Санкт-Петербургский государственный университет

Паршин Максим Алексеевич

Выпускная квалификационная работа

Управление ограничениями в символьной виртуальной машине V#

Уровень образования: бакалавриат

Направление *02.03.03 «Математическое обеспечение и администрирование информационных систем»*

Основная образовательная программа *СВ.5006.2018 «Математическое обеспечение и администрирование информационных систем»*

Профиль *Системное программирование*

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

Рецензент:
руководитель департамента анализа программ ООО «Техкомпания Хуавэй» Д.А. Иванов

Санкт-Петербург
2022

Saint Petersburg State University

Maksim Parshin

Bachelor's Thesis

Constraints management in V# symbolic virtual machine

Education level: bachelor

Speciality *02.03.03 "Software and Administration of Information Systems"*

Programme *CB.5006.2017 "Software and Administration of Information Systems"*

Profile: *Software Engineering*

Scientific supervisor:
C.Sc., docent D.A. Mordvinov

Reviewer:
software analysis teamlead at LLC Tech Company Huawei D.A. Ivanov

Saint Petersburg
2022

Оглавление

Введение **5**

1. Постановка задачи **7**

2. Обзор предметной области **8**

2.1. Символьное исполнение 8

2.2. SMT-решатели 11

2.3. Архитектура проекта V# 12

2.4. Модель символьной памяти в V# 13

3. Независимое управление ограничениями **15**

3.1. Существующие реализации 16

3.2. Подход к реализации в V# 17

4. Кэширование моделей SMT-решателя **19**

4.1. Существующие реализации 19

4.2. Подход к реализации в V# 21

5. Инкрементальное использование SMT-решателя **22**

5.1. Инкрементальность на основе стека 23

5.2. Инкрементальность на основе предпосылок 24

5.3. Существующие реализации 26

5.4. Подход к реализации в V# 26

6. Особенности реализации **28**

6.1. Независимое управление ограничениями 28

6.2. Кэширование моделей SMT-решателя 30

6.3. Инкрементальное использование SMT-решателя 31

7. Эксперименты **32**

7.1. Тестовые данные 32

7.2. Исследовательские вопросы 32

7.3. Результаты 33

Заключение **37**

Список литературы **38**

Введение

Качественное тестирование достаточно сложной, чтобы быть полезной не только для ее создателя, программы — это, как правило, трудоёмкая задача. Тестирование требует значительного количества времени, которое не всегда удаётся выделить в реальном процессе разработки. При этом даже после тестирования и отладки в программном коде могут оставаться незамеченные специалистами уязвимости, приводящие к ошибкам или нежелательному поведению только при исполнении на специфичных комбинациях входных данных. Для решения данных проблем часто применяются инструменты автоматической генерации тестов, в том числе инструменты тестирования «белого ящика» (которым доступен исходный код программы, в отличие от тестирования «черного ящика»). В основе таких инструментов лежат различные механизмы статического анализа кода, один из которых — механизм символьного исполнения.

Техники символьного исполнения кода заключаются в исполнении программы не на конкретных значениях входных данных, а на так называемых символьных переменных. Это даёт возможность для каждого пути исполнения получить логические ограничения на значения входных параметров, при выполнении которых этот путь достигается. После этого при помощи SMT[1]-решателя такие значения могут быть найдены и сгенерирован тест, реализующий данный путь.

Механизм символьного исполнения обладает таким достоинством, как теоретически полное покрытие кода, что даёт возможность гарантировать выполнение некоторых условий программой (например, что она никогда не выбрасывает Null Pointer Exception). В то же время необходимость анализировать все возможные пути исполнения, число которых растет экспоненциально, влечёт его главный недостаток — высокую вычислительную сложность. К этому также следует добавить NP-трудность задачи выполнимости формулы в теориях, к которой, как правило, сводится задача поиска значений входных данных.

[1]Satisfiability Modulo Theory, задача выполнимости формулы в теориях

В качестве примера, иллюстрирующего эффективность и сложность символьного исполнения, можно привести использование компанией Microsoft символьной машины SAGE для воспроизведения ошибки переполнения стека в ОС Windows [15]. Ошибка в коде, отвечающем за чтение файлов анимированных курсоров, приводила к возможности удаленного исполнения произвольного кода злоумышленником[2]. В отличие от других инструментов статического анализа, SAGE удалось сгенерировать файл, чтение которого приводило к ошибке, но на это понадобилось семь с половиной часов на машине с производительностью среднего ПК.

Таким образом, создание практически полезного инструмента, основанного на механизме символьного исполнения, невозможно без использования различных оптимизаций и эвристик. Существует большое количество подходов к управлению ограничениями в символьных машинах, позволяющих минимизировать число запросов к SMT-решателю, увеличить переиспользование уже полученных в процессе исполнения результатов и исключить определённые пути исполнения.

V#[3] — это символьная машина для программ на платформе .NET с открытым исходным кодом, которая разрабатывается с целью удовлетворить текущую потребность[4] в системе автоматической генерации тестов для наиболее современных фреймворков .NET и .NET Core, используя при этом наиболее современные подходы к оптимизации процесса символьного исполнения.

[2]https://docs.microsoft.com/en-us/security-updates/securitybulletins/2007/ms07-017 Дата обращения: 26.04.2022

[3]https://github.com/VSharp-team/VSharp Дата обращения: 03.05.2022

[4]https://developercommunity.visualstudio.com/t/add-intellitest-support-for-net-corestandard/359250 Дата обращения: 26.04.2022

1. Постановка задачи

Целью данной работы является реализация оптимизаций управления ограничениями в символьной виртуальной машине V#. Для достижения цели были сформулированы следующие задачи.

- Провести обзор оптимизаций управления символьными ограничениями: техник независимого управления ограничениями и кэширования моделей SMT-решателя, подходов к инкрементальному использованию SMT-решателя.
- Реализовать независимое управление ограничениями в символьной виртуальной машине V#.
- Реализовать кэширование моделей SMT-решателя в символьной виртуальной машине V#.
- Реализовать один из подходов к инкрементальному использованию SMT-решателя в символьной виртуальной машине V#.
- Провести эксперименты для определения эффективности реализованных оптимизаций.

2. Обзор предметной области

В данной главе приводится общее описание механизма символьного исполнения и рассматриваются некоторые особенности его реализации для платформы .NET в символьной виртуальной машине V#.

2.1. Символьное исполнение

Несмотря на то, что реальные символьные машины устроены с учётом таких особенностей исполняемого кода, как поддержка классов, механизмов ввода-вывода и т. д., они используют общую концепцию абстрактного **символьного исполнителя**.

Рассмотрим механизм работы символьного исполнителя на практическом примере. Функция `Foo` (листинг 1) принимает на вход три аргумента. Требуется ответить на следующие вопросы. Существуют ли конкретные значения данных аргументов, при которых функция `Foo` выбрасывает исключение (в строке 12)? И если такие значения существуют, то каковы они?

Листинг 1: Функция Foo

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

Перед исполнением кода каждому из входных параметров программы сопоставляется собственная **символьная переменная** (к примеру, аргументу x сопоставляется символьная переменная α_x (рис. 2)). После этого значения переменных[5] программы могут быть выражены через символьные переменные.

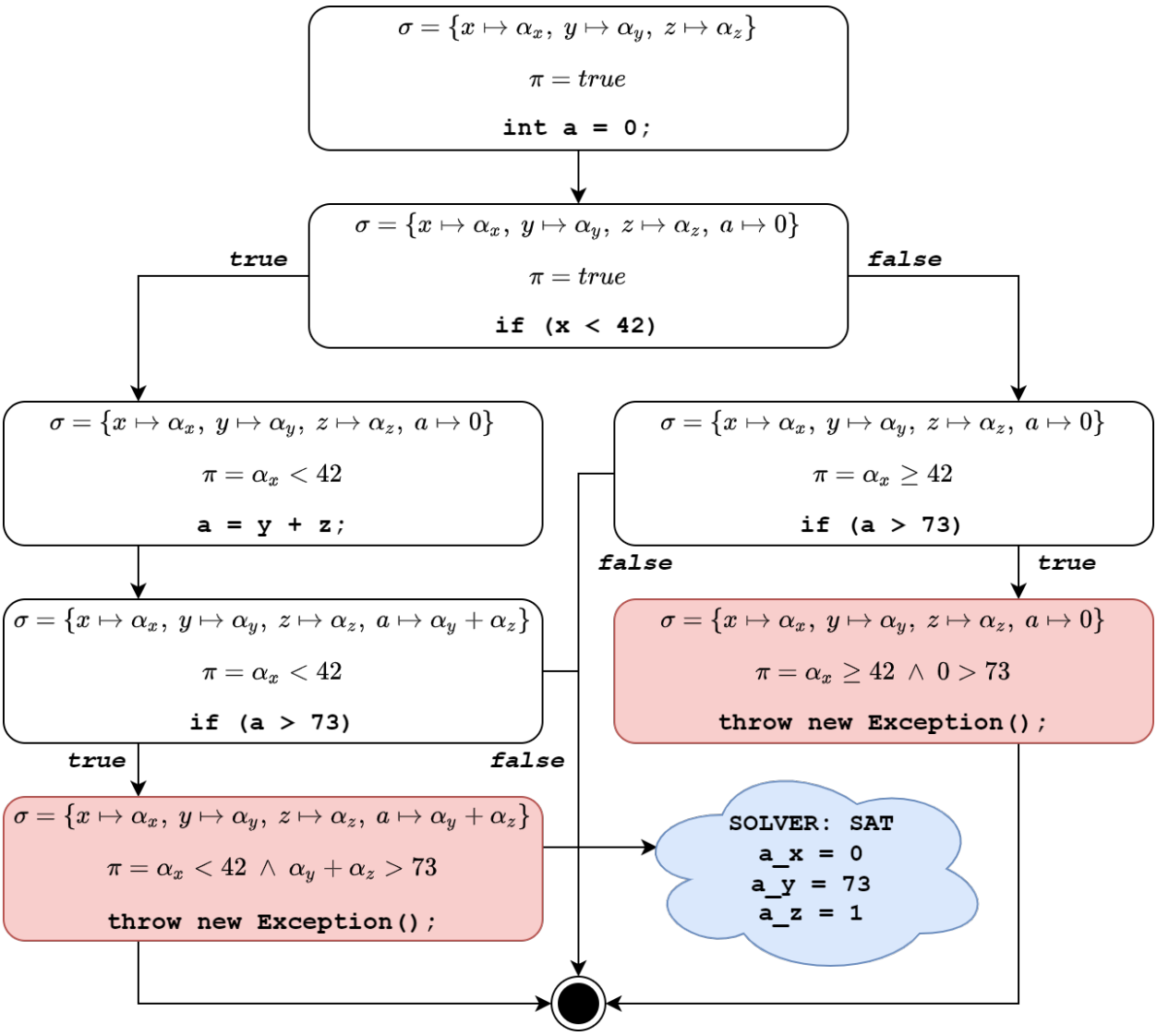

Рис. 1: Схема символьного исполнения функции `Foo`

В результате состояние символьного исполнителя можно определить как тройку $(stmt, \sigma, \pi)$ [23].

- $stmt$ — текущая исполняемая инструкция.

[5]Имеются в виду не символьные переменные, а переменные в языке программирования

- σ — **символьная память**, которая хранит в себе текущие значения переменных программы, выраженные через символьные переменные.

- π — **условие пути** [6], логическая формула, представляющая собой условие на значения символьных переменных, выполняемое на текущем пути исполнения. Таким образом, чтобы исполнилась инструкция $stmt$, необходимо, чтобы конкретные значения символьных переменных, соответствующих входным параметрам, удовлетворяли π. Начальное значение — $true$.

Символьная машина исполняет код инструкция за инструкцией, обновляя $stmt$ и σ (рис. 2). При возникновении ветвления текущее состояние исполнителя копируется для каждого пути исполнения. При этом для каждой копии за π принимается конъюнкция текущего π и условия вхождения в соответствующую «ветку». Исполнение продолжается для каждого из путей независимо.

В результате символьного исполнения функции `Foo` возникают два состояния, в которых $stmt$ = `throw new Exception();`. Если функция будет исполнена с аргументами, значения которых удовлетворяют условиям π на соответствующие им символьные переменные, будет выброшено исключение в строке 12. При этом можно заметить, что одна из формул π никогда не выполняется (поскольку содержит заведомо ложное условие $0 > 73$).

Для того, чтобы определить, выполнима ли вторая формула, используется SMT-решатель. В данном случае решатель вернёт ответ `SAT`[7] и один из наборов конкретных значений символьных переменных, на которых формула принимает значение $true$. Такой набор называется **моделью**.

Таким образом, установлено, что функция `Foo` выбрасывает исключение, если вызвать её с аргументами 0, 73 и 1.

[6] англ. *path condition*

[7] англ. *satisfiable*, выполнима

2.2. SMT-решатели

Задача поиска значений символьных переменных, при которых условие пути выполняется, может быть представлена как задача выполнимости в формулы в теориях (Satisfiability Modulo Theories, SMT) [12, 26].

Задача SMT является обобщением NP-полной задачи выполнимости булевых формул (SAT) [7] и отличается от неё тем, что формула, выполнимость которой проверяется, является формулой логики первого порядка в сигнатуре определенной теории. В частности, для того, чтобы при символьном исполнении проверять выполнимость формул над такими объектами как списки или машинные числа, решаются задачи SMT в теории массивов [22] или битовых векторов [2].

Для решения задачи SMT в определённой теории T как правило используется комбинация SAT-решателя, такого как CDCL [18, 19] или DPLL [9], и решателя данной теории. Такие комбинации носят название DPLL(T)[8] и в зависимости от «тесноты» интеграции двух алгоритмов делятся на «ленивые»[9] и «жадные»[10] [8].

Чтобы унифицировать интерфейсы различных SMT-решателей, был разработан стандарт SMT-LIB [1]. Данный стандарт помимо прочего описывает формат команд для взаимодействия с решателями в командной строке[11]:

- `assert c` — добавляет новое ограничение c;
- `check-sat` — проверяет выполнимость конъюнкции всех добавленных ограничений и возвращает ответ `sat`[12] или `unsat`[13] ;
- `check-sat c_1 ... c_i` — проверяет выполнимость конъюнкции ограничений $c_1, ..., c_i$.

[8]При этом такое название часто носят и алгоритмы, фактически использующие CDCL [13]

[9]англ. *lazy*

[10]англ. *eager*

[11]В V# взаимодействие с решателем происходит через API для .NET: https://github.com/Z3Prover/z3/tree/master/examples/dotnet Дата обращения: 03.05.2022

[12]англ. *satisfiable*, выполнима

[13]англ. *unsatisfiable*, невыполнима

Символьная виртуальная машина V# работает с SMT-решателем Z3 [27], разработанным Microsoft.

2.3. Архитектура проекта V#

Проект V# включает в себя символьную виртуальную машину для кода на IL (Intermediate Language), промежуточного языка, в который компилируются программы на высокоуровневых языках платформы .NET [10], и основанную на ней подсистему генерации тестов.

Основным языком реализации символьной виртуальной машины является F#. Пользовательские интерфейсы реализованы на языке C#.

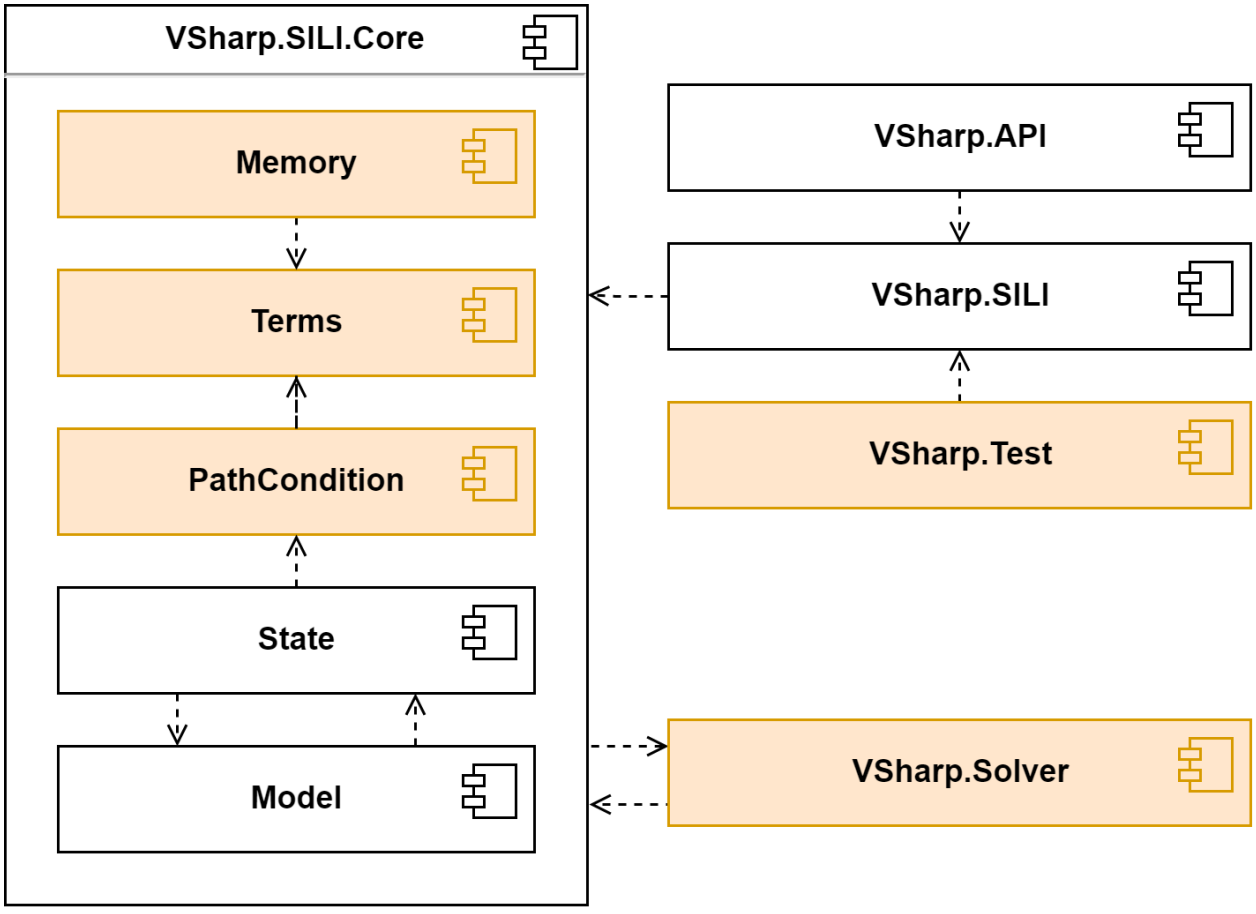

Рис. 2: Диаграмма основных модулей проекта V#. Цветом отмечены модули, затронутые в данной работе

Следует отметить следующие модули проекта.

- **VSharp.API**. Интерфейс взаимодействия конечного пользователя с символьной машиной и подсистемой генерации тестов.

- **VSharp.SILI (Symbolic Intermediate Language Interpreter)**. Интерпретатор кода на IL, который генерирует символьное представление IL (SIL, Symbolic Intermediate Language) и делегирует его исполнение модулю SILI.Core.
- **VSharp.SILI.Core**. Реализация символьного представления IL и логика его исполнения.
  - **Memory**. Операции для работы с символьной памятью, в том числе с символьными стеком и кучей.
  - **Terms**. Термы, символьные представления конструкций IL.
  - **PathCondition**. Условие пути. Представляет собой конъюнкцию набора термов-ограничений.
  - **State**. Состояние символьного исполнителя. Включает в себя среди прочего текущие условие пути, состояние символьной памяти и модель.
  - **Model**. Модель, содержащая конкретные значения символьных переменных, при которых достигается определённый путь исполнения.
- **VSharp.Solver**. Интерфейс взаимодействия с SMT-решателем Z3.
- **VSharp.Test**. Набор синтетических тестов для проверки функциональности символьной машины.

2.4. Модель символьной памяти в V#

При символьном исполнении реальных программ приходится решать вопрос о том, как представлять память программы в недетерминированном виде. Для этого применяются различные подходы, которые называются моделями символьной памяти [28].

Для символьного представления памяти в V# используется модель регионов. Данная модель позволяет учитывать, что в среде .NET определённые участки памяти априори не могут пересекаться между собой.

Например, переменная, выделенная на стеке, не может располагаться в памяти в том же месте, где выделен объект на куче, даже если её точное расположение ещё неизвестно. Аналогично не могут совпадать адреса, по которым расположены два объекта на куче, если их типы не связаны иерархией наследования. В результате сокращается множество допустимых значений символьных адресов, что уменьшает число альтернативных путей исполнения.

Регион фактически представляет собой множество объектов некоторого типа с операциями сравнения, пересечения и разности. В V# используются, например, такие регионы:

- `intervals<'a>` — объединение интервалов;
- `points<'a>` — множество точек;
- `productRegion<'a, 'b>` — декартово произведение регионов.

Все операции записи в регионы сохраняются в структуре данных, называемой деревом регионов. В узлах этого дерева лежат пары «регион-записанное значение», и поддерживаются два инварианта:

- все регионы-потомки являются подмножествами региона-предка;
- все регионы, которые являются прямыми потомками одного региона-предка, не пересекаются между собой.

Данная структура позволяет быстро получить историю всех записей, которые были сделаны в определённый регион памяти.

3. Независимое управление ограничениями

Независимое управление ограничениями — это техника оптимизации символьного исполнения, основанная на том, что множество ограничений условия пути может быть разбито на несколько непересекающихся множеств таким образом, что ограничения одного множества будут независимы от ограничений во всех других [23]. Это позволяет подавать на вход SMT-решателю не полную формулу условия пути, а более короткую формулу, состоящую только из ограничений, содержащихся в одном из подмножеств.

Рассмотрим в качестве примера следующее условие пути с целыми символьными переменными a, b и c:

$$(a > 0) \wedge (a < 73 \vee b \geq 0) \wedge (c = 42)$$

Нетрудно заметить, что для проверки выполнимости данного условия не обязательно проверять выполнимость всей формулы целиком. Вместо этого можно применить SMT-решатель для формул $(a > 0) \wedge (a < 73 \vee b \geq 0)$ и $(c = 42)$ независимо, а затем объединить получившиеся модели, если все формулы оказались выполнимы, либо сделать вывод о невыполнимости всего условия пути, если была показана невыполнимость хотя бы одной из независимых формул.

Данная оптимизация становится более полезной при совместном использовании с другими техниками. К примеру, при реальном исполнении в символьной машине V# SMT-решатель применяется не в конце какого-либо пути, как было показано в примере абстрактного исполнителя, а при возникновении нового ограничения (то есть ветвления в коде), что позволяет сразу же отбрасывать заведомо невыполнимые пути. При независимом управлении ограничениями в данной ситуации стало бы возможно проверять выполнимость только той формулы, которая соответствует подмножеству, содержащему новое ограничение.

3.1. Существующие реализации

Одним из первых инструментов, использующих независимое управление ограничениями, стала символьная машина EXE[14] [11] для кода на языке C. Независимые подмножества символьных переменных в EXE моделируются как компоненты связности графа, вершины которого соответствуют переменным. Ребро между вершинами есть в том случае, если соответствующие им переменные содержатся в одном ограничении. Для представления такого графа используется система непересекающихся множеств [16], которая обновляется при добавлении нового ограничения (то есть возникновении ветвления).

Поскольку в EXE символьные данные любого типа представляются как массивы битовых векторов, для корректного определения независимости переменных также требуется рассмотреть несколько особых случаев.

1. Чтение из массива может происходить по символьному индексу, что не позволит однозначно судить о независимости значений в нем.

2. Для каждой операции чтения из массива хранится список всех значений, которые были записаны в него до этого, что влечет косвенную зависимость данных значений.

Данные вопросы решены в EXE следующим образом: все «спорные» значения «с запасом»[15] считаются зависимыми. В первом случае в одно подмножество добавляются все элементы массива, во втором — все элементы списка.

Авторы EXE также отмечают преимущества независимого управления ограничениями: во-первых, запросы к SMT-решателю[16] часто становятся короче и «дешевле», во-вторых, гранулярность множества ограничений увеличивает число запросов к кэшу, и вызов решателя не требуется вовсе.

[14]Независимое управление ограничениями в EXE носит название «constraint independence»

[15]англ. *conservative*

[16]В EXE используется SMT-решатель STP [14]

Техника независимого управления ограничениями было заимствована у EXE как её прямым наследником KLEE [5], так и другими символьными машинами.

В символьной машине Sydr [24], разрабатываемой Институтом системного программирования Российской академии наук, для независимого управления ограничениями применяется алгоритм «вырезки»[17]. Данный алгоритм аналогично EXE выполняет поиск компоненты связности графа символьных переменных, но делает это явным образом каждый раз при возникновении ветвления, не поддерживая систему непересекающихся множеств.

Символьная машина Pex для .NET Framework также осуществляет независимое управление ограничениями [25].

3.2. Подход к реализации в V#

При определении независимости символьных переменных в V# следует учитывать, что типы в .NET разделяются на ссылочные типы и типы-значения [10]. Чтобы установить независимость переменных типов-значений, достаточно проверить, что они содержатся в различных компонентах графа переменных, как это происходит, например, в EXE. Однако для ссылочных типов только такого определения независимости переменных недостаточно. К примеру, исполняемая символьно функция на C# может получать на вход аргументы a и b типа $int[]$, которые будут обозначаться разными символьными переменными. Если рассматривать независимость переменных лишь в контексте достижимости в графе, то, например, для следующего условия пути: $(a[1] = 42) \wedge (b[1] = 73)$ будет получено разбиение на два подмножества $\{(a[1] = 42)\}$ и $\{(b[1] = 73)\}$. В реальности же $a[1]$ и $b[1]$ могут указывать на одну и ту же ячейку памяти (если в качестве аргументов a и b передан один и тот же объект), и следовательно, не являться фактически независимыми.

Модель памяти V# предоставляет возможность легко проверить,

[17] англ. *path predicate slicing*

пересекаются ли области памяти, значения из которых могут принимать переменные ссылочных типов. Для регионов определена операция пересечения, и чтобы выяснить, являются ли символьные переменные действительно независимыми, следует убедиться, что соответствующие им регионы памяти не пересекаются.

Другая особенность независимого управления ограничениями заключается в том, что SMT-решатель получает на вход лишь подмножество условия пути и возвращает в случае выполнимости модель только для символьных переменных из этого подмножества. Значения остальных символьных переменных следует взять из текущей модели, поскольку они продолжают удовлетворять условию пути.

4. Кэширование моделей SMT-решателя

Если символьный исполнитель имеет доступ к текущей модели, и в возникающем ограничении нет новых переменных, то появляется возможность избежать одного запроса к SMT-решателю при ветвлении.

Пусть в некоторый момент условием пути является формула π, и m — соответствующая модель, то есть такой набор значений переменных, при которых условие пути истинно, или $\pi(m) = true$. Когда возникает ограничение c, в котором нет новых символьных переменных, можно вычислить $c(m)$. Если $c(m) = true$, то по определению модели $(\pi \wedge c)(m) = \pi(m) \wedge c(m) = true$. Если же $c(m) = false$, то $(\pi \wedge \neg c)(m) = \pi(m) \wedge (\neg c)(m) = \pi(m) \wedge \neg(c(m)) = true$. Таким образом, для одной из ветвей исполнения m остаётся актуальной моделью, и запрос к решателю требуется сделать только для другой ветви.

Следует отметить, что данная оптимизация и независимое управление ограничениями нацелены на практически взаимоисключающие случаи. А именно: если в ограничении нет новых переменных, то применима описанная оптимизация, иначе же велика вероятность того, что новая переменная относится к новому независимому подмножеству, и применимо независимое управление ограничениями. В связи с этим данные оптимизации могут эффективно дополнять друг друга.

4.1. Существующие реализации

Оптимизация символьного исполнения, основанная на использовании ранее полученных моделей и носящая название «counter-example cache», реализована в символьной машине KLEE. Данная оптимизация подразумевает поддержание глобального кэша, хранящего сопоставления формулам удовлетворяющих им моделей. В случае, если формула невыполнима, в кэш сохраняется соответствующий флаг.

Значения, хранящиеся в кэше, используются по следующим принципам.

1. Если подмножество ограничений невыполнимо, но невыполнимо

и всё множество ограничений.

2. Если множество ограничений выполнимо, то с той же моделью выполнимо и его подмножество.

3. Если подмножество ограничений выполнимо, то велика вероятность того, что и всё множество ограничений выполнимо с той же моделью.

Для эффективной работы с множествами используется специальная структура данных на основе UB-дерева.

Проведённые авторами KLEE эксперименты показали, что число запросов к SMT-решателю при использовании оптимизации «counter-example cache» сокращается на 40%, а при одновременном использовании «counter-example cache» и независимого управления ограничениями — на 95%.

Символьное исполнение на основе конкретизации состояний[18] — подход, непосредственно основанный на хранении модели вместе с условием пути, описан в [21]. По аналогии с понятием символьной памяти вводится понятие конкретной памяти[19], в которой содержатся конкретные значения символьных переменных. Если данные значения удовлетворяют условию пути, то конкретная память называется консистентной.

При возникновении ветвления происходит подстановка значений из конкретной памяти в новое ограничение, как это было описано выше. Для ветви, в которой конкретная память оказывается неконсистентной, выполняется операция реконкретизации[20] — вызов SMT-решателя и обновление конкретной памяти согласно полученной от него модели.

Авторы SCSE реализовали данный подход в символьной машине KLEE и провели эксперименты, сравниваясь по результатам в том числе и с оптимизацией «counter-example cache». Время работы по сравнению с символьным исполнением без оптимизаций уменьшилось во всех

[18]SCSE, State Concretization based Symbolic Execution

[19]англ. *concrete store*

[20]англ. *reconcretization*

случаях (в среднем в 2,46 раза), по сравнению с «counter-example cache» — в 24 случаях из 33 (в среднем в 1,42 раза).

4.2. Подход к реализации в V#

На момент написания данной работы состояние символьного исполнителя в V# уже хранило в себе текущую модель, используемую для генерации тестов. В связи с этим реализация подхода, схожего с подходом SCSE, стала естественным шагом.

5. Инкрементальное использование SMT-решателя

Одна из особенностей использования SMT-решателя при символьном исполнении заключается в том, что формулы, подаваемые ему на вход, часто имеют значительную общую часть. Это происходит естественным образом, поскольку возникающие в процессе исполнения ограничения постепенно, или *инкрементально*, добавляются к текущему условию пути (рис. 3).

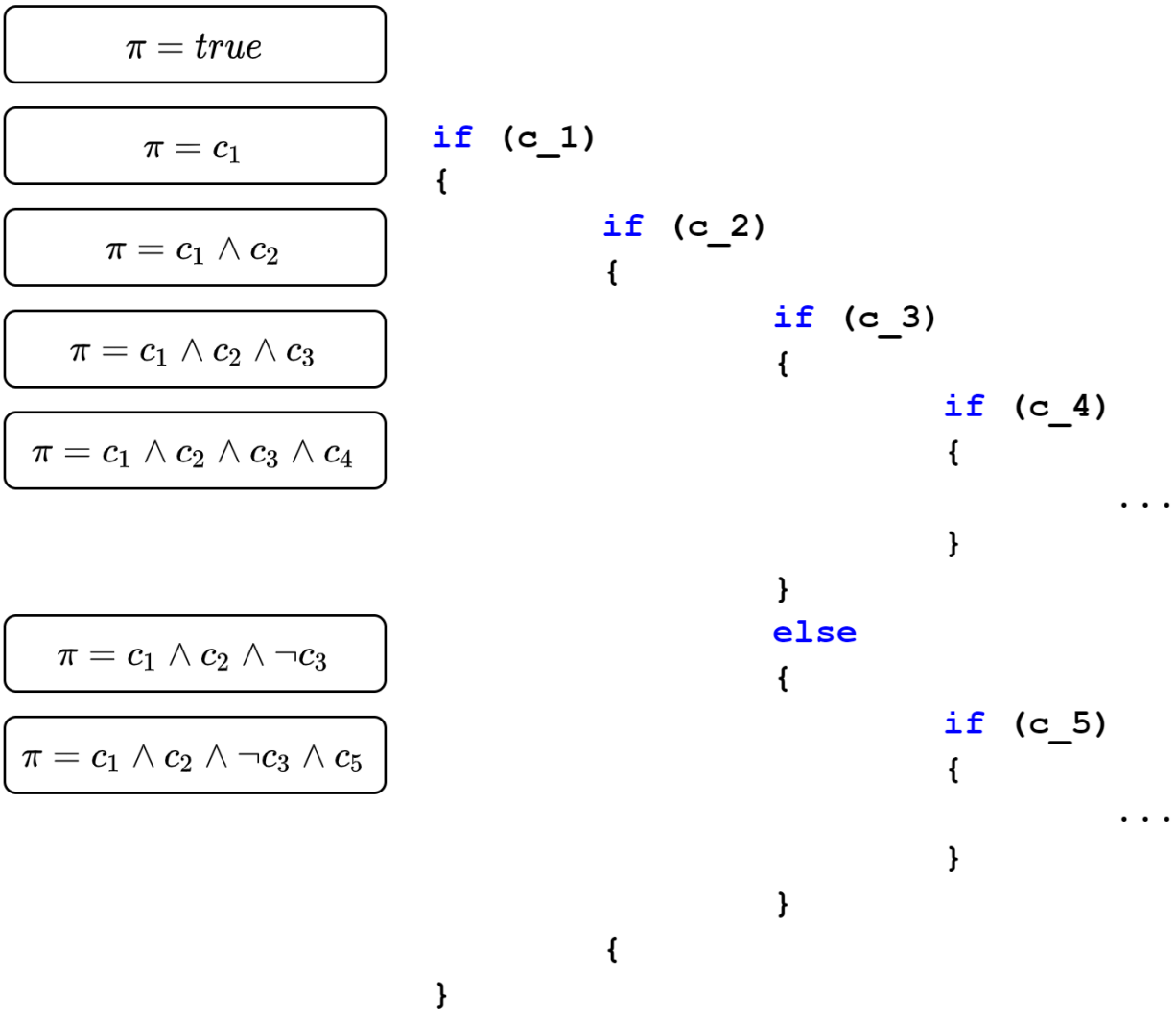

Рис. 3: Схема, показывающая изменение условия пути при возникновении ветвлений

Если не принимать во внимание данное замечание, то при символьном исполнении примера на рисунке 3 SMT-решатель получит следую-

щие команды:

```
(check-sat c_1)
(check-sat (and c_1 c_2))
(check-sat (and c_1 (and c_2 c_3)))
(check-sat (and c_1 (and c_2 (and c_3 c_4))))
(check-sat (and c_1 (and c_2 (not c_3))))
(check-sat (and c_1 (and c_2 (and (not c_3) c_5))))
```

Каждое из ограничений будет решено отдельно, и все леммы[21], выведенные в процессе решения, будут «забыты» решателем.

Однако существуют и другие подходы к использованию SMT-решателя, которые позволяют переиспользовать полученные ранее результаты, и тем самым учитывать, что ограничения имеют общую часть. В контексте SMT-решателя Z3 существует два таких подхода: на основе стека и на основе предпосылок[22] [4].

5.1. Инкрементальность на основе стека

Стандарт SMT-LIB [1] описывает команды *push* и *pop*, которые позволяют оперировать с ограничениями в манере стека. Так, команда *push* создаёт новый «кадр», и все ограничения, добавляемые далее, действуют только в контексте данного «кадра». Если вызвать команду *pop*, то «кадр» будет удалён, и решатель вернётся в исходное состояние. При использовании данного подхода взаимодействие с SMT-решателем для примера на рисунке 3 может выглядеть следующим образом:

```
(push)
(assert c_1) ; pi = c_1
(check-sat)

(push)
```

[21] Леммы — это некоторые утверждения из теории, которые доказываются в процессе работы алгоритма решения и затем используются им для определения дальнейших шагов

[22] англ. *assumptions*

```
(assert c_2) ; pi = c_1 & c_2
(check-sat)

(push)
(assert c_3) ; pi = c_1 & c_2 & c_3
(check-sat)

(push)
(assert c_4) ; pi = c_1 & c_2 & c_3 & c_4
(check-sat)

(pop) ; pi = c_1 & c_2 & c_3
(pop) ; pi = c_1 & c_2

(push)
(assert (not c_3)) ; pi = c_1 & c_2 & !c_3
(check-sat)

(push)
(assert c_5) ; pi = c_1 & c_2 & !c_3 & c_5
(check-sat)
```

Для каждого ветвления командой *push* создаётся новый «кадр», затем командой *assert* утверждается соответствующее условие и командой *check-sat* проверятся выполнимость текущего набора утверждений. При выходе из «ветки» утверждение и все доказанные в «кадре» леммы сбрасываются командой *pop* [4].

Подход на основе стека позволяет SMT-решателю переиспользовать леммы, доказанные для общей подформулы.

5.2. Инкрементальность на основе предпосылок

Другой подход к инкрементальности заключается в том, чтобы постоянно поддерживать общее множество утверждений-предпосылок и

формировать проверяемые формулы из них. При этом леммы, доказанные в процессе решения, также будут сформулированы в терминах данных утверждений, что позволит алгоритму использовать их повторно в будущем.

Взаимодействие с решателем при использовании данного подхода на рассмотренном примере (рис. 3) может выглядеть следующим образом:

```
(declare-const p_1 Bool)
(assert (=> p_1 c_1))
(check-sat p_1)

(declare-const p_2 Bool)
(assert (=> p_2 c_2))
(check-sat p_1 p_2)

(declare-const p_3 Bool)
(assert (=> p_3 c_3))
(check-sat p_1 p_2 p_3)

(declare-const p_4 Bool)
(assert (=> p_4 c_4))
(check-sat p_1 p_2 p_3 p_4)

(declare-const p_5 Bool)
(assert (=> p_5 (not c_3)))
(check-sat p_1 p_2 p_5)

(declare-const p_6 Bool)
(assert (=> p_6 c_5))
(check-sat p_1 p_2 p_5 p_6)
```

Для каждого из условий командой *declare-const* создаётся собственная булева переменная, из истинности которой следует данное условие.

Когда необходимо проверить выполнимость формулы, она выражается через введённые переменные.

5.3. Существующие реализации

Подходы к инкрементальному использованию SMT-решателя были реализованы в рамках работ [6, 17].

В частности, в [6] было проведено сравнение эффективности инкрементальности на основе стека и кэширования результатов SMT-решателя в связке с независимым управлением ограничениями, реализованных авторами в KLEE. Также была рассмотрена модификация стекового подхода, позволяющая решателю переиспользовать общие подвыражения. При использовании инкрементальных подходов время исполнения уменьшилось в среднем в пять раз по сравнению с кэшированием.

Следует отметить, что во всех экспериментах был применен обход графа потока управления в глубину.

Работа [17] рассматривает инкрементальное использование SMT-решателя в контексте композиционального символьного исполнения [23]. Авторы акцентируют внимание на том, что подход на основе стека напрямую применим только при обходе графа исполнения в глубину, но механизм композиционального символьного исполнения позволяет реализовать схожие с ним и при этом эффективные стратегии обхода. На основании данной идеи авторами была разработана символьная машина CSE. Подход на основе предпосылок также был использован.

Было проведено сравнение количества запросов к SMT-решателю для KLEE и CSE. Композициональный подход с инкрементальностью сократил число запросов примерно на 20%.

5.4. Подход к реализации в V#

Было принято решение реализовать в V# подход к инкрементальности на основе предпосылок.

Подход к инкрементальности на основе стека показывает свою эффективность при символьном исполнении [6], однако основывается на предположении о том, что обход графа потока управления происходит в глубину. На практике символьные машины, в том числе и V#, используют различные стратегии и эвристики обхода, например, обход в ширину. В дополнение к этому, в V# планируется интегрировать механизм двунаправленного символьного исполнения [20], при котором обход графа потока управления происходит в том числе и в обратном направлении. Из данных соображений следует невозможность непосредственного применения подхода к инкрементальности на основе стека в V#. Тем не менее, могут быть предложены модификации данного метода, учитывающие особенности стратегий обхода.

Другой аргумент в пользу подхода на основе предпосылок заключается в том, что для решателей некоторых теорий в Z3 подход на основе стека изначально реализован через предпосылки [4].

6. Особенности реализации

В данной главе приводятся детали реализации рассмотренных оптимизаций в символьной виртуальной машине V#.

Работа над оптимизациями велась в частях проекта, содержащих логику работы символьной машины, в связи с чем языком реализации является F#.

С целью обеспечения удобства тестирования и гибкости настройки процесса исполнения для каждой из оптимизаций при реализации была предусмотрена возможность её включения и отключения. В интерфейс командной строки генератора тестов V# добавлены ключи, соответствующие оптимизациям.

6.1. Независимое управление ограничениями

Для поддержки независимого управления ограничениями был переработан модуль *PathCondition*, содержащий логику работы с условием пути.

Листинг 2: Интерфейс условия пути

```
1  type public IPathCondition =
2      abstract Add : term →unit
3      abstract Copy : unit →IPathCondition
4      abstract ToSeq : unit →term seq
5      abstract UnionWith : IPathCondition →IPathCondition
6      abstract Map : (term →term) →IPathCondition
7      abstract IsEmpty : bool
8      abstract IsFalse : bool
9      abstract Fragments : IPathCondition seq
10     abstract Constants : term seq
```

Чтобы обеспечить возможность включения и отключения оптимизации по желанию пользователя, выделен интерфейс `IPathCondition` (листинг 2) и добавлены две его реализации: `PathCondition` и `IndependentPathCondition`. В `PathCondition` была перенесена суще-

ствующая реализация условия пути, фактически работающая как простое множество термов-ограничений — без учёта зависимостей между переменными. `IndependentPathCondition`, в свою очередь, является новой реализацией условия пути, поддерживающей независимые подмножества ограничений. Символьный исполнитель работает с абстракцией `IPathCondition`.

Следует отметить следующие методы, предоставляемые интерфейсом.

- `Add` — добавляет в условие пути новое ограничение. В `IndependentPathCondition` при этом происходит поиск символьных переменных в данном ограничении и обновление системы независимых множеств переменных.
- `Fragments` — для `IndependentPathCondition` возвращает набор экземпляров `IPathCondition`, каждый из которых содержит только ограничения из одного независимого подмножества. Для `PathCondition` возвращает исходное условие пути.

Независимые подмножества переменных и ограничений хранятся в структуре данных, напоминающей по интерфейсу систему непересекающихся множеств [16]. Каждое из множеств фактически представляет собой циклический список (рис. 4), в узлах которого содержатся термы символьных переменных. Один из узлов является «хвостовым» — он, помимо переменной, содержит в себе ссылку на соответствующее подмножество ограничений. Данный список реализован при помощи словаря `Dictionary<term, node>`, отображающего переменную на элемент размеченного объединения `node` (листинг 3). Такое представление позволяет обеспечить доступ к произвольному элементу структуры.

Листинг 3: Размеченное объединение node

```
1 type private node =
2     | Tail of term * term pset
3     | Node of term
4     | Empty
```

Для реализации проверки независимости символьных переменных ссылочных типов были внесены изменения в модуль *Memory*.

Каждая символьная переменная в V# однозначно определяется объектом, реализующим интерфейс `ISymbolicConstantSource` и указывающим на то, каким образом данная переменная хранится в символьной памяти. Например, для типов-значений, выделяемых на стеке, это экземпляр `stackReading`, а для ссылочных типов — `heapReading`, предоставляющий доступ к региону символьной памяти, по адресам из которого может располагаться объект.

Для того, чтобы иметь возможность проверять символьные переменные на косвенную независимость, в интерфейс `ISymbolicConstantSource` был добавлен метод

```
IndependentWith : ISymbolicConstantSource -> bool.
```

Данный метод возвращает *true*, если текущий экземпляр `ISymbolicConstantSource` независим с переданным ему в качестве аргумента.

Для различных источников символьных переменных метод `IndependentWith` реализован по-разному. К примеру, для `stackReading` он просто проверяет объекты на равенство, а для `heapReading` проверяет, что соответствующие регионы памяти не пересекаются.

6.2. Кэширование моделей SMT-решателя

Состояние символьного исполнителя в V# содержит в себе модель, которая в дальнейшем используется для определения входных данных при генерации теста. В связи с этим, реализация соответствующей оптимизации заключалась в подстановке значений из текущей модели в новое условие при ветвлении.

Данная оптимизация, как и независимое управление ограничениями, изменила логику ветвления: если подстановка значений из модели обращала условие в истину или ложь, то требовался только один вызов

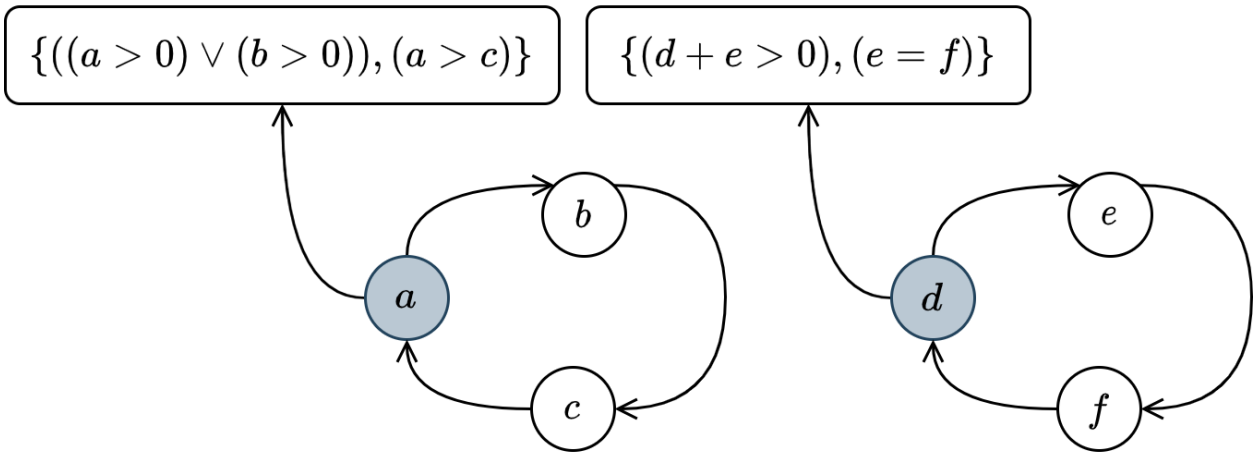

Рис. 4: Схема, показывающая способ представления независимых множеств переменных для условия пути $((a > 0) \vee (b > 0)) \wedge (a > c) \wedge (d + e > 0) \wedge (e = f)$

решателя. Для того, чтобы иметь возможность переключения реализаций ветвления в зависимости от ключей, соответствующие функции были вынесены в новый модуль *Branching*.

6.3. Инкрементальное использование SMT-решателя

При реализации инкрементальности на основе предпосылок были внесены изменения в модули *SolverInteraction*, предоставляющий общий интерфейс для работы с решателями, и *Z3*, обеспечивающий взаимодействие с API SMT-решателя Z3.

Интерфейс `ISolver`, используемый для взаимодействия с SMT-решателем, был расширен методом `CheckAssumptions`. Реализация данного метода для решателя Z3 создаёт соответствующие ограничениям предпосылки и передаёт их решателю (команда `assert`), а затем проверяет выполнимость конъюнкции предпосылок (команда `check-sat`). При этом поддерживается словарь, отображающий условия на названия соответствующих им искусственных переменных.

7. Эксперименты

Тестирование реализованных оптимизаций проводилось на тестовом стенде с характеристиками, указанными в таблице 1.

ОС	Linux Mint 20.1 Ulyssa
CPU	Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
RAM	4 Gb DDR4
Версия .NET	6.0.200

Таблица 1: Характеристики тестового стенда

7.1. Тестовые данные

На момент написания данной работы в V# ещё не был реализован механизм «concolic»[23] исполнения, в связи с чем тестирование на реальных программах практически не было возможно из-за неспособности V# работать со сторонними зависимостями в коде. Вследствие этого эксперименты проводились на синтетических тестовых данных.

Проект V# содержит набор синтетических тестов, однако большинство из них не содержит логики, порождающей существенные условия пути, что делает их недостаточно репрезентативными для проверки реализованных оптимизаций. В связи с этим в проект был добавлен новый набор тестов, порождающих длинные условия пути и делающих большое количество запросов к решателю. При реализации данных тестов был применён подход, описанный в [3] и использующий гипотезу Коллатца для создания «логических бомб»[24].

7.2. Исследовательские вопросы

Для оценки эффективности реализованных оптимизаций были сформулированы следующие вопросы.

[23]Техника, комбинирующая символьное и конкретное исполнение [23]

[24]англ. *logic bombs*

- **RQ1** Насколько уменьшается время работы SMT-решателя при применении оптимизаций?
- **RQ2** Увеличивается ли эффективность независимого управления ограничениями и кэширования моделей при их совместном применении?

7.3. Результаты

В таблицах 2 и 3 приведены общее время (в миллисекундах) символьного исполнения и генерации тестов (*total*) и время работы решателя (*check-sat*) для каждого из тестов со следующими комбинациями применённых оптимизаций:

- **no opts** — без оптимизаций;
- **ci** — с независимым управлением ограничениями;
- **cache** — с кэшированием моделей;
- **ci + cache** — с независимым управлением ограничениями и кэшированием моделей;
- **inc** — с инкрементальным использованием SMT-решателя;
- **all** — со всеми оптимизациями.

В графе *called* приведено количество запусков SMT-решателя.

Тесты *regex1, regex2* представляют собой тесты, содержащиеся в исходном тестовом наборе в V#, тесты *collatz1-collatz8* были добавлены.

При всех запусках были сгенерированы тесты, обеспечивающие стопроцентное покрытие кода[25].

[25]Тестовое покрытие измерено при помощи инструмента dotCover: https://www.jetbrains.com/help/dotcover/dotCover_Introduction.html : 27.05.2022

7.3.1. RQ1

Применение полного набора оптимизаций сократило время работы SMT-решателя во всех случаях. В лучшем случае (тест *collatz5*) получено ускорение на 71%.

Следует отметить, что для некоторых тестов (например, *regex1*) применение всех оптимизаций увеличило время работы решателя по сравнению с применением лишь некоторых.

Что касается общего времени исполнения (*total*), оно также сократилось во всех случаях, но в меньшей степени, чем время работы решателя. Это естественным образом связано с тем, что в зависимости от теста работа решателя вносит различный вклад в общее время исполнения. Оптимизации же были направлены в первую очередь на сокращение времени работы непосредственно решателя.

7.3.2. RQ2

Независимое управление ограничениями и кэширование моделей эффективно дополняют друг друга при совместном использовании. В тестах *collatz4, collatz5, regex1, regex2* самое значительное уменьшение времени работы решателя получено именно при использовании комбинации данных методов. Необходимо отметить, что к остальным тестам данное замечание неприменимо непосредственно, поскольку в них не проявило себя независимое управление ограничениями — было сформировано лишь одно независимое подмножество переменных.

case			no opts	ci	cache	ci + cache	inc	all	speedup (all, %)
collatz1	total	mean	96060	99611	37856	41802	84901	33988	65
		sd	8804	11020	2855	5457	5853	3880	
	check-sat	mean	94858	98350	36411	40304	82382	31993	66
		sd	8805	11021	2857	5454	5849	3882	
		called	422	422	276	276	422	276	
collatz2	total	mean	40522	40889	23180	22615	34616	20504	49
		sd	583	263	116	187	75	56	
	check-sat	mean	39194	39491	21543	20934	31765	18363	53
		sd	585	264	121	186	74	54	
		called	552	552	362	362	552	362	
collatz3	total	mean	7442	6926	5800	6676	12754	4572	39
		sd	676	166	20	77	2242	32	
	check-sat	mean	6683	6134	4969	5816	11934	3691	45
		sd	676	166	18	76	2243	32	
		called	78	78	59	59	78	59	
collatz4	total	mean	20067	16104	10273	6764	20286	7301	64
		sd	1189	765	421	502	673	248	
	check-sat	mean	18874	14910	8999	5479	19019	6033	68
		sd	1189	768	425	504	676	248	
		called	505	505	277	277	505	277	
collatz5	total	mean	25368	19751	13576	8643	26457	8587	66
		sd	825	419	439	333	299	482	
	check-sat	mean	23618	18089	11782	6866	24672	6867	71
		sd	823	420	441	330	299	484	
		called	969	969	505	505	969	505	
collatz6	total	mean	58966	62900	48737	56488	58323	53762	9
		sd	10759	10342	2315	7439	16396	8831	
	check-sat	mean	57985	61879	47715	55427	57243	52644	9
		sd	10761	10342	2312	7439	16401	8836	
		called	281	281	142	142	281	142	

Таблица 2: Результаты экспериментов

case			no opts	ci	cache	ci + cache	inc	all	speedup (all, %)
collatz7	total	mean	7204	7124	7308	7206	6741	6944	4
		sd	18	35	11	43	39	32	
	check-sat	mean	304	305	305	303	110	116	62
		sd	1,07	1,7	0,5	0,8	4	4	
		called	252	252	252	252	252	252	
collatz8	total	mean	22390	22279	15464	14480	21436	11479	49
		sd	1315	896	719	994	2781	1060	
	check-sat	mean	21275	21118	14281	13253	20080	10205	52
		sd	1316	896	721	991	2785	1059	
		called	280	280	193	193	280	193	
regex1	total	mean	29425	25317	27535	24954	29306	26031	12
		sd	132	117	80	103	119	140	
	check-sat	mean	3708	1552	2373	1014	3977	2470	33
		sd	19	12	11	7	23	18	
		called	9238	9238	5823	5823	9238	5823	
regex2	total	mean	8637	8111	8631	8156	8379	8265	4
		sd	24	41	43	37	28	31	
	check-sat	mean	601	280	560	264	540	552	8
		sd	4	3	3	3	5	4	
		called	1690	1690	1563	1563	1690	1563	

Таблица 3: Результаты экспериментов

Заключение

В ходе данной работы были получены следующие результаты.

- Проведён обзор оптимизаций управления символьными ограничениями: техник независимого управления ограничениями и кэширования моделей SMT-решателя, подходов к инкрементальному использованию SMT-решателя на основе стека и предпосылок.
- Независимое управление ограничениями реализовано в символьной виртуальной машине V#.
- Кэширование моделей SMT-решателя реализовано в символьной виртуальной машине V#.
- Подход к инкрементальному использованию SMT-решателя на основе предпосылок реализован в символьной виртуальной машине V#.
- Проведены эксперименты на синтетических тестовых данных.

Исходный код проекта V# является открытым и содержится в репозитории GitHub[26] (имя аккаунта — mxprshn).

[26]https://github.com/VSharp-team/VSharp Дата обращения: 03.05.2022

Список литературы

[1] The SMT-LIB Standard: Version 2.6 : Rep. / Department of Computer Science, The University of Iowa ; Executor: Clark Barrett, Pascal Fontaine, Cesare Tinelli : 2017.

[2] Barrett Clark W., Dill David L., Levitt Jeremy R. A Decision Procedure for Bit-Vector Arithmetic. — DAC '98. — New York, NY, USA : Association for Computing Machinery, 1998. — P. 522–527.

[3] Benchmarking the Capability of Symbolic Execution Tools with Logic Bombs / Hui Xu, Zirui Zhao, Yangfan Zhou, Michael R. Lyu // IEEE Transactions on Dependable and Secure Computing. — 2020. — Vol. 17, no. 6. — P. 1243–1256.

[4] Bjørner Nikolaj. How incremental solving works in Z3? // Stack Overflow. — 2016. — URL: https://stackoverflow.com/a/40427658/11933076 (online; accessed: 02.05.2022).

[5] Cadar Cristian, Dunbar Daniel, Engler Dawson. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs // Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. — OSDI'08. — USA : USENIX Association, 2008. — P. 209–224.

[6] Liu Tianhai, Araújo Mateus, d'Amorim Marcelo, Taghdiri Mana. A Comparative Study of Incremental Constraint Solving Approaches in Symbolic Execution. — 2014.

[7] Cook Stephen A. The Complexity of Theorem-Proving Procedures // Proceedings of the Third Annual ACM Symposium on Theory of Computing. — STOC '71. — New York, NY, USA : Association for Computing Machinery, 1971. — P. 151–158.

[8] DPLL(T): Fast Decision Procedures / Harald Ganzinger, George Hagen, Robert Nieuwenhuis et al. — 2004. — 06.

[9] Davis Martin, Logemann George, Loveland Donald. A Machine Program for Theorem-Proving // Commun. ACM. — 1962. — jul. — Vol. 5, no. 7. — P. 394–397.

[10] ECMA International. Standard ECMA-335 - Common Language Infrastructure (CLI). — 2012. — URL: https://www.ecma-international.org/wp-content/uploads/ECMA-335_6th_edition_june_2012.pdf (online; accessed: 01.05.2022).

[11] EXE: Automatically Generating Inputs of Death / Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski et al. // ACM Trans. Inf. Syst. Secur. — 2008. — dec. — Vol. 12, no. 2. — 38 p. — URL: https://doi.org/10.1145/1455518.1455522.

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

[13] Ganesh Vijay. SAT and SMT Solvers: A Foundational Perspective // Engineering Secure and Dependable Software Systems. — NATO Science for Peace and Security Series - D: Information and Communication Security. — 2018. — P. 29 – 60.

[14] Ganesh Vijay, Dill David. A Decision Procedure for Bit-Vectors and Arrays. — Vol. 4590. — 2007. — 01. — P. 519–531.

[15] Godefroid Patrice, Levin Michael, Molnar David. SAGE: Whitebox Fuzzing for Security Testing // ACM Queue. — 2012. — 03. — Vol. 10. — P. 20.

[16] Introduction to Algorithms / Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. — 2nd edition. — The MIT Press, 2001. — ISBN: 0262032937.

[17] Lin Yude, Miller Tim, Søndergaard Harald. Compositional Symbolic Execution: Incremental Solving Revisited // 2016 23rd Asia-Pacific Software Engineering Conference (APSEC). — 2016. — P. 273–280.

[18] Marques Silva J.P., Sakallah K.A. GRASP-A new search algorithm for satisfiability // Proceedings of International Conference on Computer Aided Design. — 1996. — P. 220–227.

[19] Marques-Silva J.P., Sakallah K.A. GRASP: a search algorithm for propositional satisfiability // IEEE Transactions on Computers. — 1999. — Vol. 48, no. 5. — P. 506–521.

[20] Mordvinov Dmitry. Property Directed Symbolic Execution. — 2021. — Spring/Summer Young Researchers' Colloquium on Software Engineering. URL: https://youtu.be/pX5qS4SbsJI (online; accessed: 03.05.2022).

[21] SCSE: Boosting Symbolic Execution via State Concretization / Huibin WANG, Chunqiang LI, Jianyi MENG, Xiaoyan XIANG // IEICE Transactions on Information and Systems. — 2019. — Vol. E102.D, no. 8. — P. 1506–1516.

[22] Stump Aaron, Barrett Clark, Dill David. A Decision Procedure for an Extensional Theory of Arrays. — 2002. — 01.

[23] 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.

[24] Sydr: Cutting Edge Dynamic Symbolic Execution / Alexey Vishnyakov, Andrey Fedotov, Daniil Kuts et al. // 2020 Ivannikov Ispras Open Conference (ISPRAS). — 2020. — Dec.

[25] Tillmann Nikolai, de Halleux Peli. Pex - White Box Test Generation for .NET // Proc. of Tests and Proofs (TAP'08). — Vol. 4966 of LNCS. — 2008. — April. — P. 134–153.

[26] de Moura Leonardo, Bjørner Nikolaj. Satisfiability modulo Theories: Introduction and Applications // Commun. ACM. — 2011. — sep. — Vol. 54, no. 9. — P. 69–77.

[27] de Moura Leonardo, Bjørner Nikolaj. Z3: an efficient SMT solver. — Vol. 4963. — 2008. — 04. — P. 337–340.

[28] Костицын Михаил Павлович. Модель символьной памяти .NET с поддержкой реинтерпретаций. Бакалаврская работа // Санкт-Петербургский государственный университет. — 2019.