Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем
Кафедра системного программирования

Батоев Константин Аланович

Композициональное символьное исполнение CIL-кода

Бакалаврская работа

Научный руководитель:
ст. преп. Д. А. Мордвинов

Рецензент:
программист ООО "ИнтеллиДжей Лабс" Д. С. Косарев

Санкт-Петербург
2019

Оглавление

Введение

Статический анализ используется для выявления некоторых свойств времени выполнения программы уже на этапе компиляции. Таким образом, могут быть обнаружены различные ошибки в коде: недостижимый код, необработанное исключение и др.

Одним из способов реализации статического анализа является статическое символьное исполнение в сочетании с проверкой моделей (model checking). Идея символьного исполнения [5] заключается в исполнении кода не над конкретными значениями, а над символьными выражениями, каждое из которых представляет типизированное множество значений, которые может принять аргумент или локальная переменная. В отличие от стандартного исполнения кода, символьное — происходит не по определенному пути, а сразу над множеством путей одновременно. При этом для каждого пути можно задать такой конкретный набор значений параметров, что реальное исполнение пройдет тот же путь. Символьное исполнение применяется как для тестирования (нахождения входных значений, влекущих нежелательный результат), так и для верификации (доказательства корректности программы). Проверка моделей используется для анализа состояний программы, полученных с помощью символьного исполнения, на отсутствие ошибок.

Взрыв множества состояний программы является одной из открытых проблем символьного исполнения [6]. Операторы языка, подразумевающие передачу управления нескольким участкам кода, например, *if, switch, for*, являются механизмами увеличения состояний программы. Например, если в программе имеется цикл, количество итераций которого является символьным значением, то мощность множества состояний программы может быть бесконечной. Более того, если язык программирования поддерживает оператор «goto», то граф потока управления (CFG) анализируемой программы может быть произвольным, например, иметь не одну, а несколько вершин, в которых может начаться исполнение цикла. Но для верификации программы необходимо проанализировать все её состояния. Поскольку существует взаимно однозначное соответствие между путями исполнения программы и путями в графе потока управления, можно свести исходную задачу получе-

ния всех состояний программы к задаче описания всех путей в графе потока управления программы.

Проект V#[1] представляет собой символьную виртуальную машину для анализа приложений платформы .NET. Он является инструментом статического композиционального символьного исполнения без раскрутки отношений перехода в программе. Композициональное символьное исполнение – это техника проведения символьного исполнения с целью переиспользования результатов исполнения [4]. Механизмом такой техники является операция композиции. Исполнение программы без раскрутки циклов стало возможно благодаря созданию рекурсивных «сущностей» для элементов, вовлеченных в рекурсию. Например, символьное исполнение рекурсивного вызова функции f заключается в создании рекурсивных состояний и рекурсивных символов, обозначающих результаты, возвращенные функцией f.

В текущей версии проекта есть интерпретатор, который символьно исполняет абстрактное синтаксическое дерево для языка C#. Однако он не поддерживает некоторые узлы абстрактного дерева, например, оператор явной передачи управления «goto». Более того, хотелось бы уметь анализировать уже скомпилированные программы без необходимости построения дерева, например, сборки пакетного менеджера NuGet. Известно, что у платформы .NET есть промежуточный язык Common Intermediate Language (CIL), представляющий собой низкоуровневый язык, в который компилируются все языки платформы .NET. Одними из инструкций языка являются инструкции условного и безусловного перехода, благодаря которым граф потока управления программы может быть произвольным. Таким образом, возникает задача создания и интеграции нового символьного интерпретатора, который смог бы работать с графом потока управления программы и получал бы все множество состояний программы.

Постановка задачи

Целью данной выпускной квалификационной работы является разработка символьного интерпретатора промежуточного языка CIL платформы

[1] https://github.com/dvvrd/VSharp

.NET для символьной виртуальной машины V#. В рамках работы были поставлены следующие задачи.

- Провести обзор алгоритмов интерпретации кода для символьного исполнения программы.
- Создать метод описания всех путей в произвольном графе потока управления с помощью введения рекурсивных символов для циклов.
- Разработать алгоритм, выполняющий композициональное символьное исполнение программы без раскрутки отношения перехода программы.
- Реализовать символьный интерпретатор языка CIL.
- Провести апробацию интерпретатора.

1. Обзор

1.1. Символьное исполнение

Символьное исполнение — это подход исполнения программ, который использует символьные значения вместо конкретных и исполняет множество путей исполнения программы вместо одного, как при стандартном исполнении. Например, (x, y, z, ...) вместо (17, "cat", $true$, ...).

```
1  void foobar(int a, int b)
2  {
3      int x = 1, y = 0;
4      if (a != 0) {
5          y = 3+x;
6          if (b == 0)
7              x = 2*(a+b);
8      }
9      assert(x-y != 0);
10 }
```

Листинг 1.1: Программа для иллюстрации символьного исполнения

На листинге 1.1 представлен фрагмент кода, взятый из работы [6]. Цель символьного исполнения — найти все входы программы, а именно значения переменных a и b, для которых не выполнится «assert». Полный перебор не применим, поскольку каждая переменная может принимать 2^{32} значений. Эту проблему и пытается решить символьное исполнение, заменяя начальные значения переменных a и b на символьные α и β.

Опишем процесс интерпретации исходного кода. Во-первых, в каждый момент времени символьное исполнение оперирует над состоянием ($stmt$, π, σ), где:

- $stmt$ — следующая инструкция для интерпретации;
- π — условие пути, представленное формулой первого порядка, описывающей ограничения, которые привели символьное исполнение в данную точку программы;

- σ — это отображение, сопоставляющее каждой переменной программы ее символьное значение.

Во-вторых, необходимо символьно исполнять инструкции языка — получать новые состояния. В представленной программе есть 3 оператора: оператор присваивания, инструкция ветвления и оператор контроля ошибок. Опишем, как ведет себя символьный интерпретатор для них:

- присваивание $x = expr$ обновляет отображение σ, которое теперь сопоставляет переменной x новое символьное выражение, вычисленное согласно $expr$;

- инструкция ветвления $if\ cond\ then\ stmt_{true}\ else\ stmt_{false}$ изменяет ограничение пути π; символьное исполнение порождает два новых состояния, у которых ограничения пути имеют вид: $\pi \wedge cond$ и $\pi \wedge \neg cond$, а следующие инструкции $stmt_{true}$ и $stmt_{false}$, соответственно;

- инструкция проверки $assert(cond)$ создает формулу $\pi \wedge \neg cond$, для которой решатель ограничений пытается найти такие конкретные значения символьных переменных, что формула станет истинной, тем самым, найдя ошибку в программе.

Наглядно описать процесс символьного исполнения программы можно с помощью дерева символьного исполнения, пример которого представлен на рисунке 1.

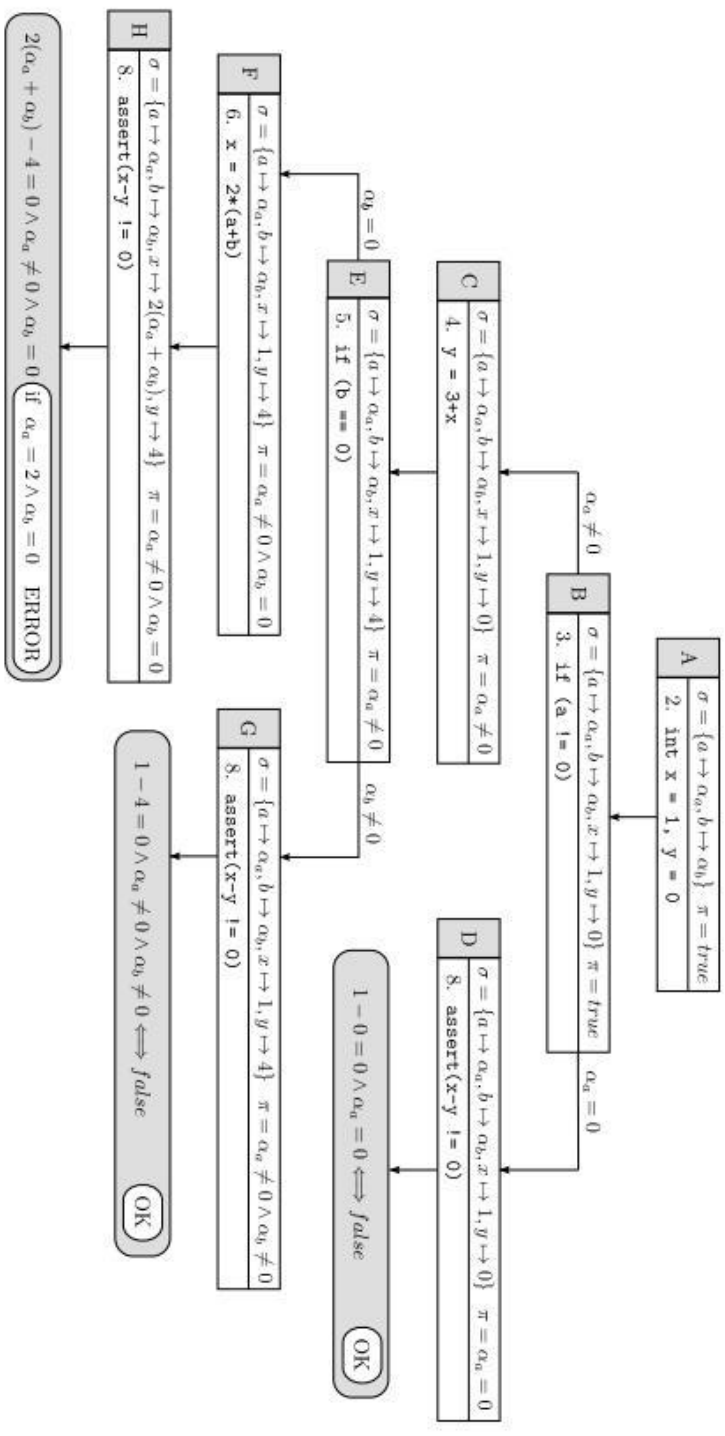

Рис. 1: Дерево символьного исполнения функции из листинга 1.1 работы [6]

На этом рисунке каждый лист соответствует результату выполнения функции на любом конкретном наборе переменных, удовлетворяющем формуле этого листа π.

1.2. Алгоритм символьного исполнения

На листинге 1.2 представлен алгоритм символьного исполнения языка, содержащего четыре инструкции, из работы [2].

1. Алгоритм оперирует с рабочим множеством состояний W. Каждую итерацию цикла (строка 3) из него при помощи функции $pickNext$ извлекается новое состояние для интерпретации.

2. Процедура $ExecuteInstruction$ на листинге 1.4 обрабатывает конструкции языка и получают множество следующих состояний S.

   - Для “присваивания” создается новое состояние, у которого метка инструкции следует за l, условие пути остается неизменным, а новое отображение s сопоставляет переменной v символьное выражение, равное e, вычисленному при помощи текущего отображения s.
   - Конструкция “ветвления” проверяет возможности принятия/непринятия перехода по условию. В случае принятия создается новое состояние, у которого метка инструкция — l', условие пути — конъюнкция старого условия и выражения e, а отображение s остается неизменным. Аналогично, в случае непринятия к множеству S добавляется состояние, у которого метка инструкции — это следующая инструкция, а условие пути — конъюнкция текущего условия пути с отрицанием e.
   - Оператор «assert» задает инварианты программы. Строка 11 проверяет нарушение инварианта. Если инвариант не нарушен, то создается новое состояние, у которого номер инструкции — это следующая инструкция.

- Оператор «halt» означает, что текущее состояние приводит к завершению интерпретируемой программы.

3. Процедура *Join* на листинге 1.3 отвечает за обновление множества состояний W, а именно, ищутся эквивалентные состояния для объединения в одно. Хорошо известны два подхода для стратегии слияния состояний.

   - Никогда не сливать состояния. Данный подход называется *динамическим символьным исполнением*. Плюс данного подхода — это относительно простая для решателя ограничений формула условия пути. В свою очередь, минусом является экспоненциальный рост рабочего множества W.
   - Всегда сливать состояния, если у них одинаковые метки инструкций l. Этот подход носит название *статического символьного исполнения*. Он решает проблему роста множества состояний. Однако структура полученного после слияния состояния усложняется, затрагивая как условие пути, так и отображение s и затрудняя работу решателя ограничений.

```
Вход    : Функция выбора следующего состояния pickNext, отношение
          эквивалентности двух состояний ∼ и номер начальной инструкции
          l_0.
Данные: Множество состояний W и множество следующих состояний S.
1 W := {(l_0, true, λv.v)};
2 while W ≠ ∅ do
3     (l, pc, s) := pickNext(W); S := ∅;
      // Символьно исполним следующую инструкцию
4     S := ExecuteInstruction(l, pc, s);
      // Объединим состояния с эквивалентными из множества W
5     forall (l', pc', s') ∈ S do
6         W := Join(W, (l', pc', s'), ∼);
7     end
8 end
9 print "no errors";
```

Листинг 1.2: Адаптированная схема символьного исполнения из работы [2]

```
Вход    : Рабочее множество W, номер инструкции l', условие пути pc',
          отображение переменных в символьные значения s', функция
          определения эквивалентности состояний ∼
Выход   : Новое рабочее множество W.
1 if ∃(l', pc'', s'') ∈ W : (l', pc'', s'') ∼ (l', pc', s') then
2     W := W \ {(l', pc'', s'')};
3     W := W ∪ {(l', pc' ∨ pc'', Merge(pc', pc'', s', s''))};
4 else
5     W := W ∪ {(l', pc', s')};
6 return W;
```

Листинг 1.3: Процедура Join добавления нового состояния в рабочее множество

```
   Вход    : Номер инструкции l, условие пути pc, отображение переменных в
             символьные значения s
   Данные: Множество следующих состояний S.
1  S := ∅;
2  switch instr(l)
3      case v := e      // присваивание
4          S := {(succ(l), pc, s[v ↦ Eval(s, e)])};
5      case if(e) goto l'      // условный переход
6          if SAT(s, pc ∧ e) then
7              S := {(l', pc ∧ e, s)};
8          if SAT(s, pc ∧ ¬e) then
9              S := S ∪ {(succ(l), pc ∧ ¬e, s)};
10     case assert(e)      // проверка
11         if SAT(s, pc ∧ ¬e) then abort;
12         else S := {(succ(l), pc, s)};
13     case halt      // завершение программы
14         print pc ;
15 return S;
```

Листинг 1.4: Процедура ExecuteInstruction символьного исполнения инструкции языка

1.3. Стратегии слияния состояний для улучшения производительности алгоритма

В работе [2] также были представлены подходы для определения функций $pickNext$ и $\sim$, которые показали хорошие результаты на практике.

- Подход, который определяет функцию эквивалентности состояний $\sim$, называется *query count estimation*. Суть заключается в определении множества "горячих переменных" («hot vaiables»), которые вероятно будут причиной нагрузки на нижележащий решатель ограничений. Формально, состояния s_1 и s_2 следует сливать, если для каждой "горячей переменной" v выполняется одно из двух:
  - $s_1[v] = s_2[v]$ и символьное выражение $s_1[v]$ является символьной

константой;

– символьные выражения $s_1[v]$ или $s_2[v]$ зависят от символьной переменной.

Множество "горячих переменных" строится с помощью введения формализма, который использует численные параметры α, β, κ, которые задаются экспериментально.

- Подход, определяющий функцию $pickNext$, называется *dynamic state merging*. Он вводит модель истории состояний, которая будет управлять выбором следующего состояния. Таким образом $pickNext$ выберет то состояние $s_1 \in W$, которое эквивалентно предку другого состояния $s_2 \in W$, с надеждой на то, что спустя несколько итераций интерпретатора, потомок s_1 станет эквивалентен s_2, чтобы можно было их слить в единое состояние.

Предоставленные результаты экспериментов показали увеличение покрытия множества состояний и время работы анализатора.

Другая работа [3] привнесла некоторые изменения к схеме алгоритма 1.2. Предлагается немного другая схема исполнения *concolic execution*, которая является сочетанием конкретного и символьного исполнения. То есть, большую часть анализа программы занимает реальное исполнение, которое при встрече инструкции ветвления переходит в режим статического символьного исполнения:

- CFGRecovery — восстанавливает часть графа потока управления для текущей инструкции;

- CFGReduce — преобразует граф потока управления в ациклический аналог (CFG_e) с возможной раскруткой циклических регионов и возвращает множество «точек перехода» (TransitionPoints);

- StaticSymbolic — статически исполняет код, соответствующий ациклическому подграфу;

- наконец, все точки перехода фильтруются предикатом достижимости.

Данная работа придерживалась такой стратегии слияния: в режиме статического символьного исполнения состояния, соответствующие одной и той же «точке перехода» всегда сливались в одно состояние. Затем для каждой достижимой «точки перехода» создается новый динамический исполнитель. В данной работе проведено разностороннее тестирование, нашедшее 2 ранее не обнаруженные ошибки в «GNU coreutils» — основных утилитах для операционных систем семейства GNU.

В итоге, можно заключить, что выбор функции эквивалентности состояний нетривиальная задача, решение которой может затронуть различные сферы, включая теорию графов, вероятностный подход, различные алгоритмические идеи и другие.

1.4. Композициональность

Представим ситуацию, когда одна и та же функция вызывается несколько раз. Конечно, можно было бы каждый раз исполнять код функции заново на переданных ей аргументах. Однако такой подход может быть неоправданным, если функция большая, и в общем случае неприменимым, если функция содержит вызов функции, находящейся в стеке вызовов, ввиду неограниченной рекурсии.

Альтернативный способ обработки вызова функции — это сначала исследовать функцию без предположений на входные аргументы и получать результат, описывающий ее полное поведение, а затем «уточнять» результат для переданных аргументов. Данный способ называется *композициональным*, а «уточнение» результата производится с помощью операции *композиции*.

Продемонстрируем идею операции композиции на примере (см. листинг 1.2). Основная функция g два раза вызывает функцию f. Сначала исполним функцию f в «изоляции» и получим результат:

$$res_f = \begin{cases} x \bmod 2 = 0, x \\ x \bmod 2 \neq 0, 2 * x \end{cases}$$

Затем применим данный результат к вызовам f внутри функции g. Значение для переменной b вычислится как композиция состояния, содержащего значение аргумента x, и результата функции f:

$$\{x \to 5\} \circ res_f = 10$$

Аналогично, получаем значение переменной c:

$$\{x \to 10\} \circ res_f = 10$$

```
1  int f(int x) {
2      if (x % 2 == 0) return x;
3      return 2*x;
4  }
5
6  int g()
7  {
8      int b = f(5);
9      int c = f(b);
10     return b + c;
11 }
```

Листинг 1.2: Программа для иллюстрации композиционального подхода

1.5. Стратегия вызова рекурсивных функций в проекте V#

Вызов функции представляет собой потенциальный источник рекурсии. Однако иногда выгоднее явно исполнить код функции еще раз, например, когда вызов функции происходит с «конкретными», а не символьными значениями аргументов. В ядре проекта создана архитектура, которая выбирает: нужно ли исполнять вызванную функцию снова, либо нужно применить вычисленные для нее результат и состояния, если она была уже исследована, либо нужно создать специальные *рекурсивные символы* для результата функции и для её состояния, если она находится на стеке вызовов (см. рис. 2).

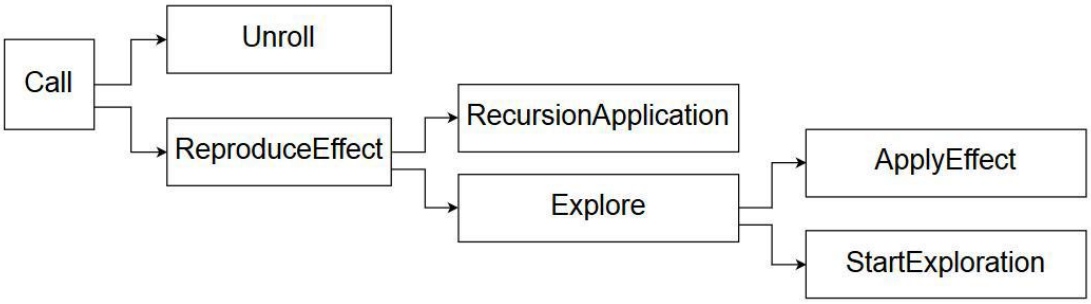

Рис. 2: Схема обработки «вызова функции» в проекте V#

Схема достаточно универсальна и конфигурируема с помощью опций. Например, можно явно задать:

- AlwaysUnroll – всегда исполнять функцию;
- NeverUnroll – исполнять функцию, если мы ее еще не вызывали; иначе — применить композицию с результатом исследования в «изоляции»;
- SmartUnroll – эвристика, определяющая стоит ли исполнять функцию, либо попытаться получить состояние исследования в «изоляции», с которым впоследствии делать композицию.

Другим источником рекурсии для программы являются циклы. Естественно было бы обрабатывать раскрутку циклов с помощью схожей схемы, как и вызов функции. Например, опция SmartUnroll раскручивала бы циклы, количество итераций которых представляет конкретное число, и создавала бы рекурсивные символы для результата и для состояния для циклов, количество итераций которых произвольно.

Таким образом, возникает необходимость в расширении схемы «вызова функции» на случай других источников рекурсии.

2. Метод описания путей в графе потока управления

Для того чтобы иметь возможность рассуждать о корректности программы, необходимо реализовать символьный интерпретатор, который бы исследовал все множество состояний программы. В распоряжении имеется механизм создания *рекурсивных* состояний, который можно применить для циклов в графе. Можно попробовать описать все пути исполнения программы, поэтому свели исходную задачу к задаче описания всех путей графа потока управления, начинающихся из стартовой вершины, с использованием рекурсивных символов для циклов в графе. Хотелось бы ввести минимальное количество таких символов, чтобы не затруднять будущую работу нижележащего решателя, но в то же время сохранить полноту и корректность описания путей.

Определение 1. *Граф потока управления — это четверка (V_G, E_G, start, exit), где:*

- V_G *— это множество вершин графа, каждая из которых соответствует инструкции программы;*
- E_G *— это множество ориентированных ребер $e = (u, v)$ графа, каждое ребро обозначает способность передачи управления от вершины u к вершине v;*
- *start — это стартовая вершина(инструкция), в которую не входит ни одно ребро графа;*
- *exit — это финальная вершина(инструкция), из которой не выходит ни одного ребра графа.*

2.1. Основные понятия

Для произвольного графа потока управления, построенного по программе, занумеруем вершины при помощи обратного обхода (Post Order DFS), который присваивает каждой вершине время посещения по формуле:

$$time(v) = |V_G| - |\{u \in V_G | \text{для вершины u уже вычислено time(u)}\}|.$$

Определение 2. *Ребро* $e = (u, v)$, *соединяющее вершины* u и v, *называется* обратным, *если* $time(u) \geq time(v)$. *Если ребро не является* обратным, *то оно называется* прямым. *Обозначим* $beg(e) = u$, *а* $end(e) = v$. *Множество обратных ребер обозначим* BE.

Определение 3. *Вершина* s *называется* рекурсивной, *если* $\exists e \in E_G$, *что* e *— обратное* и $end(e) = s$. *Множество* рекурсивных *вершин обозначим* RV.

Определение 4. *Путем* p *в графе* G *будем называть последовательность ребер* $e_1 ... e_n$ *такую, что* $\forall i \ 1 \leq i \leq n-1 \ end(e_i) = beg(e_{i+1})$. *Началом пути* $beg(p)$ *обозначим вершину* $beg(e_1)$. *Концом пути* $end(p)$ *обозначим вершину* $end(e_n)$. *Иногда удобно представлять путь в графе в виде последовательности вершин:* $v_1 \to v_2 \to ... \to v_n \to v_{n+1}$ *такой, что* $\forall i \ 1 \leq i \leq n \ v_i = beg(e_i)$ *и* $v_{i+1} = end(e_i)$.

Определение 5. *Путь, состоящий из 0 ребер, называется* пустым. *Будем обозначать его* ε.

Определение 6. *Пусть даны два пути* p_1 *и* p_2 *в графе* G. *Если* $end(p_1) = beg(p_2)$, *то можно определить конкатенацию путей* p_1 *и* p_2, *обозначаемую* $p_1 \circ p_2$ *и представляющую путь, содержащий все ребра пути* p_1, *за которыми следуют ребра* p_2.

Заметим, что пустой путь ε *является нейтральным элементов по отношению к операции конкатенации.*

Определение 7. *Дана вершина графа* v. *Дано множество путей* P_1, *что все пути заканчиваются на вершине* v. *Дано множество путей* P_2, *что все пути начинаются с вершины* v. *Тогда определим конкатенацию множеств путей* P_1 *и* P_2 *таким образом:*

$$P_1 \circ P_2 = \{p_1 \circ p_2 \quad | \quad p_1 \in P_1, p_2 \in P_2\}.$$

2.2. Метод

Введем определения, для которых докажем необходимые леммы для получения формулы всех путей в CFG.

Определение 8. *Если длина пути* $p = v_0 \to v_1 \to ... \to v_n \to v_{n+1}$ *больше, либо равна 2, то будем называть серединой пути множество вершин* $\{v_i \mid 1 \leq i \leq n\}$. *Если длина пути меньше 2, то середина пути* $-\varnothing$.

Определение 9. $\Pi(u, v, D)$ *— символ для множества путей, начинающихся в вершине* u, *заканчивающихся в вершине* v и *не проходящих через* рекурсивные *вершины множества* D *в середине пути. Это множество путей строится с помощью следующих правил.*

- *Если в графе есть ребро* (u, v), *то путь, состоящий из одного ребра, добавляется во множество путей* $\Pi(u, v, D)$ *(см. правило I).*
- *Если есть ребро* (u, t), *где* $t \neq v$ *и* $t \notin RV$, *то конкатенация ребра* (u, t) *и путей* $\Pi(t, v, D)$ *добавляется к множеству путей* $\Pi(u, v, D)$ *(см. правило II).*
- *Если есть ребро* (u, t), *где* $t \neq v$, $t \in RV$ и $t \notin D$, *то конкатенация ребра* (u, t) *и* $Rec(t, D \cup \{t\}) \circ \Pi(t, v, D \cup \{t\})$ *добавляется к множеству путей* $\Pi(u, v, D)$ *(см. правило III), где* $Rec(t, D \cup \{t\})$ *— множество циклов из* t *в* t, *не проходящих через* рекурсивные *вершины из множества* $D \cup \{t\}$ *в середине пути, а* $\Pi(t, v, D \cup \{t\})$ *— множество путей из* t *в* v, *не проходящих через* рекурсивные *вершины из множества* $D \cup \{t\}$ *в середине пути.*

$Rec(u, D)$ *— рекурсивный символ для множества циклов из вершины* u *в вершину* u, *не проходящих через вершины из множества* D *в середине пути, которые построены по правилу IV.*

- *Пустой путь* ε *принадлежит множеству* $Rec(u, D)$.
- *Конкатенация путей* $\Pi(u, u, D)$ *и циклов* $Rec(u, D)$ *добавляется к множеству циклов* $Rec(u, D)$.

$$\Pi(u,v,D) = \bigcup_{(u,v)\in E_G} \{(u,v)\} \qquad \cup \qquad (I)$$

$$\bigcup_{\substack{t\notin RV \\ t\neq v \\ (u,t)\in E_G}} (u,t) \circ \Pi(t,v,D) \qquad \cup \qquad (II)$$

$$\bigcup_{\substack{t\in RV \\ t\neq v \\ (u,t)\in E_G \\ t\notin D}} (u,t) \circ Rec(t, D\cup\{t\}) \circ \Pi(t,v,D\cup\{t\}) \qquad (III)$$

$$Rec(u,D) = \{\varepsilon\} \cup \Pi(u,u,D) \circ Rec(u,D) \qquad (IV)$$

Лемма 2.1. *$\forall p \in \Pi(u,v,D)$ начало пути $beg(p) = u$ и конец пути $end(p) = v$.*

Доказательство. Очевидно. □

Лемма 2.2. *$\forall u, v, D$ верно, что $\varepsilon \notin \Pi(u,v,D)$.*

Доказательство. Очевидно, поскольку согласно правилам (I – III), любой путь $p \in \Pi(u,v,D)$ начинается с ребра (u,t). □

Лемма 2.3. *Пусть дан путь $p = s_1 \to ... \to s_n$ такой, что $\forall i > 1 \quad s_i \neq s_1$. Тогда $p \in Rec(s_1, D) \circ \Pi(s_1, s_n, D) \iff p \in \Pi(s_1, s_n, D)$.*

Доказательство. Достаточность. Очевидно, поскольку $\varepsilon \in Rec(s_1, D)$.
Необходимость. Раскроем определение $Rec(s_1, D)$.

$$Rec(s_1,D)\circ\Pi(s_1,s_n,D) = \begin{cases} \Pi(s_1,s_n,D) \\ \Pi(s_1,s_1,D) \circ Rec(s_1,D) \circ \Pi(s_1,s_n,D) \end{cases} \qquad (2)$$

Согласно леммам 2.1 , 2.2 $\forall p \in \Pi(s_1,s_1,D)\circ Rec(s_1,D)\circ\Pi(s_1,s_n,D)$ вершина s_1 встречается в p как минимум два раза, но по условию, s_1 встречается только один раз. Следовательно, единственная возможность — $p \in \Pi(s_1,s_n,D)$. □

Лемма 2.4. *Пусть даны множество D такое, что $D \subseteq RV$, и путь $p = s_1 \to ... \to s_n$, удовлетворяющий условиям:*

1. *длина* $p \geq 1$;

2. $s_n = exit$ *или* $s_n \in D$;

3. $\forall i < n, \quad s_i \notin D \setminus \{s_1\}$.

Тогда верно следующее.

- *Если* $s_1 \notin RV$, *то* $p \in \Pi(s_1, s_n, D)$.

- *Если* $s_1 \in RV$, *то* $p \in Rec(s_1, D) \circ \Pi(s_1, s_n, D)$.

Доказательство. Индукция #1 по длине пути p. База: длина пути равна 1. Тогда $p = s_1 \to s_n \in \{(s_1, s_n)\} \subseteq \Pi(s_1, s_n, D)$. Если $s_1 \in RV$, то $p \in Rec(s_1, D) \circ \Pi(s_1, s_n, D)$, поскольку $\varepsilon \in Rec(s_1, D)$.

Индукционный переход.

- Пусть $p = s_1 \to v \to ... \to s_n$, и предположим, что вершина s_1 встречается в пути p ровно 1 раз.

  – Если вершина $v \notin RV$, то по И.П. #1, путь $v \to ... \to s_n \in \Pi(v, s_n, D)$, и, по правилу (П), имеем $p \in \Pi(s_1, s_n, D)$, так как $v \neq s_n$.

  – Пусть вершина $v \in RV$. Тогда по условию (3) можно сделать вывод, что $v \notin D$. Поскольку для всех вершин u, кроме последней, в пути $v \to ... \to s_n$ выполнено $u \notin D \cup \{v\} \setminus \{v\}$, так как среди них нет s_1, то по И.П. #1, верно, что $v \to ... \to s_n \in Rec(v, D \cup \{v\}) \circ \Pi(v, s_n, D \cup \{v\})$, а значит, по правилу (III), $p \in \Pi(s_1, s_n, D)$, так как $v \neq s_n$.

  Если вершина $s_1 \in RV$, то $p \in Rec(s_1, D) \circ \Pi(s_1, s_n, D)$, поскольку $\varepsilon \in Rec(s_1, D)$.

- Пусть $p = s_1 \to v \to ... \to s_1 \to ... \to s_n$ и $s_1 \in RV$. Обозначим $p \equiv q \circ p'$, где $q \equiv s_1 \to ... \to s_1$, а $p' \equiv s_1 \to ... \to s_n$ такой, что вершина s_1 встречается в пути p' ровно один раз, если $s_n \neq s_1$, или ровно два раза, если $s_n = s_1$. Если $s_1 \neq s_n$, то по И.П. #1, $p' \in Rec(s_1, D) \circ \Pi(s_1, s_n, D)$, откуда по лемме 2.3, $p' \in \Pi(s_1, s_n, D)$. Если $s_n = s_1$, то рассмотрим подробнее путь p'.

– $p' \equiv s_1 \to s_1$, откуда $q' \in \{(s_1, s_1)\} \subseteq \Pi(s_1, s_1, D)$.

– $p' \equiv s_1 \to v_1 \to ... \to v_m \to s_1$ такой, что $v_1 \notin RV$. Так как $\forall 1 \leq i \leq m$ верно, что $v_i \notin D$, то по И.П. #1, путь $v_1 \to ... \to v_m \to s_1 \in \Pi(v_1, s_1, D)$, а, по правилу (II), получаем, что $p' \in \Pi(s_1, s_1, D)$, так как $v_1 \neq s_1$.

– $p' \equiv s_1 \to v_1 \to ... \to v_m \to s_1$ такой , что $v_1 \in RV$. Так как $\forall 1 \leq i \leq m$ верно, что $v_i \notin D \cup \{v_1\} \setminus \{v_1\}$, то по И.П. #1, путь $v_1 \to ... \to v_m \to s_1 \in Rec(s_1, D \cup \{v_1\}) \circ \Pi(v_1, s_n, D \cup \{v_1\})$, а, по правилу (II), получаем, что $p' \in \Pi(s_1, s_1, D)$, так как $v_1 \neq s_1$.

Осталось показать, что $q \in Rec(s_1, D)$. Индукция #2 по длине q. База $q \equiv \varepsilon$ — очевидно.

Переход. Пусть $q \equiv q' \circ q''$, где $q' \equiv s_1 \to ... \to s_1$ и содержит только два вхождения s_1, а $q'' \equiv s_1 \to ... \to s_1$ является остатком пути q. Но по И.П. #2, получаем, что $q'' \in Rec(s_1, D)$. Аналогично получаем, что $q' \in \Pi(s_1, s_1, D)$. Тогда согласно правилу (IV), получаем, что $q \in Rec(s_1, D)$. Но тогда $p \equiv q \circ p' \in Rec(s_1, D) \circ \Pi(s_1, s_n, D)$.

- Пусть $p = s_1 \to v \to ... \to s_1 \to ... \to s_n$ и $s_1 \notin RV$. Покажем, что $v \neq s_n$. Пусть это не так и $v = s_n$. Тогда единственная возможность — $s_n \in D$, потому что из вершины *exit* не исходит ребер. Но в таком случае не выполняется условие (3) для пути p в вершине v, потому что $D \setminus \{s_1\} = D$. Пришли к противоречию, значит, $v \neq s_n$. Если вершина $v \notin RV$, то по И.П. #1, получаем, что $v \to ... \to s_1 \to ... \to s_n \in \Pi(v, s_n, D)$ и, по правилу (II), $p \in \Pi(s_1, s_n, D)$. Если вершина $v \in RV$, то по И.П. #1, для $D := D \cup \{v\}$ получаем, что $v \to ... \to s_1 \to ... \to s_n \in Rec(v, D \cup \{v\}) \circ \Pi(v, s_n, D \cup \{v\})$ и, по правилу (III), $p \in \Pi(s_1, s_n, D)$.

□

Теорема 2.5 (Пути в графе)**.** *$p = start \to ... \to exit$ — путь в графе G тогда и только тогда, когда $p \in \Pi(start, exit, \varnothing)$.*

Доказательство. Достаточность. Очевидно.

Необходимость.

Воспользуемся условиями леммы 2.4.

- Так как $start \neq exit$, то длина пути $p \geq 1$.
- $\forall i < n, \quad s_i \notin \varnothing \setminus \{start\}$.
- Последняя вершина равна $exit$.
- $start \notin RV$.

Тогда получаем, что $p \in \Pi(start, exit, \varnothing)$. □

Таким образом, получено описание всех путей в графе потока управления, на основе которого будет придуман алгоритм композиционального символьного исполнения без раскрутки отношения перехода.

3. Алгоритм композиционального символьного исполнения

На листинге 3.1 представлена схема алгоритма *Exec* для осуществления композиционального символьного исполнения без раскрутки отношения перехода. Стоит отметить, что для выполнения алгоритма необходимо построить граф потока управления программы и вычислить множество *рекурсивных* вершин RV. Для упрощения объяснения нового алгоритма будем предполагать, что функция выбора нового состояния $pickNext$ и функция определения эквивалентности двух состояний $\sim$ определены глобально.

Вход : Номер начальной инструкции l_0,номер конечной инструкции l_{last}, множество посещенных рекурсивных вершин D_0. Уравнения Eqs, описывающие состояния для рекурсивных участков CFG, определены глобально.

Данные: Множество состояний W и множество следующих состояний S.

```
1  W := {(l_0, true, λv.v, D_0)};
2  while W ≠ ∅ do
3  |  (l, pc, s, D) := pickNext(W); S := ∅;
   |  // Символьно исполним следующую инструкцию
4  |  S := ExecuteInstruction(l, pc, s);
   |  // Объединим состояния с эквивалентными из множества W
5  |  forall (l', pc', s') ∈ S do
6  |  |  W := NewJoin(W, l', pc', s', l_last, D_0, D);
7  |  end
8  end
9  if l_last ∈ RV then
10 |  guard_ε := true;
11 |  forall (pc, s) ∈ Eqs(l_last, D_0) do
12 |  |  guard_ε := ¬pc ∧ guard_ε;
13 |  end
14 |  Eqs(l_last, D_0) := Join(Eqs(l_last, D_0), l_last, guard_ε, λv.v, D_0);
15 else
16 |  print "no errors";
```

Листинг 3.1: Схема *Exec* композиционального символьного исполнения

Интуитивно, алгоритм соответствует множеству путей $\Pi(l_0, l_{last}, D_0)$. Он работает с множеством состояний, каждое из которых представляет собой кортеж (u, pc, s, D), где u — это номер текущей инструкции в графе потока управления, pc — условие пути до текущей инструкции, s — отображение переменных в символьные значения, D — множество встреченных на пути *рекурсивных* вершин (см. определение 3), которое регулирует передачу управления.

Главное отличие представленного алгоритма от предыдущего заключается в том, что, помимо поиска ошибок в программе, новый алгоритм строит уравнения Eqs для описания рекурсивных состояний для частей графа потока управления. Эти уравнения передаются в функцию $ExecuteInstruction$ (см. листинг 3.4) для определения выполнимости формул для условных переходов и инструкции проверки $assert$. Для того чтобы обозначить функцию-отображение переменных в символьные значения для рекурсивного состояния, вводится специальных *символ* — Rec, который параметризован текущей инструкцией l и множеством встреченных *рекурсивных* вершин D. Операция композиции двух состояний — $s \circ Rec(l, D)$ необходима, чтобы получить новое состояние, которое соответствует результату исполнения рекурсивного участка графа потока управления на состоянии, у которого отображение равно s.

Если алгоритм $Exec$ был вызван от рекурсивной вершины, то в конце своего выполнения он должен построить завершающий фрагмент уравнения на рекурсивное состояние. А именно — добавить «пустое» состояние, соответствующее исполнению, которое не вернулось в исходную инструкцию.

В алгоритм добавился вызов новой функции $NewJoin$ (см. листинг 3.2), которая решает, что делать с новым состоянием. Если исполнение дошло до номера «последней» инструкции l_{last}, то в таком случае эта инструкция является рекурсивной, и рекурсивное состояние $(l_{last}, pc', s' \circ Rec(l_{last}, D_0), D_0)$ добавляется в уравнения для *символа* $Rec(l_{last}, D_0)$. Если же номер инструкции рекурсивен, и исполнение еще не проходило через эту инструкцию, то создается рекурсивное состояние. Если для этого состояния не было постро-

ено уравнений, то запускается алгоритм $Exec(l', l', D)$.

Вход : Рабочее множество W, номер следующей инструкции l', условие пути pc', отображение переменных в символьные значения s', номер конечной инструкции l_{last}, исходное множество *рекурсивных* вершин D_0, текущее множество *рекурсивных* вершин D.
Выход : Новое рабочее множество состояний W.

```
1  if l' = l_last then
2  |   s' := s' ∘ Rec(l_last, D_0);
3  |   Eqs(l_last, D_0) := Join(Eqs(l_last, D_0), l_last, pc', s', D_0)};
4  |   return W;
5  elif l' ∈ RV ∧ l' ∈ D then  return W ;
6  elif l' ∈ RV then
7  |   D := D ∪ {l'};
8  |   s' := s' ∘ Rec(l', D);
9  |   if Eqs(l', D) ≠ ∅ then
10 |   |   Exec(l', l', D);
11 return Join(W, l', pc', s', D);
```

Листинг 3.2: Процедура NewJoin добавления нового состояния

В конце вызывается функция $Join$(см. листинг 3.3), для которой дополнительным параметром передается множество посещенных рекурсивных вершин. Очевидно, что необходимым условием эквивалентности двух состояний является равенство их множеств D.

Вход : Рабочее множество W, номер следующей инструкции l', условие пути pc', отображение переменных в символьные значения s', множество рекурсивных вершин D.
Выход : Новое рабочее множество состояний W.

```
1  if ∃(l', pc'', s'', D'') ∈ W : (l', pc'', s'') ~ (l', pc', s') ∧D'' = D  then
2  |   W := W \ {(l', pc'', s'')};
3  |   W := W ∪ {(l', pc' ∨ pc'', Merge(pc', pc'', s', s''), D )};
4  else
5  |   W := W ∪ {(l', pc', s', D)};
6  return W;
```

Листинг 3.3: Модифицированная процедура Join слияния нового состояния с имеющимся в W

```
Вход    : Номер инструкции l, условие пути pc, отображение переменных в
          символьные значения s,
          уравнения Eqs, описывающие состояния для рекурсивных участ-
          ков CFG.
Данные: Множество следующих состояний S.
1  S := ∅;
2  switch instr(l)
3      case v := e      // присваивание
4          S := {(succ(l), pc, s[v ↦ Eval(s, e)])};
5      case if(e) goto l'      // условный переход
6          if SAT(Eqs, s, pc ∧ e) then
7              S := {(l', pc ∧ e, s)};
8          if SAT(Eqs, s, pc ∧ ¬e) then
9              S := S ∪ {(succ(l), pc ∧ ¬e, s)};
10     case assert(e)      // проверка
11         if SAT(Eqs, s, pc ∧ ¬e) then abort;
12         else S := {(succ(l), pc, s)};
13     case halt      // завершение программы
14         print pc ;
15 return S;
```

Листинг 3.4: Модифицированная процедура ExecuteInstruction символьного исполнения инструкции языка

4. Архитектура и детали реализации символьного интерпретатора

4.1. Архитектура подсистемы интерпретаторов

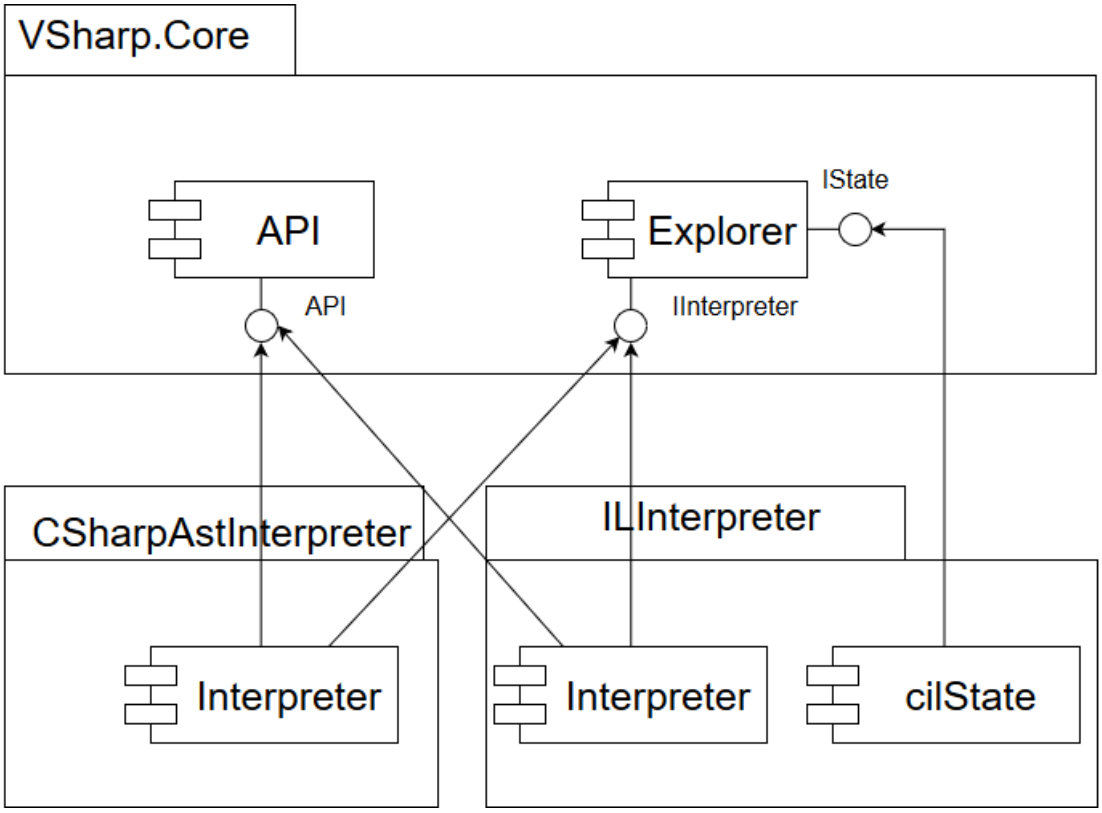

Рис. 3: Диаграмма компонентов проекта для подсистемы интерпретации

Естественным требованием при создании нового интерпретатора в ситуации, когда имеется другой интерпретатор, является разработка общей архитектуры для интерпретаторов. Диаграмма компонентов на рис. 3 представляет новую архитектуру. На ней представлены пакеты «VSharp.Core», «CSharpAstInterpreter» и «ILInterpreter». Пакет «VSharp.Core» содержит в себе компоненты «API» и «Explorer». Компонента «API» предоставляет интерфейс API, позволяющий интерпретаторам вызывать методы ядра, необходимые для осуществления символьного исполнения. Компонента

«Explorer» предоставляет интерфейсы IInterpreter и IState. Мотивация создания интерфейсов — потребность инкапсуляции единой высокоуровневой схемы алгоритма композиционального символьного исполнения 3.1 в ядре проекта «VSharp.Core» и требование не открывать «API» для таких операций как композиция и слияние двух состояний, поскольку эти операции довольно ресурсоемкие и трудозатратные. Пакет «CSharpAstInterpreter» содержит компоненту «Interpreter», которая использует интерфейс API и реализует интерфейс IInterpreter. Пакет «ILInterpreter» содержит компоненты «cilState» и «Interpreter». «cilState» реализует интерфейс IState, а «Interpreter» — интерфейс IInterpreter.

4.2. Детали реализации

Поскольку проект V# написан на языках C# и F#, и платформа .NET предоставляет модули для анализа своего скомпилированного кода (System.Reflection), реализацию[2] символьного интерпретатора языка CIL было решено проводить с помощью тех же технологий.

4.2.1. Построение графа потока управления

В конструктор интерпретатора ILInterpreter подается объект MethodBase. С помощью рефлексии можно извлечь из него тело метода, представляющее собой массив байтов. В стандарте ECMA 335 [1] описываются коды инструкций и размеры их аргументов, поэтому несложно декодировать данный массив в последовательность инструкций. По умолчанию, следующей инструкцией для исполнения является инструкция, следующая после текущей в массиве инструкций.

Инструкции потока управления (например, `br.s, switch, brfalse` и др.) могут влиять на передачу управления. Для них вычисляются все возможные следующие инструкции. Далее строится улучшенный граф потока управления, вершинами которого являются *базовые блоки*. Таким образом, полученный граф потока управления содержит меньше вершин.

[2] `https://github.com/kbatoev/VSharp/tree/ILFrontend`

4.2.2. Интерпретатор CIL

Интерпретатор ILInterpreter представляет собой реализацию интерфейса IInterpreter. Его состоянием для исследования является объект структуры cilState, которая хранит в себе элементы, специфичные для интерпретатора CIL: текущий номер вершины в графе, адрес назначения – номер рекурсивной вершины или значение для обозначения выхода из метода, объект-состояние из ядра проекта, множество посещенных *рекурсивных* вершин, которые ограничивают операцию перехода между вершинами и операционный стек, которым оперирует VES[3].

Основной интерес представляет собой реализация алгоритма обхода, который адаптирует метод описания всех путей в графе потока управления. Во-первых, в ядре проекта добавилась функция интерпретации (см. рис. 5), которая принимает объект-интерпретатор и состояние. Она задает схему интерпретации путем вызовов методов у объекта-интерпретатора и вызовов операций ядра.

Во-вторых, стратегия обработки рекурсии была расширена интерфейсом ICodeLocation (см. рис. 4). Если состояние интерпретатора CIL содержит *рекурсивную* вершину, то для нее создастся соответствующий объект ILCodePortion, реализующий интерфейс ICodeLocation. Состояние и объект ILCodePortion будут переданы в схему обработки рекурсии. В случае, если нужно раскручивать цикл, вернется то же самое состояние. Если же цикл раскручивать не нужно, то вызовется метод Invoke интерпретатора, имеющий собственную стратегию для анализа состояний цикла согласно алгоритму композиционального символьного исполнения без раскрутки отношения перехода.

4.2.3. Инструкции языка CIL

Во время работы было поддержано большинство инструкций языка CIL. Не была реализована схема обработки исключений и соответствующие ей инструкции.

[3]Virtual Execution System из стандарта ECMA 335

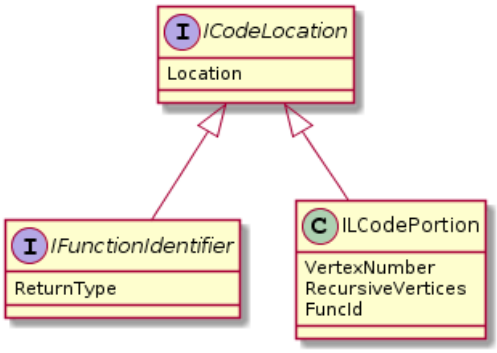

Рис. 4: Интерфейс ICodeLocation для элементов, для которых может производиться раскрутка отношения перехода

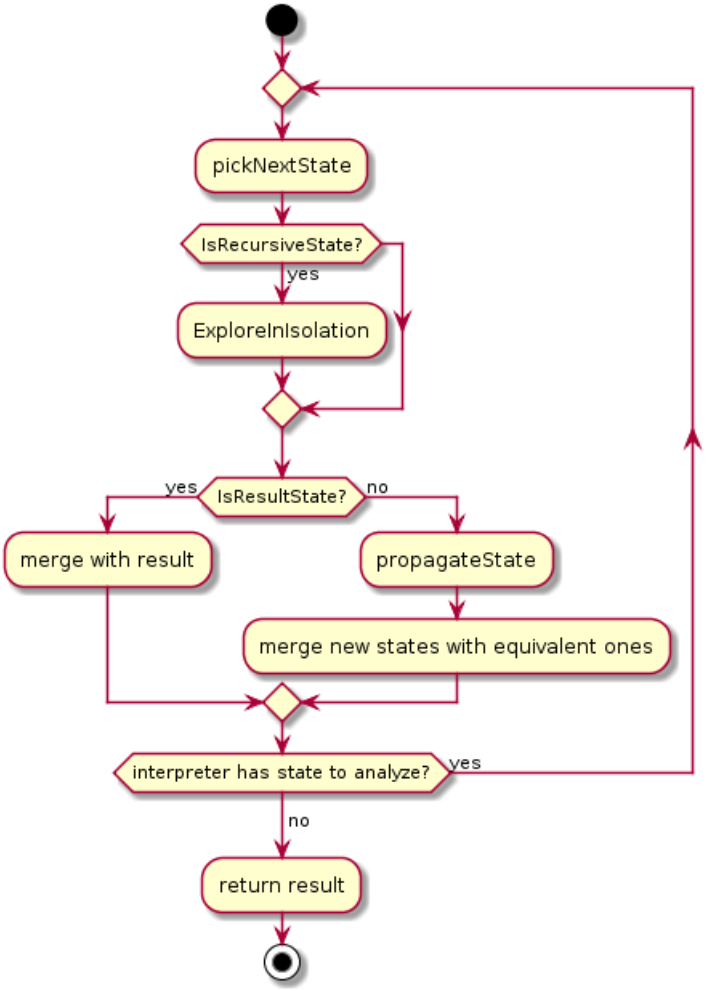

Рис. 5: Диаграмма активностей для схемы интерпретации в ядре проекта

5. Апробация

Тестирование нового интерпретатора проводилось на тестовой подсистеме проекта «VSharp.Test». Подсистема содержит тестовые наборы, затрагивающие различные конструкции и возможности языка C#: арифметику, логические операции, работу с массивами разных размерностей, генерирование исключений, тесты на классы и структуры, включающие взаимодействие со статическими членами и вызовы виртуальных методов, тесты с неограниченной рекурсией, тесты с *unsafe*-кодом, тесты со строками.

Таблица 1 показывает результаты проведенного тестирования. В наборах тестах с арифметикой и условными конструкциями была часть тестов с генерацией исключений. Поскольку схема обработки исключений для языка CIL не была реализована, то данные тесты были некорректно исполнены. По той же причине не было проведено тестирования на тестовым наборе «TryCatch», основное предназначение которого — инициирование и перехват исключений.

Количественные характеристики тестов			
Название тестового набора	Количество тестов в наборе	Количество успешно пройденных тестов	Количество инструкций CIL
Arithmetics	80	77	1968
Logics	75	75	1458
Conditional	10	6	943
Recursive	5	5	751
Lambdas	2	2	404
Generic	14	14	194
Strings	15	15	219
Unsafe	16	16	312
Typecast	19	19	969
Methods	10	10	238
Lists	9	9	1420
Всего	255	248	8876

Таблица 1: Результаты тестирования

6. Заключение

В рамках данной дипломной работы были выполнены следующие задачи.

- Проведен обзор алгоритмов символьной интерпретации программ.
- Разработан метод описания всех путей в графе потока управления, который использует множество *рекурсивных* вершин графа как ограничивающий параметр при выборе следующей вершины для передачи управления.
- Создан алгоритм для проведения композиционального символьного исполнения программы без раскрутки отношения перехода программы.
- Реализованы интерпретатор CIL-кода и общая архитектура интерпретаторов.
- Проведена апробация на тестовой подсистеме проекта, исключая тестовые наборы с обработкой исключений.

Список литературы

[1] Ecma TC39. TG3. Common Language Infrastructure (CLI). Standard ECMA-335, June 2005.

[2] Efficient state merging in symbolic execution / Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, George Candea // Acm Sigplan Notices. — 2012. — Vol. 47, no. 6. — P. 193–204.

[3] Enhancing symbolic execution with veritesting / Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, David Brumley // Proceedings of the 36th International Conference on Software Engineering / ACM. — 2014. — P. 1083–1094.

[4] Godefroid Patrice. Compositional dynamic test generation // ACM Sigplan Notices / ACM. — Vol. 42. — 2007. — P. 47–54.

[5] King James C. Symbolic execution and program testing // Communications of the ACM. — 1976. — Vol. 19, no. 7. — P. 385–394.

[6] A survey of symbolic execution techniques / Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia et al. // ACM Computing Surveys (CSUR). — 2018. — Vol. 51, no. 3. — P. 50.