Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.М07-мм

Паршин Максим Алексеевич

Автоматический двунаправленный синтез объектов для символьного исполнения

Отчёт по учебной практике
в форме «Решение»

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

Санкт-Петербург
2022

Оглавление

Введение

Практически каждая достаточно сложная 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-человеку удаётся не всегда.

Неудивительно, что исследования в области автоматического тестирования программ ведутся достаточно давно и достаточно успешно. В основе 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 — 23.12.2022

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 — 23.12.2022
[3]https://github.com/VSharp-team/VSharp, дата обращения — 23.12.2022

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 реализация автоматического синтеза объектов на основе механизма двунаправленного символьного исполнения в символьной виртуальной машине V#. Для достижения цели были сформулированы следующие задачи.

В рамках учебной практики и ВКР:

- провести обзор методов синтеза объектов, используемых в различных инструментах генерации тестов.

В рамках ВКР:

- разработать алгоритм синтеза объектов, основанный на двунаправленном исполнении;
- реализовать данный алгоритм в символьной виртуальной машине V#;
- провести эксперименты для определения эффективности реализованного алгоритма.

2. Обзор

Можно выделить два способа, с помощью которых генераторы тестов создают объекты с необходимыми свойствами [10].

- *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 [8] 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 [9], 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*.

2.4. Двунаправленное символьное исполнение

Механизм *двунаправленного символьного исполнения* [7] позволяет исследовать программу не только в прямом направлении, но и в обратном — от определённой локации в коде к точкам входа.

Пусть есть некоторая точка *target* в коде и связанная с ней формула π. Исполнение в обратном направлении от точки *target* в упрощённом виде происходит следующим образом.

- При помощи поиска в графе потока управления и в графе вызовов определяются некоторые локации в коде, из которых *target* может быть достижима.
- Из найденных точек запускается прямое символьное исполнение, приоритезированное расстоянием до точки *target*.
- Когда из некоторой точки A достигается *target*, проверяется выполнимость композиции полученного условия пути и π.
- Если композиция выполнима, то A становится новой целевой точкой, а сама композиция — новым условием π.
- Таким образом, когда исполнение достигает точки входа в программу, формула π является условием достижения первоначальной точки *target*.

Существует реализация механизма двунаправленного символьного исполнения в символьной машине *KLEE*[4]. На уровне абстракций двунаправленное исполнение интегрировано и в V#.

[4]https://github.com/misonijnik/klee/tree/bidirectional-klee-2.2, дата обращения — 25.12.2022

3. Подход к реализации

В настоящий момент в символьной виртуальной машине V# для генерации объектов используется наивный подход — создание напрямую при помощи рефлексии. Требуется реализовать генератор последовательностей методов, строящий тест по условиям пути, полученным в результате символьного исполнения.

При этом, поскольку известно определённое условие пути, для которого требуется сгенерировать тест, можно сказать, что данная задача по своей сути сходна с задачей, решаемой двунаправленным символьным исполнением. Как минимум, существующие методы генерации тестов, которые используют символьное исполнение, могут быть улучшены путём интеграции двунаправленного механизма.

Заключение

В ходе данной работы были получены следующие результаты:

- проведён обзор методов синтеза объектов, используемых в различных инструментах генерации тестов.

В рамках ВКР планируется:

- разработать алгоритм синтеза объектов, основанный на двунаправленном исполнении;
- реализовать данный алгоритм в символьной виртуальной машине V#;
- провести эксперименты для определения эффективности реализованного алгоритма.

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] Mordvinov Dmitry. Property Directed Symbolic Execution. — 2021. — Spring/Summer Young Researchers' Colloquium on Software Engi-

neering. URL: https://youtu.be/pX5qS4SbsJI (online; accessed: 03.05.2022).

[8] 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.

[9] 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.

[10] 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.