

Санкт–Петербургский государственный университет  
Кафедра системного программирования  
Группа 21.Б15-мм

*Бучин Вячеслав Андреевич*

## Гибридный решатель с поддержкой ограничений на типы

Отчёт по производственной практике  
в форме «Решение»

Научный руководитель:

доцент кафедры системного программирования, к. ф.-м. н.

Мордвинов Д. А.

Консультант:

ведущий разработчик ООО «ЭСПРИТО ДЕВЕЛОПМЕНТ»

Костицын М. П.

Санкт-Петербург

2024 г.

# Содержание

|                                                                     |    |
|---------------------------------------------------------------------|----|
| <b>Введение</b> . . . . .                                           | 3  |
| <b>Постановка задачи</b> . . . . .                                  | 5  |
| <b>1. Обзор</b> . . . . .                                           | 6  |
| 1.1. Символьное исполнение . . . . .                                | 6  |
| 1.2. SMT-решатели . . . . .                                         | 8  |
| 1.3. Проект V# . . . . .                                            | 9  |
| <b>2. Гибридный решатель с поддержкой ограничений на типы</b> . . . | 11 |
| 2.1. Взаимодействие SMT-решателя и решателя для типов . . .         | 11 |
| 2.2. UNSAT-ядро решателя для системы типов .NET . . . . .           | 13 |
| <b>3. Архитектура и детали реализации</b> . . . . .                 | 16 |
| 3.1. Архитектура . . . . .                                          | 16 |
| 3.2. Детали реализации . . . . .                                    | 19 |
| <b>4. Тестирование</b> . . . . .                                    | 21 |
| 4.1. Способ тестирования . . . . .                                  | 21 |
| 4.2. Сравнение результатов . . . . .                                | 21 |
| <b>Заключение</b> . . . . .                                         | 23 |
| <b>Список литературы</b> . . . . .                                  | 24 |

## Введение

Каждый год увеличивается объем используемого программного обеспечения. Важным этапом цикла его разработки является контроль качества. Одна из его составляющих — модульное тестирование. Для автоматизации данного процесса существует множество различных техник и подходов. Одним из которых является символьное исполнение.

*Символьное исполнение* [2, 8, 9] — техника, которая позволяет свести задачу анализа программного кода в условиях неопределенности входных данных к задаче проверки выполнимости формул логики первого порядка. Эти формулы представляют из себя ограничения, выполнение которых однозначно задает ветвь исполнения программы. Они могут быть устроены нетривиальным образом, а именно содержать конъюнкции, отрицания, а также дизъюнкции. Кроме того, среди ограничений можно выделить те, для решения которых необходимо иметь информацию о системе типов языка исследуемой программы — *ограничения на типы*.

Для их решения используются различные решатели: для ограничений, касающихся типов используются решатели для типов, а для оставшихся — SMT-решатели (Satisfiability Modulo Theories) [3, 6, 10] с применением различных теорий (битовых векторов, линейной арифметики и других). Эти инструменты позволяют не только выяснить, выполнима ли формула, но и подобрать значения переменных, для которых она выполняется, а в случае отсутствия решения предоставить набор ограничений, в которых заключено противоречие. Однако данная задача в общем случае является неразрешимой, вследствие чего решатели не всегда могут доказать наличие или отсутствие решения, а иногда и вовсе могут зависнуть. В случае существования решения ограничений для системы типов .NET его всегда можно найти, однако доказательство его отсутствия по-прежнему остается неразрешимой задачей [5].

Для организации взаимодействия между теориями при решении ограничений в SMT-решателях используются специальные алгоритмы (например, DPLL(T)). В силу того, что ограничения системы типов .NET не являются теорией SMT-решателя, по-умолчанию между ним и решателем для типов отсутствует какое-либо взаимодействие. Однако информация, полученная во

время решения ограничений на типы, может оказаться полезной в случаях, когда SMT-решатель попытается приравнять какие-то адреса между собой, присвоить в какие-то из них null, а также в некоторых других.

Ранее в рамках учебных практик был разработан и реализован решатель для системы типов платформы .NET в проекте V#. Данная производственная практика нацелена на создание гибридного решателя с поддержкой ограничений на типы путём выстраивания взаимодействия между решателем для типов и SMT-решателем.

## Постановка задачи

Целью производственной практики является создание гибридного решателя с поддержкой ограничений на типы. Для достижения цели были поставлены следующие задачи:

- изучить работу SMT-решателя в комбинации теорий,
- разработать модель управления ограничениями,
- спроектировать процесс взаимодействия решателя для типов и SMT-решателя,
- реализовать разработанную систему,
- провести тестирование полученного решения.

# 1. Обзор

В данной главе рассмотрена предметная область данной производственной практики. В разд. 1.1 подробно рассмотрена работа символьного исполнения. Разд. 1.2 посвящен SMT-решателям и их использованию в символьном исполнении, также в этом разделе обсуждается работа SMT-решателя в комбинации теорий. В разд. 1.3 показана архитектура проекта V#, в рамках которого выполняется данная производственная практика.

## 1.1 Символьное исполнение

*Символьное исполнение* — техника исследования программ, которая позволяет исполнять программный код в условиях неопределённости входных данных. Рассмотрим работу данной техники подробнее.

Изначально входные данные заменяются *символьными переменными* — абстракцией, позволяющей исследовать программу на всей совокупности значений входных данных сразу, а не на каких-то одних. Далее код программы выполняется последовательно по одной инструкции. Во время исполнения накапливается *условие пути  $pc$*  (англ. path condiditon) — формула логики первого порядка, отвечающая текущей ветви исполнения и ограничивающая возможные значения символьных переменных в ней. Изначально  $pc = \top$ . Также во время исполнения поддерживается *символьная память*. Она отвечает за хранение и модификацию символьных значений. При достижении ветвления символьная память и условие пути копируются столько раз, сколько ветвей может образоваться после ветвления. Далее каждой из ветвей в соответствующее условие пути добавляется условие попадания в нее. После модификации выясняется его выполнимость, что равносильно достижимости ветви, с помощью SMT-решателей и решателей для типов, которые будут рассмотрены позже. Затем продолжается независимое исследование достижимых ветвей.

```

public static int Abs(int x)
{
    int y = 0;
    if (x < 0) {
        y = -x;
    } else {
        y = x;
    }
    if (y < 0) {
        throw NegativeResultException();
    }
    return y;
}

```

**Листинг 1:** Программа для иллюстрации символьного исполнения

Рассмотрим работу символьного исполнения программы представленной на лист. 1.

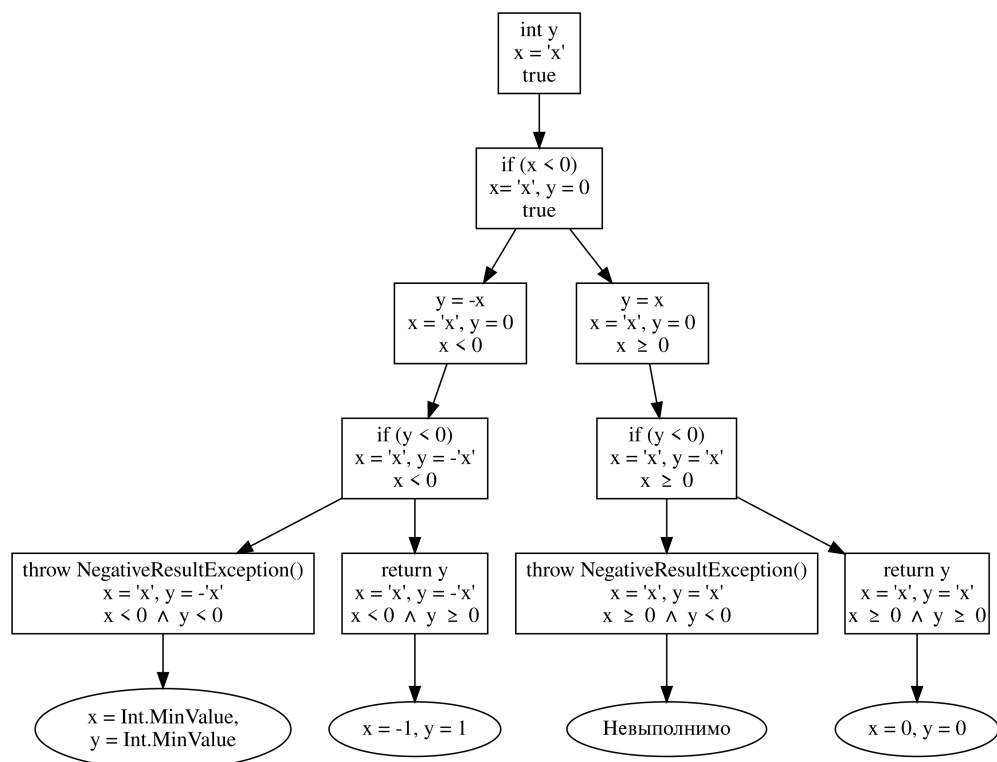


**Рис. 1:** Схема символьного исполнения функции Abs

На рис. 1 представлена схема символьного исполнения для программы

с лист. 1. В прямоугольниках на первой строке указана исполняемая инструкция, на второй — состояние символьной памяти, а на третьей — условие пути.

## 1.2 SMT-решатели

SMT-решатели (Satisfiability Modulo Theories) — это класс программных инструментов, которые автоматически решают задачу выполнимости булевых формул с добавлением теорий. SMT-решатели широко применяются в различных областях, таких как верификация программного обеспечения, анализ безопасности, автоматическое доказательство теорем и многих других. SMT-решатели могут работать с различными теориями, такими как целочисленная арифметика, линейная арифметика, теория массивов, теория списков и другие. В контексте символьного исполнения SMT-решатель является одним из решателей, используемых для проверки достижимости исследуемой ветви программы. Рассмотрим подробнее результаты работы SMT-решателя.

*SAT* Если SMT-решателю удалось найти *модель* — набор значений переменных, обращающих входную формулу в истину.

*UNSAT* Если SMT-решателю удалось доказать отсутствие решения, что равносильно противоречивости входной формулы. Также некоторые SMT-решатели в данном случае возвращают *UNSAT-ядро* (англ. UNSAT core) — минимальный набор конъюнктов, содержащих противоречие.

*UNKNOWN* В силу неразрешимости теорий или некоторых их комбинаций, SMT-решатели могут зависнуть в процессе своей работы. Поэтому в случае, когда время работы SMT-решателя превысило допустимый предел, возвращается такой результат.

Рассмотрим DPLL(T) [7] — механизм работы SMT-решателя в комбинации теорий, использующийся в таких решателях, как Z3 Theorem Prover [3], CVC4 [10]. Во время работы этого алгоритма входная формула преобразуется в SAT-формулу путём замены частей, относящихся к теориям, на булевые

переменные. Решается задача SAT, после чего происходит опрос решателей теорий. Если формула оказалась выполнимой, процесс завершается, иначе же исходная формула дополняется информацией из UNSAT-ядер решателей теорий, и процесс повторяется.

### 1.3 Проект V#

V#<sup>1</sup> — инструмент для анализа программ, он исследует CIL (Common Intermediate Language) с использованием техники символьного исполнения. Далее будет рассмотрена архитектура проекта V# (рис. 2). Цветом выделены компоненты, модифицированные в рамках данной работы.

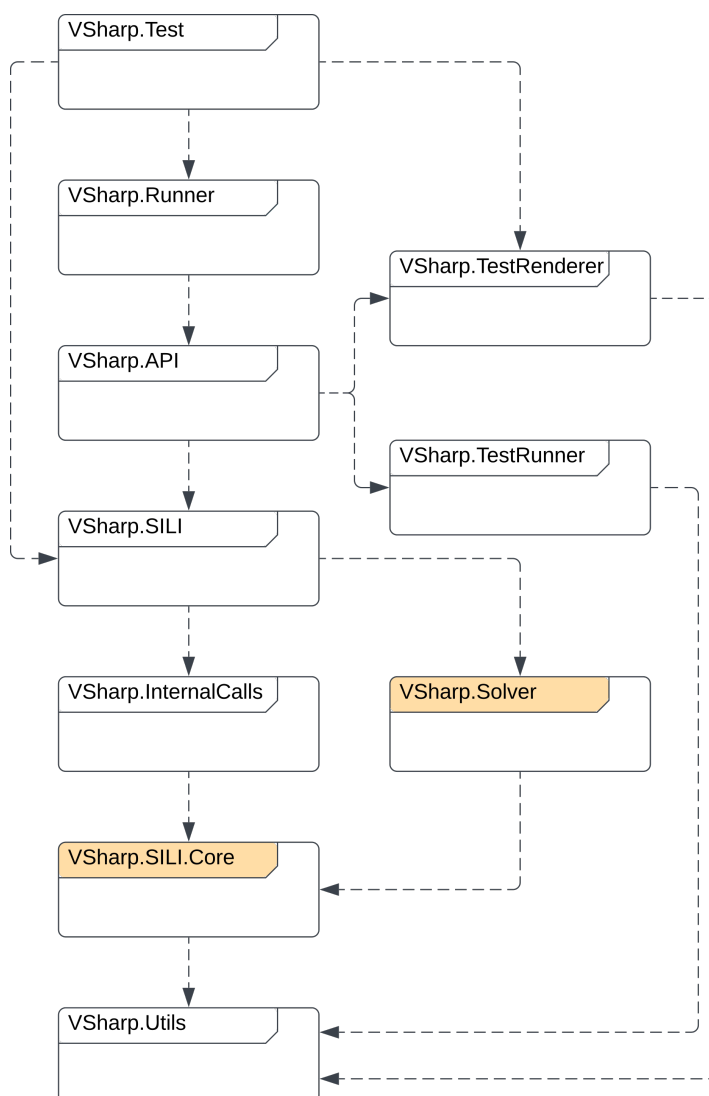


Рис. 2: Архитектура проекта V#

<sup>1</sup><https://github.com/VSharp-team/VSharp/>

Рассмотрим каждую компоненту подробнее.

***VSharp.Test*** отвечает за проведение интеграционного тестирования.

***VSharp.Runner*** предоставляет CLI (Command Line Interface) для использования V# через терминал.

***VSharp.API*** предоставляет интерфейс для использования V# из других проектов.

***VSharp.TestRenderer*** занимается генерацией кода тестов на языке C#.

***VSharp.TestRunner*** используется для запуска сгенерированных тестов.

***VSharp.SILI (Symbolic Intermediate Language Interpreter)*** является символьным интерпретатором CIL.

***VSharp.InternalCalls*** содержит символьные модели некоторых функций из CLR (Common Language Runtime), что позволяет вместо полноценного их исследования подставить готовые модели.

***VSharp.Solver*** реализует систему взаимодействия с SMT-решателем.

***VSharp.SILI.Core*** представляет из себя «ядро» проекта. В нем происходят все нетривиальные операции внутри символьной машины, например, работа с символьной памятью, решение ограничений на типы.

***VSharp.Utils*** содержит вспомогательные функции для остальных компонент.

## 2. Гибридный решатель с поддержкой ограничений на типы

Основная идея разработанного на основе алгоритма DPLL(T) подхода состоит в рассмотрении решателя для типов в качестве решателя одной из теорий.

Связано это с тем, что прямое внедрение системы типов, как теорию SMT-решателя, является трудноосуществимой задачей. Прежде всего в связи с тем, что отношение подтипирования является отношением частичного порядка, что является проблемой для SMT-решателей, а также с принципиально разным устройством систем типов различных языков программирования:

- некоторые допускают множественное наследование, некоторые нет;
- какие-то поддерживают разные формы наследования (например, C++);
- системы типов некоторых функциональных языков программирования не содержат понятия «класс» и «наследование» вовсе.

Поэтому разработанный подход фокусируется исключительно на системе типов .NET [4, 1] (но все еще может использоваться для других систем типов при наличии соответствующего решателя). Однако даже в таких ограничениях рассматриваемая задача все еще остается нетривиальной. Связано это с тем, что множество типов, среди которых происходит поиск модели, может отличаться при разных запусках решателя (например, могут отличаться пользовательские типы).

В разд. 2.1 рассмотрено взаимодействие решателя для типов и SMT-решателя, предлагаемое разработанным подходом. Для его осуществления необходимо, чтобы решатель для типов возвращал UNSAT-ядро в случае, когда данные на вход ограничения невыполнимы, этому посвящен разд. 2.2.

### 2.1 Взаимодействие SMT-решателя и решателя для типов

Разработанный алгоритм является итеративным. Рассмотрим  $i$ -ую итерацию его работы, которую можно разделить на четыре этапа:

1. вызов SMT-решателя;
2. получение информации о выполненности ограничений на типы;
3. вызов решателя для типов;
4. уточнение запроса к SMT-решателю на основе UNSAT-ядра решателя для типов.

Рассмотрим каждый из них подробнее. В начале составляется запрос  $f_i$  к SMT-решателю, в котором ограничения на типы заменяются на пропозициональные переменные. После чего происходит вызов SMT-решателя. Если он вернул UNSAT, то алгоритм завершается с результатом UNSAT. Иначе же на основе выданной им модели можно заключить, что ограничения, которым соответствует пропозициональная переменная со значением «истина», должны выполняться, а оставшиеся — нет.

Далее полученная информация об ограничениях на типы передается решателю для типов. Рассмотрим возможные результаты его работы.

**TypeSAT** В данном случае решатель для типов также возвращает собственную модель. Далее составляется модель гибридного решателя на основе модели SMT-решателя и модели решателя для типов. Алгоритм завершается с результатом SAT.

**TypeUNSAT** Этот результат означает, что решатель для типов доказал невыполнимость переданных ему ограничений. Алгоритм завершается с результатом UNSAT.

**TypeUNSATCore** В данном случае решатель для типов выдает также UNSAT-ядро, представляющее из себя формулу  $uc$ , выполненность которой необходима для выполнимости переданных ему ограничений. После чего составляется формула для следующей итерации  $f_{i+1} = f_i \wedge uc$ . Алгоритм переходит на следующую итерацию.

Рассмотрим работу гибридного решателя на примере. Имеется входная формула  $f_0 = T_1 \wedge (C \vee T_2)$ , где  $T_1, T_2$  — ограничения на типы, а  $C$  — какое-то ограничение из теорий SMT-решателя.

1. Формула  $f_0$  передается в SMT-решатель. Он возвращает SAT с моделью  $m_0 = \{\dots, T_1 = 1, T_2 = 0\}$ .
2. Полученная информация об ограничениях на типы передается в решатель для типов:  $typeSolver(T_1 \wedge \neg T_2) = TypeUNSATCore\{uc_0 = T_2\}$ .
3. На основе UNSAT-ядра решателя для типов  $f_0$  модифицируется для следующей итерации алгоритма:  $f_1 = f_0 \wedge uc_0 = T_1 \wedge (C \vee T_2) \wedge T_2$ .
4. Вызывается SMT-решатель на формуле  $f_1$ . Он возвращает SAT с моделью  $m_1 = \{\dots, T_1 = 1, T_2 = 1\}$ .
5. Вызывается решатель для типов:  $typeSolver(T_1 \wedge T_2) = TypeSAT\{tm_1\}$ .
6. Составляется общая модель  $m$  на основе модели SMT-решателя  $m_1$  и модели решателя для типов  $tm_1$ . Алгоритм завершается с результатом  $SAT\{m\}$ .

## 2.2 UNSAT-ядро решателя для системы типов .NET

Для использования решателя для типов в гибридном решателе необходимо, чтобы он возвращал UNSAT-ядро. Рассмотрим его построение для решателя для системы типов .NET.

Рассмотрим входную формулу гибридного решателя как конъюнкцию подформул. На основе этого входные данные решателя для типов можно разделить на две группы.

1. Если какая-то подформула целиком является ограничением на типы, то она обязательна для выполнения решателем для типов.
2. Если ограничение на типы входит в какую-то подформулу как ее часть (например, является операндом дизъюнкции), то его выполненность зависит от модели SMT-решателя.

В случае, когда выполнить ограничения первой группы невозможно, решатель для типов выдает `TypeUNSAT`. Однако невыполнимость ограничений второй группы не означает невыполнимость входной формулы гибридного решателя целиком, потому что при другом распределении выполненных и невыполненных ограничений на типы решение может найтись.

Также SMT-решатель может в процессе своей работы приравнять символные адреса между собой. Данное поведение является необходимым для корректности исследования программ. Например, если исследуемая функция имеет два аргумента, которые оба являются массивами, то для полного анализа необходимо рассмотреть случай, когда оба аргумента представляют ссылку на один и тот же массив. Однако SMT-решатель, приравнивая адреса, не берет во внимание типы, для них подходящие, из-за того, что не имеет информации о них. Таким образом равенство адресов в модели SMT-решателя является источником лемм.

Еще одним источником лемм для UNSAT-ядра являются конфликты с *нулевыми ссылками* (англ. *null references*). Они возникают в случаях, когда SMT-решатель приравнивает адреса в памяти к нулю, однако на эти адреса присутствуют ограничения на типы, доступом к которым SMT-решатель не обладает.

UNSAT-ядро представляет из себя дизъюнкцию лемм. Рассмотрим подробнее сами леммы и случаи, в которых они получаются. Пусть имеются  $a, b, c$  — адреса,  $T_1, T_2$  — типы,  $Cand_i$  — множество подходящих типов для адреса  $i$ , а также отношение подтипирования  $<:$ .

- $a <: T_1 = 0$  и при этом  $a = \text{null}$ . В данном случае у адреса  $a$  не должно быть надтипа  $T_1$  и при этом он должен являться нулевой ссылкой. Противоречие здесь заключается в том, что у нулевой ссылки все ссылочные типы являются надтипами (например, в переменную любого ссылочного типа можно присвоить значение `null`). Таким образом, можно заключить, что  $a \neq \text{null} \vee a <: T_1$ .
- $a >: T_2 = 1$  и при этом  $a = \text{null}$ . В данном случае у адреса  $a$  должен быть подтип  $T_2$  и при этом адрес должен являться нулевой ссылкой. Противоречие в том, что у нулевых ссылок не может быть никакого

подтипа (например, ни один класс не может наследовать `null`). Значит можно заключить, что  $a \neq \text{null} \vee \neg(a :> T_2)$ .

- SMT-решатель приравнял адреса  $a, b, c$  между собой:  $a = b = c$ . Но при этом  $\text{Cand}_a \cap \text{Cand}_b \cap \text{Cand}_c = \emptyset$ . Тогда можно заключить, что какие-то адреса не равны между собой  $a \neq b \vee a \neq c \vee b \neq c$ , или какие-то из них являются нулевыми ссылками  $a = \text{null} \vee b = \text{null} \vee c = \text{null}$ , или выполненность (согласно SMT-решателю) каких-то из ограничений второй группы является некорректной  $a <: T_1 \vee \neg(a :> T_2)$ .

### 3. Архитектура и детали реализации

Разработанный подход был реализован в проекте V# в качестве решателя, используемого символьным исполнением. Данная глава посвящена рассмотрению архитектуры получившегося решения (разд. 3.1), а также его особенностям, описание которых не входит в главу 2.

#### 3.1 Архитектура

На рис. 3 представлена архитектура гибридного решателя в проекте V#.

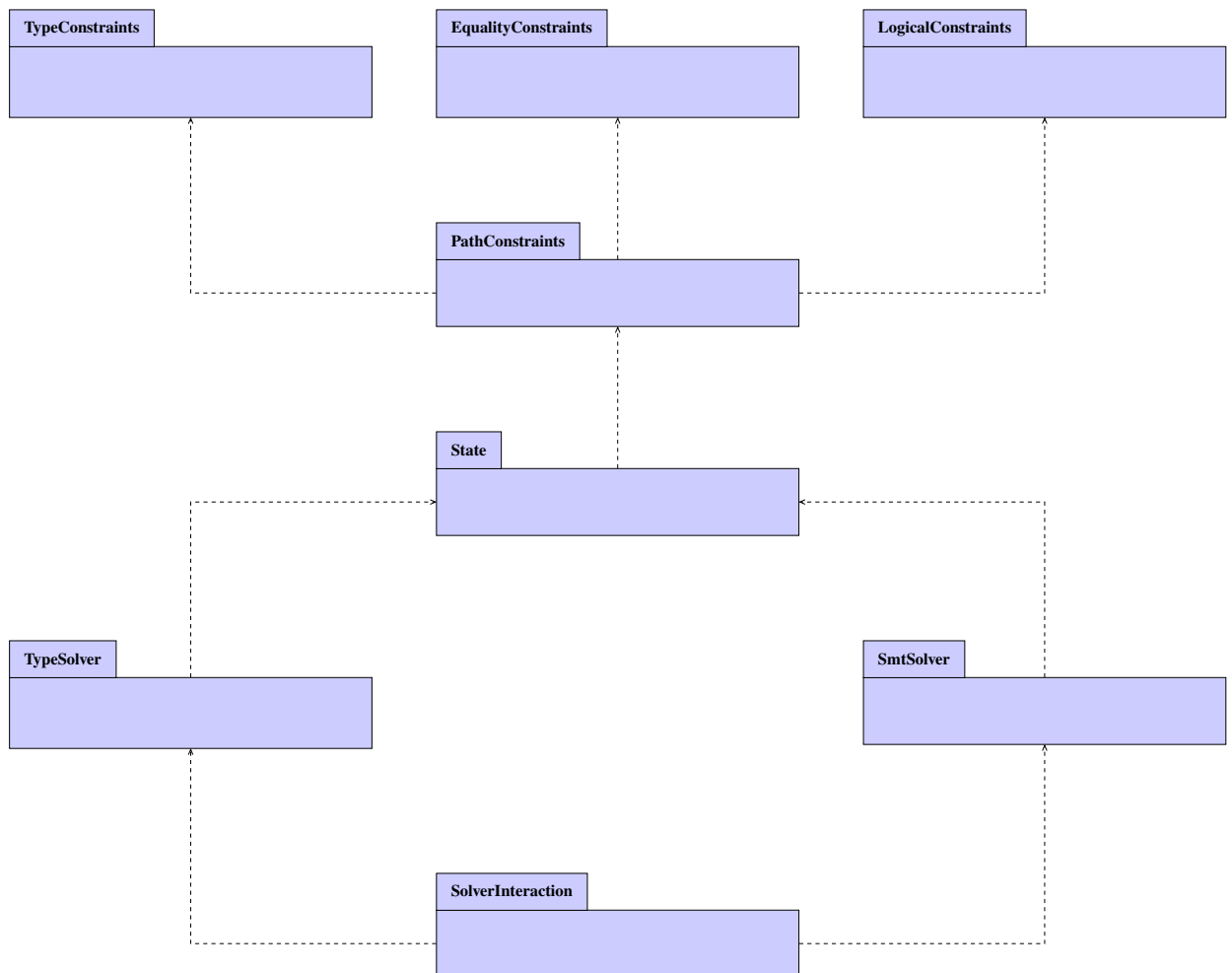


**Рис. 3:** Архитектура гибридного решателя

Рассмотрим каждую представленную компоненту подробнее. Начнем с системы, отвечающей за представление условия пути символьного исполне-

ния. Она была разработана на основе используемой ранее компоненты схожего назначения *PathCondition*.

**TypeConstraints** Данная компонента отвечает за внутреннее представление ограничений на типы. Она содержит только ограничения первой группы. Была разработана с нуля.

**EqualityConstraints** Эта компонента содержит в себе ограничения на равенство и не равенство адресов между собой, а также адресов и нулевых ссылок, была разработана с нуля.

**LogicalConstraints** Данная компонента содержит ограничения, не вошедшие в ранее рассмотренные группы, в том числе ограничения на типы второй группы. Она представляет из себя модифицированную компоненту *PathCondition*.

Компонента *State* отвечает за представление внутреннего состояния символьного исполнения. Она была модифицирована для работы с новой системой, представляющей условие пути.

Далее рассмотрим компоненты гибридного решателя.

**TypeSolver** Данная компонента содержит решатель для типов, разработанный ранее в рамках учебных практик. Он был модифицирован для построения UNSAT-ядра.

**SmtSolver** Эта компонента отвечает за взаимодействие с SMT-решателем. Она была модифицирована для получения ограничений на типы и значений соответствующих им пропозициональных переменных в модели.

**SolverInteraction** Данная компонента отвечает за взаимодействие двух решателей. Она была модифицирована таким образом, чтобы реализовывать разработанный подход их взаимодействия.

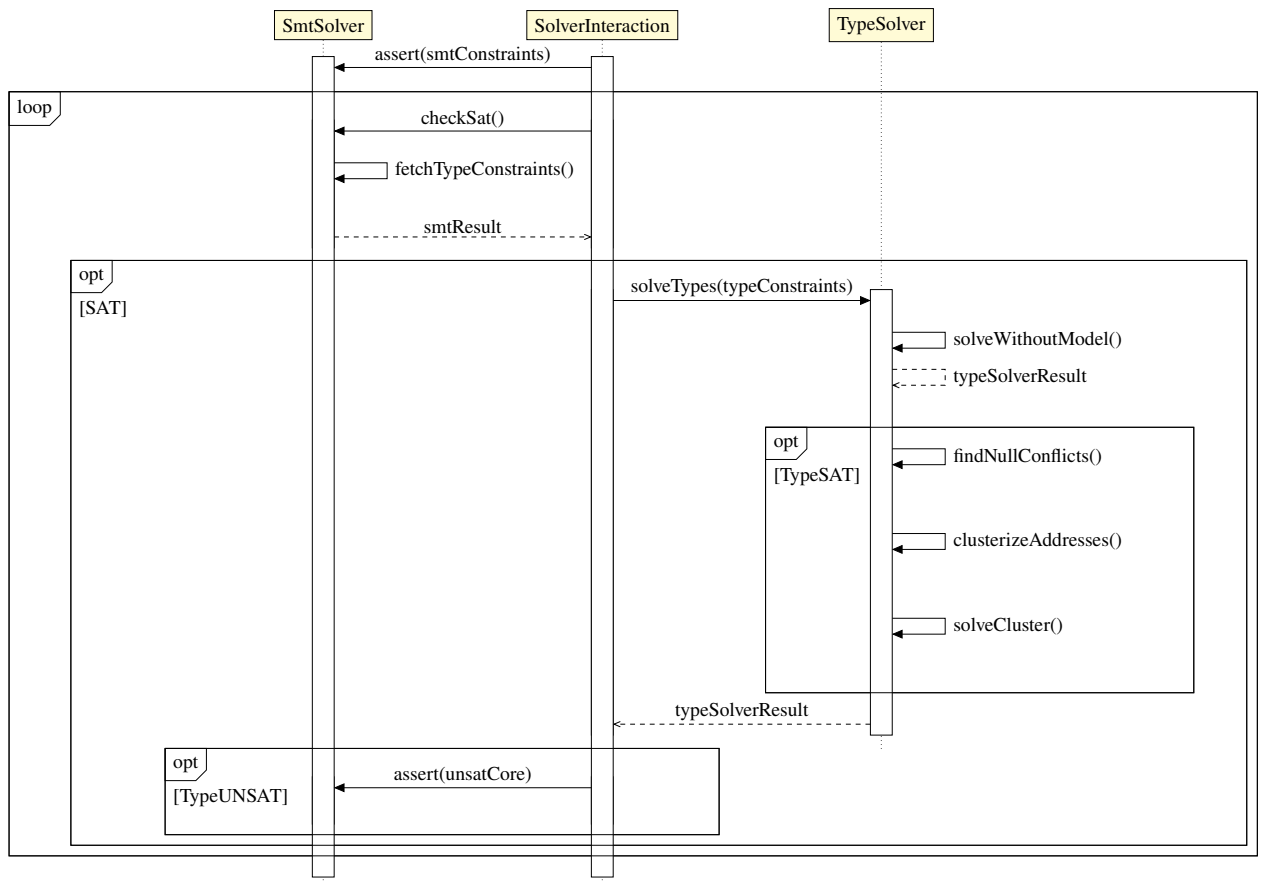


**Рис. 4:** Диаграмма взаимодействия решателя для типов и SMT-решателя

На рис. 4 представлена схема взаимодействия решателей. Рассмотрим ее подробнее. Сначала к запросу в SMT-решатель добавляются ограничения из групп *LogicalConstraints* и *EqualityConstraints* (smtConstraints на схеме). Далее взаимодействие происходит циклично. Проверяется выполнимость добавленных ограничений. В случае UNSAT взаимодействие завершается. Если же результат SAT, то во время построения модели также извлекаются ограничения на типы второй группы вместе со значениями соответствующих пропозициональных переменных. Далее происходит вызов решателя для типов. Он в свою очередь обновляет свое внутреннее состояние, решая имеющиеся ограничения первой группы. Если оказалось, что они невыполнимы, то алгоритм останавливается с результатом UNSAT. Иначе же он пытается найти конфликты нулевых ссылок, затем делит адреса на группы в зависимости от их значения в модели SMT-решателя, после чего пытается найти подходящие типы для каждой группы адресов. Если отсутствуют конфликты нулевых

ссылок, а также для каждой группы удалось подобрать тип, то решатель для типов возвращает TypeSAT, и взаимодействие завершается. Если же были обнаружены конфликты нулевых ссылок или не удалось подобрать тип для какой-либо группы, строится UNSAT-ядро. Далее оно добавляется к запросу в SMT-решатель, и цикл повторяется.

## 3.2 Детали реализации

Данная глава описывает детали реализации разработанного решения. Оно было реализовано на языке F# в проекте V#. Код реализации доступен по ссылке<sup>2</sup>.

### 3.2.1 Разделение ограничений

Реализованная система управления ограничениями с выделением ограничений на типы и ограничений на равенство адресов позволяет оптимизировать некоторые процессы при взаимодействии решателя для типов и SMT-решателя.

Во-первых, выделенные ограничения на типы можно не кодировать при составлении запроса в SMT-решатель, поскольку они всегда должны быть выполнены. Вместо этого они напрямую передаются решателю для типов.

Во-вторых, выделенные ограничения на равенство позволяют упростить UNSAT-ядро решателя для типов. Связано это с тем, что в процессе его построения многие леммы содержат в себе ограничения на равенство и не равенство адресов между собой, а также адресов и нулевых ссылок. Поэтому на основании информации из выделенных ограничений можно упростить леммы.

### 3.2.2 Ленивость решателя для типов

Возможны ситуации, при которых не существует подходящего под имеющиеся ограничения типа для адреса, однако если бы этот адрес оказался нулевой ссылкой, то никакие ограничения не были бы нарушены. Чтобы решатель для типов не признавал невыполнимыми такие формулы, он

---

<sup>2</sup><https://github.com/ch3zych3z/VSharp/tree/constraintManagement>

был модифицирован следующим образом. Реализация принимает решение о невыполнимости формулы исключительно на основании адресов, точно не являющихся нулевыми ссылками. Данное знание может быть получено либо от выделенных ограничений на равенство, либо от модели SMT-решателя.

### **3.2.3 SMT-решатель**

Работа SMT-решателя в инкрементальном режиме является большим преимуществом для разработанного подхода. Используемый в проекте V# SMT-решатель поддерживает такой режим, однако эта функциональность ранее не применялась в проекте. Поэтому модуль взаимодействия с SMT-решателем был модифицирован для ее использования.

Также был доработан процесс кодирования и декодирования результатов работы SMT-решателя. В первую очередь был реализован сбор ограничений на типы и значений соответствующих им пропозициональных переменных во время построения модели. Кроме того, был расширен жизненный цикл кэшей, которые ранее инвалидировались после каждого запроса к SMT-решателю. В реализации инвалидация кэшей происходит после каждого запроса к гибриднему решателю.

## 4. Тестирование

Данная глава посвящена тестированию разработанного решения. В разд. 4.1 рассмотрен способ тестирования, принятый в проекте V#, а также описаны группы тестов, которыми был расширен тестовый набор проекта. В разд. 4.2 производится сравнение результатов тестирования до внедрения разработанной системы и после.

### 4.1 Способ тестирования

Тестирование в проекте V# производится преимущественно интеграционными тестами, представляющими из себя методы, написанные на языке C#, для которых необходимо сгенерировать модульные тесты. Далее полученные тесты запускаются на виртуальной машине .NET и рассчитывается полученное покрытие. Если результаты запуска сгенерированных тестов совпадают с ожидаемыми, а процент покрытия равен заданному числу, то тест считается успешно пройденным.

Тестирование было произведено на тестовом наборе проекта V#, который также был дополнен двумя группами тестов: NotLazyOr и NullLaziness. Первый проверяет работу символьного исполнения на программах с операцией «неленивого или» (оператор «|»), данный оператор ранее не поддерживался вовсе, однако благодаря гибриднему решателю стало возможным исследовать программы с его использованием. Второй в свою очередь представляет из себя программы, в которых присутствуют адреса, на типы которых в процессе исследования накладываются противоречивые ограничения, но при этом в качестве значения адреса допускается null.

### 4.2 Сравнение результатов

Разработанное решение прошло все тесты тестового набора проекта V#. На табл. 1 представлено сравнение получаемого тестового покрытия до и после внедрения разработанной системы. Представленное сравнение содержит тестовые группы, в которых задействован решатель для типов. В табл. 1 в первом столбце указано название тестовой группы. Во втором — покрытие,

полученное до внедрения разработанной системы, а в третьем — после.

| Название группы       | Покрытие «Old» | Покрытие «New» |
|-----------------------|----------------|----------------|
| MethodParameters      | 100            | 100            |
| MockRelocation        | 100            | 100            |
| DeepPropagating       | 100            | 100            |
| Nested                | 100            | 100            |
| NoSolution            | 100            | 100            |
| GenericCandidates     | 100            | 100            |
| Arrays                | 100            | 100            |
| ClassesPropertyAccess | 100            | 100            |
| ClassesHierarchy      | 100            | 100            |
| VirtualCall           | 100            | 100            |
| NotLazyOr             | 0              | 100            |
| NullLaziness          | 0              | 100            |

**Таблица 1:** Результаты тестирования разработанной системы

Полученные результаты свидетельствуют о том, что после внедрения разработанного решения тестовое покрытие, получаемое с помощью символического исполнения, не уменьшилось. Кроме того, оно увеличилось за счет поддержания новых случаев, таких как «ленивое или» и противоречащих ограничений на адрес, значение которого допускает null.

## Заключение

В ходе работы были получены следующие результаты.

- Изучен алгоритм DPLL(T), используемый SMT-решателями для работы в комбинации теорий.
- Разработана модель управления ограничениями, включающая в себя ограничения на типы и ограничения на равенство.
- Спроектирован процесс взаимодействия решателя для типов и SMT-решателя на основе DPLL(T) алгоритма.
- Разработанная система реализована в проекте V# на языке F#.
- Проведено тестирование полученного решения на тестовом наборе проекта V#.

## Список литературы

- [1] Alkhoun Osama. Language Specification. — 2011. — P. 77–92.
- [2] Cadar Cristian and Sen Koushik. Symbolic execution for software testing: three decades later // Communications of the ACM. — 2013. — Vol. 56, no. 2. — P. 82–90.
- [3] De Moura Leonardo and Bjørner Nikolaj. *Z3: An efficient SMT solver* // International conference on Tools and Algorithms for the Construction and Analysis of Systems / Springer. — 2008. — P. 337–340.
- [4] Ecma TC39. TG3. Common Language Infrastructure (CLI). Standard ECMA-335, June 2005.
- [5] Misonizhnik Aleksandr and Mordvinov Dmitry. On Satisfiability of Nominal Subtyping with Variance // 33rd European Conference on Object-Oriented Programming (ECOOP 2019) / Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. — 2019.
- [6] Niemetz Aina and Preiner Mathias. Bitwuzla at the SMT-COMP 2020 // arXiv preprint arXiv:2006.01621. — 2020.
- [7] Nieuwenhuis Robert, Oliveras Albert, and Tinelli Cesare. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL (T) // Journal of the ACM (JACM). — 2006. — Vol. 53, no. 6. — P. 937–977.
- [8] Păsăreanu Corina S and Visser Willem. A survey of new trends in symbolic execution for software testing and analysis // International journal on software tools for technology transfer. — 2009. — Vol. 11, no. 4. — P. 339–353.
- [9] Baldoni Roberto, Coppa Emilio, D’elia Daniele Cono, Demetrescu Camil, and Finocchi Irene. A survey of symbolic execution techniques // ACM Computing Surveys (CSUR). — 2018. — Vol. 51, no. 3. — P. 1–39.

- [10] Barrett Clark, Conway Christopher L, Deters Morgan, Hadarean Liana, Jovanović Dejan, King Tim, Reynolds Andrew, and Tinelli Cesare. *Cvc4* // International Conference on Computer Aided Verification / Springer. — 2011. — P. 171–177.