, p_0)$ — состояние алгоритма, а $\sigma = (\pi, \mu)$ — состояние символьного исполнения метода m_{n+1}. Определим *композицию* $s_1 = s_0 \circ \sigma$ (Рис. 1) как состояние алгоритма (sx_1, A_1, p_1), полученное следующим образом:

1: $sx_1 \leftarrow (m_{n+1}, m_n, ..., m_2, m_1, M)$
2: $A_1 \leftarrow A_0 \setminus A^-_{m_{n+1}} \cup A^+_{m_{n+1}}$
3: $p_1 \leftarrow wp(\pi, p_0)$

В первой строке к текущему суффиксу добавляется новый метод m_{n+1}. Во второй строке множество аргументов обновляется: из него вычитается множество $A^-_{m_{n+1}}$ — множество аргументов, которые были инициализированы вызовом m_{n+1}, и к нему добавляется множество $A^+_{m_{n+1}}$ аргументов метода m_{n+1}. В третьей строке за новое предусловие суффикса принимается слабейшее предусловие wp для текущего предусловия суффикса и условия пути π.

Процесс работы

Общая схема алгоритма представлена на рис. 2.

SymbolicMethodExplorer — символьная машина, осуществляющая символьное исполнение кода методов, из которых могут строиться последовательности. Получает на вход дескрипторы методов, которые необходимо ис-

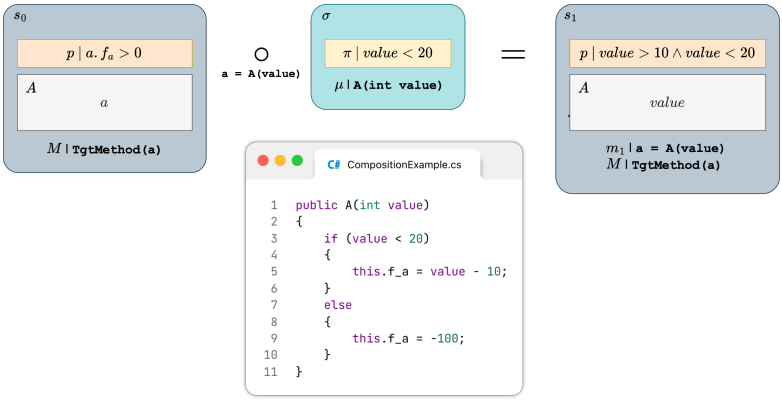

Рис. 1: Схема композиции

полнить, и в асинхронном режиме возвращает завершённые состояния символьного исполнения по мере исследования.

Queue — очередь, хранящая состояния алгоритма и соответствующие им состояния символьного исполнения, полученные от *SymbolicMethodExplorer*. Из очереди можно получить пару из состояния алгоритма и состояния символьного исполнения метода-кандидата на присоединение к последовательности.

Composer осуществляет композицию полученной из *Queue* пары состояний.

Пусть (sx_n, A_n, p_n) — результат композиции. Если p_n невыполнимо, то алгоритм продолжает работу. Если p_n выполнимо, и при этом $A_n = \emptyset$ или содержит только аргументы примитивных типов (например, *int, flow, char* и др.), то алгоритм завершает работу. В таком случае искомая последовательность найдена, аргументы примитивных типов могут быть найдены с помощью SMT-решателя. Если же это условие не выполняется, но p_n выполнимо, то s_n добавляется в очередь.

SequenceElementSelector осуществляет выбор методов-кандидатов на присоединение к текущему суффиксу последовательности. Для этого он производит анализ методов на то, какие поля ими используются.

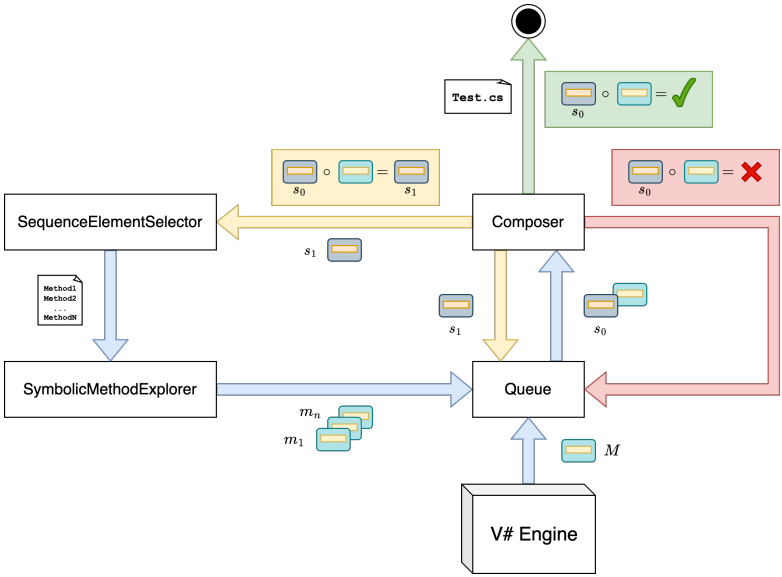

Рис. 2: Общая схема алгоритма

Реализация и эксперименты

Прототип предложенного алгоритма был реализован в символьной виртуальной машине V# на языке F#.

Эксперименты проводились на методах из проектов *btcpayserver*[5] и *openra*[6] (Табл. 1). На данный момент множество методов, из которых строится последовательность, ограничено конструкторами классов и методами, устанавливающими значения свойств[7]. В связи с этим, в тестовую выборку были включены только методы, последовательности для которых могли быть сгенерированы с учётом этих ограничений.

Машина, на которой проводились эксперименты — *MacBook Pro* с процессором *Apple M1 Pro* (8 ядер) и 16 ГБ RAM. ОС — *macOS 12.5.1*.

В начале для каждого метода из выборки запускалась символьная машина с лимитом времени 120 секунд на метод, затем для каждого из полученных состояний запускался алгоритм с лимитом времени 15 секунд на состояние. Медианное время для каждого из этапов приведено в табл. 1. Медианное вре-

[5]https://github.com/btcpayserver/btcpayserver, дата обращения — 14.05.2024
[6]https://github.com/OpenRA/OpenRA, дата обращения — 14.05.2024
[7]англ. *setters*, «сеттеры»

Количество методов	57
Количество состояний символьного исполнения	214
Медианное время исследования одного метода	19 сек
Медианное время генерации последовательностей для одного метода	1 сек

Таблица 1: Характеристики тестовой выборки и результаты экспериментов

мя поиска последовательностей для состояний символьного исполнения метода составило около 5% от медианного времени символьного исполнения.

На рис. 3 приведена гистограмма распределения длин сгенерированных последовательностей. 74% найденных последовательностей состоят из двухтрёх методов. В 49% случаев был исчерпан лимит времени на поиск последовательностей. Это может быть связано с тем, что искомой последовательности не существует, или с ограниченным пространством поиска. Идентификация данных случаев требует дополнительной трудоёмкой работы, и запланирована на будущее. Наличие ошибочных запусков говорит о возникновении в процессе исследования случаев, не поддержанных в текущей реализации алгоритма или ядре символьной машины V#. Данные случаи включают в себя инициализацию «моковых» объектов символьной машиной, а также вызовы «внешних»[8] методов в среде .NET в процессе символьного исполнения.

Рис. 3: Гистограмма распределения длин сгенерированных последовательностей

[8]англ. extern methods

Заключение

В представленной работе предложен алгоритм поиска инициализирующих последовательностей методов для состояний символьного исполнения, основанный на вычислении слабейших предусловий. Проведены эксперименты с прототипом алгоритма, реализованным в символьной машине V#. В 43% случаев были получены человекочитаемые тесты по состояниям символьного исполнения. Медианное время работы алгоритма составило около 5% от медианного времени символьного исполнения.

Список литературы

1. *Baldoni R.*, *Coppa E.*, *D'Elia D. C.*, *Demetrescu C.*, *Finocchi I.* A Survey of Symbolic Execution Techniques // ACM Comput. Surv. — New York, NY, USA, 2018. — Т. 51, № 3.
2. *Biere A.*, *Heule M.*, *Maaren H. van*. Handbook of Satisfiability. — IOS Press, 2009. — (Frontiers in Artificial Intelligence and Applications). — ISBN 9781607503767.
3. *Braione P.*, *Denaro G.*, *Mattavelli A.*, *Pezzè M.* Combining symbolic execution and search-based testing for programs with complex heap inputs // Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. — Santa Barbara, CA, USA : Association for Computing Machinery, 2017. — С. 90—101. — (ISSTA 2017). — ISBN 9781450350761.
4. *Martena V.*, *Orso A.*, *Pezzè M.* Interclass testing of object oriented software //. — 02.2002. — С. 135—144. — ISBN 0-7695-1757-9.
5. *Thummalapenta S.*, *Xie T.*, *Tillmann N.*, *Halleux J.*, *Su Z.* Synthesizing Method Sequences for High-Coverage Testing //. Т. 46. — 10.2011. — С. 189—206.
6. *Xie T.*, *Marinov D.*, *Schulte W.*, *Notkin D.* Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution // Tools and Algorithms for the Construction and Analysis of Systems / под ред. N. Halbwachs, L. D. Zuck. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2005. — С. 365—381. — ISBN 978-3-540-31980-1.
7. *Zhang Y.*, *Zhu R.*, *Xiong Y.*, *Xie T.* Efficient Synthesis of Method Call Sequences for Test Generation and Bounded Verification // Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. — , Rochester, MI, USA,Association for Computing Machinery, 2023. — (ASE '22). — ISBN 9781450394758.

8. *Мисонижник А. В.*, *Костюков Ю. О.*, *Костицын М. П.*, *Мордвинов Д. А.*, *Кознов Д. В.* Генерация слабейших предусловий программ с динамической памятью в символьном исполнении // Научно-технический вестник информационных технологий, механики и оптики. — 2022. — Т. 22, № 5. — С. 982—991.