Санкт-Петербургский государственный университет

Программная инженерия
Кафедра системного программирования

Костицын Михаил Павлович

Модель символьной памяти .NET с поддержкой реинтерпретаций

Бакалаврская работа

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

Рецензент:
программист ООО «Интеллиджей Лабс» Д. С. Косарев

Санкт-Петербург
2019

Оглавление

Введение 3

1. Постановка задачи 5

2. Обзор 6
   - 2.1. Символьное исполнение 6
   - 2.2. SMT-решатели 8
   - 2.3. Модели памяти 10
   - 2.4. Инструменты анализа программ, поддерживающие реинтерпретацию данных 13
   - 2.5. Проект V# 16

3. Модель памяти 17
   - 3.1. Структуры данных в платформе .NET 17
   - 3.2. Операции с памятью в платформе .NET 20
   - 3.3. Представление данных в символьной памяти 24
   - 3.4. Моделирование операций с памятью в рамках безопасного контекста 31
   - 3.5. Моделирование операций с памятью в рамках небезопасного контекста 37

4. Архитектура и детали реализации 46
   - 4.1. Архитектура 46
   - 4.2. Детали реализации 50

5. Тестирование 52

Заключение 54

Список литературы 55

Введение

С каждым годом сложность программных продуктов возрастает, и, соответственно, задача обеспечения их надёжности и корректности становится всё более востребованной. Для её решения используются активно развивающиеся техники из области верификации. Одной из таких техник является символьное исполнение [14, 15, 22, 23, 29, 32, 33].

Некоторые современные верификаторы [14, 15, 22, 32] используют технику символьного исполнения для сведения задачи анализа программного кода с неопределёнными входными данными к проверке выполнимости формул логики первого порядка. Для такой проверки чаще всего используются SMT-решатели (Satisfiabilty Modulo Theories). Данные инструменты позволяют определить, выполнима ли формула, записанная в поддерживаемых решателями теориях (например, теория линейной арифметики, битовых векторов и некоторые другие). Таким образом в процессе сведения анализа программного кода к проверке выполнимости формул логики необходимо моделировать семантику программы в теориях логики первого порядка, поддерживаемых решателями.

В современных языках программирования, таких как C, C++, C#, Java и других, память программы достаточно полно отражает состояние этой программы, и операции с памятью имеются почти в каждой нетривиальной программе. Соответственно, разработка метода моделирования операций с памятью для последующего использования в SMT-решателе является важной и актуальной задачей верификации. Метод моделирования операций с памятью (т. е. *модель памяти*) определим, согласно статье [34], как формальное представление указателя и ссылки, а также формализацию результата операций над ними с использованием логических формул. Разработка такого метода позволила бы доказывать, что в программе, например, отсутствуют: утечки памяти, некорректные доступы к памяти (например, выход за границы массива), неопределённые поведения и т.д. Однако задача разработки модели памяти является нетривиальной, а её решение зависит от семантики

конкретного языка программирования.

Важной особенностью платформы .NET является возможность использования динамической памяти, т.е. памяти заранее не ограниченного размера. Помимо этого в платформе .NET существует два контекста исполнения кода: безопасный (англ. safe) и небезопасный (англ. unsafe). В случае безопасного контекста, низкоуровневыми операциями с памятью управляет CLR (Common Language Runtime). Когда контекст исполнения является небезопасным, пользователю становятся доступны операции с указателями, как в языке C, которые могут привести к *реинтерпретации данных*.

Под реинтерпретацией понимается рассмотрение данных при помощи структуры[1], отличной от той, которой они изначально задавались. Такого поведения можно добиться с помощью адресной арифметики и приведения типов указателей, например, прочитав массив из элементов типа `int` указателем с типом `byte*`.

Данная работа выполнена в рамках проекта V# кафедры системного программирования СПбГУ. Данный проект направлен на создание символьной виртуальной машины для платформы .NET с целью верификации программ.

[1]в широком смысле, от слова «структурированный»

1. Постановка задачи

Целью данной работы является создание модели символьной памяти для платформы .NET с поддержкой реинтерпретации данных для верификации .NET-программ, основанной на SMT-решателях. Для её достижения были поставлены следующие задачи:

- выполнить обзор литературы о моделировании операций с памятью при помощи логики первого порядка, рассмотреть существующие инструменты анализа программ, поддерживающие реинтерпретацию данных;
- создать модель памяти для платформы .NET с поддержкой реинтерпретации данных;
- реализовать созданную модель памяти в символьной виртуальной машине V#;
- провести тестирование функциональности полученного решения.

2. Обзор

В данной главе представлен обзор используемых техник анализа, подходов к моделированию операций с памятью, а также инструментов анализа программ, поддерживающих реинтерпретацию данных. В разделе «V#» описан проект в рамках которого выполнена данная работа.

2.1. Символьное исполнение

Символьное исполнение [14, 15, 22, 23, 29, 30, 32, 33] — техника, которая позволяет исполнять программный код в условиях неопределённости входных данных, исследуя все ветки выполнения программы. При конкретном исполнении функции из лист. 1, условие (`x` > `y`) будет либо ложным, либо истинным, следовательно будет выполнена только одна ветка. При символьном исполнении этой функции входные данные (т. е. `x` и `y`) будут заменены на *символы* — абстракции над конкретными значениями, при этом обе эти ветки будут исполнены, а результат исполнения каждой ветки будет защищен соответствующим условием попадания в неё. Такое условие будем называть *условием пути*.

Условие пути является одним из элементов состояния интерпретатора, в которое также входит и *символьная память* — представление состояния памяти при символьном исполнении. В начале исследования функции условие пути равно `true`. При встрече ветвления текущая ветка исполнения разбивается на две, условия пути которых равны условиям попадания в ветку `then` и `else` соответственно.

```
public static int MaxInt(int x, int y) {
    int maxInt = 0;
    if (x > y) {
        maxInt = x;
    }
    else {
        maxInt = y;
    }
    return maxInt;
}
```

Листинг 1: Пример функции для символьного исполнения

Одним из видов символьного исполнения является *статическое символьное исполнение*, результатом которого является значение, которое вернула функция, а также состояние символьной памяти. Главной особенностью данного вида является *слияние* результатов исполнения веток, полученных разветвлении одной ветки. После слияния получается результат исполнения (т. е. значение и символьная память), вбирающий в себя все изначальные ветки, полученный благодаря введению новых синтаксических конструкций, например $ite(condition, thenTerm, elseTerm)$. Результатом исполнения функции из примера в таком случае будет значение $ite(\mathtt{X} > \mathtt{Y}, \mathtt{X}, \mathtt{Y})$ и символьная память $M = \{\mathtt{x} \mapsto \mathtt{X}, \mathtt{y} \mapsto \mathtt{Y}, \mathtt{maxInt} \mapsto ite(\mathtt{X} > \mathtt{Y}, \mathtt{X}, \mathtt{Y})\}$, где `X` и `Y` являются символьными значениями переменных `x` и `y` соответственно.

Для назначения входным данным символьных значений может использоваться метод *ленивой инстанциации* [17] (англ. lazy instantiation). Главная идея данного метода заключается в инициализации данных по необходимости, т. е. вначале символьного исполнения функции все входные данные помечаются как неинициализированные. При первом использовании таких данных происходит их инициализация. Если инициализируется переменная ссылочного типа, то в неё недетерминированно помещаются: значение `null`, ссылка на новый объект с неинициализированными полями, ссылка на ранее созданный объект. В случае инициализации переменной простого типа, в неё помещается символьное значение соответствующего ей типа. В данном методе во время инициализации значений используется условие пути для проверки, что по-

лученное значение может находиться в данной переменной. Благодаря такому подходу возможно символьное исполнение в условиях отсутствия знаний о некоторых простых ссылочных локациях (т. е. ссылочных локациях у которых известное количество полей). В частности, это позволяет исследовать функции с некоторыми рекурсивными структурами данных без указания априорных ограничений на их размер.

Одним из главных минусов техники символьного исполнения является *взрыв путей исполнения*, т. е. экспоненциальный рост количества исследуемых веток. Важно отметить, что некоторые из этих веток могут быть недостижимы, условия пути в таких ветках будут невыполнимыми.

Для проверки достижимости веток выполнения современные верификаторы [14, 15, 22, 30, 32] используют SMT-решатели, которые могут проверять, выполнима ли формула пути определённой ветки: если нет, то исполнение такой ветки прекращается.

2.2. SMT-решатели

SMT-решатели [6, 11, 28] являются инструментами для автоматизированной проверки выполнимости логических формул в теориях. На вход эти инструменты принимают формулу логики первого порядка с функциональными, предикатными и константными символами из сигнатуры заранее заданных теорий. Если формула выполнима, то в качестве результата решатель возвращает `SAT`[2] и модель, которая интерпретирует формулу истинно. Если формула невыполнима, и решателю удалось это доказать, то он возвращает `UNSAT`. Помимо этих результатов, решатель может вернуть `UNKNOWN` или зависнуть, так как некоторые теории или их комбинации являются неразрешимыми.

Среди теорий, поддерживаемых решателями, можно выделить следующие: теория линейной целочисленной арифметики, линейной вещественной арифметики, битовых векторов, неинтерпретированных функций, а также массивов.

[2]Выполнима (англ. satisfiable)

Теория линейной целочисленной арифметики. Сигнатура данной теории включает в себя целые числа, операции сложения и вычитания, а также предикаты равенства и меньше. Данная теория является разрешимым фрагментом арифметики.

Теория линейной вещественной арифметики. Сигнатура данной теории логики первого порядка содержит вещественные числа, операции сложения и вычитания, предикаты равенства и меньше. Описанная теория является разрешимой.

Теория битовых векторов. В сигнатуру данной теории входят числа, каждое из которых представляет битовый вектор фиксированной длины (представления чисел в машинной арифметике), предикаты равенства и меньше, операции сложения и произведения, а также следующие битовые операции: «и», «или», «исключающее или», «не», «сдвиг влево», «сдвиг вправо», «конкатенация двух бит-векторов», «взятие подвектора». Данная теория является разрешимой [2], так как сводится к задаче выполнимости формул логики высказываний.

Теория неинтерпретированных функций. В сигнатуру данной теории, которую также называют свободной, входят произвольные функциональные символы с известной арностью, а также предикат равенства. Данная теория является неразрешимой, так как можно выполнимых формул в логике первого порядка не рекурсивно перечислимо.

Теория массивов. Сигнатура описываемой теории содержит в себе сигнатуру линейной арифметики, дополняя её двумя операциями: чтением элемента массива по указанному индексу (select) и записью значения в элемент массива по указанному индексу (store). Данная теория является неразрешимой [9] в общем случае теорией логики первого порядка, которая используется для выражения суждений о массивах.

2.3. Модели памяти

Модель памяти [34] — это формальное представление указателя и ссылки, а также формализация результата операций над ними с использованием логических формул. Среди существующих на данный момент методов моделирования операций с памятью можно выделить две группы подходов: модели для для высокоуровневого анализа памяти и для низкоуровневого. Далее отдельно отдельно разберём обе эти группы подхов. Более подробный обзор существующих моделей памяти приведён в статье [34].

2.3.1. Модели высокоуровневого анализа памяти

Модели высокоуровневого анализа памяти направлены на анализ рекурсивных структур данных заранее неогрниченного размера, однако в общем случае не поддерживают массивы, адресную арифметику и приведения типов указателей. Среди таких моделей выделим LISBQ [19].

LISBQ. Данный способ моделирования операций с памятью основан на LISBQ (Logic of Interpreted Sets and Bounded Quantification) — логике интерпретируемых множеств и ограниченной квантификации. Этот метод используется в дедуктивной верификации, например, в инструменте HAVOC [8]. Для моделирования поведения программы в логике первого порядка модель памяти LISBQ использует теорию линейной целочисленной и вещественной арифметики. Главными минусами данной модели являются отсутствие поддержки массивов, адресной арифметики, приведений типов указателей. Помимо этого основной особенностью этого метода является необходимость аннотаций со стороны пользователя для достижения *точного анализа* (т. е. порождаемые формулы описывают все поведения программы и только их), что влечёт неприменимость данной модели для автоматизированного анализа.

2.3.2. Модели низкоуровневого анализа памяти

Модели памяти, выполняющие низкоуровневый анализ памяти, в отличие от моделей высокоуровневого анализа памяти, сфокусированы на анализе массивов, приведении типов указателей, адресной арифметики, однако не поддерживают произвольные свойства рекурсивных структур данных. Первой среди таких моделей рассмотрим *модель памяти на основе анализа алиасов.*

Модель памяти на основе анализа алиасов [1]. Данный метод моделирования операций с памятью выполняет анализ синонимичных указательных выражений, что позволяет находить указатели, которые могут ссылаться на одну область памяти. Такая модель памяти была использована в инструменте автоматической верификации BLAST [27]. Данный метод ориентирован на области памяти заранее ограниченного размера (например, структуры или значения простых типов). Однако области памяти заранее неограниченного размера (например, массивы переменной длины) в общем случае не поддерживаются, а именно не поддерживается проверка произвольных свойств таких областей [34]. Также необходимо отметить, что проверка некоторых простых свойств рекурсивных структур данных может быть выполнена в рамках данной модели. Для кодирования в SMT-решатель этот метод использует теории линейной вещественной арифметики и неинтерпретированных функций. Формулы, порождаемые данным методом, которые моделируют результат операций с памятью, описывают упрощенный результат, дополненный ложными знаниями, из-за чего происходят многочисленные ложные срабатывания. Данная особенность говорит о неприменимости такого моделирования для задачи проверки произвольных свойств программы. Помимо этого, модель памяти с использованием анализа алиасов не подходит для проверки произвольных свойств программ платформы .NET, так как в этих программах можно выделить область памяти заранее не ограниченного размера (например, массив). Для анализа памяти таких программ используются модели для областей памяти заранее неограниченного размера, среди которых *типизи-*

рованная модель.

Типизированная модель [26]. Основной идеей данной модели, используемой в дедуктивной верификации, является сужение области возможных локаций, на которые может указывать указатель, благодаря знаниям о типе локаций и указателя. Если типы локации и указателя совпадают, то указатель может указать на данную локацию. Такая информация о типах предоставляется при помощи аннотаций пользователя, что означает неприменимость для автоматизированной верификации. Такая модель памяти была реализована в инструменте дедуктивной верификации VCC2 [25]. Для моделирования операций с памятью в логике первого порядка типизированная модель памяти использует следующие теории: теорию массивов, линейной целочисленной и вещественной арифметики. Главным недостатком данной модели является отсутствие поддержки произвольных свойств рекурсивных структур данных. Помимо этого, данный метод ориентирован на дедуктивную верификацию, а значит неприменим в автоматической верификации. Улучшением этого метода моделирования операций с памятью является *модель Бурсталла-Борната.*

Модель Бурсталла-Борната [8]. Главной идеей данного способа моделирования, применяемого в области частично автоматизированной дедуктивной верификации, является раздельное представление состояния памяти для компонентов составных объектов, иначе говоря, вся моделируемая память программы является массивом, элементы которого — это элементы составных объектов. В качестве теорий логики первого порядка для моделирования поведения программы данная модель использует те же теории, что и типизированная модель памяти, т. е. теорию массивов, линейной целочисленной и вещественной арифметики. Основной особенностью такого метода является достижение точного анализа путём требования аннотаций со стороны пользователя, что неприменимо в случае автоматической верификации. Отсутствие поддержки произвольных свойств рекурсивных структур данных, как

и в случае с типизированной моделью, является основным минусом. Расширением такой модели памяти является *модель памяти с регионами.*

Модель памяти с регионами [16]. Ключевой идеей данной модели памяти для дедуктивной верификации является уменьшение пространства возможных локаций для определённого указателя с помощью добавления понятия *регионов. Регионы* — такие непересекающиеся множества указательных выражений, что элементы из разных множеств не могут адресовать пересекающиеся области в памяти. Последняя модификация данной модели была представлена в работе [20] М. У. Мандрыкина и А. В. Хорошилова. Модель памяти с регионами, как и модель Бурсталла-Борната, использует теорию массивов, линейной целочисленной и вещественной арифметики для моделирования операций программы с памятью. Данная модель относится к моделям памяти для дедуктивной верификации, т. е. основывается на аннотациях пользователя, что невозможно в случае автоматической верификации. Недостатки данной модели типичны для моделей низкоуровневого анализа памяти, т. е. отсутствует поддержка произвольных свойств рекурсивных структур данных [34].

2.4. Инструменты анализа программ, поддерживающие реинтерпретацию данных

Среди существующих на данный момент анализаторов программ, поддерживающих реинтерпретацию данных, можно выделить два типа: инструменты, выполняющие автоматический анализ программ, а также инструменты дедуктивной верификации, выполняющие точный анализ при наличии аннотаций со стороны пользователя, число строк которых, зачастую, может превышать число строк программы. Одним из успешных анализаторов первого типа можно называть инструмент Predator [31].

2.4.1. Инструмент Predator

Данный инструмент является автоматическим анализатором С-кода, который многократно побеждал [3, 4, 5] на соревнованиях SV-COMP, посвящённых анализу программ, в секциях Memory Safety и Heap (низкоуровневый анализ памяти). Его основная особенность — совмещение техник символьного исполнения и абстрактной интерпретации [10] в домене *символьных графов памяти* (англ. symbolic memory graphs, SMG) [12]. С помощью SMG можно выражать свойства вложенных дву- и односвязных списков, часто встречающихся, например, в коде ядра Linux. Благодаря этому инструмент Predator хорошо зарекомендовал себя не только на соревнованиях, но и в анализе реальных проектов, таких как драйвера ядра Linux, lvm2, менеджер памяти Firefox и других.

Недостатком данного инструмента можно назвать отсутствие поддержки произвольных свойств структур, определённых пользователем. При необходимости расширения области анализа на новые структуры, например, на деревья, необходимо значительно расширить базовые алгоритмы анализатора.

Касаемо реинтерпретаций, анализ памяти в инстурументе Predator можно назвать *анализом формы кучи* (англ. shape analisys), т. е. в сложных случаях информация из графа памяти удаляется, что делает модель упрощённой, содержащей ложные знания о программе, из-за чего происходят ложные срабатывания.

2.4.2. Инструмент Pex

Данный инструмент автоматического анализа программ на платформе .NET [24] предназначен для генерации тестового покрытия. Анализатор Pex базируется на динамическом символьном исполнении [18], при этом объединяет в себе техники конкретного и символьного исполнения, стараясь выполнить все линейные участки кода с конкретными значениями, подходящими для этих веток, тем самым теряя большое количество поведений исследуемой программы. Такое свойство делает инструмент Pex неприменимым для проверки произвольных свойств

программ. Этот инструмент доступен для разработчиков программ на платформе .NET, так как является частью среды разработки Visual Studio 2015 Enterprise под названием IntelliTest. Главным недостатком данного подхода является взрыв путей исполнения, т. е. экспоненциальный рост количества исследуемых веток.

2.4.3. Инструмент VCC3

Главной особенностью данного инструмента дедуктивной верификации C программ [7] является реализованная в нём модель памяти Бурсталла-Борната. Благодаря такой модели памяти верификатор способен на основе аннотаций пользователя выполнять точный анализ программ с массивами, приведением типов указателей и адресной арифметики. Свои основные недостатки при анализе операций с памятью, среди которых реинтерпретация данных, этот инструмент перенимает из реализованной в нём модели памяти, а именно отсутствие поддержки произвольных свойств рекурсивных структур данных при анализе.

2.4.4. Инструмент Jessie

Данный инструмент [21] является инструментом дедуктивной верификации C-кода. Моделирование операций с памятью инструмент Jessie осуществляет с помощью вышеупомянутой модели памяти с регионами, тем самым выполняя точный анализ памяти программы при наличии аннотаций пользователя. Среди достоинств этого верификатора можно выделить следующие: наличие точного анализа памяти, поддержка массивов, приведения типов указателей и адресной арифметики. Данные преимущества инструмента, подобно анализатору VCC3, получены благодаря используемой модели памяти, как и его недостаток: отсутствие анализа произвольных свойств рекурсивных структур данных.

2.5. Проект V#

Данный проект[3] нацелен на создание композиционального верификатора программ на платформе .NET, выполняющего точный анализ с помощью техники статического символьного исполнения и доказательства теорем над дизъюнктами Хорна. Архитектура данного проекта представлена на рис. 1.

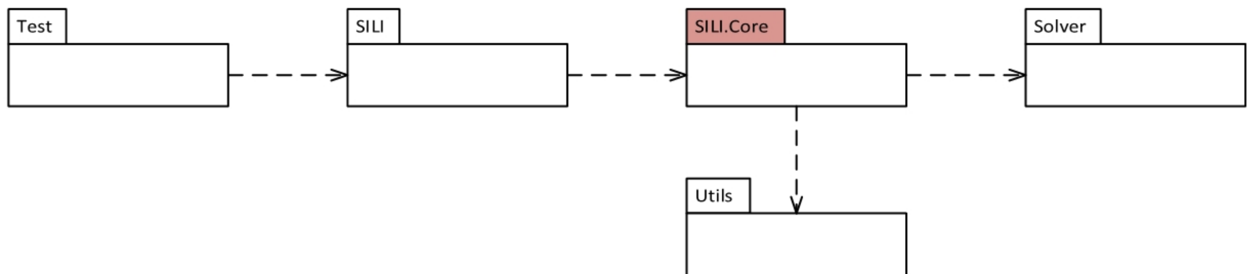

Рис. 1: Архитектура проекта V#

Рассмотрим отдельно каждую подсистему данного проекта и функции, которые она выполняет.

- Test — система тестирования, включающая в себя тесты для проверки корректности работы символьной виртуальной машины.
- SILI (Symbolic Intermediate Language Interpreter) — система интерпретатора, выполняющая символьное исполнение .NET кода с помощью вызова методов из API, предоставляемого SILI.Core.
- SILI.Core — ядро системы, содержащее в себе всю нетривиальную работу символьного исполнения, например, операции с символьной памятью программы, массивами, строками.
- Solver — система взаимодействия с решателем, выполняющая кодирование результатов работы символьной виртуальной машины в термины теорий логики первого порядка.
- Uitls — вспомогательная система, содержащая необходимые для проекта структуры данных и примитивы работы с ними.

Во время данной работы были затронуты все системы проекта, однако преобладающая часть работы была сконцентрирована в ядре системы, которое на рисунке выделено цветом.

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

3. Модель памяти

Данная глава описывает разработанный метод моделирования операций с памятью в платформе .NET. В этом методе можно выделить два этапа: выполнение операций в символьной виртуальной машине и последующее кодирование результатов этих операций в SMT-решатель. В начале данной главы рассматривается устройство памяти в платформе .NET, а именно структуры данных и существующие операции с ними. Далее отдельно описано представление этих структур данных в символьной виртуальной машине и выполнение операций с ними.

3.1. Структуры данных в платформе .NET

Платформа .NET содержит четыре блока данных.

1. Стек — область памяти, имеющая структуру LIFO. В этом блоке размещаются параметры и переменные метода, которые представляют примитивные типы, структуры, а также массивы, выделенные на стеке с помощью инструкции `stackalloc`.
2. Динамически распределяемая память. В данном блоке размещаются ссылочные типы (например, экземпляры классов пользователя, массивы, строки и т. д.).
3. Статическая память — область памяти для хранения статических членов классов и структур.
4. Пул интернирования — хэш-таблица, используемая для хранения строк, над которыми было произведено интернированние.

Далее отдельно рассмотрим структуры данных, которые могут находится внутри одного из этих блоков. Начнём описание представления данных с простых типов (англ. simple types), после чего будут рассмотрим более сложные структуры данных.

3.1.1. Простые типы

Согласно спецификации языка платформы .NET [13], т. е. CIL (Common Intermediate Language), простые типы (англ. simple types) состоят

из численных типов (англ. numeric types) и типа `bool`. Ниже перечислены все простые типы вместе с размером, который они занимают в байтах:

- sbyte (1);
- byte (1);
- short (2);
- ushort (2);
- int (4);
- uint (4);
- long (8);
- ulong (8);
- char (2, в стандарте Unicode);
- float (4);
- double (8);
- decimal (16);
- bool (1).

3.1.2. Структуры

Структуры, как и простые типы, входят в группу «тип-значение» (англ. value types) однако имеют уже нетривиальное устройство, похожее на случай класса, например, содержат поля, методы, но не поддерживают наследование. В качестве полей структуры могут выступать любые типы, в том числе массивы. Если поле является массивом, то в памяти структуры будет лежать ссылка на него, так как массив является ссылочным типом. Однако при наличии небезопасного контекста (т. е. ключевого слова `unsafe`) полем структуры может быть массив фиксированного размера (англ. fixed sized buffer), элементы которого расположены внутри памяти структуры, а не по ссылке. Ограничениями такого массива являются линейность, индексация с нуля, принадлежность типа элементов к простым типам, а также наличие ключевого слова `fixed` в объявлении.

Особенностью структур в платформе .NET является возможность при объявлении структуры с помощью атрибута StructLayout указать

конкретный способ размещения её полей в памяти, а именно: последовательное размещение (sequential), конкретное смещение каждого поля внутри структуры (explicit), или автоматический (auto — не специфицированный способ, позволяющий CLR (Common Language Runtime) располагать поля так, чтобы добиться максимально эффективного использования памяти). Необходимо отметить, что конкретное размещение (explicit) допускает пересечение полей друг с другом, что ведёт к реинтерпретации данных. В качестве варианта по умолчанию используется последовательное размещение полей структуры. Помимо размещения полей в памяти атрибут `StructLayout` позволяет при помощи поля Size установить размер структуры больше, чем суммарный размер её полей. В таком случае при выделении структуры будет предоставлена дополнительная память. С помощью данного атрибута (а именно его поля Pack) также можно изменить способ выравнивания полей структуры.

3.1.3. Классы

Экземпляр класса является схожим со структурой представлением данных, главным отличием которого является принадлежность к группе ссылочных типов, т. е. расположение в динамической памяти. Так же, как и в случае со структурой, при объявлении класса пользователя можно указать определённый способ размещения полей класса в памяти с помощью того же атрибута `StructLayout`. Отличием работы этого атрибута для класса является использование автоматического режима размещения полей в качестве варианта по умолчанию.

3.1.4. Массивы

В платформе .NET существует класс `System.Array`, от которого нельзя явно унаследоваться, однако любой привычный тип массива, например, int[], является его наследником. В данном классе находятся общие операции для массивов любого типа. Помимо этих операций, элементом класса `System.Array` является метод `CreateInstanse`, кото-

рый позволяет во время исполнения программы создать массив любой размерности, указав при этом нижние границы и длины каждой из размерностей.

3.1.5. Строки

Строка в платформе .NET представляют собой неизменяемый ссылочный тип, а именно экземпляр класса `System.String`, особенностью которого является сравнение по содержимому строки, а не по ссылке. Внутри данного класса отдельно содержится информация о длине и содержимом строки, т. е. `System.String` содержит два поля: длину и массив символов строки.

3.2. Операции с памятью в платформе .NET

Как уже было упомянуто, в платформе .NET существует два контекста операций с памятью: безопасный (англ. safe) и небезопасный (англ. unsafe). В последующих разделах отдельно для каждого контекста рассмотрим возможные операции с памятью.

3.2.1. Операции с памятью в рамках безопасного контекста

В начале данного раздела рассмотрим три основных операции: выделение, чтение и запись. После этого отдельно будет рассмотрены операции упаковки (англ. box) и распаковки (англ. unbox), а также механизм интернирования строк.

Выделение. Эта операция размещает указанные данные в памяти. Рассматривая данную операцию подробнее, разобьём её на три случая: выделение типа-значения (структура, простые типы) на стеке, выделение ссылочного типа (массива, строки, экземпляра класса) в динамической памяти, выделение класса или структуры со всеми статическими полями в статической памяти.

Чтение. Операция чтения позволяет получить значение, которое уже существует в памяти. В данном случае можно выделить следующие применения операции: чтение значения простого типа, чтение структуры, чтение поля структуры, чтение поля экземпляра класса, чтение элемента массива, чтение значения статического поля.

Запись. Данная операция позволяет обновить уже существующее в памяти значение и применяется в одном из следующих случаев: обновление значения простого типа, обновление структуры, запись значения в поле структуры, запись значения в поле экземпляра класса, запись значения в элемент массива, запись значения в статическое поле.

Упаковка/распаковка. Данные операции позволяют конвертировать экземпляр типа-значения в ссылочный тип `object` и обратно. Операция упаковки состоит из двух действий: выделение специального объекта в динамической памяти и копирование значения в поле value этого объекта. В случае операции распаковки происходит следующее: вначале проверяется, является ли объект упакованным экземпляром заданного типа-значения, после чего значение поля value этого объекта копируется в переменную типа-значения.

Интернирование строк. Интернирование — механизм, который позволяет одинаковым объектам представлять собой одну область в памяти, т. е. индивидуальные данные хранятся в памяти только в одном экземпляре. Платформа .NET предоставляет данный механизм для случая строк. Для реализации такого механизма в памяти программы находится хэш-таблица, называемая *пулом интернирования*, ключами которой являются хэши строк, а значениями — (списки) ссылки на строки. По умолчанию интернирование производится только для литеральных строк, и выполняется оно следующим образом: во время JIT-компиляции литеральные строки последовательно заносятся в хэш-таблицу, после чего, на этапе выполнения ссылки на эти литеральные строки присваиваются из данной таблицы. Для интернирования нелите-

ральных строк в платформе .NET существуют операции явного взаимодействия с пулом интернирования: `String.Intern`, `String.IsInterned`. Первая операция позволяет поместить строку в хэш-таблицу. Данная операция производится следующим образом: если указанная строка уже находится в пуле интернирования, метод возвращает ссылку на эту строку, в противном случае — добавляет строку в пул и возвращает ссылку на неё. Вторая операция (`String.IsInterned`) проверяет, находится ли строка в хэш-таблице: если указанная строка присутствует в пуле, то возвращается ссылка на неё из данной таблицы, иначе возвращается `null`.

3.2.2. Операции с памятью в рамках небезопасного контекста

Наличие небезопасного контекста (т. е. ключевого слова `unsafe`) позволяет в явном виде использовать указатели для взаимодействия с памятью. Основными операциями в данном случае будут следующие: создание указателя на область памяти, адресная арифметика, приведение типа указателя, разыменование указателя. Данные основные операции позволяют совершать реинтерпретацию данных программы следующим образом: создание указателя на область памяти, прибавление или вычитание из этого указателя числа и/или приведение типа этого указателя, разыменование получившегося указателя. Вначале отдельно разберём основные операции в рамках небезопасного контекста, после чего рассмотрим операцию `stackalloc`.

Создание указателя. Согласно спецификации CIL (Common Intermediate Language, язык платформы .NET [13]), тип указателя должен удовлетворять грамматике (1), при условии, что struct type не является обобщенным типом и содержит только поля типа unmanaged type. Помимо данных ограничений для создания указателя на область в динамической памяти, необходимо использовать ключевое слово `fixed`, которое не позволяет сборщику мусора перемещать данную область памяти.

$$\begin{aligned} \textit{pointer type} &::= \textit{unmanaged type}^* \mid \textit{void}^* \\ \textit{unmanaged type} &::= \textit{simple type} \mid \textit{enum type} \mid \textit{pointer type} \mid \textit{struct type} \end{aligned} \quad (1)$$

Адресная арифметика. В данный класс операций входит прибавление числа к указателю, вычитание числа из указателя, результатами которых являются указатели того же типа на новую локацию, вычитание указателей друг из друга, результат которого — это значение типа `long`, а также сравнение указателей друг с другом.

Приведение типов указателей. В платформе .NET тип любого указателя можно привести к `void*`, любому типу указателя, а также к целым типам (англ. integral types): `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`. Результат приведения типа указателя к одному из целых типов является не специфицированным (implementation-defined), т. е. зависит от архитектуры, версии CLR и т.д. Необходимо отметить, что операция приведения типа указателя никак не изменяет значение этого указателя.

Разыменование указателя. Данная операция возвращает данные, которые адресует указатель T*, рассмотренные с помощью типа T. Разыменование void* влечёт ошибку во время компиляции, в то время, как применение этой операции к указателю null приводит к не специфицированному поведению (implementation-defined behavior). Также к такому поведению приводит разыменование указателя, адресующего область памяти, часть которой не имеет известного размещения. Примером такой ситуации является разыменование области памяти за границами массива, который располагается на стеке. Однако таким случаем не является разыменование указателя, адресующего память за границами массива фиксированного размера, который располагается внутри структуры с конкретным размещением полей, при условии то-

го, что адресуемая область памяти не выходит за границы структуры.

Stackalloc. Так как массив является ссылочным типом, он выделяется в динамической памяти. Однако в небезопасном контексте операция stackalloc позволяет разместить массив на стеке. На данный массив действуют те же ограничения, что и на массив фиксированного размера, а именно: линейность, индексация с нуля, принадлежность типа элементов к простым типам.

3.3. Представление данных в символьной памяти

Согласно модели памяти Бурсталла-Борната, элементы составных объектов (например, поля структур, элементы массивов), представляются отдельно в моделируемой памяти. Разработанный способ моделирования операций с памятью модифицирует данную идею, обозначая содержимое составных объектов как семантически отделённую область памяти, которую далее будем называть *логическим блоком*. Помимо содержимого составных объектов, такой областью памяти также являются следующие блоки данных: динамическая и статическая память, стек и пул интернирования. Заметим, что внутри логического блока «динамическая память» может лежать логический блок «содержимое структуры», внутри которого также может располагаться логический блок «элементы массива фиксированного размера». Данное замечание демонстрирует *иерархичность* логических блоков, т. е. наличие иерархии, построенной на основе вложенности одного блока в другой. Следовательно, представление логического блока в символьной виртуальной машине также должно обладать этим свойством. Символьная память, в частности представление логического блока, удовлетворяющего заданному свойству иерархичности, будет описана далее. Данная символьная память оперирует термами, т. е. языком символьной виртуальной машины, который будет описан в разделе 3.3.2.

3.3.1. Символьная память

Представлением всей памяти программы в целом является разработанная символьная память M, которая состоит из следующих элементов:

- S (stack);
- DM (dynamic memory);
- SM (static memory);
- IP (interning pool).

Представлением логического блока «стек» анализируемой программы является элемент S (stack), который описывается с помощью *символьного стека*.

Определение 1. *Символьный стек* — это частичная функция $\varsigma : stackKey \to term$.

Представлением логического блока «динамическая память» является элемент символьной памяти DM (dynamic memory), который описывается частным случаем *символьной кучи*, адресами которой являются термы (`LOC` — это term). Термы описаны в разделе 3.3.2.

Определение 2. *Символьная куча* — это частичная функция $\sigma : \texttt{LOC} \to term$.

Определение 3. Пустая куча ϵ — это частичная функция с областью действия $dom(\epsilon) = \emptyset$.

Представлением логического блока «статическая память» служит элемент SM (static memory), который также является частным случаем символьной кучи, адреса которой — это типы, имеющие статические поля. InterningPool — внутреннее представление хэш-таблицы «пул интернирования» анализируемой программы, которое также описывается символьной кучей, ключи которой — это значения строк, над которыми было произведено интернирование. Любой элемент символьной памяти (S, DM, SM, IP) далее будем называть *верхнеуровневым символьным блоком*.

Основной особенностью разработанной символьной памяти помимо иерархичности является *поддержка неопределённости данных*, которая выражается в двух свойствах. Первым является наличие символьных значений, о которых пойдёт речь в следующем разделе, в качестве локаций. Второе свойство — это выполнение операций с памятью в условиях отсутствия знаний о некоторых локациях. Символьная память может не знать о существующих локациях, например, если в анализируемую функцию передаётся связный список, длина которого заранее не известна, а значит и число звеньев, выделенных в памяти, не определено во время анализа. Данная ситуация показана на лист. 2. Про способ выполнения операций с локациями, о которых отсутствует информация при анализе, будет упомянуто в разделах, описывающих операции с памятью.

```
public static int LastValue(LinkedList<int> list)
{
    return list.Last.Value;
}
```

Листинг 2: Пример функции со связным списком

3.3.2. Язык символьной виртуальной машины

При символьном исполнении программы внутри логических блоков может возникать неопределённость данных, например, при наличие символьных значений внутри. *Термы* — внутренний язык символьной виртуальной машины, который задаётся следующей грамматикой:

$$
\begin{aligned}
\mathit{term} &::= \texttt{NOP} \mid \mathit{error} \mid \texttt{CONCRETE} \mid \texttt{SYMBOL} \\
&\mid \mathit{ref} \mid \mathit{ptr} \mid \mathit{expr} \mid \mathit{array} \mid \mathit{struct} \mid \mathit{union} \\
\mathit{error} &::= \texttt{ERROR}(\text{название ошибки}) \\
\mathit{fql} &::= \langle \mathit{topLevelLocation}, \mathit{path} \rangle \\
\mathit{topLevelLocation} &::= \texttt{Null} \mid \texttt{SLoc}(\texttt{LOC}) \mid \texttt{DMLoc}(\texttt{LOC}) \\
&\mid \texttt{SMLoc}(\texttt{LOC}) \mid \texttt{IPLoc}(\texttt{LOC}) \\
\mathit{path} &::= \langle \mathit{pathSegment} \rangle^{*} \\
\mathit{pathSegment} &::= \texttt{SF}(\texttt{LOC}) \mid \texttt{AI}(\texttt{LOC}) \\
&\mid \texttt{ALB}(\texttt{LOC}) \mid \texttt{AL}(\texttt{LOC}) \\
\mathit{ref} &::= \texttt{REF}(\mathit{fql}) \\
\mathit{ptr} &::= \texttt{PTR}(\mathit{fql}, \mathit{offset}) \\
\mathit{expr} &::= \texttt{EXPRESSION}(\texttt{operation}, \langle \mathit{term} \rangle^{+}) \\
\mathit{array} &::= \texttt{ARRAY}(\texttt{dimension}, \texttt{elements}, \texttt{lower bounds}, \texttt{lengths}) \\
\mathit{struct} &::= \texttt{STRUCT}(\texttt{fields}) \\
\mathit{union} &::= \texttt{UNION}(\langle \mathit{guard}, \mathit{term} \rangle^{*}) \\
\mathit{guard} &::= \top \mid \bot \mid \mathit{expr}
\end{aligned}
$$

Вначале разберём простые случаи терма: `NOP` обозначает отсутствие операции, `ERROR` является представлением исключений программы, `CONCRETE` – конкретное значение простого типа (например, 3 или ’a’), `SYMBOL` — это символьной значение, т. е. абстракция над конкретным значением, используемая в символьном исполнении, `LOC` — адрес области памяти. Благодаря свойству иерархичности символьной памяти любую локацию программы можно представить с помощью *fql* (fully qualified location), которая состоит из адреса внутри верхнеуровневого символьного блока (*topLevelLocation*), а также последовательности адресов внутри представлений логических блоков составных объектов (*path*). Верхнеуровневыми локациями могут быть следующие: `Null`, `SLoc` (stack location, т. е. локация на стеке), `DMLoc` (dynamic memory location,

т. е. локация в динамической памяти), `SMLoc` (static memory location, т. е. локация в статической памяти), `IPLoc` (interning pool location, т. е. локация в пуле интернирования). Примером верхнеуровневой локации является `SMLoc`(*ClassWithStaticFields*), где у класса *ClassWithStaticFields* имеются статические поля. В качестве элементов последовательности адресов внутри представлений логических блоков могут выступать следующие: `SF` (struct field, т. е. поле структуры), `AI` (array index, т. е. индекс в массиве), `ALB` (array lower bound, т. е. нижняя граница массива по указанной размерности), `AL` (array length, т. е. длина массива по указанной размерности). Примером такой последовательности может быть (`SF`(*AnotherStruct*), `SF`(*field*)), где у структуры, лежащей в верхнеуровневом символьном блоке, есть поле *AnotherStruct*, по которому лежит структура с полем *field*. Таким образом, терм `REF`, используя *fql*, обозначает ссылки на область памяти (например, для ссылочных типов). Указатели в виртуальной машине представлены с помощью терма `PTR`, который включает *fql*, сдвиг в байтах от данной локации и тип, с которым рассматривается данная область памяти. Более сложными случаями терма (*составными*) являются `STRUCT`, описывающий экземпляры классов и структур, а также `ARRAY`, который является внутренним представлением виртуальной машины для массива анализируемой программы. Данные представления составных объектов будут подробнее рассмотрены далее, также как и представление строки. Терм `UNION` — это синтаксическая конструкция статического символьного исполнения, позволяющая объединять результаты символьного исполнения веток программы. Элементами терма `UNION` являются результаты символьного исполнения отдельных веток выполнения программы в виде термов, которые защищены логическими формулами условий пути *guard*, обеспечивающими попадание в эти ветки, т. е. $x = \texttt{UNION}\big(\langle g_1, v_1\rangle, \ldots, \langle g_n, v_n\rangle\big)$ т.и т.т. $(g_1 \wedge x = v_1) \vee \ldots \vee (g_n \wedge x = v_n)$.

Представление экземпляров классов и структур. Терм `STRUCT` описывает экземпляры классов и структур в символьной виртуальной машине. Данный терм внутри себя содержит символьную кучу `fields`,

которая является представлением полей структуры или класса. Для сохранения информации о размещении полей (указанной с помощью атрибута `StructLayout`), адреса символьной кучи `fields` — это названия полей вместе с их отступом от начала структуры, т. е. $\langle \mathit{fieldName}, \mathit{offset} \rangle$. Благодаря поддержке неопределённости данных в символьной памяти, решается задача символьного исполнения кода в условиях полиморфизма. Покажем данную задачу на примере программы, изображённой на лист. 3. В качестве аргумента функции `PolyAdd` передаётся экземпляр обобщённого типа `T`, рассматриваемый с помощью терма `STRUCT`, о полях которого информации нет, т. е. символьная куча `fields` пуста (опр. 3). После динамического приведения типов знания о полях появляются, а в куче инициализируются необходимые для исполнения значения. В случае, если полем является массив фиксированного размера, внутри символьной кучи `fields` по соответствующему адресу добавляется терм `ARRAY`, о котором далее пойдёт речь.

```
class ExampleStruct
{
    public int x;
}
class AnotherStruct
{
    public int a;
}
public static T PolyChange<T>(T poly, int value)
{
    if (poly is ExampleStruct)
    {
        ExampleStruct example = poly as ExampleStruct;
        example.x = value;
    }
    if (poly is AnotherStruct)
    {
        AnotherStruct example = poly as AnotherStruct;
        example.a = value;
    }
    return poly;
}
```

Листинг 3: Пример ad-hoc-полиморфизма для случая класса

Представление массивов. Так как существует общий класс для всех типов массивов, называемый `System.Array`, который позволяет во время исполнения программы создать массив произвольной размерности с заданными пользователем нижними границами и длинами, необходимо добавить неопределённость размерности, длин и нижних границ по каждой из размерностей. С этой целью терм `ARRAY` состоит из размерности, которая так же является термом, символьной кучи (lengths), содержащей длины по каждой размерности, символьной кучи с нижними границами (lower bounds) и, наконец, символьной кучи с элементами массива (elements). В примере на лист. 4 демонстрируется необходимость неопределённости данных для терма `ARRAY`.

```
public static System.Array RetSystemArray1(System.Array arr)
{
    if (arr is int[])
    {
        var arrOne = arr as int[];
        arrOne[1] = 5;
    }
    else if (arr is int[,])
    {
        var arrOne = arr as int[,];
        arrOne[1,1] = 7;
    }
    return arr;
}
```

Листинг 4: Пример ad-hoc-полиморфизма для случая массива

Представление строк. Строка в символьной виртуальной машине представляется как терм `STRUCT`, одним элементом которого является терм `ARRAY`, содержащий символы строки, а другим — его длина. В случае конкретной строки длина представлена в виде терма `CONCRETE`, а в случае символьной — `SYMBOL`.

3.4. Моделирование операций с памятью в рамках безопасного контекста

В данном разделе в терминах языка символьной виртуальной машины описано выполнение операций с символьной памятью в рамках безопасного контекста. Среди данных операций: выделение объекта, чтение, запись, упаковка и распаковка, а также интернирование.

3.4.1. Выделение объекта

Наиболее простой операцией является «alloc» (выделение) – добавляет в символьный блок значение с указанным ключом. Так как любой верхнеуровневый логический блок программы представляется либо с помощью символьной кучи, либо символьного стека, то необходимо описать данную операцию для двух этих случаев.

Определение 4. *Выделение* терма v в символьной куче (символьном стеке) ρ по локации $y \notin dom(\rho)$ — это символьная куча (символьный стек) $alloc(\rho, y, v)$, такая(-ой), что для всех $x \in dom\big(alloc(\rho, y, \cdot)\big) = dom(\rho) \cup \{y\}$,

$$\big(alloc(\rho, y, v)\big)\big(x\big) \stackrel{\text{def}}{=} \begin{cases} v, & x = y \\ \rho(x), & x \neq y \end{cases}$$

В случае выделения класса или структуры со статическими полями в статической памяти, ключом является тип этого класса или структуры. Если выделение происходит внутри динамической памяти, тогда ключ – это `CONCRETE` терм, который до этого ещё не встречался в качестве локации символьной кучи. При выделении строки внутри пула интернирования, в качестве ключа используется терм `STRUCT` самой строки.

3.4.2. Чтение значения

Так как любая локация выражается через *fql*, а значит ссылка на эту будет выглядеть как терм $\texttt{REF}(\mathit{fql})$, то необходимо описать только операцию чтения для терма $\texttt{REF}$ из символьной памяти M. Далее при описании операций чтения и записи будем использовать следующие сокращения: $[x_1; x_n] \stackrel{\text{def}}{=} (x_1, \ldots, x_n)$, $\texttt{NRE} \stackrel{\text{def}}{=} \texttt{ERROR}(\mathit{NullReferenceException})$. Также будут использоваться стандартные обозначения деления нацело (т. е. div), и взятие остатка от деления (т. е. mod).

Определение 5. *Чтение* локации по ссылке *ref* из символьной памяти M определяется следующим образом:

$$\mathit{read}(M, \mathit{ref}) \stackrel{\text{def}}{=} \begin{cases} \texttt{NRE}, & \mathit{ref} = \texttt{Ref}(\langle \texttt{Null}, \mathit{xs} \rangle) \\ \mathit{readStack}\big(S(M), x, \mathit{xs}\big), & \mathit{ref} = \texttt{REF}(\langle \texttt{SLoc}(x), \mathit{xs} \rangle) \\ \mathit{readHeap}\big(\mathit{DM}(M), x, \mathit{xs}\big), & \mathit{ref} = \texttt{REF}(\langle \texttt{DMLoc}(x), \mathit{xs} \rangle) \\ \mathit{readHeap}\big(\mathit{SM}(M), x, \mathit{xs}\big), & \mathit{ref} = \texttt{REF}(\langle \texttt{SMLoc}(x), \mathit{xs} \rangle) \\ \mathit{readHeap}\big(\mathit{IP}(M), x, \mathit{xs}\big), & \mathit{ref} = \texttt{REF}(\langle \texttt{IPLoc}(x), \mathit{xs} \rangle) \end{cases}$$

В данном определении функции $S(M)$, $\mathit{DM}(M)$, $\mathit{SM}(M)$ и $\mathit{IP}(M)$ обозначают взятие соответствующего элемента из символьной памяти M.

Определение 6. *Чтение* символьной кучи σ определяется следующим образом:

$$\mathit{readHeap}(\sigma, x, \mathit{xs}) \stackrel{\text{def}}{=} \texttt{UNION}\Big(\big\{\big\langle x = l, \mathit{readTerm}\big(\sigma(l), \mathit{xs}\big)\big\rangle \mid l \in \mathit{dom}(\sigma)\big\} \cup \big\langle \bigwedge_{l \in \mathit{dom}(\sigma)} x \neq l, \mathit{readTerm}\big(\mathit{LI}(x), \mathit{xs}\big)\big\rangle\Big)$$

В данном определении участвует функция ленивой инстанциации (англ. lazy instantiation) LI(x). Идея данной функции является модификацией идеи ленивой инстанциации, описанной в разделе 2.1. Данная функция по указанному адресу создаёт так называемую «ленивую

ячейку», т. е. локацию, о которой информация до этого отсутствовала. Благодаря использованию данной функции при описании операций чтения и записи обеспечивается поддержка символьной памятью неопределённости данных.

Определение 7. *Создание ленивой ячейки* по локации x определяется следующим образом:

$$LI(x) \stackrel{\text{def}}{=} \begin{cases} \mathtt{STRUCT}(\epsilon), & isClassOrStruct(x) \\ \mathtt{ARRAY}(\mathtt{SYMBOL}, \epsilon, \epsilon, \epsilon), & isArray(x) \\ \mathtt{SYMBOL}, & isSimpleType(x) \end{cases}$$

В данном определении функция $isClassOrStruct(x)$ возвращает истинное значение, если тип локации x является типом структуры или класса, в противном случае данная функция возвращает ложное значение. Функция $isArray(x)$ определяет, является ли тип локации x типом массива, а функция $isSimpleType(x)$ — является ли тип x простым типом.

Определение 8. *Чтение* символьного стека ς определяется следующим образом:

$$readStack(\varsigma, x, path) \stackrel{\text{def}}{=} \begin{cases} readTerm(\varsigma(x), path), & x \in dom(\varsigma) \\ readTerm(LI(x), path), & x \notin dom(\varsigma) \end{cases}$$

Определение 9. *Чтение* значения из составного терма t определяется следующим образом:

$$readTerm\big(t, [x_1; x_n]\big) \stackrel{\text{def}}{=}$$

$$\stackrel{\text{def}}{=} \begin{cases} t, & n = 0 \\ readHeap\big(fields(t), x, [x_2; x_n]\big), & n \neq 0 \land x = \mathtt{SF(x)} \\ readHeap\big(elems(t), x, [x_2; x_n]\big), & n \neq 0 \land x = \mathtt{AI(x)} \\ readHeap\big(LBs(t), x, [x_2; x_n]\big), & n \neq 0 \land x = \mathtt{ALB(x)} \\ readHeap\big(lens(t), x, [x_2; x_n]\big), & n \neq 0 \land x = \mathtt{AL(x)} \end{cases}$$

В данном определении были использованы следующие функции:

- *fields*(t) — возвращает символьную кучу `fields` из структуры t;
- *elems*(t) — возвращает кучу `elements` из терма массива t;
- *LBs*(t) — возвращает кучу `lower bounds` из терма массива t;
- *lens*(t) — возвращает кучу `lengths` из терма массива t.

3.4.3. Запись значения

Операция записи описывается аналогично операции чтения, изменяя при этом символьную память M.

Определение 10. *Запись* терма v в символьную память M по ссылке *ref* определяется следующим образом:

$$write(M, ref, v) \stackrel{\text{def}}{=}$$

$$\stackrel{\text{def}}{=} \begin{cases} withDM\Big(M, alloc\big(DM(M), a, \mathtt{NRE}\big)\Big), & ref = \mathtt{REF}(\langle \mathtt{Null}, xs\rangle) \\ withStack\Big(M, writeStack\big(S(M), x, v, xs\big)\Big), & ref = \mathtt{REF}(\langle \mathtt{SLoc}(x), xs\rangle) \\ withDM\Big(M, writeHeap\big(DM(M), x, v, xs\big)\Big), & ref = \mathtt{REF}(\langle \mathtt{DMLoc}(x), xs\rangle) \\ withSM\Big(M, writeHeap\big(SM(M), x, v, xs\big)\Big), & ref = \mathtt{REF}(\langle \mathtt{SMLoc}(x), xs\rangle) \\ withIP\Big(M, writeHeap\big(IP(M), x, v, xs\big)\Big), & ref = \mathtt{REF}(\langle \mathtt{IPLoc}(x), xs\rangle) \end{cases}$$

В данном определении функции *withStack*(M, s), *withDM*(M, dm), *withSM*(M, sm) и *withIP*(M, ip) возвращают символьную память M с заменённым соответствующим элементом (S, DM, SM, IP соответственно).

Определение 11. *Запись* значения v в символьную кучу σ — это символьная куча $writeHeap(\sigma, y, v, path)$, такая что для всех
$x \in dom\big(writeHeap(\sigma, y, \cdot, path)\big) = dom(\sigma) \cup \{y\}$,

$$\big(writeHeap(\sigma, y, v, path)\big)(x) \stackrel{\text{def}}{=} \mathtt{UNION}\Big(\big\langle x = y, writeTerm\big(LI(x), path, v\big)\big\rangle, \big\langle x \neq y, writeTerm\big(\sigma(x), path, v\big)\big\rangle\Big)$$

Определение 12. *Запись* значения v в символьный стек ς — это символьный стек $writeStack(\varsigma, y, v, path)$, такой что для всех
$x \in dom\big(writeStack(\varsigma, y, \cdot, path)\big) = dom(\sigma) \cup \{y\}$,

$$\big(writeStack(\varsigma, y, v, path)\big)(x) \stackrel{\text{def}}{=} \begin{cases} writeTerm\big(\varsigma(x), path, v\big), & x = y \\ writeTerm\big(LI(x), path, v\big), & x \neq y \end{cases}$$

Определение 13. *Запись* значения v в составной терм t определяется следующим образом:

$$writeTerm\big(t, [x_1; x_n], v\big) \stackrel{\text{def}}{=}$$
$$\stackrel{\text{def}}{=} \begin{cases} v, & n = 0 \\ withFields\Big(t, writeHeap\big(fields(t), x, [x_2; x_n]\big)\Big), & n \neq 0 \wedge x_1 = \mathtt{SF}(x) \\ withElems\Big(t, writeHeap\big(elems(t), x, [x_2; x_n]\big)\Big), & n \neq 0 \wedge x_1 = \mathtt{AI}(x) \\ withLBs\Big(t, writeHeap\big(LBs(t), x, [x_2; x_n]\big)\Big), & n \neq 0 \wedge x_1 = \mathtt{ALB}(x) \\ withLens\Big(t, writeHeap\big(lens(t), x, [x_2; x_n]\big)\Big), & n \neq 0 \wedge x_1 = \mathtt{AL}(x) \end{cases}$$

В данном определении функции $withFields(t, fields)$, $withElems(t, elems)$, $withLBs(t, LBs)$, $withLens(lens)$ обозначают терм t с заменённым соответствующим элементом (*fields*, *elements*, *lowerbounds*, *lengths* соответственно).

3.4.4. Упаковка/распаковка

Операции упаковки (box) и распаковки (unbox) сводятся к ранее описанным операциям взаимодействия с символьной памятью M.

Определение 14. *Упаковка* терма x при символьной памяти M определяется следующим образом:

$$box(M, v) \stackrel{\text{def}}{=} withDM\Big(M, alloc\big(DM(s), v, v\big)\Big)$$

Определение 15. *Распаковка* терма x при символьной памяти M определяется следующим образом:

$$unbox(M, v) \stackrel{\text{def}}{=} read(s, \langle DMLoc(x), empty\rangle)$$

В данном определении используется сокращение $empty \stackrel{\text{def}}{=} [x_1; x_n]$, где $n = 0$.

3.4.5. Интернирование

Операции механизма интернирования (т. е. `String.Intern` и `String.IsInterned`) сводятся к операциям чтения и записи с пулом интернирования, т. е. символьной кучей. Вначале опишем операции взаимодействия с данной кучей.

$$readIP(IP, str, li) \stackrel{\text{def}}{=} \texttt{UNION}\Big(\big\{\big\langle str = l, IP(l)\big\rangle \mid l \in dom(IP)\big\} \cup \big\langle \bigwedge_{l \in dom(IP)} str \neq l, li\big\rangle\Big)$$

Определение 16. *Запись* значения $strRef$ в символьную кучу «пул интернирования» IP по ключу str — это символьная куча $writeIP(IP, str, strRef)$, такая что для всех $x \in dom\big(writeIP(IP, str, \cdot)\big) = dom(IP) \cup \{str\}$,

$$\big(writeIP(IP, str, strRef)\big)\big(x\big) \stackrel{\text{def}}{=} \begin{cases} IP(x), & x \in dom(IP) \\ strRef, & x \notin dom(IP) \end{cases}$$

С помощью описанных операций далее определим функции *intern* и *isInterned*.

$$intern(M, strRef) \stackrel{\text{def}}{=} \langle newRef, withIP(M, newPool) \rangle$$

В данном определении использовались следующие сокращения:
$str \stackrel{\text{def}}{=} read(M, strRef)$, $newRef \stackrel{\text{def}}{=} readIP(IP(M), str, strRef)$,
$newPool \stackrel{\text{def}}{=} writeIP(IP(M), str, strRef)$.

$$isInterned(M, strRef) \stackrel{\text{def}}{=} readIP(IP(M), str, null)$$

В данном определении использовалось сокращение
$null \stackrel{\text{def}}{=} \texttt{REF}(\texttt{Null}, empty)$.

3.5. Моделирование операций с памятью в рамках небезопасного контекста

В данном разделе в терминах языка символьной виртуальной машины описано выполнение операций с символьной памятью в рамках небезопасного контекста. Среди данных операций: создание указателя, адресная арифметика, приведение типов указателей, выделение массива на стеке, а также разыменование указателя. Последняя операция, т. е. разыменование указателя, используется в двух случаях: чтение и запись по указателю. Оба этих случая будут рассмотрены.

Создание указателя. Данная операция в символьной сводится к построению *fql* и дальнейшему созданию терма $\texttt{PTR}(fql, 0)$.

Адресная арифметика. Данная операция сводится к арифметике чисел, которые находятся внутри терма $\texttt{PTR}(fql, offset)$, т. е. *offset*.

Приведение типов указателей. Данная операция сводится к изменению типа у терма PTR, с которым рассматривается область памяти.

Stackalloc. Операция выделения массива на стеке сводится к выделению терма `ARRAY` на стеке.

3.5.1. Разыменование указателя

Для описания данной операции в грамматику термов добавляется два случая: $\mathit{Slice}(\mathit{term}, \mathit{from}, \mathit{to})$, $\mathit{Combine}(\langle \mathit{term} \rangle^*)$. Терм $\mathit{Slice}(x, \mathit{from}, \mathit{to})$ позволяет взять часть терма x с байта *from* по байт *to*. Терм $\mathit{Combine}(\langle \mathit{term} \rangle^*)$ позволяет последовательно совместить несколько термов в единую последовательность байт. Далее будут использоваться следующие сокращения: $\mathit{empty} \stackrel{\text{def}}{=} [x_1; x_n]$, где $n = 0$, $\texttt{UB} \stackrel{\text{def}}{=} \texttt{ERROR}(\mathit{UndefinedBehavior})$.

Определение 17. *Чтение* области символьной памяти M по указателю p определяется следующим образом

$$\mathit{ptrRead}(M, p) \stackrel{\text{def}}{=}$$

$$\stackrel{\text{def}}{=} \begin{cases} \texttt{UB}, & p = \texttt{PTR}(\langle \texttt{Null}, xs \rangle, \mathit{offset}) \\ \mathit{readPtrTerm}(\mathit{read}(M, \mathit{LB}), \mathit{byte}, \mathit{ptrSize}), & p = \texttt{PTR}(\mathit{fql}, \mathit{offset}) \end{cases}$$

В данном определении используется следующие сокращения: $\mathit{LB} \stackrel{\text{def}}{=} \texttt{REF}(\mathit{discardLast}(\mathit{LBfql}(\mathit{fql})))$, $\mathit{ptrSize} \stackrel{\text{def}}{=} \mathit{sizeOfPtr}(p)$, $\mathit{byte} \stackrel{\text{def}}{=} \mathit{getByte}(\mathit{LBfql}(\mathit{fql}), \mathit{offset})$. Функция $\mathit{LBfql}(\mathit{fql})$ принимает *fql* и возвращает его от начала и включительно до первого элемента $\texttt{AI}(\mathit{index})$ или $\texttt{SF}(\mathit{field})$, где у поля *field* известное смещение (sequential или explicit). При отсутствии элментов $\texttt{AI}(\mathit{index})$ или $\texttt{SF}(\mathit{field})$, *fql* остаётся без изменений. Функция $\mathit{sizeOfPtr}(p)$ возвращает размер области памяти, на которую указывает p в байтах, что возможно благодаря ограниченному набору типов, на которые может указывать указатель. Помимо этого в определении участвует функция $\mathit{getByte}(\mathit{LBfql}, \mathit{offset})$, которая возвращает байт, на который указывает p, относительно начала логического блока, который находится по ссылке *LB*.

$$discardLast(fql) \overset{\text{def}}{=} \begin{cases} fql, & fql = \langle topLevel, empty \rangle \\ \langle topLevel, [x_1; x_n - 1] \rangle, & fql = \langle topLevel, [x_1; x_n] \rangle \end{cases}$$

$$readPtrTerm(t, byte, ptrSize) \overset{\text{def}}{=}$$
$$\overset{\text{def}}{=} \begin{cases} ptrStructRead(fields, byte, size, ptrSize), & t = \mathtt{STRUCT}(fields) \\ ptrArrayRead(e, byte, esize, size, ptrSize), & t = \mathtt{ARRAY}(d, e, lb, l) \\ ptrSimpleRead(t, byte, size, ptrSize), & isSimpleType(t) \end{cases}$$

В данном определении используются следующие сокращения: $size \overset{\text{def}}{=} sizeOf(t)$, $esize \overset{\text{def}}{=} sizeOfElems(t)$. Использованная функция $sizeOf(t)$ возвращает размер терма t в байтах, $sizeOfElems(t)$ возвращает размер элементов массива t в байтах, $isSimpleType(t)$ возвращает истину, если t является простым типом.

$$ptrSimpleRead(t, byte, size, ptrSize) \overset{\text{def}}{=}$$
$$\overset{\text{def}}{=} \mathtt{UNION}\Big(\big\{\big\langle byte = b, Slice(t, b, b + ptrSize)\big\rangle \mid b \in [0; size - ptrSize]\big\} \cup \big\langle byte < 0 \lor byte > size - ptrSize, \mathtt{UB}\big\rangle\Big)$$

$$ptrStructRead(fields, byte, size, ptrSize) \overset{\text{def}}{=}$$
$$\overset{\text{def}}{=} \mathtt{UNION}\Big(\big\{\big\langle byte = b, readStructBytes\big(fields, b, size, ptrSize\big)\big\rangle \mid b \in [0; size - ptrSize]\big\} \cup \big\langle byte < 0 \lor byte > size - ptrSize, \mathtt{UB}\big\rangle\Big)$$

$$
\begin{aligned}
&\mathit{readStructBytes}(\mathit{fields}, \mathit{byte}, \mathit{size}, \mathit{ptrSize}) \stackrel{\text{def}}{=} \\
&\stackrel{\text{def}}{=} \mathit{Combine}(\big\{\mathit{SliceIfNeed}(\mathit{fields}(x), \mathit{getOffset}(x), \mathit{byte}, \mathit{ptrSize}) \\
&\qquad\qquad \mid x \in \mathit{dom}(\mathit{fields}), \mathit{cAlign}(x)\big\})
\end{aligned}
$$

В данном определении используется следующее сокращение:
$\mathit{cAlign}(x) \stackrel{\text{def}}{=} \mathit{correctAligned}(\mathit{fields}(x), \mathit{getOffset}(x), \mathit{byte}, \mathit{ptrSize})$.

$$
\mathit{getOffset}(\langle \mathit{fieldName}, \mathit{offset}\rangle) \stackrel{\text{def}}{=} \mathit{offset}
$$

$$
\begin{aligned}
&\mathit{correctAligned}(t, \mathit{offset}, \mathit{byte}, \mathit{ptrSize}) \stackrel{\text{def}}{=} \\
&\qquad \stackrel{\text{def}}{=} (\mathit{byte} \leqslant \mathit{offset} + \mathit{sizeOf}(t)) \wedge (\mathit{byte} + \mathit{ptrSize} \geqslant \mathit{offset})
\end{aligned}
$$

$$
\mathit{SliceIfNeed}(t, \mathit{offset}, \mathit{byte}, \mathit{ptrSize}) \stackrel{\text{def}}{=} \mathit{ptrTermRead}(t, l, r - l)
$$

В данном определении используются следующие сокращения:
$l \stackrel{\text{def}}{=} \mathit{max}(\mathit{byte}, \mathit{offset}) - \mathit{offset}$,
$r \stackrel{\text{def}}{=} \mathit{min}(\mathit{byte} + \mathit{ptrSize}, \mathit{offset} + \mathit{sizeOf}(t)) - \mathit{offset}$.

$$
\begin{aligned}
&\mathit{ptrArrayRead}(\mathit{elems}, \mathit{byte}, \mathit{esize}, \mathit{size}, \mathit{ptrSize}) \stackrel{\text{def}}{=} \\
&\stackrel{\text{def}}{=} \mathit{ite}(\mathit{byte} < 0 \vee \mathit{byte} + \mathit{ptrSize} > \mathit{size}, \mathtt{UB}, \\
&\qquad \mathit{IntoArray}(\mathit{elems}, \mathit{ind}, \mathit{delta}, \mathit{esize}, \mathit{ptrSize}))
\end{aligned}
$$

В данном определении были использованы следующие сокращения:
$\mathit{ite}(c, t, e) \stackrel{\text{def}}{=} \mathtt{UNION}\big(\langle c, t\rangle, \langle \neg c, e\rangle\big)$, $\mathit{ind} \stackrel{\text{def}}{=} \mathit{byte} \operatorname{div} \mathit{esize}$,
$\mathit{delta} \stackrel{\text{def}}{=} \mathit{byte} \bmod \mathit{esize}$.

$$IntoArray(elems, ind, delta, esize, ptrSize) \stackrel{\text{def}}{=}$$
$$\stackrel{\text{def}}{=} \texttt{UNION}\Big(\big\{\big\langle delta = b, readArrayBytes\big(elems, ind, b, esize, ptrSize\big)\big\rangle \mid b \in [0; esize]\big\}\Big)$$

$$readArrayBytes(elems, ind, delta, esize, ptrSize) \stackrel{\text{def}}{=}$$
$$\stackrel{\text{def}}{=} Combine(leftPart \cup midPart \cup rightPart)$$

В данном определении используются следующие сокращения:
$leftPart \stackrel{\text{def}}{=} ptrTermRead(readAI(elems, ind), delta, l)$,
$l \stackrel{\text{def}}{=} min(ptrSize, esize - delta)$, $n \stackrel{\text{def}}{=} (delta + ptrSize)$ div $esize$,
$midPart \stackrel{\text{def}}{=} \{readAI(elems, i) \mid i \in [ind + 1; ind + n - 1]\}$,
$rightPart \stackrel{\text{def}}{=} ptrTermRead(readAI(elems, ind + n), 0, rightDelta)$,
$rightDelta \stackrel{\text{def}}{=} (delta + ptrSize)$ mod $esize$.

$$readAI(elems, ind) \stackrel{\text{def}}{=} readHeap(elems, ind, empty)$$

Определение 18. *Запись* значения v в символьную память M по указателю p определяется следующим образом

$$ptrWrite(M, p, v) \stackrel{\text{def}}{=}$$
$$\stackrel{\text{def}}{=} \begin{cases} M, & p = \texttt{PTR}(\langle \texttt{Null}, xs\rangle, offset) \\ write(M, LB, newLB), & p = \texttt{PTR}(fql, offset) \end{cases}$$

В данном определении используется следующие сокращения:
$LB \stackrel{\text{def}}{=} \texttt{REF}(discardLast(LBfql(fql)))$, $ptrSize \stackrel{\text{def}}{=} sizeOfPtr(p)$,
$byte \stackrel{\text{def}}{=} getByte(LBfql(fql), offset)$,
$newLB \stackrel{\text{def}}{=} writePtrTerm(read(M, LB), byte, v)$.

$$writePtrTerm(t, byte, v) =$$

$$= \begin{cases} ptrStructWrite(fields, byte, size, v), & t = \texttt{STRUCT}(fields) \\ ptrArrayWrite(e, byte, esize, size, v, t), & t = \texttt{ARRAY}(d, e, lb, l) \\ ptrSimpleWrite(t, byte, size, v), & isSimpleType(t) \end{cases}$$

В данном определении использовались следующие сокращения:
$size \stackrel{\text{def}}{=} sizeOf(t),\ esize \stackrel{\text{def}}{=} sizeOfElems(t).$

$$ptrSimpleWrite(t, byte, size, v) \stackrel{\text{def}}{=}$$
$$\stackrel{\text{def}}{=} \texttt{UNION}\Big(\big\{\big\langle byte = b, updateTermPart(t, b, size, v)\big\rangle$$
$$\mid b \in [0; size - sizeOf(v)]\big\}$$
$$\cup \big\langle byte < 0 \vee byte > size - sizeOf(v), \texttt{UB}\big\rangle\Big)$$

$$updateTermPart(t, byte, size, v) \stackrel{\text{def}}{=}$$
$$\stackrel{\text{def}}{=} Combine(\{Slice(t, 0, byte), v, Slice(t, byte + sizeOf(v), size)\})$$

$$ptrStructWrite(fields, byte, size, v) \stackrel{\text{def}}{=}$$
$$\stackrel{\text{def}}{=} \texttt{UNION}\Big(\big\{\big\langle byte = b, \texttt{STRUCT}\big(writeStructBytes(fields(x), b, size, v)\big)\big\rangle$$
$$\mid b \in [0; size - sizeOf(v)]\big\}$$
$$\cup \big\langle byte < 0 \vee byte > size - sizeOf(v), \texttt{STRUCT}(fields)\big\rangle\Big)$$

Определение 19. *Запись* значения v по байту $byte$ в символьную кучу структуры $fields$ — это символьная куча
$writeStructBytes(fields, byte, size, v)$, такая что для всех

$x \in dom\big(writeStructBytes(fields, byte, size, \cdot)\big) = dom(fields),$

$$\big(writeStructBytes(fields, byte, size, v)\big)\,(x) \stackrel{\text{def}}{=}$$
$$\stackrel{\text{def}}{=} \begin{cases} updateFieldPart(fields(x), getOffset(x), byte, size, v), & cAlign(x) \\ fields(x), & \neg cAlign(x) \end{cases}$$

В данном определении использовалось сокращение:
$cAlign(x) \stackrel{\text{def}}{=} correctAligned(fields(x), getOffset(x), byte, sizeOf(v))$.

$$updateFieldPart(t, offset, byte, size, v) \stackrel{\text{def}}{=}$$
$$\stackrel{\text{def}}{=} writePtrTerm(t, max(byte, offset) - offset, vpart)$$

В данном определении используется сокращение
$vpart \stackrel{\text{def}}{=} ptrTermRead(v, start, size)$, $start \stackrel{\text{def}}{=} max(byte, offset) - byte$,
$size \stackrel{\text{def}}{=} min(byte + sizeOf(v), offset + sizeOf(t)) - max(byte, offset)$.

$$ptrArrayWrite(elems, byte, esize, size, v, t) \stackrel{\text{def}}{=}$$
$$\stackrel{\text{def}}{=} ite(byte < 0 \lor byte + sizeOf(v) > size, t, updatedArray)$$

В данном определении использовалось сокращение
$updatedArray \stackrel{\text{def}}{=} writeIntoArray(elems, ind, delta, esize, v, t)$,
$ind \stackrel{\text{def}}{=} byte \operatorname{div} esize$, $delta \stackrel{\text{def}}{=} byte \operatorname{mod} esize$.

$$writeIntoArray(elems, ind, delta, esize, v, t) \stackrel{\text{def}}{=}$$
$$\stackrel{\text{def}}{=} \texttt{UNION}\Big(\big\{\big\langle delta = b, withElems(t, updatedElems)\big\rangle \mid b \in [0; esize]\big\}\Big)$$

В данном определении использовалось сокращение
$updatedElems \stackrel{\text{def}}{=} writeArrayBytes(elem, ind, b, esize, v)$.

$$writeArrayBytes(elems, ind, delta, esize, ptrSize) \stackrel{\text{def}}{=}$$
$$\stackrel{\text{def}}{=} writeAI(newMidLeft, endInd, updatedRight)$$

В данном определении использовались следующие скоращения:
$newLeft \stackrel{\text{def}}{=} writeAI(elems, ind, updatedLeft)$,
$updatedLeft \stackrel{\text{def}}{=} writePtrTerm(oldLeft, delta, leftVPart)$,
$newMidLeft \stackrel{\text{def}}{=} writeInd(newLeft, ind + 1, endInd - 1, esize, delta, v)$,
$updatedRight \stackrel{\text{def}}{=} writePtrTerm(oldRight, 0, rightVPart)$,
$oldLeft \stackrel{\text{def}}{=} readAI(elems, ind)$, $endInd \stackrel{\text{def}}{=} ind + end$,
$leftVPart \stackrel{\text{def}}{=} ptrTermRead(v, 0, min(ptrSize, esize - delta))$,
$right \stackrel{\text{def}}{=} ptrSize - rightDelta$, $ptrSize \stackrel{\text{def}}{=} sizeOf(v)$,
$rightVPart \stackrel{\text{def}}{=} ptrTermRead(v, right, rightDelta)$,
$rightDelta \stackrel{\text{def}}{=} (delta + ptrSize) \bmod esize$,
$oldRight \stackrel{\text{def}}{=} readAI(elems, endInd)$,
$end \stackrel{\text{def}}{=} (delta + ptrSize) \operatorname{div} esize.$

$$writeInd(elems, ind, endInd, esize, delta, v) \stackrel{\text{def}}{=}$$
$$\stackrel{\text{def}}{=} \begin{cases} updated, & ind = endInd \\ writeInd(updated, ind + 1, endInd, esize, delta, v), & ind \neq endInd \end{cases}$$

В данном индуктивном определении использовались сокращения:
$vpart \stackrel{\text{def}}{=} ptrTermRead(v, ind * esize - delta, ind * esize - delta + esize)$,
$updated \stackrel{\text{def}}{=} writeAI(elems, ind, vpart)$.

$$writeAI(elems, ind, v) \stackrel{\text{def}}{=} writeHeap(elems, ind, v, empty)$$

3.5.2. Кодирование в SMT-решатель

В данном разделе будет описан способ кодирования внутреннего языка символьной виртуальной машины (см. раздел 3.3.2) в язык SMT-решателей, т. е. теории логики первого порядка. Отметим, что благода-

ря описанию операций, в условии пути могут появляться только термы простых типов, которые и кодируются в решатель. В разработанном кодировании можно выделить два случая: трансляция в условиях безопасного контекста и небезопасного контекста.

Кодирование в безопасном контексте. В данном случае в программе отсутствуют операции с указателями, а значит, нет необходимости транслировать термы `Slice`$(term, from, to)$ и `Combine`$(\langle term\rangle^*)$. Таким образом, в качестве теории логики первого порядка была выбрана теория линейной арифметики. Наиболее простыми случаями для кодирования являются термы `CONCRETE` и `SYMBOL`, которые транслируются в конкретные числа и константы (const — неизвестные данные, для которых необходимо найти интерпретацию) соответственно. Терм `UNION`$\big(\langle g_1, v_1\rangle, \ldots, \langle g_n, v_n\rangle\big)$ кодируется с помощью введения новой переменной x и добавления ограничений на неё $(g_1 \wedge x = v_1) \vee \ldots \vee (g_n \wedge x = v_n)$.

Кодирование в небезопасном контексте. В данном случае в программе могут появляться операции с указателям, а, следовательно, термы `Slice`$(term, from, to)$ и `Combine`$(\langle term\rangle^*)$. В качестве теории логики первого порядка была выбрана теория битовых векторов, в которой вышеупомянутые термы представляются тривиальным образом. Терм `Slice`$(term, from, to)$ кодируется как операция «взятие подвектора», а терм `Combine`$(\langle term\rangle^*)$ представляется с помощью нескольких операций «конкатенация двух бит-векторов». Остальные простые случаи терма кодируются следующим образом: `CONCRETE` — число в теории битовых векторов, `SYMBOL` — константа в теории битовых векторов, `UNION`$\big(\langle g_1, v_1\rangle, \ldots, \langle g_n, v_n\rangle\big)$ — новая переменная x со следующими ограничениям: $(g_1 \wedge x = v_1) \vee \ldots \vee (g_n \wedge x = v_n)$.

4. Архитектура и детали реализации

В данной главе описывается архитектура разработанной системы «модель памяти», а также её реализация в символьной виртуальной машине V#.

4.1. Архитектура

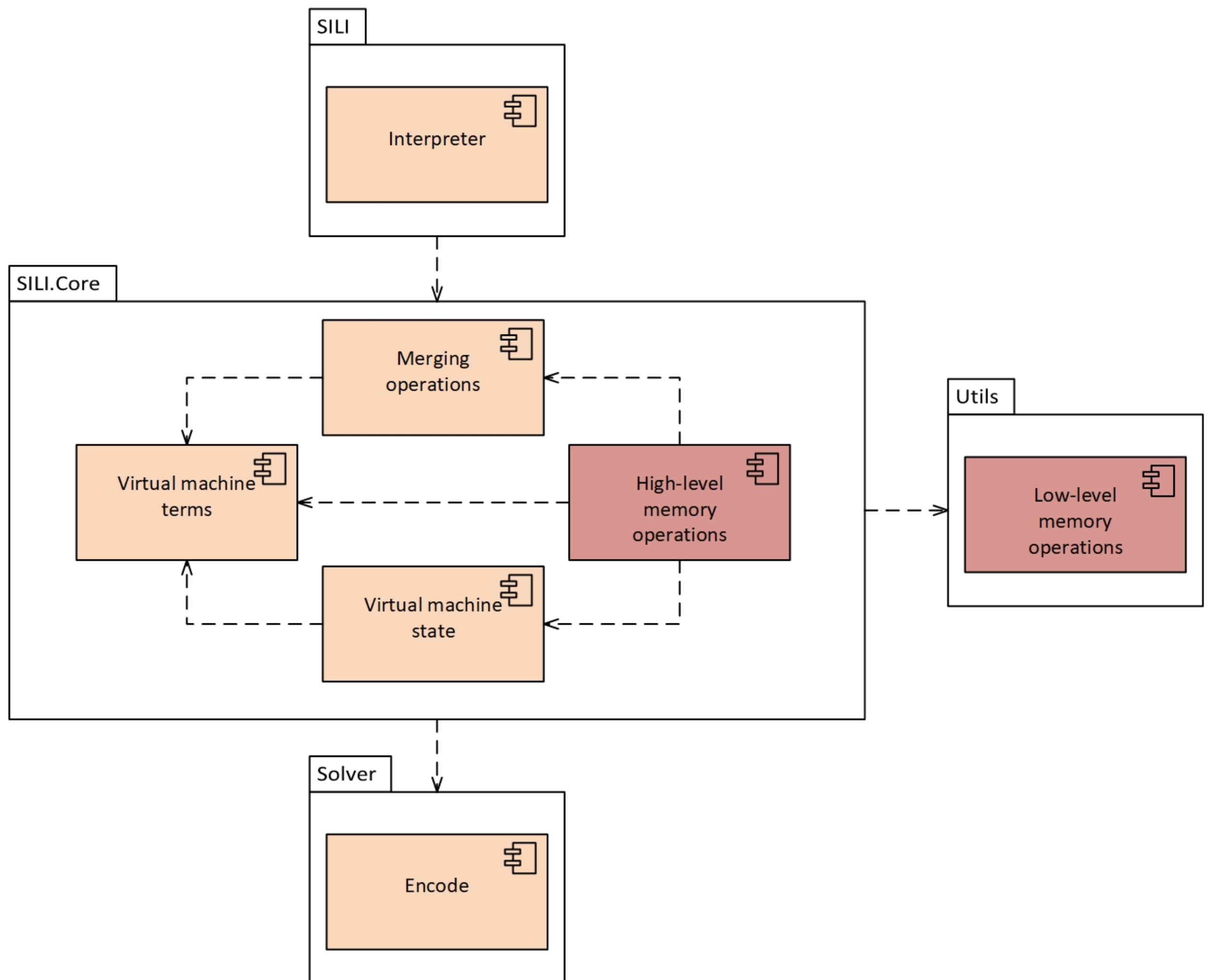

Рис. 2: Архитектура объемлющей подсистемы проекта V#

Разработанная модель символьной памяти является частью проекта V#, которая затрагивает большинство уже существующих составляющих. На рис. 2 показано, как система операций с памятью встроена в архитектуру символьной виртуальной машины. Вначале опишем вза-

имодействие проекта V# и системы «модель памяти», после чего подробнее рассмотрим данную разработанную систему.

Interpreter. Данный модуль выполняет символьное исполнение кода программы с помощью функций ядра системы (SILI.Core). При встрече взаимодействия с памятью программы, интерпретатор вызывает необходимую функцию из модуля «High-level memory operations», который является основной частью разработанной системы.

Virtual machine state. В данном модуле описано состояние символьной виртуальной машины. Частью этого состояния является символьная память.

Virtual machine terms. Данный модуль описывает внутренний язык символьной виртуальной машины. Элементы языка термов, в частности, находятся внутри символьной памяти.

Merging operations. Данная компонента обеспечивает слияние состояний и термов с помощью введённой ранее синтаксической конструкции `UNION`. Описанное слияние используется при объединении результатов операций с памятью.

Encode. Во время интерпретации кода возникает необходимость проверки достижимости ветки выполнения, т. е. выполнимости условия пути, что осуществляется с помощью запроса к решателю. Данный модуль выполняет кодирование таких запросов в язык решателя. В частности, результаты взаимодействия с символьной памятью транслируются в логические формулы с помощью данного модуля.

4.1.1. Система операций с символьной памятью

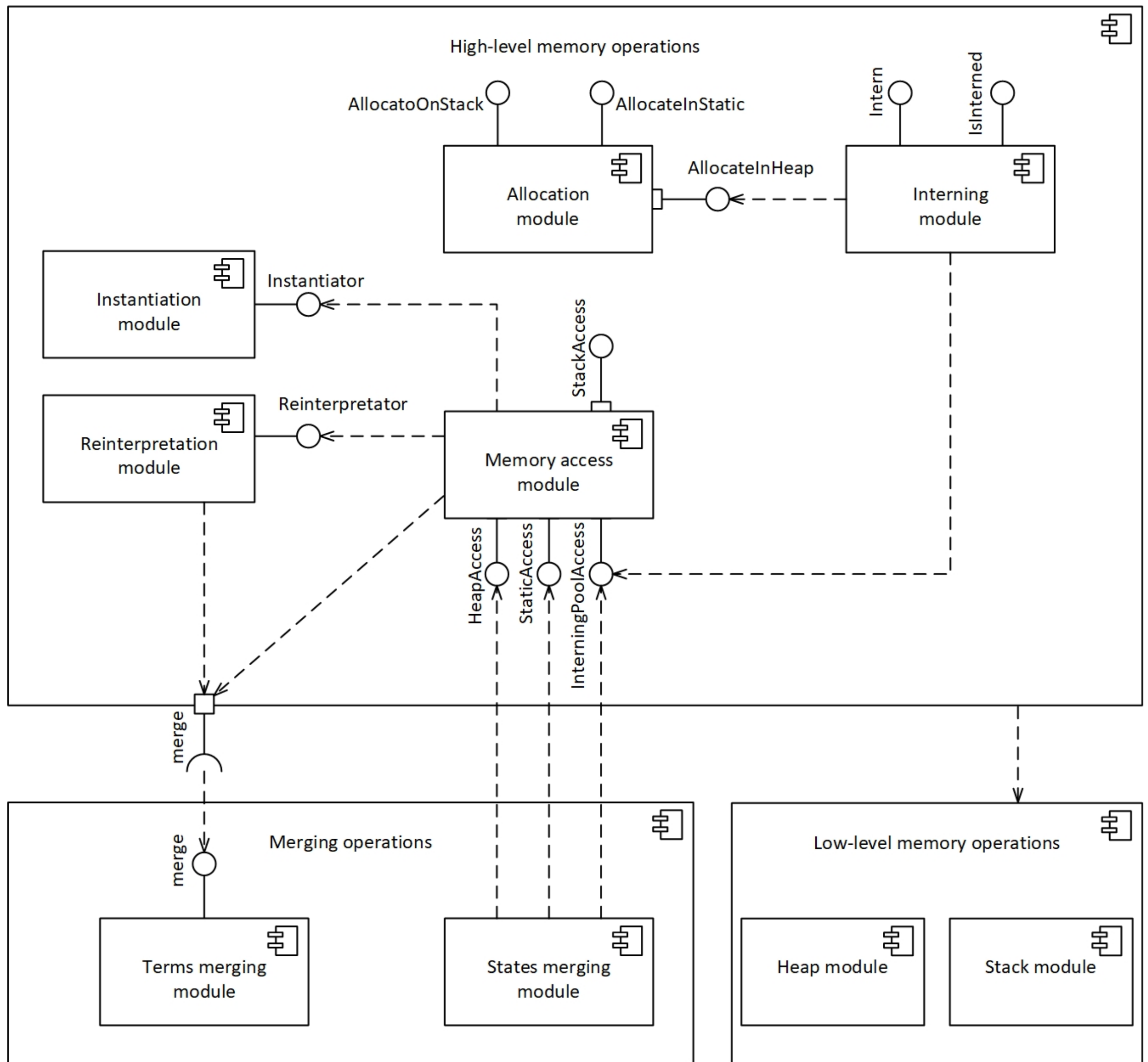

Рис. 3: Архитектура системы «модель памяти»

Разработанная система, архитектура которой представлена на рис. 3, состоит из двух частей:

- блока «Низкоуровневые операции с памятью» (англ. low-level memory operations), в котором находятся описанные в предыдущих разделах структуры «символьная куча» и «символьный стек», а также примитивы для работы с ними (например, проверка наличия ключа);
- блока «Высокоуровневые операции с памятью» (англ. high-level

memory operations), выполняющего все взаимодействия с символьной памятью анализируемой программы на основе примитивов блока «Низкоуровневые операции с памятью».

Heap module. Данная компонента блока «Низкоуровневые операции с памятью» описывает структуру «символьная куча» в символьной виртуальной машине, а также примитивы взаимодействия с ней.

Stack module. Данная компонента блока «Низкоуровневые операции с памятью» описывает структуру «символьная куча» в символьной виртуальной машине, а также примитивы взаимодействия с ней.

Instantiation module. Данная компонента отвечает за создание значения по-умолчанию (англ. default value), а также символьного значения, т. е., в частности, реализует функцию $LI(x)$.

Reinterpretation module. Данная компонента позволяет совершать чтение и запись по указателю, предоставляя возможность совершать реинтерпретацию данных. Во время работы этого модуля используется функция слияния термов «merge» из компоненты «Terms merging module».

Memory access module. Данная компонента является основным элементом в разработанной системе. Она обеспечивает работу операций чтения и записи в общем: как по ссылке, так и по указателю. Данные операции предоставляются для использования в интерпретаторе. Для работы этих операций компонента агрегирует операции из модуля реинтерпретаций, а также использует функцию $LI(x)$ из компоненты «Instantiation module». Помимо компонент из блока «Высокоуровневые операции с памятью» в модуле доступа к памяти также используется функция слияния термов из компоненты «Terms merging module», находящейся в блоке «Merging operations». В разработанной системе слияние символьных куч реализовано через операцию чтения кучи. Для

этого компонента «Memory access module» предоставляет функции чтения различных символьных куч для компоненты слияния состояний виртуальной машины «States merging module».

Allocation module. Данная компонента описывает операцию выделения в символьной памяти, предоставляя наружу следующие функции: выделение на стеке, в динамической и статической памяти.

Interning module. Данная компонента описывает операции механизма интернирования, используя функцию взаимодействия с пулом интернирования из компоненты «Memory access module», а также функцию выделения в динамической памяти из компоненты «Allocation module».

4.2. Детали реализации

Реализация созданной модели памяти составляет 1215 строк кода на языке F# в проекте V#. Реализованная модель памяти имеет два режима работы:

- безопасный контекст;
- небезопасный контекст.

Безопасный контекст. Особенностью данного режима является кодирование термов простых типов, полученных в результате выполнения операций с памятью, в теорию линейной арифметики. Благодаря такому кодированию анализ программ получается быстрее, чем в режиме «небезопасный контекст» (так как известны эффективные алгоритмы проверки формул теории арифметики на выполнимость), однако сильно упрощённый в случае наличия операций с указателями. Упрощаются все операции, которые ведут к реинтерпретации данных. В случае использование данного режима, кодирование происходит в решатель Z3.

Небезопасный контекст. В случае использования данного режима работы, при запросе к решателю кодирование всех термов производится в теорию битовых векторов. Этот режим работы обеспечивает точный анализ программ с указателями, однако работает не так эффективно, как режим «безопасный контекст». В качестве решателя в данном режиме используется инструмент ELDARICA, который хорошо зарекомендовал себя в проверке на выполнимость логических формул теории битовых векторов.

5. Тестирование

Разработанная модель была реализована в проекте V#, в следствие чего были затронуты части всего проекта. Для проверки функциональности разработанной системы были написаны тесты на языке C#. Данные тесты являются методами, которые необходимо исполнить в символьной виртуальной машине. С помощью данных методов тестировалось выполнение операций со структурами, классами, массивами, строками, а также указателями (некоторые из них вели к реинтерпретации данных). В табл. 1 представлены результаты тестирования. Данные результаты показывают, что система успешно проходит тестирование функциональности и может использоваться в проекте V# как модель памяти.

Название теста	Результат	Верный результат
Struct.TestPolyChange	Безопасно	Безопасно
Struct.TestMutateFieldFromSymbolicStruct	Безопасно	Безопасно
Struct.MutateFieldOfSymbolicStructWithConcrete	Небезопасно	Небезопасно
Class.CreateSymbolicField	Безопасно	Безопасно
Arrays.TestMutateWithConcrete	Безопасно	Безопасно
Arrays.TestMutateWithSymbolic	Безопасно	Безопасно
Arrays.TestConcreteLowerBoundException	Небезопасно	Небезопасно
Arrays.TestMutateLessConcreteDimLowerBound	Небезопасно	Небезопасно
Arrays.TestMutateLessSymbolicDimLowerBound	Небезопасно	Небезопасно
Arrays.TestMutateGreaterConcreteDimUpperBound	Небезопасно	Небезопасно
Arrays.TestMutateGreaterSymbolicDimUpperBound	Небезопасно	Небезопасно
Arrays.TestMutateSystemArrayWithOneDimsArray	Безопасно	Безопасно
Arrays.TestMutateSystemArrayWithTwoDimsArray	Безопасно	Безопасно
Arrays.TestMutateSystemArrayWithThreeDimsArray	Безопасно	Безопасно
Arrays.MutateLastValueWithConcrete	Безопасно	Безопасно
Arrays.MutateLastValueWithSymbolic	Безопасно	Безопасно
String.TestNullLength	Небезопасно	Небезопасно
String.CreateStringOfConcreteCharArray	Безопасно	Безопасно
String.LenOfConcreteCharArrayString	Безопасно	Безопасно
String.TestLiteralIsInterned	Безопасно	Безопасно
String.TestConcreteInternOfNotInterned	Безопасно	Безопасно
String.TestNotInternedConcreteStringIsInterned	Безопасно	Безопасно
Unsafe.TestSizeOfFixedSizedBuffer	Безопасно	Безопасно
Unsafe.TestSizeOfStruct	Безопасно	Безопасно
Unsafe.TestDoubleIndirection	Безопасно	Безопасно
Unsafe.TestInderectionOfAdressOf	Безопасно	Безопасно
Unsafe.TestPointerDifference	Безопасно	Безопасно
Unsafe.TestReinterpretateStructField	Безопасно	Безопасно
Unsafe.TestReinterpretateArrayOfTwoBytes	Безопасно	Безопасно
Unsafe.TestSumReinterpretedArrayOfBytesEqualsZero	Безопасно	Безопасно
Unsafe.TestReinterpretateStackAlloc	Безопасно	Безопасно

Таблица 1: Результаты экспериментов

Заключение

В ходе работы были получены следующие результаты:

- выполнен обзор следующих моделей памяти: модель на основе анализа алиасов, типизированная модель памяти, модель памяти Бурсталла-Борната, модель с регионами, LISBQ; сделан обзор инструментов анализа программ, поддерживающих реинтерпретацию данных: Predator, Pex, VCC3, Jessie;
- создана иерархическая модель памяти для платформы .NET с поддержкой реинтерпретации данных на основе теории битовых векторов и линейной арифметики;
- созданная модель памяти реализована на языке F# в рамках проекта V#;
- проведено тестирование функциональности полученного решения.

Список литературы

[1] Andersen Lars Ole. Program analysis and specialization for the C programming language : Ph. D. thesis / Lars Ole Andersen ; University of Cophenhagen. — 1994.

[2] Barrett Clark W, Dill David L, Levitt Jeremy R. A decision procedure for bit-vector arithmetic // Proceedings 1998 Design and Automation Conference. 35th DAC.(Cat. No. 98CH36175) / IEEE. — 1998. — P. 522–527.

[3] Beyer Dirk. *Software verification and verifiable witnesses* // International Conference on Tools and Algorithms for the Construction and Analysis of Systems / Springer. — 2015. — P. 401–416.

[4] Beyer Dirk. *Reliable and reproducible competition results with benchexec and witnesses (report on SV-COMP 2016)* // International Conference on Tools and Algorithms for the Construction and Analysis of Systems / Springer. — 2016. — P. 887–904.

[5] Beyer Dirk. *Software verification with validation of results* // International Conference on Tools and Algorithms for the Construction and Analysis of Systems / Springer. — 2017. — P. 331–349.

[6] Bjørner Nikolaj, Phan Anh-Dung, Fleckenstein Lars. *νZ-an optimizing SMT solver* // International Conference on Tools and Algorithms for the Construction and Analysis of Systems / Springer. — 2015. — P. 194–199.

[7] Böhme Sascha, Moskal Michał. Heaps and data structures: A challenge for automated provers // International Conference on Automated Deduction / Springer. — 2011. — P. 177–191.

[8] Bornat Richard. Proving pointer programs in Hoare logic // International Conference on Mathematics of Program Construction / Springer. — 2000. — P. 102–126.

[9] Bradley Aaron R, Manna Zohar, Sipma Henny B. What's decidable about arrays? // International Workshop on Verification, Model Checking, and Abstract Interpretation / Springer. — 2006. — P. 427–442.

[10] Cousot Patrick, Cousot Radhia. Abstract interpretation and application to logic programs // The Journal of Logic Programming. — 1992. — Vol. 13, no. 2-3. — P. 103–179.

[11] De Moura Leonardo, 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.

[12] Dudka Kamil, Peringer Petr, Vojnar Tomáš. *Byte-precise verification of low-level list manipulation* // International Static Analysis Symposium / Springer. — 2013. — P. 215–237.

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

[14] Enhancing Symbolic Execution by Machine Learning Based Solver Selection / Sheng-Han Wen, Wei-Loon Mow, Wei-Ning Chen et al.

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

[16] Hubert Thierry, Marché Claude. Separation analysis for deductive verification // Heap Analysis and Verification (HAV'07). — 2007. — P. 81–93.

[17] Khurshid Sarfraz, Păsăreanu Corina S, Visser Willem. Generalized symbolic execution for model checking and testing // International Conference on Tools and Algorithms for the Construction and Analysis of Systems / Springer. — 2003. — P. 553–568.

[18] Kolawa Adam K, Salvador Roman. *Method and system for generating a computer program test suite using dynamic symbolic execution of Java programs.* — 1998. — US Patent 5,784,553.

[19] Lahiri Shuvendu, Qadeer Shaz. Back to the future: revisiting precise program verification using SMT solvers // ACM SIGPLAN Notices / ACM. — Vol. 43. — 2008. — P. 171–182.

[20] Mandrykin Mikhail, Khoroshilov Alexey. A memory model for deductively verifying Linux kernel modules // International Andrei Ershov Memorial Conference on Perspectives of System Informatics / Springer. — 2017. — P. 256–275.

[21] Moy Yannick. Automatic modular static safety checking for C programs : Ph. D. thesis / Yannick Moy ; Paris 11. — 2009.

[22] Pasareanu Corina S, Phan Quoc-Sang, Malacaria Pasquale. Multi-run side-channel analysis using Symbolic Execution and Max-SMT // 2016 IEEE 29th Computer Security Foundations Symposium (CSF) / IEEE. — 2016. — P. 387–400.

[23] Sethu Ramesh. *Systems and methods to reverse engineer code to models using program analysis and symbolic execution.* — 2018. — Mar. 22. — US Patent App. 15/268,011.

[24] Tillmann Nikolai, de Halleux Jonathan. *Pex–White Box Test Generation for. NET* // Tests and Proofs. — 2008. — P. 134–153.

[25] VCC: A practical system for verifying concurrent C / Ernie Cohen, Markus Dahlweid, Mark Hillebrand et al. // International Conference on Theorem Proving in Higher Order Logics / Springer. — 2009. — P. 23–42.

[26] A precise yet efficient memory model for C / Ernie Cohen, Michał Moskal, Stephan Tobies, Wolfram Schulte // Electronic Notes in Theoretical Computer Science. — 2009. — Vol. 254. — P. 85–103.

[27] The software model checker b last / Dirk Beyer, Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar // International Journal on Software Tools for Technology Transfer. — 2007. — Vol. 9, no. 5-6. — P. 505–525.

[28] *Cvc4* / Clark Barrett, Christopher L Conway, Morgan Deters et al. // International Conference on Computer Aided Verification / Springer. — 2011. — P. 171–177.

[29] *Dependence guided symbolic execution* / Haijun Wang, Ting Liu, Xiaohong Guan et al. // IEEE Transactions on Software Engineering. — 2017. — Vol. 43, no. 3. — P. 252–271.

[30] *KLOVER: Automatic Test Generation for C and C Programs, Using Symbolic Execution* / Hiroaki Yoshida, Guodong Li, Takuki Kamiya et al. // IEEE Software. — 2017. — Vol. 34, no. 5. — P. 30–37.

[31] *Predator shape analysis tool suite* / Lukáš Holík, Michal Kotoun, Petr Peringer et al. // Haifa Verification Conference / Springer. — 2016. — P. 202–209.

[32] *Symbolic optimization with SMT solvers* / Yi Li, Aws Albarghouthi, Zachary Kincaid et al. // ACM SIGPLAN Notices / ACM. — Vol. 49. — 2014. — P. 607–618.

[33] *Veritesting Challenges in Symbolic Execution of Java* / Vaibhav Sharma, Michael W Whalen, Stephen McCamant, Willem Visser // ACM SIGSOFT Software Engineering Notes. — 2018. — Vol. 42, no. 4. — P. 1–5.

[34] Мандрыкин Михаил Усамович. Моделирование памяти Си-программ для инструментов статической верификации на основе SMT-решателей : Ph. D. thesis / Михаил Усамович Мандрыкин ; Диссертация на соискание ученой степени кандидата физико-математических наук ИСП РАН, Москва. — 2016.