

Санкт-Петербургский государственный университет

ПОЖАРСКИЙ Роман Николаевич

Выпускная квалификационная работа

Символьная виртуальная машина .NET на основе платформы USVM

Уровень образования: бакалавриат

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

Основная образовательная программа *СВ.5162.2021 «Технологии программирования»*

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

Консультант:
ведущий разработчик, ООО «ИИнтелКод» Костицын М. П.

Рецензент:
старший академический консультант, ООО «Коулмэн Тех» Соболев В.

Санкт-Петербург
2025

Saint Petersburg State University

Roman Pozharskiy

Bachelor's Thesis

USVM-based symbolic virtual machine for .NET

Education level: bachelor

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

Programme *CB.5162.2021 "Programming Technologies"*

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

Consultant:
lead software engineer at "IntelCode" M.P. Kosticyn

Reviewer:
senior academic consultant at "Coulmen Tech" V. Sobol

Saint Petersburg
2025

Оглавление

Введение	4
1. Постановка задачи	5
2. Обзор	6
2.1. Символьное исполнение	6
2.2. SMT-решатели	10
2.3. Инструменты автоматической генерации тестов	12
2.4. Модели памяти	13
2.5. Платформа USVM	17
3. Архитектура	19
3.1. Архитектура проекта USVM-NET	19
3.2. Архитектура символьной машины	21
3.3. Модель памяти	22
3.4. Система валидации тестов	27
4. Особенности реализации	30
4.1. Структуры	30
4.2. Исключения	32
4.3. Валидатор тестов	32
5. Тестирование	34
Заключение	38
Список литературы	39

Введение

Символьное исполнение [15] — техника анализа программного обеспечения, идея которой заключается в исполнении кода функции на специальных константах, символах, которые представляют абстракцию над всеми возможными входными данными. Код программы исполняется со специальной операционной семантикой, а выполнимость условий, в которых участвуют символы, проверяется при помощи SMT-решателей (Satisfiability Modulo Theories). Результаты символьного анализа могут быть полезны для верификации программ [10, 12] и автоматической генерации тестового покрытия [11].

Известной проблемой символьного исполнения является взрыв путей [13], который обусловлен тем, что число возможных сценариев исполнения программы растёт экспоненциально, поэтому реализация инструментов символьного исполнения требует эффективной модели символьной памяти, управления ограничениями, а также использования эффективных структур данных.

USVM (Universal Symbolic Virtual Machine) [1] является платформой, реализующей ядро символьного исполнения на основе композиционной модели памяти, что позволяет переиспользовать данное ядро при реализации анализа для новых языков программирования. Сейчас платформа поддерживает исследование Java, Kotlin, Python, Go, Typescript.

Данная работа посвящена созданию символьной виртуальной машины для .NET на основе платформы USVM. По сравнению с предыдущим проектом для .NET — V# [2], использование универсальной платформы существенно упрощает разработку символьной виртуальной машины, предоставляя развитую символьную память, управление ограничениями и библиотеку для взаимодействия с SMT-решателями. Однако для использования USVM требуется реализовать трансляцию IL-байткода в трёхадресное представление, а также платформу для получения информации о .NET коде во время анализа программы в JVM окружении. Это было сделано вне рамок данной работы.

1. Постановка задачи

Целью работы является создание системы автоматической генерации тестов на основе символьного исполнения для платформы .NET при помощи платформы USVM. Для её выполнения были поставлены следующие задачи:

1. провести обзор инструментов для генерации тестов и способов моделирования символьной памяти с небезопасным контекстом;
2. спроектировать архитектуру символьной виртуальной машины USVM-NET;
3. реализовать символьную виртуальную машину USVM-NET;
4. провести интеграционное тестирование символьной машины.

2. Обзор

В этом разделе будет с примерами будет описана техника символьного исполнения, его абстракции, применимость результатов, проблемы 2.1. Далее пойдет речь о SMT-решателях, неотъемлемой части символьного исполнения 2.2. Также будут описаны подходы в индустрии к созданию инструментов автоматической генерации тестов 2.3 и моделированию символьной памяти 2.4. В последнем подразделе 2.5 будет более подробно описана платформа USVM, ее компоненты, необходимые для реализации символьного анализа.

2.1. Символьное исполнение

Символьное исполнение — техника анализа программного обеспечения, которая позволяет исполнить код функции в условиях неопределённости входных данных: вместо аргументов функции подставляются символьные константы, представляющие абстракцию над всеми возможными входными данными. При анализе виртуальная машина работает как с символьными объектами — параметрами функций и их производными, например, результатами вызова других функций на символьных аргументах, так и с конкретными объектами, выделенными, например, через оператор `new` в .NET. Код программы (байткод, трёхадресное представление и т.п.) исполняется со специальной операционной семантикой.

Важным аспектом этой семантики является задание поведения при исполнении логических операций. Если все аргументы операции конкретные, то значение операции можно вычислить непосредственно. Если же проверяется выполнимость условий, в которых участвуют символьные объекты, символьная машина кодирует условие в формулы логики первого порядка в теориях и делает запрос к SMT-решателю. Результатом проверки решателя может быть **SAT** и модель, конкретные значения параметров исследуемой функции, обращающие проверяемую формулу в истину; **UNSAT**, что означает, что не существует конкретных данных, на которых проверяемая формула будет истинной, или **UNKNOWN**.

Существует несколько стратегий вызова SMT-решателей: раннее вычисление (eager evaluation) и ленивое (lazy evaluation). Суть раннего вычисления заключается в отправке запроса к решателю сразу при встрече условия, что позволяет не тратить ресурсы на исследование недостижимых веток. При ленивом вычислении проверка выполнимости условий делается не сразу, а, например, в конце анализа для фильтрации недостижимых веток. На практике чаще всего применяется раннее вычисление.

Стадию символьного исполнения характеризуют следующие параметры:

1. текущая инструкция pc (program counter);
2. условие пути π , которая представляет минимальное предусловие, обеспечивающее определённый путь исполнения. В условие пути добавляются выражения из условных операторов (if, switch);
3. символьная память σ , сопоставляющая локациям в программе значения, находящиеся в этих локациях.

При помощи символьного исполнения выясним, может ли броситься исключение в результате исполнения функции на листинге 2.1. На рисунке 1 представлено дерево символьного исполнения этой функции. Символьный анализ начинается со стартового состояния s с пустой памятью, pc указывает на первую инструкцию метода, условие пути π есть `true`. Модель, соответствующую начальному состоянию в данном случае можно получить даже без вызова решателя: можно подставить значения параметров по умолчанию, положив, что $x = 0$.

```
1 public int Abs(int x) {
2     var r = x;
3     if (x < 0) {
4         r = -x;
5     }
6     if (r < 0) {
7         throw new Exception();
8     }
9     return r;
10 }
```

Листинг 2.1: Функция абсолютного значения

1. первая инструкция записывает в локальную переменную r значение символьной константы x ;
2. далее идёт проверка условия $x < 0$, в котором участвует символьная константа x , поэтому делается запрос к решателю. Это условие выполнимо, решатель предоставляет модель. При этом отрицание условия выполняется на стартовой модели $x = 0$, и s разветвляется на s_1 и s_2 , $\pi_1 = x < 0$, $\pi_2 = \neg(x < 0)$;
3. в состоянии s_1 выполняется инструкция $r = -x$, а состояние s_2 сразу переходит к инструкции в строке 6;
4. состояние s_1 проверяет возможность разветвиться на $r < 0$. Это условие действительно выполнимо при $x = -2147483648$ (минимальное значение `int`), в этом случае $r = -x = x < 0$, так как происходит переполнение 4-байтного целого числа. s_1 ветвится на s_3 и s_4 . В состоянии s_2 условие $r < 0$ оказывается невыполнимым с учётом памяти $r = x$ и $\pi = \neg(x < 0)$, и поэтому модель удовлетворяет условию $\neg(r < 0)$, и s_2 не делает ветвлений;
5. состояние s_3 выполняет инструкцию в строке 7 и бросает исключение, состояние s_2 и s_4 выполняют последнюю инструкцию метода, возвращая r .

Результатом символьного исполнения являются финальные состояния, соответствующие разным веткам исполнения программы, у каждого

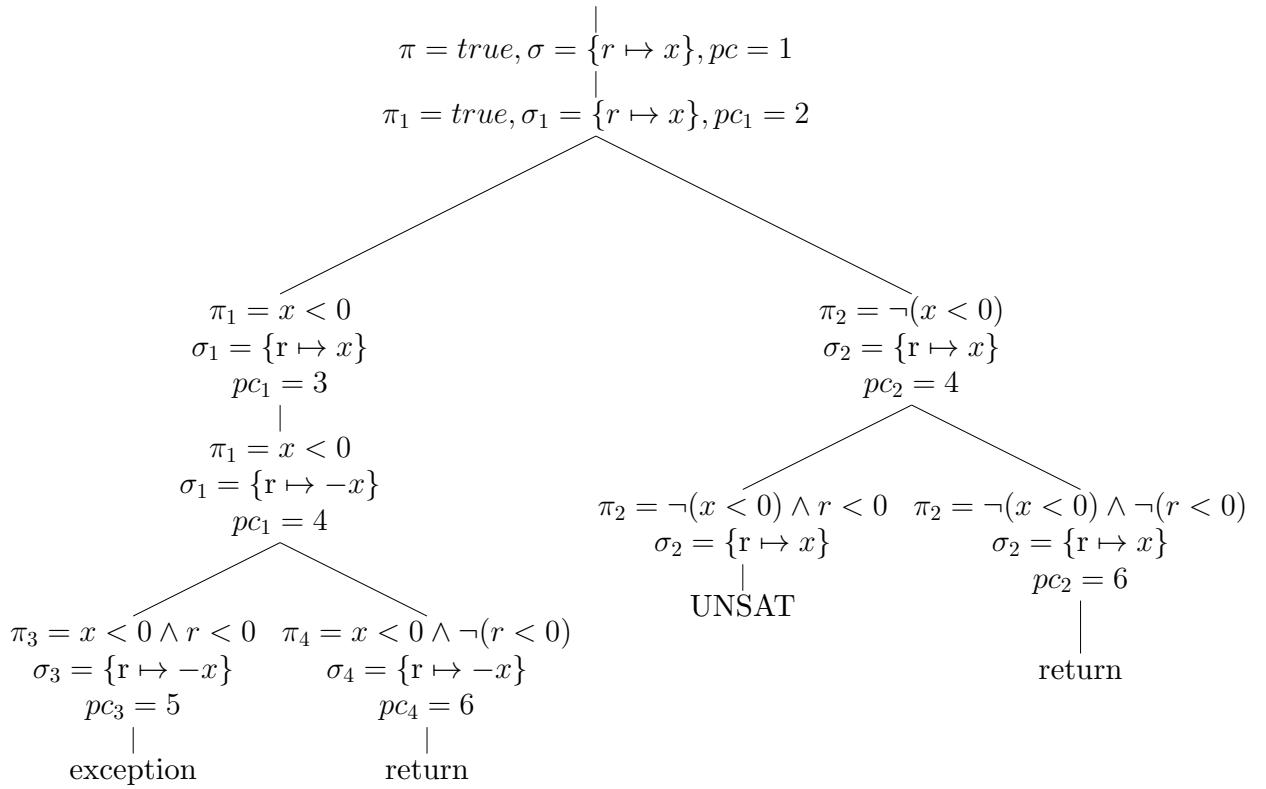


Рис. 1: Дерево символьного исполнения

состояния есть модель, которая содержит конкретные данные для обеспечения специфичной ветки исполнения. Таким образом, при помощи символьного исполнения можно решать вопросы достижимости определённых участков программы с некоторой входной точки. Эти данные можно использовать при поиске уязвимостей, доказательстве свойств программы, автоматической генерации тестов, реализации «умного» отладчика и т.д.

Несмотря на то, что инструменты символьного исполнения стараются исследовать каждое достижимое состояние, исследовать все ветви исполнения на полезных сложных программах почти никогда не удаётся, так как число возможных путей исполнения растёт экспоненциально из-за циклов с символьной границей, особенно если они содержат ветвления. Символьных состояний становится слишком много, и их проблематично не только исследовать до конца, но и просто хранить в памяти. В коде функции на листинге 2.2 есть 2 проблемы для символьного исполнения: во-первых, неизвестно сколько раз нужно исполнить тело цикла, и, во-вторых, в теле цикла есть ветвление по символьному объекту, и уже

только на 10 итерации будет 1024 состояния.

Описанная проблема носит известное название взрыва путей (path explosion) [13] при анализе программ. Однако в индустрии используются ограничения на символьные границы циклов, появляются эвристики, оптимизации, позволяющие исследовать сложные программы.

```
public void Foo(int n) {  
    for (i = 0; i < n; i ++) {  
        if (F(i)) {  
            ...  
        } else {  
            ...  
        }  
    }  
}  
  
bool F(int x) { ... }
```

Листинг 2.2: Взрыв путей

2.2. SMT-решатели

При чувствительным к путям анализе программ возникает необходимость проверять выполнимость условий. Для этого в символьном исполнении используют SMT-решатели, которые поддерживают специальные теории (линейная арифметика, массивы, строки и т.д.) В символьном исполнении SMT-решатели используются для поиска входных данных, удовлетворяющих условию пути π . Семантика исследуемой программы кодируется в формулы логики первого порядка в теориях, поддерживаемых решателями. Разрешающие процедуры решателей проверяют эти формулы и дают один из 3 результатов: SAT и модель, UNSAT или UNKNOWN. Модель декодируется в программные сущности (числа, структуры, классы, массивы). Разные теории лучше или хуже справляются с кодированием определенных конструкций: например, целые числа фиксированной длины удобно кодировать при помощи бит-векторов, а если кодировать их в линейную арифметику, придется

«вручную» выражать свойства, связанные с переполнением и знаком. SMT-решатели позволяют проверять выполнимость формулы, в которой используется несколько теорий: наиболее распространенным является метод комбинации теорий Nelson-Open. Теории имеют разную сложность (в большинстве случаев, от NP-полной до неразрешимой), и при неудачном выборе комбинации теорий для кодирования, инструмент анализа может стать полностью непригодным для исследования реального кода. Известно, что комбинация разрешимых теорий является разрешимой.

Далее будут приведены некоторые теории, поддерживаемые современными SMT-решателями.

2.2.1. Линейная целочисленная арифметика (LIA)

Теория в сигнатуре $\{\mathbb{Z}, 0, 1, +, \leq\}$ позволяет записывать системы линейных целочисленных уравнений. Не является особенно важной для верификации программ, поскольку числа в компьютере имеют фиксированный размер и особую семантику, для которой лучше подходит теория бит-векторов.

2.2.2. Бит-векторы

Теория предоставляет битовые вектора фиксированного размера и ряд типичных операций: сложение, умножение, битовые операции, расширение, вырезка. Теория бит-векторов поддерживает знаковые операции с числами в формате two's complement, поэтому в неё удобно кодировать целочисленные ограничения. Также в теории бит-векторов удобно представлять объекты для моделирования небезопасных операций с указателями и реинтерпретацией, что очень актуально для платформы .NET.

2.2.3. Теория массивов

Массивы абстрактных сортов (типов) с функциями `select(array, index)` (чтение из массива по индексу) и `store(array, index, value)`, которая возвращает новый массив, полученных из `array` записью `value` по индексу

index. В теорию массивов удобно кодировать операции с стандартными коллекциями Array, List, Set, Map.

К современным state-of-the-art SMT-решателям можно отнести Z3 [3], Yices [4], CVC5 [5], Bitwuzla [6].

2.3. Инструменты автоматической генерации тестов

Существующие инструменты, позволяющие автоматически генерировать тесты для платформы .NET, часто используют подходы, совмещающие конкретное и символьное исполнение в том или ином виде, поскольку конкретное исполнение может задавать путь для символьного исполнения, или же сильно ускорить анализ в некоторых случаях.

2.3.1. Инструмент V#

Инструмент V# [2] анализирует код при помощи динамического символьного исполнения. V# поддерживает подсистему конкретной памяти и исполняет функции в конкретном режиме, если ни один из её аргументов не зависит от символьных данных, что позволяет существенно ускорить время анализа. По финальным состояниям символьной машины генерируются Unit-тесты в формате AAA (Act, Arrange, Assert) на языке C#.

2.3.2. Инструмент IntelliTest

Инструмент IntelliTest [16] также базируется на динамическом символьном исполнении, однако же конкретное исполнение используется для другой цели. IntelliTest использует инструментирование исходного кода для параллельного исполнения кода в конкретном и в символьном режиме. При этом конкретное исполнение «направляет» символьное: SMT-решатель не используется во время исполнения программы, он используется только для поиска входных данных, отвечающих еще не пройденным ветвям исполнения. IntelliTest является частью среды разработки Visual Studio Enterprise и также генерирует Unit-тесты в

формате AAA. В связи с выбором динамического символьного исполнения для анализа программы, обладает низкой инкрементальностью и может пропустить «интересные» ветви исполнения при неразрешимости ограничений на произвольном этапе исследования.

2.4. Модели памяти

Модельная семантика[17] языка программирования определяет комбинацию теорий, в которые кодируются программные объекты и непосредственно алгоритмы кодирования. В поддерживающих синонимичные указатели языках, к которым относится большинство императивных языков, необходимо формализовать понятие адреса (указателя) и возможные операции над ним. Фрагмент модельной семантики, формализующий эти аспекты, называется *моделью памяти*. Итак, модель памяти символьной машины — это формальное описание типичных операций с памятью: чтения, записи, выделения.

Наиболее сложно моделировать операции в языках с динамически выделяемой памятью (Java, C#, Go, C/C++) из-за доступа по символьным адресам, когда нужно определить все объекты, которые потенциально могут быть прочитаны по адресу и отразить информацию об изменении в случае записи. Для полноты анализа символьная машина должна проводить анализ указателей (pointer analysis, points-to-analysis, alias analysis): для каждого адреса необходимо определить множество переменных, которые могут ссылаться на этот же адрес.

Функция Foo с листинга 2.3 возвращает 1, если вызывающий в качестве аргументов передал один и тот же массив, поэтому запись по адресу $a[0]$ должна быть учтена при чтении $b[0]$.

```
public int Foo(int[] a, int[] b) {
    Debug.Assert(b[0] != 1)
    a[0] = 1;
    if (b[0] == 1) return 1;
    return 0;
}
```

Листинг 2.3: Aliasing

Анализ псевдонимов затрудняется полиморфизмом подтипов, параметрическим полиморфизмом, позволяющим передавать в функцию, ожидающую аргумент типа `List<object>`, список строк `List<string>`.

Далее будет приведено несколько подходов к моделированию операций с памятью в символьном исполнении на основе статьи [14]. Подходы можно классифицировать по

1. используемым SMT-теориям для представления объектов;
2. полноте (некоторые модели не поддерживают динамическую память, вложенные сложные объекты, реинтерпретацию объектов);
3. способ организации объектов в памяти (линейный, сегментный).

2.4.1. Инструмент KLEE

KLEE [7] — символьная виртуальная машина с открытым кодом, написанная на C++ для анализа IR-кода платформы LLVM. В KLEE каждый объект в памяти представлен массивом бит-векторов (ABV), а сама память является массивом объектов. При доступе к объекту по символьному указателю, SMT-решатель определяет все возможные локации, на которые может ссылаться указатель, и исходное состояние разветвляется на каждое возможное конкретное значение адреса. Так как в каждом состоянии символьный адрес конкретизируется, исчезает необходимость проводить анализ псевдонимов, однако при таком подходе при доступе по символьным указателям появляется огромное количество символьных состояний. Так как каждый объект представляется в виде массива бит-векторов, у двух состояний может быть совершенно разное размещение объектов в памяти, что сильно усложняет объединение символьных состояний с целью сэкономить ресурсы на исполнении инструкций для состояний с похожими путями исполнения.

2.4.2. Инструмент BINSEC

Инструмент динамического символьного исполнения на бинарном уровне, написанный на OCaml. Модель памяти BINSEC [8] рассматри-

вает всю память как единый массив бит-векторов, операции с памятью делегируются функциям `Select` и `Store` теории массивов. Недостатками такого подхода к моделированию памяти, где вся логика доступа к памяти делегирована решателю, являются высокая нагрузка на решатель и невозможность проводить анализ псевдонимов.

2.4.3. Платформа ANGR

Платформа ANGR [9] представляет набор Python библиотек для анализа бинарного кода, в том числе и при помощи символьного исполнения. ANGR транслирует бинарный код в промежуточный язык VEX, предоставляя символьную виртуальную машину, не привязанную к конкретной архитектуре. Символьная память ANGR представляет набор конкретных адресов, которым сопоставлены объекты, представленные бит-векторами (BV).

ANGR идёт на компромисс между полнотой и скоростью, используя модель памяти, где каждая запись по символьному указателю конкретизируется до наибольшего возможного адреса, которому может быть равен указатель. При этом часть потенциально интересных ветвей исполнения с записью по меньшим адресам отсеивается. Чтение реализовано в более гибком виде: сначала определяется множество адресов, которые может затрагивать символьный указатель p . Если число возможных чтений меньше некоторого настраиваемого параметра n , то результатом чтения является выражение $\text{Union}((g_1, v_1), \dots, (g_m, v_m))$, $m < n$, g_i — защитные условия, содержащие условие равенства p и локации, в которой находится v_i , и самих значений v_i .

2.4.4. Инструмент V#

V# — символьная виртуальная машина для платформы .NET, анализирует непосредственно CIL. Подход V# к моделированию символьной памяти основывается на разбиение всей памяти на непересекающиеся области, *регионы памяти*. В регионе памяти хранятся записи по адресам, которые потенциально могут указывать на один объект в памяти.

В проекте реализован алгоритм анализа псевдонимов, который сопоставляет ключу записи соответствующий регион. Имеются следующие регионы (приведены с идентификаторами для анализа псевдонимов)

1. содержимое массивов (тип элемента);
2. нижние границы массивов (тип элемента);
3. длины массивов (тип элемента);
4. поля классов (название, тип поля, тип класса, в котором объявлено поле);
5. статические поля (тип).

Модель с регионами предоставляет компромисс между слишком частым разветвлениями состояний и уровнем неопределённости: неопределённость существует в рамках региона. К её преимуществам можно отнести быстрый доступ к релевантным для операции объектам благодаря трансляции ключа чтения/записи в регион, поддержка анализа псевдонимов и удобство кодирования запроса к решателю благодаря поддержке иерархии записей внутри региона.

2.4.5. Инструмент USVM

Модель памяти USVM является идеологическим наследником $V\#$, дополняя её и решая некоторые важные проблемы. Наиболее значимые отличия:

1. в USVM реализован более точный алгоритм поиска псевдонимов, основанный на тривиальном наблюдении, что символьный адрес не может быть равен конкретному, и никакие конкретные адреса не могут быть равны друг другу; также модифицированный алгоритм решает проблемы с параметрическим полиморфизмом и приведениями типов, когда, например, в функцию, ожидающую `Object[]`, вызывают с аргументом `String[]`: при чтении из `Object[]` необходимо учитывать все записи в массивы с ссылочным типом элемента;

2. в USVM, в отличие от V#, отсутствует подсистема конкретной памяти: все записи транслируются в регионы;
3. в V# доступ к памяти осуществляется через высокоуровневый интерфейс символьного состояния, в USVM имеется API для памяти и даже к регионам, что позволяет произвести операцию по указанному ключу: в USVM нет единой инстанции, через которую гарантированно проходят все операции с памятью.

2.5. Платформа USVM

USVM (Universal Symbolic Virtual Machine) — универсильная символьная виртуальная машина, которая предоставляет платформу для символьного анализа исходного кода на разных языках программирования, предварительно транслируя исходный код в трёхадресное представление. На данный момент на основе ядра USVM реализованы символьные машины для анализа Java, Kotlin, Python, Go, Typescript. Несмотря на отличительные особенности каждой платформы, многие компоненты символьной машины можно реализовать универсальным образом: это касается управления ограничениями, взаимодействия с решателем, подсистеме выбора состояний, общей модели памяти. Ядро реализовано таким образом, чтобы на его основе можно было спроектировать символьную машину с моделью памяти на основе регионов, добавив поддержку особенностей целевой платформы через расширения базовой реализации.

Диаграмма компонентов платформы USVM представлена на рисунке 2, далее приведены пояснения:

- **USVM** содержит специфичные для символьного исполнения абстракции (машина, интерпретатор, состояние, память, ограничения);
- **JacoDB** — библиотека, позволяющая получать информацию о программе во время выполнения. Типы, атрибуты, поля, мето-

ды, их инструкции, хранятся в базе данных, и главная задача JacoDB — организовать эффективный доступ к ней;

- **KSMT** предоставляет API к нескольким высокопроизводительным SMT-решателям (Z3, CVC5, Yices, Bitwuzla) и реализует около 200 упрощений символьных выражений.

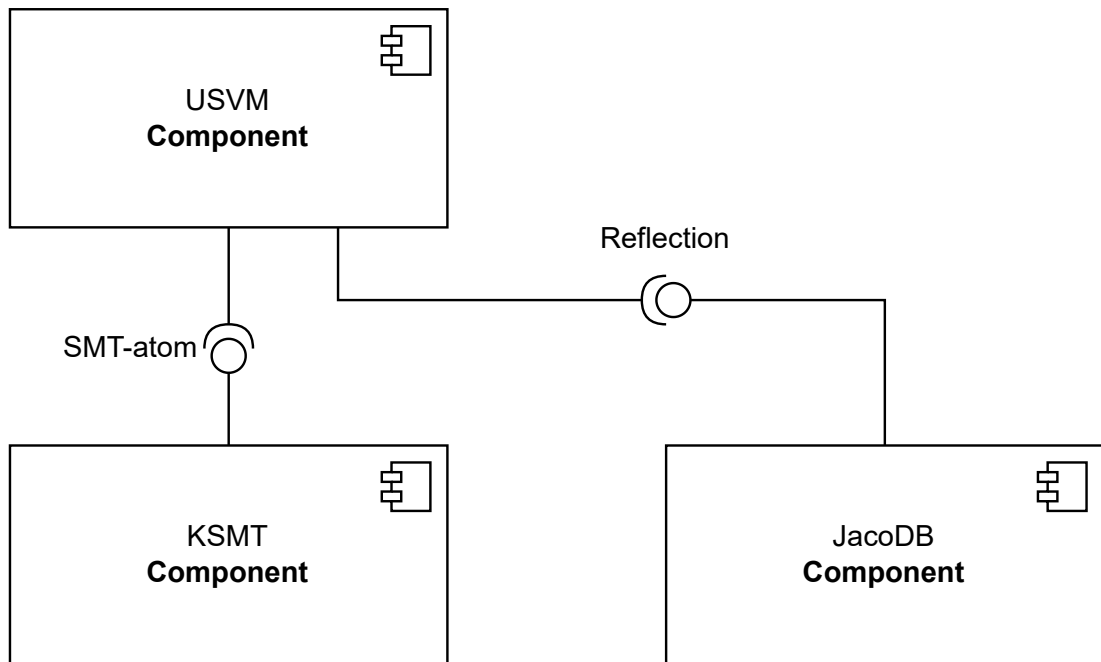


Рис. 2: Диаграмма компонентов USVM

Для реализации символьной .NET машины, необходимо реализовать построение трёхадресного кода исследуемой сборки, сохранения метаданных о коде в JacoDB для .NET и реализовать символьный интерпретатор инструкций в USVM. Тема текущей работы относится к проектированию и реализации символьной виртуальной машины в рамках компоненты USVM.

3. Архитектура

В разделе 3.1 представлена архитектура проекта USVM-NET. Далее будет описана архитектура символьной виртуальной машины USVM-NET-SE 3.2. В разделе 3.3 приводится организация памяти USVM-NET. В последнем разделе 3.4 будет описана подсистема для генерации тестов и валидации результатов работы символьной машины.

3.1. Архитектура проекта USVM-NET

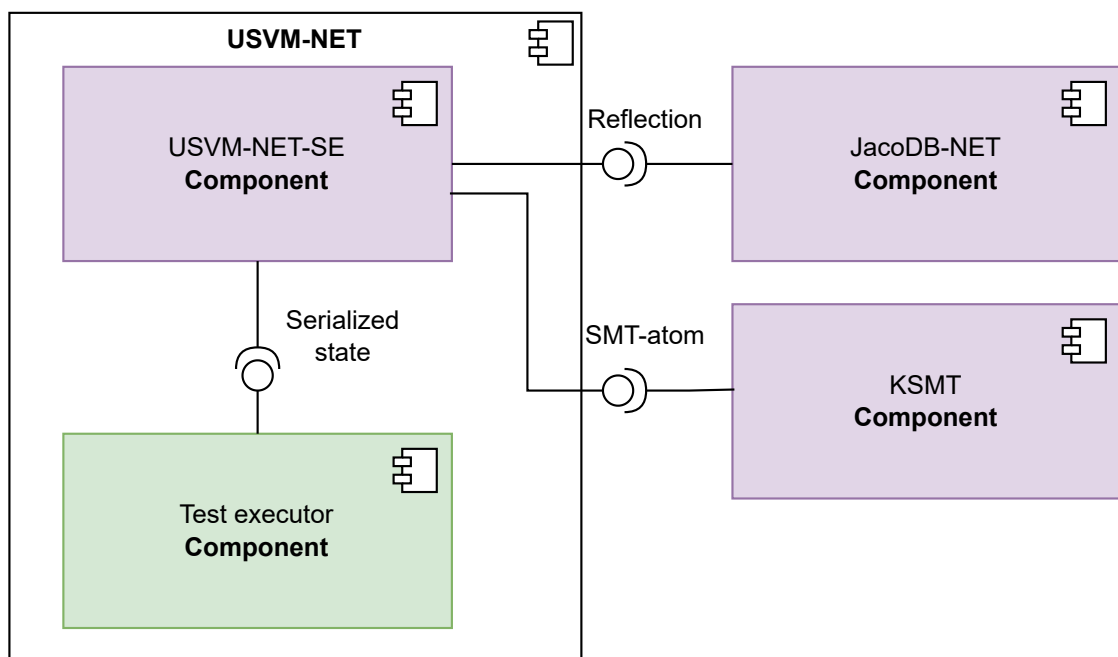


Рис. 3: Архитектура проекта USVM-NET

На рисунке 3 представлена архитектура проекта USVM-NET. Зелёным цветом выделены компоненты, реализованные на .NET, а компоненты, выделенные фиолетовым цветом, написаны на Kotlin:

1. в **USVM-NET-SE** реализован символьный анализ кода;
2. финальные состояния символьного исполнения сериализуются в тесты, которые воспроизводит **Test executor**;

3. **JacoDB-NET** строит трёхадресный код исследуемой сборки, хранит и предоставляет эффективный доступ к метainформации о коде: типы, методы, блоки для обработки исключений;
4. **KSMT** предоставляет API к SMT-решателям.

В рамках данной работы были созданы компоненты USVM-NET и валидатор тестов Test executor.

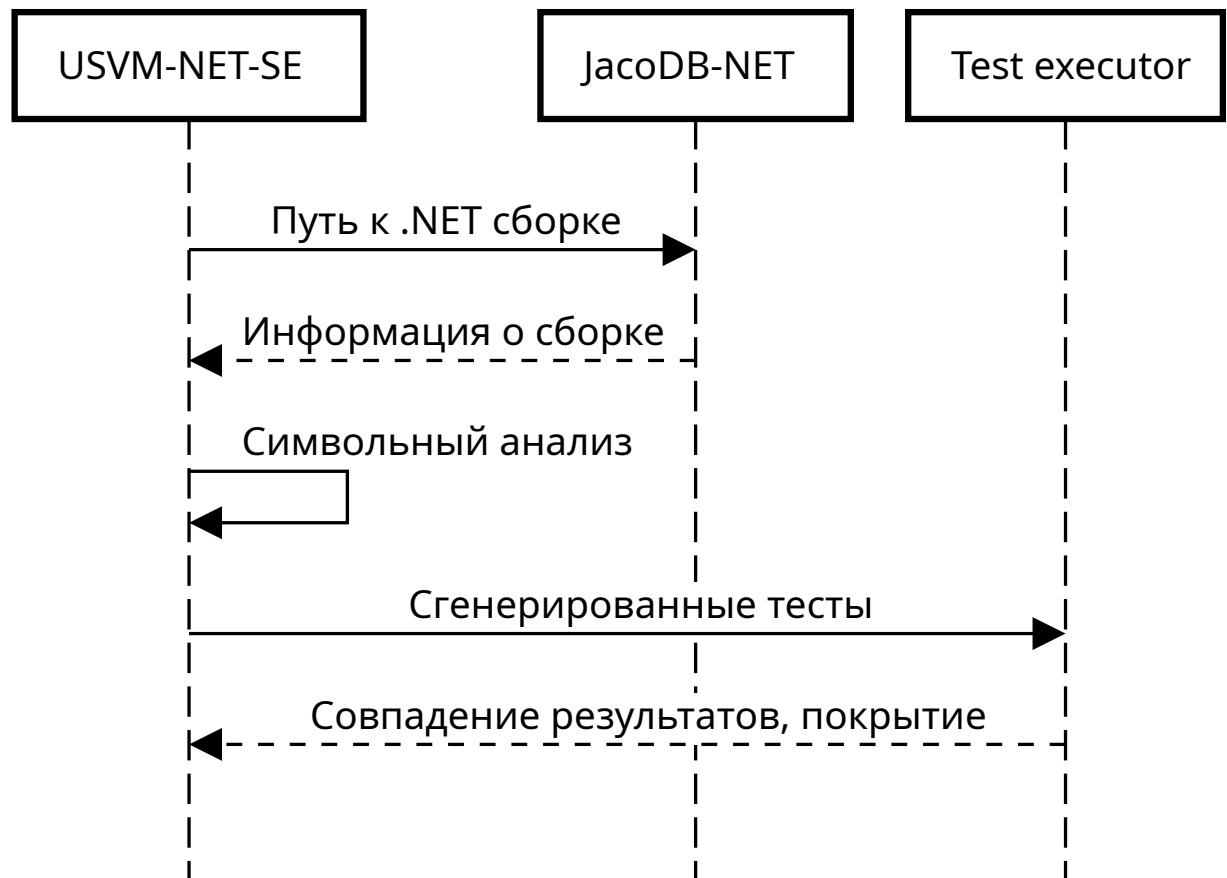


Рис. 4: Диаграмма последовательностей анализа USVM-NET

На рисунке 4 представлена диаграмма последовательностей генерации тестов на основе символьного исполнения:

1. символьная машина получает путь к сборке для анализа;
2. символьная машина отправляет запрос на получение информации о сборке в библиотеку JacoDB-NET;

3. символьная машина проводит анализ и возвращает список финальных состояний;
4. по результату и параметрам метода создаётся тест в некотором DSL, который затем воспроизводит Test executor в .NET процессе;
5. Test executor возвращает ответ с соответствием результатов символьного и конкретного исполнения и покрытием.

3.2. Архитектура символьной машины

На рисунке 5 представлена архитектура символьной машины USVM-NET-SE, которая на высоком уровне абстракции схожа с любой другой символьной машиной на основе USVM.

ILMachine — символьная машина, которая поддерживает множество состояний **states**, из которых **PathSelector** на основе множества алгоритмов может выбирать следующее к исполнению состояние. Интерпретатор **ILInerpreter** выполняет одну инструкцию состояния. **ILInterpreter** оперирует инструкциями трёхадресного кода **ILStatement**, например, **Assign**, **Call**, **Goto**, **Return**, **Throw**. **ILExprResolver** конвертирует выражения трёхадресного кода в символьные выражения **UExpr**.

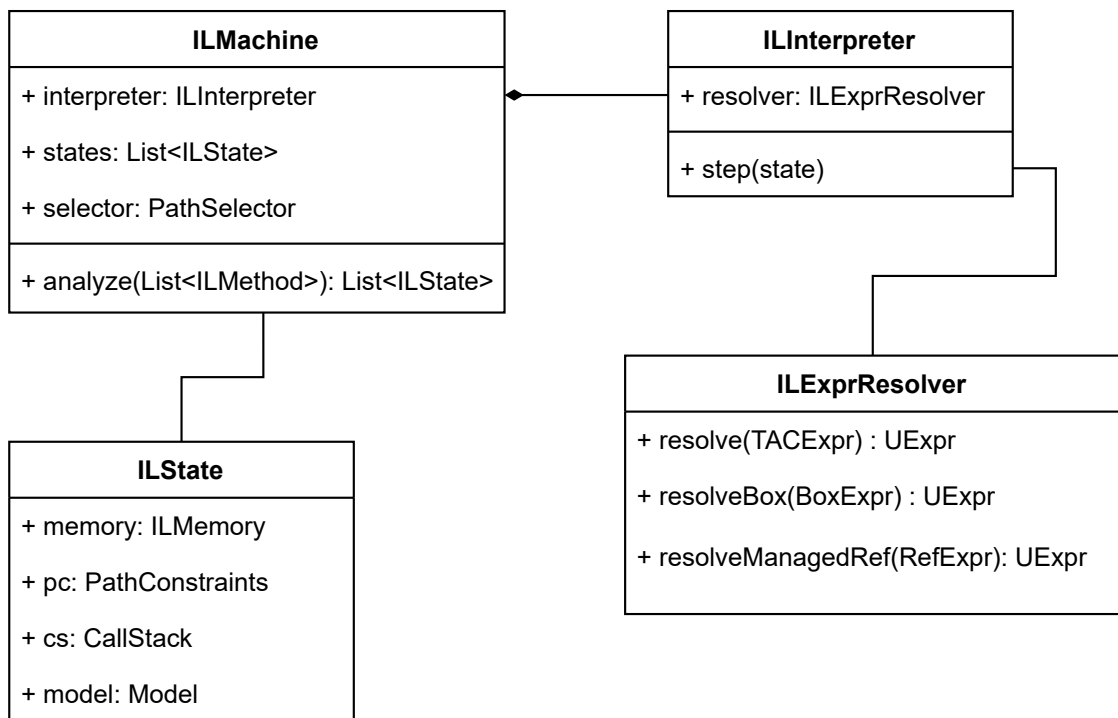


Рис. 5: Архитектура символьной машины

Все особенности и семантика платформы .NET сконцентрированы в компонентах **ILInterpreter** и **ILExpResolver**.

3.3. Модель памяти

Ядро USVM предоставляет композиционную модель памяти на основе регионов, и в нём уже реализованы регионы для доступа к одномерным массивам и полям классов. Однако особенности платформы .NET требуют реализации специфичных регионов для доступа к статическим полям, byref-параметрам и значимым типам на куче после операции `box`. Устройство новых регионов будет описано в подразделе 3.3.1. Так как .NET программы могут использовать небезопасный код с реинтерпритацией данных, в символьной памяти необходимо моделировать и небезопасные операции, этому аспекту посвящен подраздел 3.3.2.

3.3.1. Моделирование операций в безопасном контексте

В USVM записи хранятся в регионах, реализациях интерфейса **UMemoryRegion**, у которых есть идентификаторы **UMemoryRegionId**. Записи и чтения из регионов осуществляются по ключам **ULValue**, уникальным для каждого региона. На рисунке 6 показана архитектура этих регионов, где добавленные сущности выделены зеленым цветом.

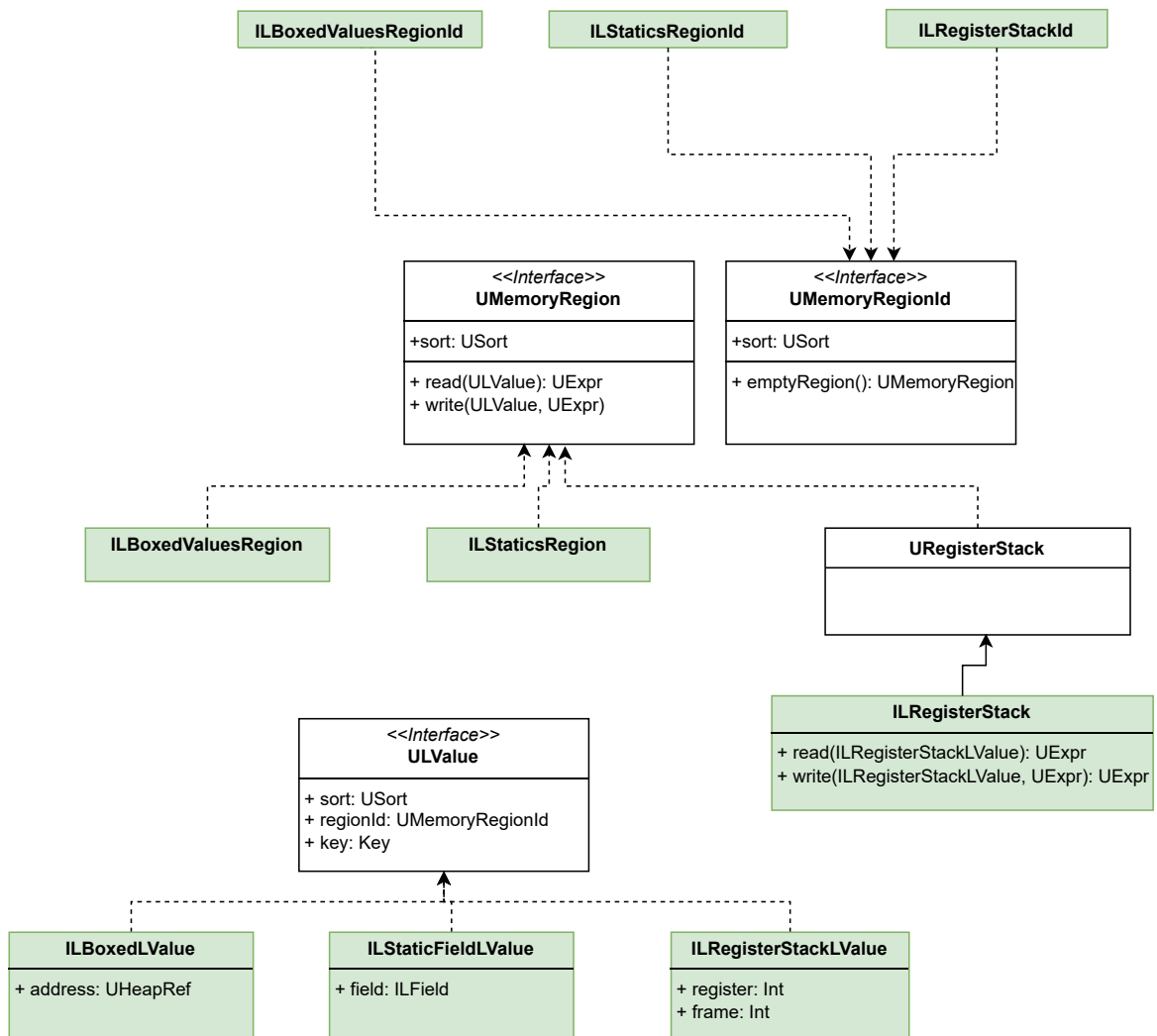


Рис. 6: Регионы памяти USVM-NET

Для значимых типов на куче после операции **box** был реализован регион **ILBoxedValuesRegion** с идентификатором **ILBoxedValuesRegionId**, доступ к которому осуществляется

по ключам **ILBoxedValue**, который состоит из адреса объекта.

Статические поля классов в USVM-NET хранятся в регионе **ILStaticsRegion** с идентификатором **ILStaticsRegionId** и ключом **ILStaticFieldLValue**, который хранит поле **ILField**.

Так как .NET поддерживает управляемые рантаймом указатели (например, **byref** параметры), которые можно передавать как аргументы функций, для USVM-NET был реализован **ILRegisterStack**, являющийся наследником базового **URegisterStack**, который позволял лишь обращаться к регистрам (переменным) на последнем фрейме. Идентификатором этого стека является **ILRegisterStackId**, а ключами служат **ILRegisterStackLValue**, которые помимо номера регистра, содержат и номер фрейма регистра.

USVM-NET использует стандартную реализацию регионов и ключей из ядра для доступа к полям классов и массивов.

3.3.2. Моделирование небезопасных операций

В этом разделе будет представлена архитектура спроектированной модели памяти с поддержкой указателей и реинтерпретаций на основе подхода $V\#$. Её ключевыми особенностями являются:

- небезопасный доступ к памяти выражен с помощью безопасного;
- обобщённость, что позволит переиспользовать её при реализации символьной машины для анализа LLVM (C/C++) на основе USVM.

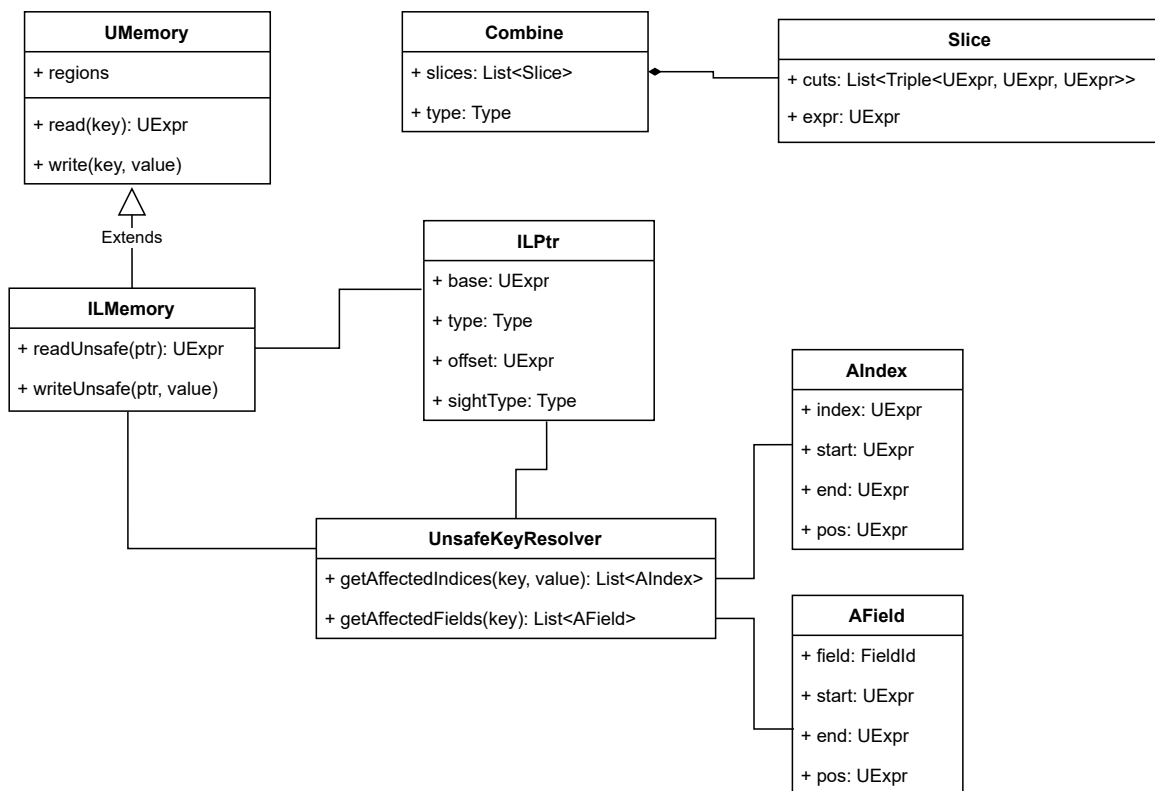


Рис. 7: Архитектура небезопасной памяти USVM-NET

На рисунке 7 представлена архитектура компонентов, обеспечивающих небезопасный доступ к памяти.

UMemory — универсальное хранилище регионов памяти, доступ к которому осуществляется по ключам, на основе которых проводится анализ псевдонимов. UMemory используется в качестве памяти во всех символьных виртуальных машинах.

Для USVM-NET используется расширение UMemory — **ILMemory**, которая предоставляет доступ к тем же регионам по указателям **ILPtr**, которые состоят из **base**, выражения, на которое взят указатель; его типа **type**; отступа **offset** и типа, с которым рассматривается выражение **sightType**.

Как уже было отмечено в преимуществах предлагаемой модели, небезопасные операции сводятся к безопасным при помощи **UnsafeKeyResolver**, который определяет безопасные ключи доступа, которые могут быть задействованы во время небезопасной операции. После определения списка безопасных ключей, которыми могут быть

индексы массива **AIndex**, поля классов и структур **AField**, переменные на стеке, составляется **Combine** в качестве результата чтения, или, в случае записи, в регионы памяти транслируется информация об изменении при помощи новых выражений **Slice** и **Combine**, которые получены из значения **value** операции записи.

Теория бит-векторов используется для кодирования символьных выражений, полученных при помощи небезопасного взаимодействия с памятью.

Символьные выражения **Slice** и **Combine** представляют результаты небезопасного доступа к памяти. Так как доступ по указателю с отступом сводится к доступу по безопасным ключам, для хранения информации о таких затронутых небезопасной операцией объектах используется **Slice**. **Combine** — это более высокоуровневый результат небезопасной операции.

1. **Slice** состоит из атомарного выражения *expr*, и его рассматриваемой части: *start*, *end*, *position* (внутри **Combine**). Части представлены в виде списка, поскольку небезопасные операции могут наслаиваться друг на друга. Если при подстановке конкретных данных **Slice** не корректен (например, имеет отрицательную длину), последующие куски инвалидируются;
2. **Combine** хранит список **Slice** и тип, с которым рассматривается набор **Slice**.

Например, результатом чтения из массива $a : \text{Int}[]$ с символьным отступом i по указателю **long*** будет **Combine** типа **long** с тремя **Slice**:

- $a[i/4 + 0], [i - i/4 * 4 - 0, i - i/4 * 4 + 8, i/4 * 4 - i + 0]$
- $a[i/4 + 1], [i - i/4 * 4 - 4, i - i/4 * 4 + 4, i/4 * 4 - i + 4]$
- $a[i/4 + 2], [i - i/4 * 4 - 8, i - i/4 * 4 + 0, i/4 * 4 - i + 8]$

Представленная модель основывается на подходе $V\#$ к моделированию небезопасных операций с памятью, и при анализе релизации

этого подхода было выявлено несколько недостатков. В частности, имела место неэффективность при кодировании запроса к решателю из-за излишнего переворота бит-векторов, а также алгоритм определения задействованных ключей массива не учитывал крайние ситуации, когда запись по указателю размером больше одного байта могла повлиять на две части составного объекта. Эти недостатки были устранены в пулл-реквесте <https://github.com/VSharp-team/VSharp/pull/331> и учтены при реализации модели для USVM-NET.

3.4. Система валидации тестов

Для того чтобы генерировать тесты по результатам символьного исполнения и валидировать их, была создана система интеграционного тестирования всей символьной машины, архитектура которой изображена на рисунке 8.

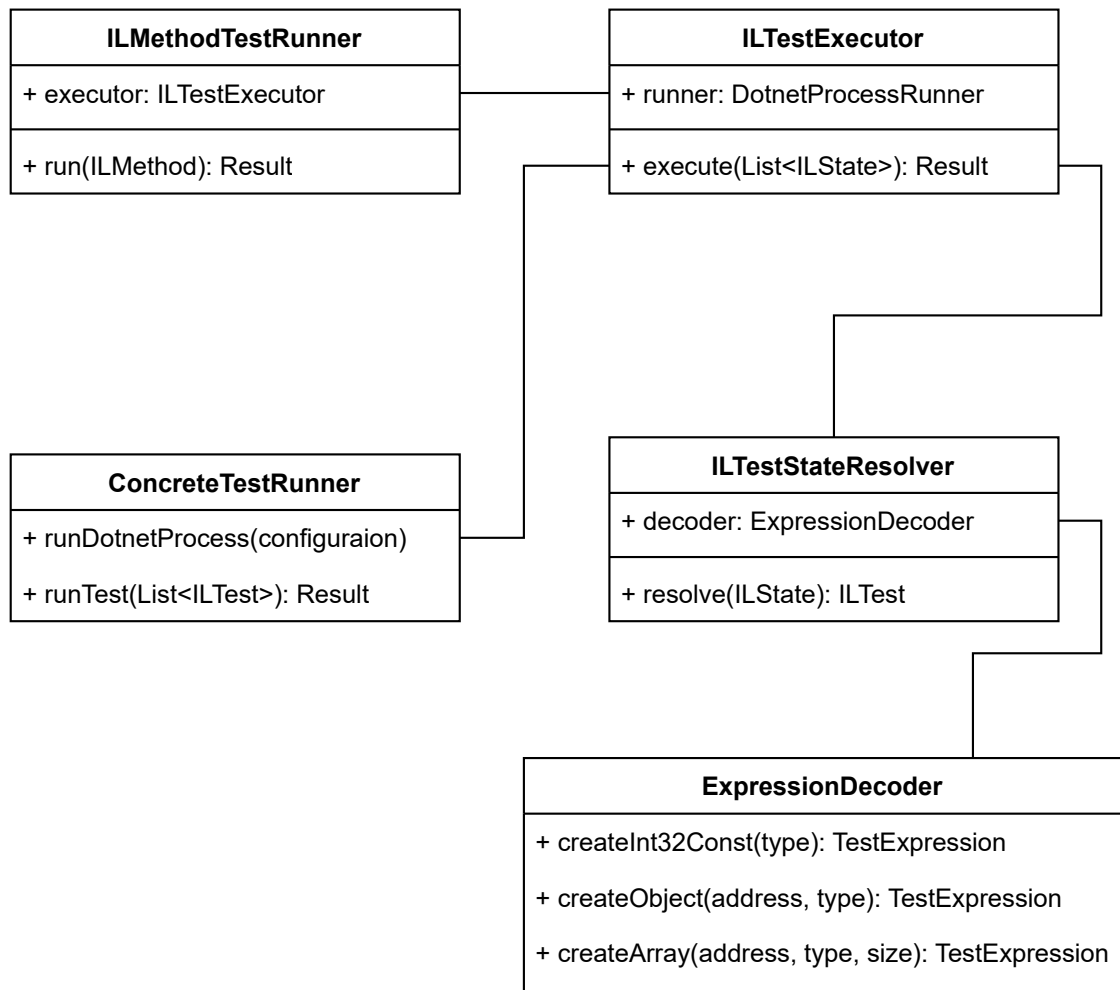


Рис. 8: Архитектура подсистемы валидации генерируемых тестов

Сначала символьная машина получает запрос на анализ метода через функцию `run` сущности **ILMethodTestRunner**.

Проводится символьный анализ, результатом которого является множество финальных состояний с результатами исполнения (возвращаемое значение или необработанное исключение). Эти состояния далее отправляются в **ILTestExecutor**: сначала состояния при помощи **ILTestStateResolver** сериализуются в **ILTest**. **ILTestStateResolver**, имея модель и память состояния, воссоздаёт метаинформацию о результате символьного исполнения и параметрах исследуемого метода через **ExpressionDecoder**, сериализуя их в сущности **TestExpression** некоторого DSL (Domain Specific Language):

- примитивные объекты (численные типы, строки);
- создание объекта класса или структуры;
- создание массива;
- запись в поле класса или структуры;
- запись в массив.

ConcreteTestRunner запускает .NET сервер, который десериализует `ILTest`, создаёт через рефлекссию параметры метода, вызывает его и сравнивает ожидаемый результат символьного исполнения с результатом конкретного исполнения.

Для похожих целей валидации символьного исполнения в проекте `V#` был написан инструмент, позволяющий измерять покрытие метода, исполненного конкретно по тестам, созданным аналогичным образом. Инструмент работает как профилятор произвольного .NET процесса, который перед JIT-компиляцией вставляет специальные инструкции, благодаря которым можно понять, какие инструкции исследуемого метода были исполнены, что критично для определения недостижимых веток исполнения. В проекте `USVM-NET` был переиспользован этот инструмент.

При воспроизведении тестов, которые задействуют небезопасные операции с памятью, запись по указателю в защищённую память может завершиться бросанием исключения `AccessViolationException`, которое нельзя поймать `catch` конструкцией. Это означает, что процесс, в котором работает сервер, воспроизводящий тесты, завершится с ошибкой. По этой причине интерпретатор помечает состояния, которые соответствуют сценариям, где может выброситься это исключение, и `Test executor` эти тесты не воспроизводит.

4. Особенности реализации

В данной главе приведены несколько важных особенностей реализации: поддержка особенностей целевой платформы в рамках ограничений USVM: структуры 4.1, обработка исключений 4.2. Также приведены особенности реализации валидатора состояний 4.3.

4.1. Структуры

По умолчанию структуры в .NET выделяются на стеке, однако впоследствии могут оказаться на куче, например, если они хранятся в коллекциях на куче или подвергаются операции `box`. Поэтому для доступа к полям структуры невозможно определить универсальный регион памяти и соответствующий ключ. Поэтому структуры удобно моделировать как объекты классов, которые всегда хранятся на куче и имеют ссылку `UHeapRef`. В USVM поля классов хранятся в разных регионах, доступ к которым осуществляется по ключам, которые состоят из ссылки на объект и поля, то же самое можно делать и с полями структур. Однако необходимо сохранить семантику структур как типов-значений: если структура передаётся как аргумент функции или присваивается в локальную переменную, выделяется новый адрес и поля структуры копируются. Регионы памяти устроены так, что изменение структуры с новым адресом не будут влиять на структуру, из которой была скопирована новая.

При описанном подходе возникают проблемы с символьными структурами. Поскольку в символьной машине у структур, независимо от локации, появляется адрес, который может быть символьным, и модель SMT-решателя может положить адреса произвольных структур одинаковыми. У структур возникает проблема псевдонимов, описанная в обзоре 2.3. В примере на листинге 4.1 структуры $a.s$ и $b.s$, как экземпляры значимых типов, могут быть одним и тем же объектом, только если $a = b$ (ссылочное равенство), однако для SMT-решателя доступы к полям объектов кодируются как `Select(address, updatesArray)` в теории массивов, где `address` — адрес объекта, поле которого читается,

`updatesArray` — массив, получающийся в результате трансляции региона памяти USVM для записей в поле в более простые выражения KSMT. Таким образом, даже если $a \neq b$, $a.s$ и $b.s$ для SMT-решателя — это чтения из одного и того же массива по индексам a и b , которые решатель в модели может сделать одинаковыми при отсутствии дополнительных ограничений. Это означает, что при $a \neq b$, запись в `a.s.age` может изменить `b.s.age` в символьной памяти.

```
public struct Person
{
    public int age;
    public string name;
}
public class ClassWithStruct
{
    public Person s;
    public int x;
}
public int StructsAsClassFieldsAliasing(ClassWithStruct a, ClassWithStruct b)
{
    if (a != b) {
        a.s.age = 10;
        b.s.age = 20;
        if (a.s.age == 20) {
            return -1;
        }
        return 0;
    }
    return 1;
}
```

Листинг 4.1: Aliasing

Чтобы не допустить синонимичность символьных структур, было придумано следующее решение: на уровне символьного интерпретатора у всех типов структур появляется искусственное поле `location` типа `Object`, `s` при чтении или записи в поле структуры `s` в условие пути добавляется ограничение `s.location = structLocation`, где `structLocation` — локация, из которой была считана структура `s`. В примере на листинге 4.1 в качестве `structLocation` будут выступать ссылки a и b . Это ограничение не позволит SMT-решателю положить

$a.s = b.s$, если $a \neq b$, так как в противном случае `a.s.location = a = b.s.location = b`.

4.2. Исключения

В .NET алгоритм обработки исключений состоит из двух этапов. Во время первого прохода (first bypass) рантайм ищет подходящий обработчик исключения, не снимая фреймы со стека вызова, при этом для поиска обработчика может исполняться код: в CIL имеются фильтры, которые соответствуют конструкции `catch (Exception) when (condition)` в C#. Вычисление `condition` может вызывать другие функции, и если фильтр бросает необработанное исключение, он считается непройденным, и ищется следующий обработчик предыдущего исключения. После того, как был найден обработчик исключения, начинается этап второго прохода (second bypass), когда перед тем, как исполнять обработчик, необходимо исполнить `finally` блоки из защищённых блоков, которые не смогли обработать исключения.

Описанный механизм обработки исключений обязывает хранить текущий рассматриваемый фрейм в `ILRegistersStack`, поскольку во время первого прохода фильтры исполняются без снятия стековых фреймов. Так как код в фильтрах может вызывать другие функции, после их исполнения необходимо восстанавливать текущий рассматриваемый фрейм вызывающей функции, поэтому стек вызовов `UCallStack` для USVM-NET был модифицирован для хранения этого фрейма.

4.3. Валидатор тестов

Для сериализации финальных символьных состояний использовался протокол сериализации Protobuf, для взаимодействия с сервером использовался gRPC, который зачастую использует Protobuf для сериализации данных. Такой стек технологий был выбран по следующим причинам

1. скорость и размер данных. Protobuf — протокол бинарной сериализации, допускающий сжатие;

2. компилятор `protoc` может генерировать код для C# и Kotlin;
3. возможность выразить полиморфизм сущностей DSL при помощи специального класса `Any`. Класс `Any` в `Protobuf` позволяет упаковать байты сообщения и исходный тип в одну сущность, а компилятор генерирует удобный код для рефлексии, благодаря которому можно автоматически конвертировать байты в исходный тип без явного разбора всех возможных типов сообщений. Если же сериализовывать Java-объекты с иерархией наследования в формат `JSON` или `XML`, библиотека сериализации добавит дополнительное поле о типе времени выполнения сериализуемого объекта, однако эта информация бесполезна для `.NET` сервера при десериализации, поэтому при сериализации в `JSON` или `XML` потребовалось бы хранить `Enum` с типом сущности, а при десериализации делать громоздкий разбор лексем `JSON`.

Для воспроизведения тестов запускается ASP.NET сервер в профилируемом процессе для сбора информации о покрытии. Инструмент `V#` для воспроизведения одного теста запускал такой же профилируемый процесс, что сильно нагружало операционную систему, так как проект насчитывает более 1000 интеграционных тестов. Таким образом, в USVM-NET воспроизведение тестов сделано гораздо эффективнее.

5. Тестирование

Тестирование проводилось в формате интеграционного тестирования всей символьной машины на основе валидатора тестов. В проекте V# использовалась аналогичная система тестирования, и за 5 лет существования проекта было написано свыше 1000 интеграционных тестов, часть которых была переиспользована для валидации USVM-NET. Было выбрано несколько категорий тестов на основании возможностей, поддерживаемых USVM-NET: сейчас не поддерживается исследование программ, где нужно разрешать условия на обобщенные типы, многомерные массивы, ограничение символьных параметров циклов, Enum-ы, делегаты. Были выбраны следующие тесты:

1. арифметические операции над целыми числами и числами с плавающей точкой, конверсии между ними;
2. логические операции;
3. массивы примитивов, классов, структур, копирования массивов;
4. небезопасные операции с примитивами, классами, структурами, в том числе с пересекающимися полями;
5. boxing и unboxing примитивных типов, структур;
6. контроль потока управления при обработке исключений;
7. структуры, их копирование, передача по значению и по ссылке;
8. классы, доступ к полям, рекурсивные объекты;
9. виртуальные и обычные вызовы функций.

Характеристики машины и версии библиотек, использованные для проведения эксперимента, приведены на рисунке 9.

Laptop	Victus by HP Laptop 16-e012ur
CPU	AMD Ryzen 5 5600H
OS	Ubuntu 24.04 LTS
RAM	32 GB DDR4
JDK version	corretto-11
Samples .NET version	7
Test executor .NET version	8

Рис. 9: Конфигурационный стенд

Построение трёхадресного кода для сборки с тестами составляет 5.58 ± 0.16 секунд. Этот параметр выносится отдельно, поскольку при дальнейшем развитии проекта планируется сохранять на диске результат построения трёхадресного кода, что очень целесообразно для стандартной библиотеки `mscorlib`, содержащей несколько тысяч классов. На момент проведения эксперимента JasoDB-NET не строил трёхадресный код сборки `mscorlib` с целью ускорения времени разработки и тестирования, однако данное условие наложило существенные ограничения на возможности анализа, поэтому для эксперимента были выбраны методы, для исполнения которых не нужен трёхадресный код `mscorlib`.

Результаты эксперимента приведена в таблице на рисунке 10.

Категория	Кол-во методов	Время анализа, с		Сгенерировано тестов		Воспроизведенные, %		Покрытие, %	
		USVM-NET	V#	USVM-NET	V#	USVM-NET	V#	USVM-NET	V#
Arithmetic	106	5.16 ± 0.15	2.50 ± 0.18	147	130	98	100	81	91
Logic	75	2.37 ± 0.05	0.52 ± 0.01	75	75	100	100	98	98
Arrays	36	3.84 ± 0.15	3.76 ± 0.11	163	112	96	100	70	96
Unsafe	28	3.24 ± 0.08	2.80 ± 0.14	49	50	96	100	82	93
Boxing	18	2.61 ± 0.22	0.1 ± 0.05	19	20	84	100	80	94
Exceptions	18	2.83 ± 0.15	0.81 ± 0.01	34	36	100	100	68	80
Structs	9	2.56 ± 0.08	0.69 ± 0.03	15	12	100	100	90	90
Classes	8	2.34 ± 0.06	0.69 ± 0.01	15	11	100	100	83	83
Calls	5	2.09 ± 0.12	0.67 ± 0.02	12	6	100	100	85	74

Рис. 10: Результаты эксперимента

Как можно видеть, даже за исключением времени на построение трёхадресного кода, время символьного анализа USVM-NET во всех категориях уступает инструменту V#. Это связано, во-первых, с тем, что в USVM-NET метаданные о программе хранятся в базе данных, и на небольших методах, где к данным обращаются несколько раз, может быть незаметен выигрыш от кэшей внутри JasoDB-NET, и, во-вторых, с

тем, что $V\#$ имеет подсистему конкретной памяти, позволяющий исполнить метод конкретно, если все его аргументы не являются символьными объектами, в то время как USVM-NET всегда исполняет инструкции через символьный интерпретатор. Столбец **сгенерированных тестов** показывает количество сгенерированных тестов для категории методов. В большинстве категорий USVM-NET сгенерировал больше тестов, несмотря на то, что USVM-NET поддерживает меньше возможностей языка. Это обусловлено тем, что интерпретатор USVM-NET чаще пытается разветвить состояние на выброс исключений. Столбец **воспроизведенных** тестов показывает процент сгенерированных тестов, ожидаемый результат символьного исполнения у которых совпадает с результатом воспроизведения в валидаторе тестов. Столбец **покрытие** показывает процент покрытых инструкций исследуемых методов. Во всех наборах тестов, кроме Calls и Arrays, можно считать покрытие инструмента $V\#$ эталонным. Несмотря на то, что USVM-NET генерирует больше тестов, покрытие оказывается меньше. Это объясняется текущими ограничениями поддерживаемых возможностей платформы машины USVM-NET. Если обратить внимание на последнюю категорию Calls с виртуальными вызовами методов, можно увидеть, что USVM-NET выигрывает в этой категории и по количеству сгенерированных тестов, и по покрытию. Это обусловлено реализацией виртуального вызова для символьных объектов в $V\#$, которая может упускать ветки из-за устройства символьной памяти. Реализация виртуального вызова в USVM-NET благодаря более продвинутой символьной памяти эти ветви не упускает.

Выводы

Несмотря на необходимость построения трёхадресного кода для исследования .NET на платформе USVM и задержки при первичном обращении к метаданным в базе во время анализа, эти недостатки нивелируются кэшированием результата построения. Среди значительных плюсов можно отметить базирование анализа .NET кода на гораздо более развитом ядре символьного исполнения USVM, возможность ис-

пользовать развитую библиотеку взаимодействия с SMT-решателями KSMT, которая, в отличие от $V\#$, поддерживающего лишь Z3, даёт возможность использовать еще и Yices, CVC5, Bitwuzla.

Заключение

Были достигнуты следующие результаты:

1. проведен обзор инструментов для генерации тестов и способов моделирования символьной памяти с небезопасным контекстом, для USVM-NET был выбран подход, основывающийся на подходе инструмента $V\#$, для генерации тестов и моделирования небезопасных операций;
2. спроектирована архитектура символьной виртуальной машины USVM-NET;
3. выполнена реализация символьной виртуальной машины USVM-NET без поддержки дженериков у символьных объектов, делегатов, многомерных массивов, Enum-ов, отслеживания границ циклов с символьным параметром;
4. проведено интеграционное тестирование символьной машины.

Код символьной виртуальной машины с генератором тестов и валидатором тестов доступны по ссылкам <https://github.com/oveeernight/usvm/tree/il-machine-enhancement> и <https://github.com/oveeernight/test-executor> соответственно.

Список литературы

- [1] URL: <https://github.com/UnitTestBot/usvm> (дата обращения: 9 апреля 2025 г.).
- [2] URL: <https://github.com/VSharp-team/VSharp> (дата обращения: 30 декабря 2024 г.).
- [3] URL: <https://www.microsoft.com/en-us/research/project/z3-3/> (дата обращения: 24 мая 2025 г.).
- [4] URL: <https://yices.csl.sri.com/> (дата обращения: 24 мая 2025 г.).
- [5] URL: <https://cvc5.github.io/> (дата обращения: 24 мая 2025 г.).
- [6] URL: <https://bitwuzla.github.io/> (дата обращения: 26 мая 2025 г.).
- [7] URL: <https://github.com/klee/klee> (дата обращения: 30 декабря 2024 г.).
- [8] URL: <https://github.com/binsec/binsec> (дата обращения: 30 декабря 2024 г.).
- [9] URL: <https://github.com/angr/angr> (дата обращения: 30 декабря 2024 г.).
- [10] Dannenberg R.B., Ernst G.W. Formal Program Verification Using Symbolic Execution // [IEEE Transactions on Software Engineering](#). — 1982. — Vol. SE-8, no. 1. — P. 43–52.
- [11] Godefroid P. Test generation using symbolic execution // [Leibniz International Proceedings in Informatics, LIPIcs](#). — 2012. — 01. — Vol. 18. — P. 24–33.
- [12] King James C. Symbolic execution and program testing // [Commun. ACM](#). — 1976. — . — Vol. 19, no. 7. — P. 385–394. — URL: <https://doi.org/10.1145/360248.360252>.

- [13] Krishnamoorthy Saparya, Hsiao Michael, Lingappan Loganathan. [Tackling the Path Explosion Problem in Symbolic Execution-Driven Test Generation for Programs](#). — 2011. — 01. — P. 59 – 64.
- [14] Memory models in symbolic execution: key ideas and new thoughts / Luca Borzacchiello, Emilio Coppa, Daniele Cono D’Elia, Camil Demetrescu // [Software Testing, Verification and Reliability](#). — 2019. — Vol. 29, no. 8. — P. e1722. — <https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1722>.
- [15] A Survey of Symbolic Execution Techniques / Roberto Baldoni, Emilio Coppa, Daniele Cono D’elia et al. // [ACM Comput. Surv.](#) — 2018. — . — Vol. 51, no. 3. — 39 p. — URL: <https://doi.org/10.1145/3182657>.
- [16] Tillmann Nikolai, Halleux Jonathan "Peli" de. Pex - White Box Test Generation for .NET // Proc. of Tests and Proofs (TAP’08). — Vol. 4966 of LNCS. — Springer Verlag, 2008. — April. — P. 134–153. — URL: <https://www.microsoft.com/en-us/research/publication/pex-white-box-test-generation-for-net/>.
- [17] М. У. Мандрыкин В. С. Мутилин. Обзор подходов к моделированию памяти в инструментах статической верификации // Труды Института системного программирования РАН. — 2017. — P. 195–230. — URL: [https://doi.org/10.15514/ISPRAS-2017-29\(1\)-12](https://doi.org/10.15514/ISPRAS-2017-29(1)-12).