Министерство образования и науки Российской Федерации
Московский физико-технический институт (государственный университет)

Физтех-школа радиотехники и компьютерных технологий
Кафедра микропроцессорных технологий в интеллектуальных системах управления

Выпускная квалификационная работа магистра

Разработка средств инструментации бинарного файла многоязыковой виртуальной машины.

Автор:

Студент М01-206б группы

Назаров Константин Олегович

Научный руководитель:

Добров Андрей Дмитриевич

Научный консультант:

Солдатов Антон Анатольевич

Москва 2024

Аннотация

Разработка средств инструментации бинарного файла многоязыковой виртуальной машины.

Назаров Константин Олегович

Инструментация байткода является распространенной техникой для изменения поведения или аттрибутов программы. Эта техника может использоваться в целях профилировки, мониторинга, а также для динамического изменения поведения программы даже после сборки приложения. Существующие библиотеки для бинарной инструментации кода рассчитаны на инструментацию бинарных файлов для какого-либо языка программирования. В ходе данной работы был разработан фреймворк для работы с бинарными файлами многоязыковой платформы. Было проведено сравнение с существующими аналогами, а также показаны возможности инструментации кода, полученного из исходных файлов на разных языках.

Содержание

1 Введение

Изменение семантики программы посредством манипуляции ее бинарным кодом является распространенной техникой, используемой в современной разработке. Она используется для добавления к существующему проекту элементов рефлексии или имплементации парадигм аспектно-ориенторованного программирования (далее АОП). Зачастую, целью подобных изменений является решение существующих проблем традиционного объектно-ориентированного программирования (далее ООП). Несмотря на то, что применение ООП позволяет увеличить продуктивность разработки за счет переиспользования существующих компонент посредством таких механизмов как полиморфизм и наследование, соблюдение этих принципов не всегда возможно. Например в проектах, в которых невозможно реализовать централизованную стратегию разработки и развития всех компонент, таких как внешние библиотеки. Конечный пользователь не имеет контроля над совместимостью компонент из разных библиотек, а также не всегда имеет доступ к исходному коду, что приводит к проблеме обеспечения функциональной совместимости (интероперабельности) между компонентами. Для решения этой проблемы ООП предлагает написание классов-оберток, которые бы обеспечивали интерфейсную совместимость. Данный подход приводит к проблемам поддержки кодовой базы, а также обеспечения выполнения контрактов между компонентами. Бинарная манипуляция является альтернативным решением проблем интероперабельности. Вместо того, чтобы создавать новые классы-обертки, пользователь может модифицировать оригинальные определения в библиотеке [1]. Изменения происходят на стороне пользователя после поставки компоненты, и не требуют доступа к исходному коду.

Помимо обеспечения интероперабельности между компонентами, бинарная манипуляция также применяется для обеспечения принципа разделения обязанностей, являющегося важной частью АОП. Соблюдение этого принципа заключается в отделении разработки функциональной части

приложения от имплементации нефункциональных контрактов, таких как обеспечение безопасности, поддержка единой модели распространения приложения и дргуих. Для достижения этой цели недостаточно простого изменения интерфейсов или иерархии классов, и необходимо иметь инструменты для модификации тел уже существующих функций [2].

Помимо упомянутых выше, существует еще множество областей применения бинарной инструментации. Ниже приведен более подробный разбор уже существующих задач, а также рассмотрены некоторые другие области применения данной техники.

1.1 Интероперабельность

Интероперабельность - это способность двух или более компонент программного обеспечения работать вместе несмотря на различия в языке, интерфейсе или исполняющей платформе [3]. В рамках данной работы остановимся на рассмотрении кросс-интерфейсной интероперабельности и того, как задача бинарной манипуляции связана с ней.

Существует два основных механинзма для реализации интероперабельности: стандартизация интерфейсов (Interface Standartization) и интерфейсный мост (Interface Bridging). Первый подход заключается в создании единого интерфейса и последующего приведения интерфейсов участников взаимодействия к нему. Второй подход состоит в создании взаимооднозначного соответствия между интерфейсами всех участников взаимодействия [4]. Стандартизация интерфейсов зачастую явялется предпочтительным решением, поскольку является наиболее масштабируемым. Действиетльно, для данного подхода нужно построить всего $n + m$ соответствий, где n - число интерфейсов первого участинка, и m - число интерфейсов второго участника взаимодействия. Для второго подхода число соответствий будет равно $m*n$. Недостатком же стандартизации интерфейсов является то, что она значительно усложняет дальнейшее расширение интерфейсов, а также поддержку новых функциональностей, остутствующих в языке на момент стандартизации интерфейсов. Промышленным примером системы ин-

тероперабельности с стандартизованным интерфейсом является стандарт Component Object Model COM/OLE компании Microsoft. Данный стандарт решает проблему межпроцессного взаимодействия путем предоставления единого бинарного интерфейса для представления объектов в системе [5]. Альтернативным подходом к решению проблемы кросс-интерфейсной интероперабельности явялется предоставление механизма для изменения интерфейсов при их загрузке в систему.

В рамках платформы JVM для языка Java данный подход был реализован в фреймворке BCA [1]. Данный фреймворк работает следующим образом: он предоставляет пользователю интерфейс для написания собственного трансформатора (Modifier), который принимает считанный бинарный файл Java (далее класс-файл, class-file), проводит над ним указанные пользователем манипуляции, после чего возвращает измененный файл и передает его верификатору JVM. При помощи бинарной манипуляции достигается интерфейсная совместимость по месту использования, что гарантирует сохранение совместимости используемых компонент, а также позволяет отложить манипуляции до непосредственно загрузки бинарного файла на платформу, что приводит к уменьшению дополнительной нагрузки на виртуальную машину.

Более подробно фреймворк BCA будет рассмотрен в разд. 3.

1.2 АОП

Выше упоминалось, что ООП не всегда способно обеспечить выполнение принципа разделения обязанностей. Можно без труда заметить, что применение объектно-ориентированного подхода приводит к тому, что некоторые участки кода повторяются во многих классах, поскольку функциональность, реализуемая данным кодом, присуща им всем. Особенно это заметно, когда речь заходит об имплементации контрактов или не функциональных свойств системы, таких как модель обработки ошибок, проверка пост- и предусловий или аутентификация и проверка прав доступа. Для решения этих проблем была предложена парадигма АОП [6]. Основной идеей

АОП стала мысль, что программист должен иметь возможность раздельно судить о функциональных и не функциональных свойствах системы.

С точки зрения АОП функиональное требование - это некоторое требование, позволяющее реализовать какую-либо концепцию, возможность или вид функциональности. Аспект же - это код, реализующий заданное функциональное требование, который запускается другими функциональными требованиями в различных ситуациях [7]. Если функциональное требование не было выделено в аспект, то функциональность этого требования будет вызываться явно в коде, реализующем другой вид требования, что приводит к спутыванию (`cross-cutting`) этих требований. Кроме того, необходимость вызовов кода функционального требования в различных местах, приводит к рассеиванию реализации требования по многим несвязным участкам системы. В рамках АОП такая функциональность называется сквозная функциональность. Таким образом, АОП является парадигмой программирования, основанной на идее отделения сквозной функциональности от основной для улучшения разбиения программы на модули. АОП ставит своей целью разработать механизм модуляризации сквозной функциональности в объектно-ориентированных системах, предоставляя языковые средства для отделения функциональности от аспектов. Действительно, наличие модели программирования, которая предполагает реализацию таких требований единожды и их использование только в одном месте программы позволяет значительно сократить время разработки, а также улучшить читаемость и отлаживаемость кода.

В парадигме АОП каждое приложение можно поделить на классы и аспекты. Модульность в приложении достигается следующим способом: основная функциональность реализуется в виде классов, в то время как сквозная - аспектами.

Следствием того, что классическое ООП плохо справляется с проблемой отделения аспектов является широкое использование в промышленности фреймворков по типу `BREX` [8]. Данный фреймворк изолирует сегмен-

ты кода относящиеся к бизнесс-логике проекта, что помогает эффективнее менять и адаптировать правила, которым должен следовать продукт. Для достижения этой цели `BREX` производит анализ проекта в три этапа: классификация переменных (Variable Classification), идентификация бизнесс-логик (Business Rule Identification) и отображение бизнесс-логик (Business Rule Representation).

Работу данного фреймворка можно проиллюстрировать на слеюющем примере. Пусть имеется приложение, описывающее поведение людей и животных. В данном приложении имплементированы две основные функциональности. Одна является бизнесс-логикой проекта и описывает, как взаимодействия между хищниками и жертвами влияет на общую популяцию. Вторая используется для хранения статистик всех акторов участвующих в симуляции. Ниже на рис. 1 приведена схема приложения, из которой можно понять устройство проекта.

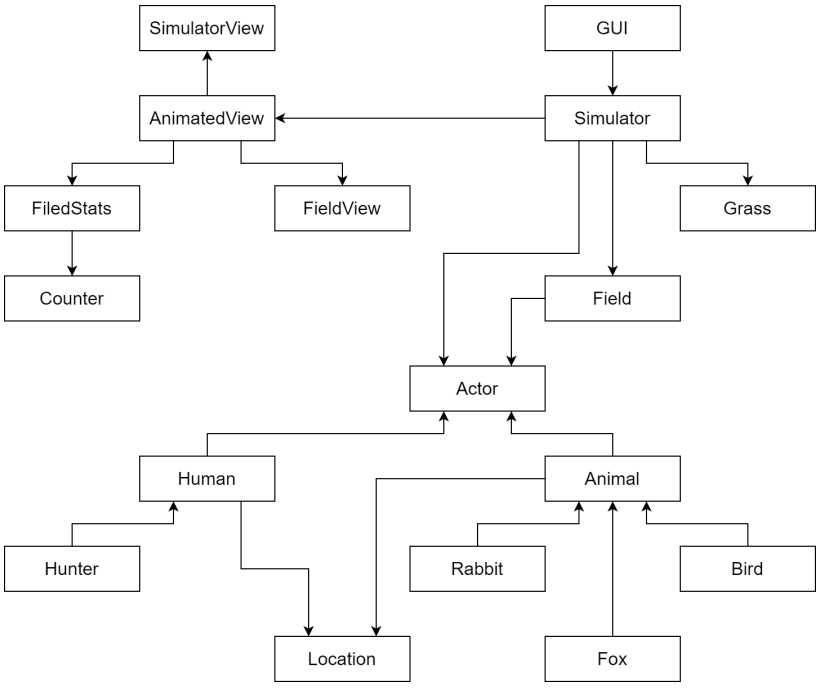

Рис. 1: Схема зависимостей приложения.

После применения каждой из описанных выше стадий, фреймворк выделит поетнциальные участки, относящиеся к сквозным функциональностям, присутствующим в программе. В качестве результата фреймворк генерирует диаграмму найденных связей, пример которой приведен ниже.

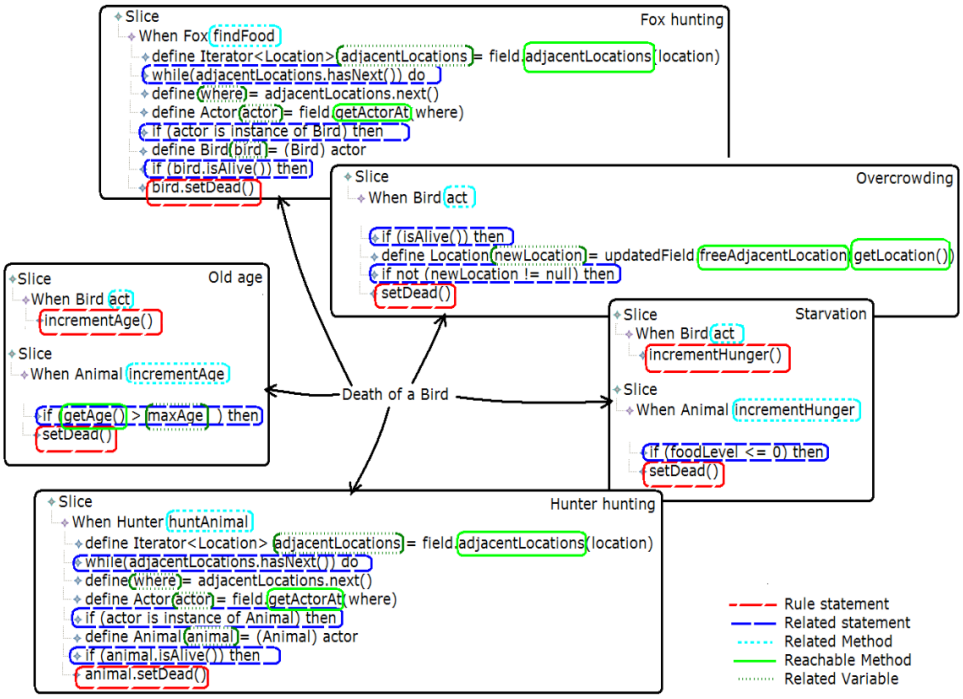

Рис. 2: Результат работы фреймворка - потенциальные причины смерти птицы.

Схема на рис. 2 является текстовым представлением всех условий, которые приводят к смерти птиц в данной модели. На изображении выделены различные сгенерированные бизнесс-логики, контролирующие смерть птиц, которые были обнаружены фреймворком. Данный анализ может помочь установить все явные и неявные контракты приложения, однако он не решает изначальной проблемы: кодовая база все еще не модуляризована и имеет множество сквозных функциональностей. Ниже речь пойдет о том, как внедрение элементов АОП позволяет повысить модульность проекта и избежать спутывания кода.

Поскольку широкое распространение парадигма АОП получила в языке `Java`, для него были написаны многие фреймворки, привносящие элементы АОП в язык и платформу `JVM`. Популярность АОП на платформе `JVM` объясняется в том числе относительной открытостью элементов процесса сборки к расширению. Например пользователь имеет возможность встроить свои элементы логики на многих важных этапах компиляции, таких как загрузка считанного файла в виртуальную машину или построение дерева абстрактного синтаксиса. Еще одной причиной распространенности

парадигмы АОП среди `Java` приложений является наличие в языке механизма аннотаций, позволяющего помечать пользовательскими данными место определения класса, функции или поля, а также место использования типа. Данные аннотации будут сохранены в бинарный класс-файл и позже могут быть считаны как компилятором, так и во время исполнения кода рантаймом виртуальной машины посредством языковых средств рефлексии. Наличие механихма аннотаций делает естественным внедрение логик, основанных на аннотировании языковых сущностей. Более того, в самом языке `Java` определены несколько встроенных аннотаций, которые также можно считать зачатками аспекто-ориентированности в языке. Например аннотация *@Deprecated* означает, что сущность, помеченная этой аннотацией больше не является поддерживаемой, и ее использование должно быть ограничено. Встречая использование сущности, помеченной этой аннотацией, компилятор будет генерировать соответствующее предупреждение, что будет приводить к падению процесса сборки при должных настройках [9].

```
1 /**
2  * @deprecated
3  * explanation of why it was deprecated
4  */
5 @Deprecated
6 static void deprecatedMethod() { }
```

Листинг 1: Пример использования аннотации `Deprecated`

Выше в Листинг 1 приведен пример использования аннотации `Deprecated`. Любое дальнейшее использование метода `deprecatedMethod` будет сопровождаться предупреждением от компилятора.

Примером того, как бинарная манипуляция может помочь с достижением модуляризации аспектов может служить следующий пользовательский сценарий, полученный из промышленной кодовой базы. Чтобы обеспечить выполнение требований к безопасности, при сборке приложения под определенное окружение разработчик может захотеть убедиться, что перед любым применением некоторого набора функций будет вызываться

служебная процедура, проверяющая все необходимые для обеспечения безопасности условия. Исходя из описанного выше, наивная имплементация подобнойлогики приведет к пересечению аспекта, отвечающего за обеспечение безопасности, с основной функциональностью приложения. Чтобы избежать этого и обеспечить модульность контракта безопасности используется следующая АОП архитектура, основанная на применении пользовательских аннотаций бинарной манипуляции.

```
1  // Declaration of the annotation that will be used to specify
2  // safe wrappers for the library API.
3  @interface CallSiteReplacement {
4     String targetClass();
5     String methodName();
6  }
7
8  // Class that is responsible for providing safe wrappers for the library API.
9  public class ApiControl {
10     // Presence of this annotation means that we need to replace all calls to
11     // the method ‘foo‘ from the class ‘com.xxx.Api‘ with the calls to
12     // ‘ApiControl.WrapApiFoo‘
13     @CallSiteReplacement(targetClass="com.xxx.Api", methodName="foo");
14     public static void wrapApiFoo(target: com.xxx.Api, ...) {
15         // Logic that ensures safe use of the ‘foo‘ API.
16     }
17 };
```

Листинг 2: Пример решения задачи замены функции в месте вызова при помощи АОП

Рассмотрим Листинг 2. В нем приведен пример того, как может быть реализована АОП архитектура, обеспечивающая вызов безопасной обертки для библиотечного метода `foo` класса `com.xxx.Api`. Класс `ApiControl` реализует функционал, проверяющий все необходимые параметры для безопасного вызова конкретных функций библиотеки. Каждый метод этого класса - это безопасная обертка одной из библиотечных функций. Аспектом в даной архитектуре является механизм замены мета вызова на безопасный при определенных параметрах сборки. Данный аспект, как и каждый, состоит из двух частей - точки соединения (join point) и совета (advice).

Точка соединения - это точка в выполняемой программе, где следует применить совет. В данном случае точкой соединения будет являться каждая конкретная инстанциация аннотации `CallSiteReplacement`. В ней указаны два параметра: `targetClass` и `methodName`. Они определяют то, вызовы какого метода должен быть заменены на безопасную обертку. Советом же в данной схеме будет являться логика преобразования бинарного файла, находящая все функции, аннотированные аннотацией `CallSiteReplacement`, и заменяющая все вызовы методов, указанных в аргументах аннотации на вызов аннотированной функции. Ниже в Листинг 3 приведен пример того, как выглядит код до и после применения описанных выше трансформаций.

```
1  // Before AOP.
2  public class MyClass {
3      public static void main() {
4          com.xxx.Api api = GetApi();
5          api.foo(...);
6      }
7  };
8
9  // After AOP.
10 public class MyClass {
11     public static void main() {
12         com.xxx.Api api = GetApi();
13         ApiControl.wrapApiFoo(api, ...);
14     }
15 };
```

Листинг 3: Демонстрация работы аспекта

В результате применения данной схемы реализации удается достигнуть модуляризации логики замены вызова функции на ее обертку, что избавляет кодовую базу от сквозных функциональностей, тем самым улучшая качество кода и стоимость его поддержки.

1.3 Рефлексия

Рефлексия - это способность системы или программы инспектировать и модифицировать собственную структуру во время исполнения. Для

языка `Java` это означает изменение поведения загруженных классов, интерфейсов, методов и полей во время исполнения. Данная возможность позволяет программному обеспечению динамически изменять собственное поведение в зависимости от параметров времени исполнения. Применение рефлексии в производстве облегчает разработку и дальнейшую поддержку программ, предоставляя средства для гибкой интеграции со сторонними кодовыми базами и для отделения логики конфигурирования поведения программы в зависимости от параметров окружения в во время исполнения.

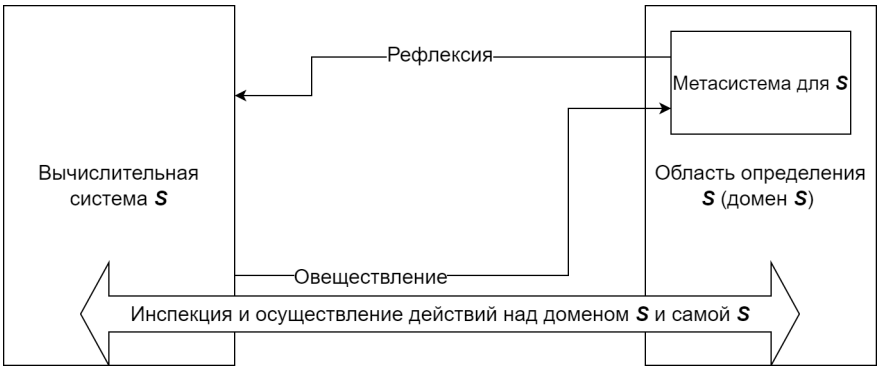

Рис. 3: Рефлексивная вычислительная система.

Как показано на рис. 3, в своем наиболее общем виде рефлексивная система может быть определена как система, удовлетворяющая следующим требованиям. Во-первых, система `S` должна иметь в своем домене собственное представление, называемое метасистемой. Это представление должно быть доступно для инспекции и манипуляции. Во-вторых, существует связь между системой `S` и ее представлением. Любое изменение в системе должно приводить к соотвествтуеющему изменению в метасистеме, и наоборот [10]. Тоесть система `S` должна быть овеществлена (reified) в свое представление прежде чем метасистема может начать оперировать. Затем метасистема инспектирует и манипулирует системой средствами рефлексии используя представление `S`. Данные связи отражены подписанными стрелками на рис. 3.

Несмотря на потенциальные плюсы полноценной рефлексии, многие решения, приедоставляемые стандартными библиотеками весьма ограни-

чивают пользовательля в возможности изменения объектов или же вовсе запрещают модификацию. Подобное решение связано с тем, что неконтролируемое испольование возможносетй рефлексии может привести к коду, о поведении которого очень сложно судить, что только ухудшает качество разработки. Однако существуют сценарии, в которых подобный подход наоборот приводит более прозрачной логике проекта. Примером такого сценария может служить задача отслеживания изменения состояния. Для достижения этой цели наиболее естественным решением будет является переопределение операции доступа к состоянию в рантайме, нежели чем написание отдельной программы, проходящейся по всем методам бинарного класс-файла и инструментирующей все операции доступа к состоянию. Помочь с имплементацией этого подхода может бинарная манипуляция. Существует множество фреймворков, использующих бинарную манипуляцию для имплементации расширений к языку. Примером подобного фреймворка может служить `Kava`, подробнее о которой будет рассказано в разд. 3.

1.4 Конфигурация и оптимизация

Бинарная манипуляция также широко применяется для имплементации пользовательских оптимизаций и конфигиурации параметров кода в зависимости от параметров сборки. Как правило, такие пользовательские настройки делаются при помощи препроцессора, который способен менять структуру приложения на уровне исходного кода. Однако многие высокоуровневые языки не обладают доступным пользователю препроцессором. В качестве альтернативного решения зачастую используются фреймворки бинарной манипуляции, позволяющие изменить результирующий бинарный код аналогично тому, как препроцессор меняет исходный. Примером подобной пользовательской настройки может служить Листинг 5

```
1 // Configuration class
2 public class Configuration {
3     public static Boolean debug = False;
4 }
5
```

```
// User written code
public static class MyClass {
    public static void foo() {
        if (Configuration.debug) {
            ...
        }
        ...
    }
}
```

Листинг 4: Пример использования класса с конфигурацией проекта

Для того, чтобы убрать из поставляемого пользователю бинарного файла все следы кода, используемого для отладки, авторы библиотек пишут свои собственные расширения к компилятору, использующие бинарную манипуляцию. Целью данного преобразования является удаления `true` ветки всех подобных условных операторов.

```
// After binary manipulations
public static class MyClass {
    public static void foo() {
        // ‘true‘ branch eliminated
        ...
    }
}
```

Листинг 5: Результат работы преобразования

Данный подход является очень дешевым и распространенным способом настройки поведения приложения, хоть и не дает всех плюсов, обеспечиваемых парадигмой АОП.

1.5 Актуальность

Выше была показана востребованность инструментов бинарной манипуляции. Действительно, многие бизнесс-логики больших и даже средних компаний построены на применении описанных техник, особенно в таких языках как `Java` и `Kotlin`, где подобные инструменты имеют долгую историю развития. Поэтому, чтобы оставаться конкурентноспособной, любая платформа, намеревающаяся выйти на рынок разработки мобильных при-

ложений, обязана предоставить свои средства для решения проблем интероперабельности компонент, поддержки принципов АОП и анализа программ посредством рефлексии или инспекции бинарных файлов. Более того, решение должно быть разработано с учетом всех особенностей платформы и предоставлять свои преимущества по сравнению с конкурирующими. В данной работе были рассмотрены дизайн и реализация библиотеки для манипуляции бинарными файлами многоязыковой платформы. Были рассмотрены особенности проектированя, связанные с необходимостью поддержки бинарных файлов, полученных из исходных файлов на разных языках, проведено сравнение с существующими решениями на других платформах, а также показано, как разработанный инструмент способен эффективно решать задачи, встречающиеся в реальных бизнесс-логиках.

2 Постановка задачи

Данная работа ставит своей целью предоставить возможность манипуляции бинарным файлом многоязыковой платформы. В данной работе будет рассмотен вопрос о представлении языковых сущностей в рамках библиотеки, а также о разработке интерфейса для их инспекции и изменения. Помимо этого, данная работа затронет вопрос интероперабельности между различными языками. Будет предложена архитектура обобщенного пользовательского интерфейса для инспекции и модификации зависимостей между сущностями из разных языков. Вопрос манипуляции бинарными инструкциями, а также графом инструкций будет затронут лишь поверхностно.

Имея в виду области применения техники бинарной манипуляции, а также поставленную цель, можно сформулировать следующие задачи, решаемые в данной работе:

- Спроектировать и разработать библиотеку для инспекции и манипуляции бинарными файлами платформы.
- В рамках данной библиотеки реализовать удобный интрефейс для написания сложных анализов тел функций.
- Предоставить единый интерфейс для использования интероперабельности между различными языками платформы.
- Предоставить общий интерфейс для работы с бинарными файлами полученными из разных языков.
- Поддержать возможность представления одного и того же языка разными кодировками бинарного файла, чтобы единообразно работать с бинарными файлами, получанными из одного исходного файла различными фронтендами.

Исходя из дальнейших планов развития платформы, к решению задачи были выдвинуты следующие требования:

- Имплементированная библиотека должна быть общего назначения, чтобы позже на ее основе имплементировать специализированные фреймворки и инструменты.

- Имплементированная библиотека должна иметь возможность быть обернутой в высокоуровневые обертки для различных языков, чтобы предоставить пользователям платформы удобный интерфейс для инспекции и манипуляции их бинарными файлами.

- Интерфейс для работы с телами функций должен скрывать от пользователя наличие служебных инструкций платформы, чтобы облегчить использование библиотеки.

- Интефрейс для работы с бинарным файлом должен скрывать от пользователя детали имплементации бинарного файла, чтобы уменьшить зависимость пользователя от версионирования формата файла.

Как видно из постановки задач и требований к ним, в данной работе не будут рассмотрены вопросы расширения средств языков или их стандартной библиотеки.

3 Обзор существующих решений

Одним из способов модификации программы является изменение ее семантики посредством рефлексии [11], однако многие языки, в том числе `Java` не предоставляют поддрежки для изменения семантики программы. Например, в `Java` класс `Class` объявлен `final`, что предотвращает специализацию данного класса с целью изменения семантики языковых механизмов, как это например возможно в `SmallTalk`. Исходя из этого, необходимо либо вносить изменения на уровне виртуальной машины, принося в жертву портабельность (как например делают мета-объектные протоколы (`Meta Object Protocol, MOP`), такие как `Metaxa` и `IguanaJ`), либо вносить изменения на уровне бинарного кода. Любые изменения исходного кода не рассматриваются, т.к. зачастую к нему нет доступа. Именно поэтому было сделано множество различных предложений по трансформации байткода, отличающихся по уровню абстракции сущностей, с которыми работает пользователь, по выразительной мощности и по гранулярности разрешенных преобразований.

В текущей главе будут рассмотрены существующие решения для решения задачи бинарной манипуляции, а также будет проведен анализ этих решений с точки зрения их области применения и архитектуры: какие из описанных во введении задач бинарной манипуляции они решают и каким образом интергируются в потребительские приложения.

3.1 java.lang.classfile

Пакет `java.lang.classfile` является нативной библиотекой для работы с бинарными класс-файлами в языке `Java` [12]. Мотивацией для разработки данного решения является необходимость периодически обновлять формат бинарного класс-файла. Поскольку до недавнего времени не существовало нативного решения для манипуляции бинарными файлами, разработчикам компонент приходилось сначала ждать обновления выбранного имим фреймворка для бинарной манипуляции прежде чем переходить

на новую версия файла. С введением `java.lang.classfile` разработчики стандартной библиотеки будут иметь возможность предостаивть интерфейс для раобты с новым форматом файла одновременно с выходом новой версии. Это позволит сторонним разработчикам инструментов и фреймворков автоматически поддерживать новейшую версию класс-файла и не зависеть от сторонних инструментов.

Отличительными особенностями `java.lang.classfile` являются следующие архитектурные решения. Во-первых, все сущности этого пакета иммутабельны. Для обновления какой либо из сущностей требуется создать новую. Во-вторых, аналогично уже существующим фреймвокам типа `ASM` [13], данное API имеет деревовидную структуру, что позволяет разработчикам при необходимости быстрее адаптировать свой код под нативное решение платформы. Помимо этого, данный пакет производит все вычисления лениво, тоесть тяжелые операции по типу чтения и парсинга производятся только для тех компонент, с которыми пользователь напрямую взаимодействует. Это позволяет улучшить производительность при загрузке больших классов, что в производстве встречается очень часто. Еще одной особенностью `java.lang.classfile` является то, что хоть данное API и работает на уровне абстракции бинарного файла, оно не дает возможности пользователю напрямую изменять поля, относящиеся к деталям имплементации виртуальной машины, такие как `Constant Pool`, `Stack Map` и другие, что позволяет сделать применение данного API более безопасным.

Ниже в Листинг 6 представлен пример использования данной библиотеки для модификации существующего класса, а именно удаление из него всех методов, начинающихся со слова "*debug*".

```
1  ClassModel classModel = ClassFile.of().parse(bytes);
2  byte[] newBytes = ClassFile.of().build(classModel.thisClass().asSymbol(),
3      classBuilder -> {
4          for (ClassElement ce : classModel) {
5              if (!(ce instanceof MethodModel mm
6                      && mm.methodName().stringValue().startsWith("debug"))) {
7                  classBuilder.with(ce);
```

```
8               }
9           }
10      });
```

Листинг 6: Удаление методов отладки из класса при помощи библиотеки `java.lang.classfile`

3.2 BIT

Архитектура фреймворка `BIT` основана на наблюдении, что для достижения большей части необходимых изменений достаточно инструментации только очень ограниченного количества ключевых локаций, таких как пролог и эпилог функции, начало и конец базового блока, а также до и после конкретной инструкции. Поэтому `BIT` предоставляет интерфейс для вызова методов в каждой из этих ключевых точек [14]. Как видно из описания, данный фреймворк работает с абстракциями бинарного файла, что дает возможность увеличить гранулярность инструментации.

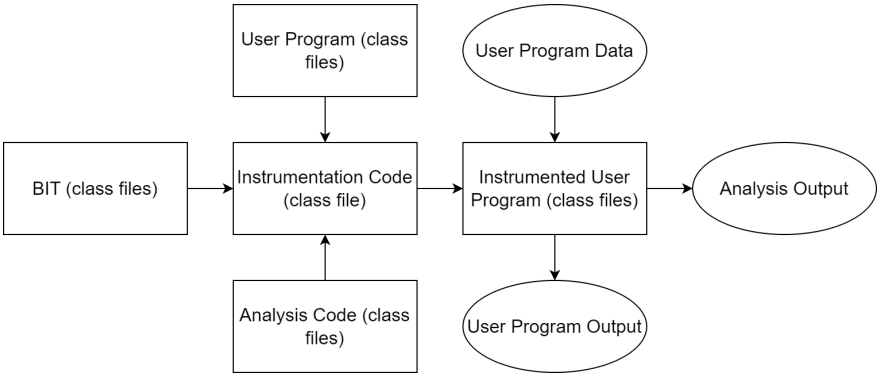

Рис. 4: Архитектура фреймворка `BIT`.

На рис. 4 проиллюстрирован процесс использования фреймворка `BIT`. Сначала пользователь пишет приложение, которое в последствии захочет инструментировать. Затем он пишет непосредственно код для инструментации и код для анализа бинарных файлов, используя классы и методы, предоставляемые фреймворком. Когда пользователь запустит код для инструментации, виртуальной машиной будут считаны бинарные файлы пользовательского приложения, после чего в нужные места пользовательского приложения будут вставлены вызовы кода для анализа. В результате

получится инструментированное приложение, которое может впоследствии быть исполнено виртуальной машиной Java. Исполнение инструментированного приложения порождает как вывод оригинальной пользовательской программы, так и вывод анализа. Вставленные вызовы к функциям анализа ни коим образом не влияют на семантику приложения, и его вывод должен остаться неизменным.

Таким образом, BIT - это инструмент для создания специализированных инструментов для наблюдения и анализа поведения программ во время исполнения. BIT позволяет пользователям добавлять производить анализ в любом месте бинарного кода, а также получать динамическую информацию о программе во время исполнения. Подобные возможности делают BIT хорошим вариантом для имплементации профилировщиков и других схожих инструментов.

Ниже приведен пример того, как выглядит инструментация на фреймворке BIT. В Листинг 7 и Листинг 8 показано, каким образом будет выглядеть код для подсчета условных переходов во время исполнения.

```
1   // filenameIn = ...; filenameOut = ...;
2   ClassInfo ci = new ClassInfo(filenameIn);
3   for (Enumeration e=ci.getRoutines().elements();e.hasMoreElements(); ){
4       Routine routine = (Routine) e.nextElement();
5       Vector instructions = routine.getInstructions();
6       for (Enumeration b = routine.getBasicBlocks().elements(); b.hasMoreElements(); ) {
7           BasicBlock bb = (BasicBlock) b.nextElement();
8           Instruction instr = (Instruction)instructions.elementAt(bb.getEndAddress());
9           short instr_type = InstructionTable.InstructionTypeTable[instr.getOpcode()];
10          if (instr_type == InstructionTable.CONDITIONAL_INSTRUCTION) {
11              instr.addBefore("BranchPrediction", "Offset", new Integer(instr.
                    getOrigOffset()));
12              instr.addBefore("BranchPrediction", "Branch", new String("BranchOutcome"));
13          }
14      }
15      String method = new String(routine.getMethod());
16      routine.addBefore("BranchPrediction", "EnterMethod", method);
17      routine.addAfter("BranchPrediction", "LeaveMethod", method);
18  }
```

```
19    ci.write(filenameOut);
```

Листинг 7: Подсчет условных переходов на BIT. Код инструментации

Заметим, что методы `addBefore` и `addAfter` добавляют вызовы к функциям, объявлены пользователем в коде анализа Листинг 8.

```
1   public BranchPrediction {
2       static Hashtable branch = null;
3       static int pc = 0;
4       public static void EnterMethod(String s) {
5           branch = new Hashtable();
6       }
7       public static void LeaveMethod(String s) {
8           System.out.println("stat for method: " + s);
9           for (Enumeration e = branch.keys(); e.hasMoreElements(); ) {
10              // Log results
11          }
12      }
13      public static void Offset(int offset) {
14          pc = offset;
15      }
16      public static void Branch(int brOutcome) {
17          Branch b = (Branch) branch.get(pc);
18          if (b == null)
19              b = new Branch();
20          if (brOutcome == 0)
21              b.taken++;
22          else
23              b.not_taken++;
24      }
25  }
```

Листинг 8: Подсчет условных переходов на BIT. Код анализа

3.3 BCA

Фреймворк BCA помогает упростить составление связей между объектами путем смещения многих важных решений (например таких как имена методов или подтипирование) с времени разработки на время интеграции компонент, тем самым позволяя программистам адаптировать и применять

даже сторонние библиотеки.

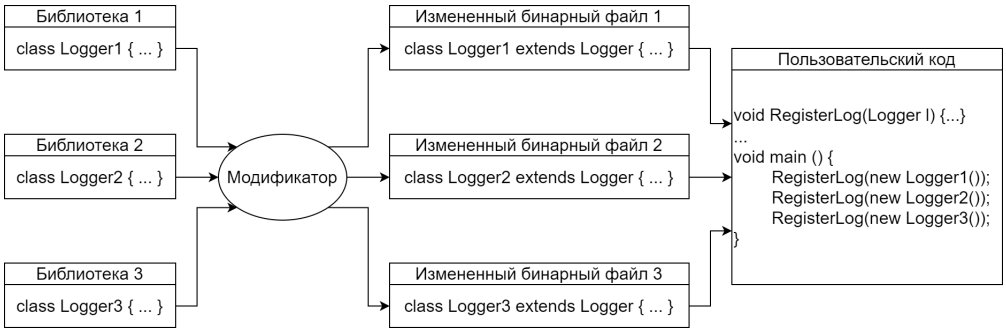

Рис. 5: Пример использования фреймворка BCA для достижения интероперабельности сторонних компонент.

BCA изменяет компоненты во время их загрузки в виртуальную машину. Адаптация компонент происходит уже после того, как они были получены программистом, и их внутренняя структура модифицируется напрямую по месту использования. Вместо того, чтобы писать классы-обертки, происходит непосредственная модификация оригинального класса. Путем изменений бинарных файлов BCA удается достигнуть гибкости изменения исходного кода без сопуствующих этому недостатков:

- Нет необходимости доступа к исходным файлам, что позволяет использовать фреймфорк даже на сторонних библиотеках
- Модификация может быть отложена до времени загрузки, что позволяет применять их по месту использования.
- В сравнении с модификацией исходного файла приводит к меньшему приросту времени загрузки.
- Модификация бинарного файла очень гибкая, и позволяет легко производить многие операции, такие как добавление нового метода, переименовывание, изменение подтипирования и иерархии классов.

Общая структура фреймворка BCA и его интеграция в виртуальную машину Java проиллюстрированы на рис. 6.

Когда загрузчик класс-файлов считывает бинарный файл, он сторит внутреннее представление для этого файла. Во время обычного исполнения

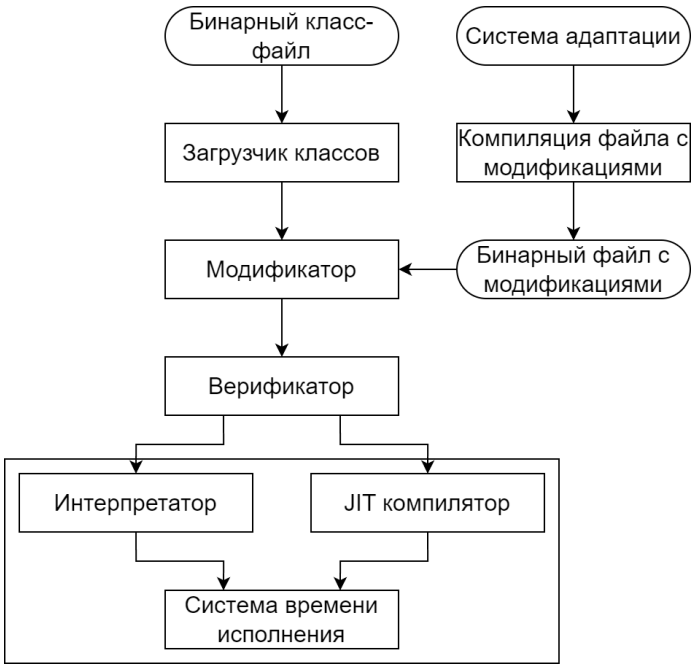

Рис. 6: Архитектура фреймворка BCA.

виртуальной машины Java это внутреннее представление сразу передается верификатору. BCA встраивает между этими двумя этапами, перенаправляя промежуточное представление модификатору, применяющему все необходимые трансформации к класс файлу. Все трансформации описаны в файле с модификациями (Delta File), который считывается модификатором при запуске виртуальной машины.

После модификации изменное представление класса передается верификатору, который проверяет код на соответствие спецификации JVM с целью выявления недопустимых операций. После успешной верификации промежуточное представление передается в часть JVM, отвечающую за время исполнения. BCA не требует никаких изменений ни в веривикатор JVM, ни в часть, отвечающую за время исполнения, что делает его применение более портабельным.

Чтобы гарантировать консистентность изменений, системе адаптации нужно иметь доступ ко всем файлам, которые могут быть потенциально затронуты изменениями. В случае, если весь проект хранится локально, то может быть произведена статическая адаптация компонент, в ходе которой BCA может просто считать все файлы и заранее применить к ним необходи-

мые изменения, получив при этом цельное измененное приложение. Статическая адаптация имеет недостаток в том, что ей требуется дополнительное место на хранение измененного приложения, что приводит к большему потреблению дискового пространства. Однако статическая адаптация полностью убирает какие либо дополнительные затраты при загрузке классов, т.к. вссе трансформации уже были применены. Более того, такая адаптация позволяет публиковать код без необходимости изменений среды на стороне конечного пользователя.

Альтернативой статической адаптации является динамическая адаптация. Динамическая адаптация работает по схеме, приведенной на рис. 6, тоесть производит все манипуляции во время исполнения. Такой подход приводит к дополнительным расходам во время загрузки, что приводит к увеличению общего времени исполнения. Также динамическая адаптация требует того, чтобы файлы с модификациями распространялись вместе с приложением, и конечный пользователь использовал систему с предустановленным `BCA`. С другой стороны, динамическая адаптация не требует дополнительного места на хранение модифицированной копии приложения. Более того, в данном подходе не требуется апиорное знание всвех методов, которые будут загружены в приложении, что позволяет поддержать более сложные сценарии с динамической загрузкой классов (например стандартный метод `Class.forName` позволяет искать класс по имени, производя динамическую загрузку, если до этого ее не было произведено. Поскольку аргумент данной функции - строка, то не всегда есть возможность статически вычислить, чему будет равно ее значение, и статической адаптации будет недостаточно [1]).

3.4 Kava

Примером фреймворка, расширящего стандартные средства рефлексии языка является `Kava` [15] [16]. Большинство фреймворков для манипуляции байткодом предоставляют объектно-ориентированное представление для элементов бинарного класс-файла таких как методы, типы, инструк-

ции и других. После чего пользователю предлагается написать отдельную программу, описывающую, как переписать класс файл. Главным недостатком данного подхода является то, что для того чтобы им возпользоваться, пользователь должен обладать знаниями о байткода и формате бинарного файла платформы. Альтернативныс является подход, оперирующий на языковом уровне, что несколько обезопасит и облегчит использование программистом бинарного кода. Поэтому `Kava` предлагает модель на основе поведенческой рефлексии, позволяющая изменять поведение приложения без необходимости изменения имплементации всей программы.

Для представления языковых сущностей `Kava` использует метаобъекты (`Meta Object Protocol`). При их помощи поведение сущностей во время исполнения, например доступ к полю, вызов метода и другие, могут быть переопределены. Метаобъекты конструируются при помощи овеществления (reification) рантайм модели объектов. Например, метод овеществляется как объект класса `Method`.

Рис. 7: Архитектура фреймворка `Kava`.

Как видно из рис. 8, Kava реализована при помощи инструментирования класс файла кодом, передающим контроль метаобъекту, ассоциированному с объектом. Для манипуляции самим байткодом Kava использует сторонний низкоуровневый фреймворк BCEL. При помощи него Kava инструментирует класс-файл во время его загрузки, что позволяет пользователю перехватывать и переопределять такие события как доступ к полю класса, вызов метода, вызов конструктора или выброс программой исключения.

Ниже в Листинг 9 представлен простой пример работы с Kava. В нем пользователь хочет перехватить исполнение метода, пожтому он переопределяет процесс исполнения функций платформой, исполняющей Java. Перед исполнением функции будут выведены ее имя и дополнительная информация.

```
1  public class TracingMetaObject implements MetaObject {
2      public ExecutionContext beforeExecuteMethod(ExecutionContext context) {
3          System.out.println("tracing " + context.getMethodName());
4      }
5  };
```

Листинг 9: Объявление метаобъекта в Kava

Для достижения желаемого результата, метаобъект должен быть привязан к классу, вызов методов которого необходимо отслеживать. Код, приведенный в Листинг 10, показывает как добиться отслеживания вызовов всех методов класса Test.

```
1  bind {
2      class Test metaclass-is TracingMetaObject {
3          any-method(any-parameters) {
4              execute;
5          }
6      }
7  }
```

Листинг 10: Привязка класса к метаобъекту в Kava

Для сравнения, имплементация эквивалентной логики при помощи классического фреймворка для работы с класс-файлами приведен ниже в

Листинг 11

```
1  public class TraceMethod implements Constants {
2      private static Method traceMethod(Method m) {
3          // create the byte code to insert
4          // find insertion point
5          // insert byte code at beginning of method
6          // recalculate stack size
7          // return modified method
8      }
9      public static void main(String[] argv) {
10         // parse the class
11         // get constant pool
12         // generate necessary constants
13         // foreach method:
14         // call traceMethod
15         // write out modified class
16     }
17 }
```

Листинг 11: Реализация аналогичной функциональности при помощи классического фреймворка для работы с класс-файлом

У данного подхода есть несколько недостатков в сравнении с реализацией через расширенный механизм рефлексии. Во первых, программист должен самостоятельно генерировать инструкции, которые необходимо вставить в бинарное представление методов, а также самостоятельно обходить класс-файл чтобы найти нужную точку вставки нового кода. Вдобавок, вставленный код может быть верифицирован только при повторной загрузке класс-файла. При помощи расширения механизма рефлексии бинарной манипуляцией, этих недостатков удается избежать.

3.5 AspectJ

`AspectJ` является расширением для языка `Java`, привнося в него поддержку элементов аспекто-ориентированного программирования. Данное расширение вносит в язык следующие понятия и сущности [17]:

- Точки соединения (`Join Points`) - однозначно определенные точки в исполнении программы.

- Срез (`Pointcut`) - ссылка на набор точек соединения и значения, определенные в них.
- Совет (`Advice`) - функционал, который определяет поведение в точке соединения.
- Аспект (`Aspect`) - модуль, реализующий сквозную функциональность. Аспект отвечает за применение некоего совета в точках соединения, которые определяются некоторым срезом

Далее будет приведен краткий обзор каждого из этих понятий.

Модель точки соединения является критичным элементом в любой аспекто-ориенттированной системе. Данная модель предоставляет общую "систему координат позволяющую координировать исполнение аспектов. `AspectJ` имплементирует модель, в которой точки соединения - это некоторые одозначно определенные точки в исполнении программы. В грубом приближении точки соединения в `AspectJ` можно представлять как узлы в графе потока управления. Получается, что поток управления проходит через каждую точку соединения дважды: первый раз на пути в поддерево, и второй раз - на обратном пути. `AspectJ` выделяет следующие виды точек соединения:

- Вызов метода или конструктора - данная точка соединения находится на стороне вызывающего объекта, сразу до вызова метода или конструктора.
- Принятие вызова метода или конструктора - данная точка соединения находится на стороне вызываемого объекта, и находится строго до вызова метода / конструктора.
- Исполнение метода или конструктора - данная точка соединения находится в вызываемом методе или конструкторе
- Чтение или запись поля

- Вызов обработчика исключений
- Инициализация класса
- Инициализация объекта

Срез, как уже было сказано, это набор точек соединения с, возможно, некоторыми значениями из контекста исполнения в этих точках соединения. `AspectJ` определяет несколько примитивных срезов, которые пользователь может комбинировать, чтобы создавать свои виды срезов. Например срез $receptions(voidPoint.setX(int))$ объединяет все точки соединения вида "Принятие вызова метода или конструктора в которых сигнатура принимаемого метода равна $voidPoint.setX(int)$. Срезы могут комбинироваться с помощью логических операторов, как показано ниже:

```
1   receptions(void Point.setX(int)) || receptions(void Point.setY(int))
```

Листинг 12: Срез, следящий за координатой точки `x` или `y`.

Пользователь также может определять собственные срезы:

```
1   pointcut moves():
2       receptions(void FigureElement.incrXY(int, int)) ||
3       receptions(void Line.setP1(Point)) ||
4       receptions(void Line.setP2(Point)) ||
5       receptions(void Point.setX(int)) ||
6       receptions(void Point.setY(int));
```

Листинг 13: Пользовательский срез, следящий за координатой точки `x` или `y`.

Совет - это функциональный механизм, использующийся для декларации логики, вызываемой в каждой из точек соединения в срезе. `AspectJ` определяет три вида советов - `before`, `after` и `around`. Совет определяется путем соотнесения куска кода с срезом и промежутком времени, относительно которого должен будет быть применен код в каждой из точек соединения. Например совет:

```
1   after(): moves() {
2       flag = true;
3   }
```

Листинг 14: Совет, отмечающий изменения точки `x` или `y`.

определяет совет в срезе `moves()`. Этот совет выставляет флаг каждый раз когда исполнение выйдет из среза `moves()`.

Аспекты - это модули, определяющие сквозную логику. Декларация аспекта ведет себя как декларация класса. Внутри него могут быть определены срезу, советы и другие декларации, допустимые внутри класса. Ниже приведен пример декларации аспекта, следящего за тем, была ли фигура передвинута.

```
1  aspect MoveTracking {
2      static boolean flag = false;
3      static boolean testAndClear() {
4          boolean result = flag;
5          flag = false;
6          return result;
7      }
8      pointcut moves():
9          receptions(void FigureElement.incrXY(int, int)) ||
10         receptions(void Line.setP1(Point)) ||
11         receptions(void Line.setP2(Point)) ||
12         receptions(void Point.setX(int)) ||
13         receptions(void Point.setY(int));
14     after(): moves() {
15         flag = true;
16     }
17 }
```

Листинг 15: Аспект, отслеживающий изменения фигур.

Советы аспекта чем-то похожи на методы в том смысле что имеют доступ к всем членам аспекта. В данном случае совет `after()` может спокойно работать со статическим полем `flag`.

Как видно из примеров, применение `AspectJ` позволяет имплементировать сквозную логику в виде отдельных модулей, тем самым явно выражая ее в одном месте. Благодаря этому, различные инструменты имеют возможность понимать сквозную логику проекта и давать программисту релевантные подсказки.

3.6 Javassist

Подобно `AspectJ`, `Javassist` является расширением языка `Java`, с той лишь разницей, что `Javassist` ставит своей целью расширить возможности встроенного функционала рефлексии. Данный вреймворк, как и `BCA`, может быть применен как статически, так и динамически. `Javassist` был разработать с целью упростить и безопасить процесс инструментации финарного файла и убрать необходимость рядовому разработчику проникать в детали работы виртуальной машины. Поэтому для имплементации этого фреймворка был выбран уровень абстракции исходного кода. Этот выбор приводит к уменьшению гибкости интерфейса, но с другой стороны позволяет сделать его более удобным для разработчика. Также в `Javassist` существует набор методов, позволяющих работать с байткодом напрямую. Помимо предоставления абстракций на уровне исходного кода, `Javassist` автоматически обновляет места использования модифицированных сущностей. Например, если имя класса было изменено, то оно будет изменено и в тех местах, где это имя используется.

Для работы с исходным кодом `Javassist` не использует полноценный java компилятор, поскольку полноценная компиляция `Java` кода довольно медленный процесс, и как правило, для работы функций фреймворка требуется скомпилировать только тело метода, не создавая при этом классов и другие вспомогательные структуры, которые присутствуют в бинарном класс-файле. Для обеспечения быстродействия компиляции, `Javassist` использует собственный компилятор, транслирующий только тело метода и вставляет его байткод в уже сущестующий метод.

Для обеспечения безопасности трансформаций, `Javassist` позволяет производить только ограниченный набор изменений. Например, в классе невозможно удалить поле или метод, можно только изменить тело уже существующего метода, или нельзя изменять сигнатуру существующего метода.

Приведем пример пользования данного фреймворка для инструмен-

тации кода. Рассмотрим пример, в котором пользователь выводит значения аргументов функциии. Пусть есть метод, описанный в Листинг 16.

```
1 public class Point {
2     int x, y;
3     void move(int dx, int dy) {
4         x += dx; y += dy;
5     }
6 }
```

Листинг 16: Исходный класс

Для вставки логгирующего кода, пользователь должен написать последовательность инструкций, описанную на Листинг 17.

```
1     // ...
2     ClassPool pool = ClassPool.getDefault();
3     CtClass cc = pool.get("Point");
4     CtMethod m = cc.getDeclaredMethod("move");
5     m.insertBefore("{ System.out.println($1); System.out.println($2); }");
6     cc.writeFile();
7     // ...
```

Листинг 17: Код, добавляющий логгирование

3.7 ASM

Аналогично BCA, ферймворк ASM [13] поддерживает как статический режим трансформации, так и динамический, являющийся для ASM основным режимом работы. Поэтому как имплементация фреймворка, так и его пользовательский интерфейс заточены под то, чтобы уменьшить дополнительные расходы на работу с класс-файлами во время загрузки.

ASM поддерживает два варианта пользовательского интерфейса: событийный и объектно-ориентированный. Рассмотрим каждый из них по отдельности.

Событийный пользовательский интерфейс является изначальным интерфейсом ASM и призван решить главную проблему объекто-ориентированного представления: создание большого числа объектов для репрезентации элементов класса при загрузке, что приводит к падению производительности

и увеличению расходов памяти виртуальной машины. Данный интерфейс реализован при помощи шаблона посетитель (`Visitor Design Pattern`). Применение данного шаблона позволяет изменять как бинарный код методов, так и структуру класса без необходимости создания объекта на каждую из сущностей, что позволяет `ASM` значительно уменьшить накладные расходы при загрузке классов.

Объектно-ориентированный интерфейс представляет класс в виде дерева объектов, в котором каждый объект осуществляет некую сущность принадлежащую классу, такую как сам класс, его поле, метод, и другие. Каждый объект содержит ссылки на объекты, осуществляющие принадлежащие ему сущности. Помимо этого, объектно-ориентированный интерфейс предоставляет функционал для преобразования древовидного представления в событийное и обратно. Другими словами, объектно-ориентированный интерфейс реализован поверх событийного интерфейса.

Каждый из этих интерфейсов имеет как свои преимущества, так и свои недостатка. Например, как было сказано выше, событийный интерфейс работает быстрее и требует меньше памяти, чем объектно-ориентированный. Однако такой интерфейс пригоден только для написания достаточно простых преобразований, поскольку в любой момент доступен только один из элементов класса, соответствующий текущему посещаемому элементу. Объектно-ориентированный интерфейс решает эту проблему, предоставляя пользователю доступ сразу ко всей модели класса, но ценой дополнительных расходов на построение дерева объектов.

Сравним эти два интерфейса на следующем примере. Пусть пользователю нужно создать код для метода в Листинг 18

```
1 public class MyClass {
2     int fld = 1;
3     public int getFld() {
4         return this.fld;
5     }
6 }
```

Листинг 18: Необходимый метод.

Используя событийный интерфейс, создание инструкций данного метода достигается следующей последовательностью вызовов Листинг 19.

```
1  mv.visitCode();
2  mv.visitVarInsn(ALOAD, 0);
3  mv.visitFieldInsn(GETFIELD, "pkg/MyClass", "fld", "I");
4  mv.visitInsn(IRETURN);
5  mv.visitMaxs(1, 1);
6  mv.visitEnd();
```

Листинг 19: Пример использования событийного интерфейса.

Ту же самую логику можно выразить при помощи объектно-ориентированно интерфейса Листинг 20.

```
1  MethodNode mn = new MethodNode(/* ... */);
2  InsnList il = mn.instructions;
3  il.add(new VarInsnNode(ILOAD, 1));
4  LabelNode label = new LabelNode();
5  il.add(new JumpInsnNode(IFLT, label));
6  il.add(new VarInsnNode(ALOAD, 0));
7  il.add(new VarInsnNode(ILOAD, 1));
8  // ...
```

Листинг 20: Пример использования объектно-ориентированного интерфейса.

Интерфейс для работы с инструкциями автоматически осуществляет поддержку таких служебных сущностей как constant pool. Проведение анализа потока управления осуществляется при помощи отдельной функциональности `ASM`, позволяющей строить графовое представление функций. Анализ же потока данных осуществляется при абстрактной интерпретации, позволяющей смоделировать все возможные пути исполнения инструкций в методе для получения всех возможных значений ее аргументов.

3.8 Анализ существующих решений

Все рассмотренные в данном разделе решения можно сравнивать по множеству признаков, таких как уровень абстракции сущностей, назначение фреймворка, требуется ли глубокое знание бинарного кода для работы с ним, и дргуим. В ходе данной работы такое сравнение было проведено, в

результате чего была получена следующая таблица:

Фреймворк	Абстракция	Требует знания байткода	Интеграция	Специализация
AspectJ	Язык	Нет	Расширение языка	Поддержка АОП
BIT	Бинарный файл	Нет	Библиотека	Вставка прологов и эпилогов методов
ASM	Бинарный файл	Да	Библиотека	Общего назначения
Javassist	Язык	Нет	Библиотека	Общего назначения
Kava	Язык	Нет	Расширение языка	Рефлексия
BCA	Язык	Нет	Библиотека	Модификация иерархии классов

Рис. 8: Сравнение рассмотренных фреймворков.

Изучив ее и требования, выдвинутые при построении задачи, был сделан вывод, что ни одно из существующих решений не выполняет в полной мере поставленные перед данной работой задачи, из-за чего требуется более детальное рассмотрение каждого из фреймворков по отдельности и выделение из его архитектуры необходимых нам аспектов.

4 Исследование и построение решения задачи

В предыдущей главе были рассмотрены различные решения для задачи бинарной манипуляции. На основе их анализа составим более детальный план решения задач, обозначенных в разд. 2.

Задача разработки библиотеки для инспекции и манипуляции бинарными файлами разбивается на две части: разработка интерфейса для работы с бинарными инструкциями и разработка интерфейса для работы с метаданными бинарного файла. как говорилось раньше, в данной работе подробно будет рассмотрена только вторая часть интерфейса, подробное описание которго будет в разд. 5.

Задача реализации интерфейса, удобного для написания сложных анализов тел функций сводится к выбору промежуточного представления для тел функций и написанию интерфейса, имплементирующего его. Отдельной задачей является проектировние и имплементация механизма, обеспечивающего выполнение требования скрывать от пользователя наличие служебных инструкций платформы. Данные задачи можно решать непосредственно, и подробный процесс будет описан далее в разд. 5.

Предоставление интерфейса для использования интреоперабельности между языками платформы является самодостаточной задачей, решение которой будет представлено в разд. 5.

Задача разработки общего интерфейса для бинарных файлов, полученных из исходных файлов написанных на разных языках, вместе с требованием поддержать возможность представления одного и того же языка разными кодировками бинарного файла, а также с требованием скрыть от пользователя детали имплементации бинарного файла говорит о том, что в разрабатываемой библиотеке должен использоваться уровень абстракции языковых сущностей. Таким образом, задача сводится к проектированию необходимых языковых абстракций и механизма овеществления различных кодировок в единое представление. Отдельной задачей является дизайн и имплементация интерфейса поверх выделенных абстракций. Для импле-

ментации решения данной задачи был полезен анализ фреймворка `BCA`, интерфейс которого позволяет пользователю работать напрямую с языковыми понятиями, такими как наследование, избегая необходимости вникать в детали бинарного файла. Также был полезен анализ фреймворка `ASM`, который показал удобство ограниченного применения шаблона посетителя для анализа метаданных бинарного файла.

5 Описание практической части

После того, как все большие задачи из разд. 2 были разбиты на подзадачи, которые можно рассматривать непосредственно, перейдем к описанию их решения.

Чтобы удовлетворить требованию предоставить возможность обертки библиотеки в языки более высокого уровня, для написания публичного интерфейса был выбран язык `C`. Главным плюсом этого языка в данном случае является наличие нативных интерфейсов для него у многих современных языков программирования. Примером таких интерфейсов может служить `JNI` (`Java Native Interface`) [18] у языка `Java` или `N-API` (`Node API`) [19] для языка `JavaScript`. Данные интерфейсы позволяют пользователю писать код, изменяющий параметры время исполнения. Например, `JNI` позволяет регистрировать функции, написанные на языке `C`, в качестве встроенных (`native`) функций платформы, которые в последствии можно вызывать из языка `Java`.

Поскольку для написания пользовательского интерфейса был выбран язык `C`, в котором нет возможности явно указывать приватные и публичные поля структур, то было принято решение имплементировать все сущности библиотеки как непрозрачные указатели. Эта техника позволяет разработчикам библиотеки не открывать пользователям деталей имплементации структур, вынуждая их использовать исключительно предоставленный библиотекой интерфейс. Необходимый эффект достигается путем вынесения определений структур в исходные файлы, что оставляет только декларации в заголовочных файлах. Таким образом, пользователь, подключивший себе заголовочный файл библиотеки, имеет только декларации библиотечных типов, в то время как имплементация библиотеки имеет доступ к полноценным определениям. Однако у данной техники есть один большой недостаток - отсутствие value-семантики у библиотечных типов в пользовательском интерфейсе. Действительно, поскольку пользователю недоступны определения типов, то пользовательский интерфейс может со-

держать только указатели на них. В рамках языка `C` это приводит к дополнительным вопросам про владение указаелем: кто аллоцировал память для объекта по указателю и чья обязанность ее освободить? В рамках данной библиотеки было решено проводить все аллокации строго на внутренних аренах библиотеки, и выдавать пользователю указатели на них. Это снимает с пользователя необходимость что-либо аллоцировать или освобождать на своей стороне для работы с интерфейсом библиотеки.

Общая архитектура библиотеки представлена на рис. 9.

Рис. 9: Общая схема работы библиотеки.

Входной точкой пользовательского интерфейса библиотеки является бинарный файл. Он может быть получен после компиляции пользователем кода при помощи любого из возможных фронтендов платформы. Во время загрузки библиотекой бинарного файла происходит построение промежуточного представления всех структур, овеществялющих языковые абстракции, закодированные в бинарном файле. Далее, пользователь имеет возможность работать с этим представлением посредством двух интерфейсов. Первый позволяет работать с метаданными файла, такими как типы, определения функций и другие. Второй интерфейс служит для работы с телами функций, а также бинарными инструкциями.

5.1 Фронтенд

Для начала кратко опишем архитектуру фронтенда многоязыковой платформы, в рамках которой разрабатывается описываемая библиотека.

Рассмотрим то, каким образом генерируется бинарный файл, какова его структура, и как он в дальнейшем исполняется.

Для того чтобы поддержать исполнение нескольких языков программирования в рамках одной плафтормы, необходимо решить задачу компиляции их исходных и бинарных файлов в представление, поддерживаемое платформой. С этой целью на платформе были имплементированы несколько фронтендов, каждый из которых компилирует исходный или бинарный входной файл в бинарный формат платформы. Благодаря единому бинарному формату в рамках платформы становится возможным написание таких инструментов, как описанный в данной работе, а также достигается бесшовная интероперация между различными языками.

Однако для эффективных компиляции и исполнения различных языков требуется разный по семантике набор бинарных инструкций. Это требование возникает из-за отличного поведения сущностей во время компиляции и во время исполнения. Проиллюстрируем это рассмотрением двух больших групп исполняемых на виртуальных машинах (`managed`) языков: статических и динамических. Примером статического `managed` языка могут являться `Java` или `Kotlin`. Данные языки статически типизированы, тоесть тип в них не является частью значения, поэтому например все обращения к полям объектов в них могут быть вычислены во время компиляции кода. Это позволяет компилятору и загрузчику осущесвтлять множество анализов и верифицировать код на предмет выполнения языковых гарантий, таких как связанные с модификаторами доступа сущностей. Для поддержки возможности верификации кода даже после компиляции исходных файлов требуется сохранение всей необходимой информации на уровне бинарных инструкций, что влияет на проектирование набора инструкций в виртуальных машинах, исполняющих эти языки программирования. Рассмотрим процесс обращения к полю класса в языке `Java`. Скомпилировав код, показанный в Листинг 21 компилятором `JVM` получим последовательность бинарных инструкций, показанную в Листинг 22.

```
1   public class MyClass {
```

```
2       public int a = 1;
3       public static int b = 2;
4   };
5   public class Main {
6       public static void main() {
7               MyClass my = new MyClass();
8               int res = my.a + MyClass.b;
9       }
10  }
```

Листинг 21: Исходный код языка Java.

```
1   public class MyClass
2   public int a;
3       descriptor: I
4       flags: ACC_PUBLIC
5   public static int b;
6       descriptor: I
7       flags: ACC_PUBLIC, ACC_STATIC
8
9   public class Main
10  Constant pool:
11      ...
12      #7 = Class #8 // MyClass
13      #8 = Utf8 MyClass
14      ...
15      #10 = Fieldref #7.#11 // MyClass.a:I
16      #11 = NameAndType #12:#13 // a:I
17      #12 = Utf8 a
18      #13 = Utf8 I
19      #14 = Fieldref #7.#15 // MyClass.b:I
20      #15 = NameAndType #16:#13 // b:I
21      #16 = Utf8 b
22      ...
23  public static void main();
24      ...
25      getfield #10 // Field MyClass.a:I
26      getstatic #14 // Field MyClass.b:I
27      iadd
28      ...
```

Листинг 22: Бинарный код виртуальной машины JVM.

Как видно из листинга бинарного кода Листинг 22, обращения к полям класса `MyClass` в методе `main` имеют всю необходимую типовую информацию, необходимую для проверок даже после компиляции. Действительно, поскольку аргументом инструкций доступа к полю является полный дескриптор необходимого поля, загрузчик имеет возможность найти это поле среди загружаемых классов и проверить все флаги доступа. Подобное проектирование инструкций позволяет имплементировать безопасное и надежное исполнение программ даже после применения к коду бинарных манипуляций.

Классическим примером динамического managed языка программирования является `JavaScript`. Поскольку этот язык динамический, тип в нем является частью значения переменной, и не может быть выведен фронтендом во время компиляции. Более того, из-за того, что объект в данном языке - это коллекция пар ключ-значение, из который можно как удалять, так и добавлять пары, то ни о какой верификации инструкций во время загрузки не может быть и речи. Более того, меняется семантика доступа к полю объекта, что находит свое отражение в наборе инструкций, которым оперируют виртуальные машины, исполняющие данный язык. Проиллюстрируем отличия в семантике доступа к полю объекта на примере бинарного кода виртуальной машины машины `V8`. Написав код, аналогичный Листинг 21, получим следующий результат.

```
1  class MyClass {
2      static a = 1;
3      b = 2;
4  }
5  function main() {
6      let my = new MyClass();
7      let res = my.b + MyClass.a
8  }
9  main()
```

Листинг 23: Исходный код языка `JavaScript`.

```
1  [generated bytecode for function: main (0x0e93e5d98a29 <SharedFunctionInfo main>)]
2  Constant pool
```

```
3       ...
4       0: ... <String[7]: #MyClass>
5       1: ... <String[1]: #b>
6       2: ... <String[1]: #a>
7       ...
8   Bytecode
9       ...
10      ThrowReferenceErrorIfHole [0]    // #MyClass
11      Star2
12      Construct r2, r0-r0, [0]
13      Star0
14      GetNamedProperty r0, [1], [3]    // #b
15      Star2
16      LdaImmutableCurrentContextSlot [2]
17      ThrowReferenceErrorIfHole [0]    // #MyClass
18      Star3
19      GetNamedProperty r3, [2], [5]    // #a
20      Add r2, [2]
21      ...
```

Листинг 24: Бинарный код виртуальной машины `V8`.

Как видно из Листинг 24, в динамическом языке совершенно другая семантика обращения к полю. Во-первых, обращение к статическому и к динамическому полю выражается единой инструкцией - `GetNamedProperty`. Во-вторых, никакой информации, которая могла бы быть полезна при верификации не может быть сохранено и не сохраняется в виду модели объекта в данном языке. Также, поскольку в динамических языках тип является свойством значения, в общем случае фронтенд не может вывести, что складываются два целочисленных значения. Поэтому в отличие от `Java`, где использовалась специализированная инструкция `addi`, в Листинг 24 используется общая операция `Add`.

В виду этих и еще многих различий семантик исполнения различных языков, на платформе имплементировано несколько расширяемых наборов инструкций (`ISA`), каждый из которых исполняется на одной из виртуальных машин платформы в зависимости от исходного языка приложения.

Поскольку приложения могут быть написаны на нескольких языках

программирования, то после линковки всех бинарных файлов приложения может оказаться так, что в рамках одного бинарного файла будет присутствовать бинарное представление сущностей из разных языков.

Это является большой проблемой для пользователей, пытающихся прочитать и проинтерпретировать бинарный файл, поскольку разные фронтенды могут по-разному кодировать необходимые ему языковые сущности в рамках одного бинарного формата. Так например фронтенду для виртуальной машины, имплементирующей спецификацию языка `JavaScript ECMA-262` нет смысла кодировать пользовательский класс никак иначе, кроме как функцию, в то время как фронтенду языка `Java` нет смысла кодирвать класс иначе как структуру с фиксированными разметкой и модификаторами доступа. Аналогично, фронтенду языка `JavaScript` нет необходимости сохранять информацию о типах или о сигнатурах функций, но для исполнения языка `Java` эта информация является критичной. Кроме того, некоторые фронетнды могут использовать структуры бинарных файлов для хранения имплементационных деталей, не имеющих прямого отображения на исходный код. Таким образом мы получаем ситуацию, когда каждая сущность в бинарном файле может быть интерпретирована по-разному в зависимости исходного языка и использованного фронтенда. Отсюда возникает требование скрыть такие сложные детали имплементации бинарного формата от пользователя.

Далее будут отдельно рассмотрены две части пользовательского интерфейса: интерфейс для работы с метаданными бинарного файла и интерфейс для работы с бинарными инструкциями. При рассмотрении каждого из них будет представлена его архитектура, а также то, как ее имплементация решает обговоренные выше подзадачи. Подробно в данной работе будет рассмотрен только интерфейс для работы с метаданными файла.

5.2 Интерфейс для работы с метаданными бинарного файла

Как уже было сказано ранее, для имплементации интерфейса для работы с метаданными бинарного файла был выбран уровень абстрак-

ции языковых сущностей. Данное решение было продиктовано желанием скрыть от пользователей платформы сложное устройство бинарного файла, а также убрать необходимость пользователю разбираться в деталях имплементации различных фронтендов платформы.

Однако данное решение требует аккуратного выбора набора языковых сущностей, с которыми будет работать библиотека, поскольку их должно быть достаточно чтобы выразить все необходимые языковые абстракции поддерживаемых языков, но при этом они должны быть достаточно общими, чтобы не создавать ситуации, когда пользователь путается, какую языковую сущность ему необходимо применить в том или ином случае.

Для выделения таких сущностей было проведено сравнение распространенных managed языков программирования. Сравнение проводилось в два этапа: сначала языки сравнивались по набору поддерживаемых языковых сущностей, затем сравнивалась их семантика: какие операции к ним применимы, а какие - нет.

В результате сравнения языков `JavaScript`, `TypeScript`, `Java`, `Kotlin` и нативного статического языка платформы был выделен следующий набор сущностей, достаточный для того, чтобы оперировать этими языками.

- Класс (`Class`) - общее обозначение для всех структур данных.
- Функция (`Function`) - общее обозначение для всех видов функций. Анонимных функций, методов, свободных функций и дргуих.
- Поле (`Field`) - общее обозначение для полей чего-либо. Класса, интерфейса, пространства имен и других.
- Перечисление (`Enum`) - обозначение для типов перечисления.
- Пространство имен (`Namespace`) - обознчание для пространств имен.
- Аннотация (`Annotation`) и Декоратор (`Decorator`) - являются взаимоисключающими понятиями, поскольку обозначают статический и динамический механизмы добавления метаданных к декларациям.

Поскольку, как уже было обозначено ранее, один бинарный файл может содержать сущности из нескольких разных языков, в рамках библиотеки было решено адаптировать следующую модель бинарного файла. Бинарный файл - это коллекция сущностей, называемых модуль (`Module`). Модуль же - это именованная коллекция языко-специфичных деклараций, у которой есть свой публичный интерфейс, и свой список внешних зависимостей. Поскольку каждый язык имеет собственный механизм работы с внешними зависимостями, а также способ обозначения публичного интерфейса, было решено ввести абстракиции, обобщающие понятия внешней зависимости и сущности из публичного интерфейса модуля. Ими являются дескриптор импорта `Import Descriptor` и дескриптор экспорта `Export Descriptor`. Целью введения этих дескрипторов является отделение понятия внешней зависимости и публичного интерфейса от языко-специфичной нагрузки и предоставление некоего общего интерфейса для работы с ними, оставив при этом возможность работать с ними в контексте конкретного языка. С помощью понятий модуль, дескриптор импорта и дескриптор экспорта возможно описать общий механизм отношений между модулями и сущнстями из модулей, что позволяет библиотеке единообразно обрабатывать сценарии интероперабельности и сценарии работы с внешними зависимостями в рамках одного языка. Данный механизм является строго необходимым, поскольку различия в механизмах взаимодействия с внешними сущностями у различных языков во время компиляции и во время исполнения могут радикально отличаться. Так, например, для языка `Java` каждая импортируемая сущность может быть разрешена в класс, метод или поле класса во время компиляции проекта. После чего данные сущности могут быть сохранены в бинарном файле с пометкой `external`. Во время исполнения программы будут загружены класс-файлы с их определениями и код будет иметь возможность быть исполеннным. Из-за такой имплементации работы с внешними сущностями, на инструкций работа с внешним и локальным классом абсолютно не отличается. Однако для тако-

го языка как JavaScript ни о каком разрешении имен до начала времени исполнения речи идти не может, поэтому вся информация о внешних зависимостях и публичном интерфейсе согласно спецификации ECMA-262 хранится в отдельной для каждого модуля таблице. Данный подход очевидно отражается на наборе инструкций, с которыми работает виртуальная машина, имплементирующая данную спецификацию. Среди них появляются инструкции, работающие непосредственно с вхождениями в этой таблице. Целью введения понятий дескриптора импорта и дескриптора экспорта как раз и является сокрытие подобных деталей имплементации от пользователя и вынесение понятий внешней зависимости и публичного интерфейса модуля в языко-независимую чать библиотеки, чтобы упростить интерфейс работы с интероперабельностью.

Также, чтобы отразить факт того, что поддерживаемые платформой языки будут скомпилированы в один из ограниченного числа наборов инструкций, поддерживаемых виртуальными машинами платформы, было введено понятие Target, соединяющее в себе как исходный язык, так и спользуемый при его компиляции набор инструкций. Например один и тот же код на языке Java может быть представлен совершенно разными наборами инструкций на разных платформах, таких как JVM и ART. В рамках нашей библиотеки этот факт отразился бы в определении двух Target-ов для языка Java: lib_Core_Target_Java_ART и lib_Core_Target_Java_JVM.

Анализ семантики выделенных выше языковых сущностей показал, что функционал, применимый к данным сущностям можно разбить на две основные группы: общий и языко-специфичный.

Общий функционал над сущностями библиотеки имеет одно и то же семантическое значение для всех языков. Например взятие имени у функции или у класса. Также, в общий функционал входят те функции, для которых можно однозначно определить поведение во всех языках. В них входят функции-перечислители, принимающие функцию-посетитель в качестве аргумента (как например функция ClassEnumerateFields или

`ClassEnumerateMethods`). Для языков, в которых данные операции имеют смысл (например `Java`), происходит вызов функции-посетителя, а для языков, в которых операции смысла не имеют (например `JavaScript`), просто ничего не происходит, и функция-посетитель не вызывается ни разу.

Таким образом, основная часть пользовательского интерфейса состоит из следующих сущностей:

- Объявления всех библиотечных типов. В эти объявления входят как служебные типы, такие как тип, строка и тп, так и обозначенные выше языковые сущности. Всем подобным типам соответствуют имена вида `lib_Core_XXX`, где `Core` означает принадлежность к основной части интерфейса, а `XXX` заменяется на имя сущности, описываемой типом, например `lib_Core_Class` или `lib_Core_Type`.
- Все функции-перечислители для объявленных типов. Поскольку поведение таких функций было однозначно определено выше для всех возможных языков, они могут стать частью общего интерфейса.
- Все функции-геттеры, для которых возможно определить однозначеное поведение для всех возможных языков.
- Интерфейс для модификации всех языко-независимых сущностей: `lib_Core_I` `lib_Core_String`, `lib_Core_Type`, `lib_Core_Value` и других.

Задачей языко-специфичного интерфейса является определить, какие из основных сущностей релевантны для конкретного языка, а также определить функционал, применимый только к ним. Языко-специфичный интерфейс состоит из множества языковых расширений, каждое из которых определяет специфику отдельного языка. Каждое расширение определяет свой набор `lib_<LANG>_XXX` сущностей, специализирующих `lib_Core_XXX` сущности из основной части библиотеки. Каждое из расширений определяет только те сущности, которые ему нужны. Так, например, расширению `JavaScript` не нужно будет определять свой `lib_JavaScript_Field`, поскольку понятие поля в данном расширении не имеет смысла. Таким обра-

зом, внутренняя имплементация структуры `lib_Core_Class` будет выглядеть следующим образом:

```
1  struct lib_Core_Class {
2      /* Used to determiine correct type of ‘impl‘. */
3      lib_Core_Module* m;
4      /* ... */
5      union {
6          lib_JS_Class* JS;
7          lib_Java_Class* Java;
8          /* ... */
9      } impl;
10 };
```

Листинг 25: Внутренняя имплементация типа `lib_Core_Class`

Расширения, как уже было сказано выше, определяют языко-специфичный функционал только над специализированными сущностями из основной части интерфейса. Каждое расширение имплементирует языковой интерфейс для одного или более `Target`-ов, поскольку языковые метаданные определяются исходным языком, а не выбранным набором инструкций. Для работы с функционалом расширения пользователям необходимы функции-конверторы между сущностями основной части и сущностями расширения:

```
1  struct lib_Java_Api {
2      /* ... */
3      CoreClassGetJavaClass, // Converts lib_Core_Class* to lib_Java_Class*
4      JavaClassGetCoreClass, // Converts lib_Java_Class* to lib_Core_Class*
5      /* ... */
6  };
```

Листинг 26: Пример функций-конверторов.

Поскольку большая часть модификаций сущностей требует языко-специфичного набора параметров, в рамках библиотеки было решено вынести весь интерфейс для модификации языковых сущностей в расширения, где модификации будут определены для канкретных специализаций `lib_<LANG>_XXX`.

Также, языковые расширения содержат методы для инспекции метаданных всех `lib_<LANG>_XXX` сущностей, поскольку большая часть мета-

данных является языко-специфичной, и для функций-геттеров невозможно определить общего поведения. Функции-геттеры, вынесенные в основную часть не дублируются в расширении.

Поскольку механизм работы с внешними сущностями определяется парой языков (экспортируемый-экспортирующий или импортируемый-импортирующий), каждое расширение должно предоставить собственный интерфейс для работы с импортами и экспортами. Этот интерфейс должен включать в себя добавление и удаление внешних зависимостей, а также добавление и удаление сущностей из публичного интерфейса модуля. Еще одной важной частью интерфейса является извлечение из дескрипторов языко-специфичных данных. Так, для языка Java мы можем извлечь из дескриптора импорта `lib_Java_Class`, `lib_Java_Field`, `lib_Java_Function`. Тем самым, каждое расширение определяет возможности своего языка к интероперабельнсти, задавая пары языков, между которыми возможно построение отношений экспорт-импорт или импорт-экспорт. Это позволяет определять правила преобразований между различными механизмами работы с внешними сущностями в частном порядке для каждой конкретной пары.

Резюмируя сказанное выше, роль языковых расширений состоит в следующем: основная часть пользовательского интерфейса для работы с метаданными предоставляет общий интерфейс, позволяющий писать общий код для работы с любым исходным языком; этот интерфейс строится поверх внутренних и публичных функций расширений. Расширения предоставляют языко-специфичный функционал, работающий только с сущностями из расширения.

Ниже представлен пример того, как может выглядеть расширение для языка `TypeScript`.

```
1  /* Declaring language-specific entities. */
2  struct lib_TS_Module;
3  struct lib_TS_Class;
4  struct lib_TS_Function;
5  struct lib_TS_Decorator;
```

```
6   struct lib_TS_Namespace;
7   /* etc. */
8
9   struct lib_Api_TS
10  {
11      /* lib_TS_Module */
12      /*
13       * Note, how pair AddImport/AddExport defines,
14       * which languages are supported for interop with extension's language.
15       */
16      ModuleAddImportFromJS(lib_TS_Module* importing, lib_JS_Module* imported, ...
            payload);
17      ModuleAddExportFromJS(lib_TS_Module* exporting, lib_JS_Module* exported, ...
            payload);
18      ModuleAddImportFromTS(lib_TS_Module* importing, lib_TS_Module* imported, ...
            payload);
19      ModuleAddExportFromTS(lib_TS_Module* exporting, lib_TS_Module* exported, ...
            payload);
20      /* etc. */
21      ModuleRemoveImport(lib_TS_Module* m, lib_Core_ImportDescriptor* id);
22      ModuleRemoveExport(lib_TS_Module* m, lib_Core_ExportDescriptor* ed);
23      TSModuleGetCoreModule(lib_TS_Module* m),
24      CoreModuleGetTSModule(lib_Core_Module* m),
25
26      /* lib_TS_Namespace */
27      TSNamespaceGetConstructorFunction(lib_TS_Namespace* ns),
28      TSNamespaceGetCoreNamespace(lib_TS_Namespace* ns),
29      CoreNamespaceGetTSNamespace(lib_Core_Namespace* ns),
30
31      /* lib_TS_Enum */
32      /* lib_TS_Class */
33      /* lib_TS_Function */
34      /* etc. */
35  };
```

Листинг 27: Примерный код расширения `Typescript`.

Комбинируя использование функций из общего интерфейса и функций из языко-специфичных расширений, пользователь может писать безопасную общкю логику, расширяемую по необходимости языко-специфичными

действиями. Ниже приведен пример такого кода. Пусть перед пользователем стоит задача обойти все функции в бинарном файле. Он знает, что в языке `TypeScript` пространство имен представлено немедленно-вызываемой функцией `IIFE (Immediately Invoked Function Expression)`, которую пользователь также хочет обработать как обычную функцию. Данную логику можно выразить следующим образом:

```
1   static bool NamespaceEnumerator(lib_Core_Namespace* ns, void* data) {
2       /* ... */
3
4       /*
5        * I, as user, know that TS namespace can contain expressions,
6        * and this logic is not present in the ‘Core‘ part. So language-specific
7        * behaviour is needed in order to process namespace’s constructor.
8        */
9       lib_Core_Module* m = api_core->NamespaceGetModule(ns)
10      if (lib_Core_Target_TS == api_core->ModuleGetTartget(m)) {
11          auto ts_namespace = api_ts->CoreNamespaceGetTSNamespace(ns);
12          auto ts_ns_ctor = api_ts->NamespaceGetConstructor(ts_namespace);
13          auto ns_ctor = api_ts->TSFunctionGetCoreFunction(ts_ns_ctor);
14          (void)FunctionEnumerator(ns_ctor, data);
15      }
16
17      /* ... */
18  }
```

5.3 Интерфейс для работы с бинарными инструкциями

Как уже было сказано ранее, в рамках рассматриваемой платформы имплементировано несколько наборов инструкций, поддерживаемых различными виртуальными машинами. Все языки программирования, поддерживаемые на платформе транслируются в один из этих наборов инструкций в зависимости от используемого фронтенда. Каждый из имеющихся на платформе наборов инструкций является регистровым, с одним выделенным регистром, называемым аккумулятором, использующимся как неявный регистр для сохранения возвращаемого значения. Данный дизайн инструкций позволяет уменьшить размер инструкций за счет того, что нет

необходимости кодировать регистр для возвращаемого значения, что приводит к уменьшению размера кода вцелом. Операндами инструкций могут являться как регистры, так и конкретные (`immediate`) значения.

Поскольку одним из требований к имплементации рассматриваемой библиотеки является удобство имплементации сложных анализов тел функций, что сводится к предоставлению пользователям механизмов для анализа потока управления и потока данных, в рамках библиотеки не было предложено интерфейса для работы с непосредственным представлением инструкций как они хранятся в бинарном файле. Действиетльно, для того, чтобы провести анализ потока управления или данными по последовательности регистровых инструкций пользователю придется писать на своей стороне весьма сложные анализы. Более того, если пользователь захочет добавить свою собственную инструкцию в последовательность, то ему придется писать свой алгоритм аллокации регистров, чтобы избежать конфликтов с уже используемыми. Чтобы избежать данных затруднений, было предложено переиспользовать компилятор платформы для предоставления пользовательского интерфейса для работы с телами функций. Данный компилятор является очень гибким, и уже успешно переиспользуется в имплементации `JIT`, `AOT` компиляторов, а также в имплементации оптимизатора бинарного кода.

Компилятор преобразует последовательность инструкций бинарного файла в `SSA` граф из них, попутно добавляя необходимые компилятору служебные инструкции. `SSA` граф является одинм из лучших представлений для анализа тел функций, поскольку анализ потока управления и анализ потока данных получаются бесплатно "из коробки и именно на данном представлении проводится большинство описанных в литературе анализов. Еще одним преимуществом переиспользования компилятора является простота поддержки компиляторных проходов (например таких как построение дерева доминаторов) в пользовательском интерфейсе. Поскольку все они уже имплементированы для внутреннних нужд компилятора, библио-

теке остается только написать безопасную для пользователя обертку.

Для отделения языко-независимой структуры графа от конкретных наборов инструкций, публичный интерфейс для работы с бинарными инструкциями был разделен на две основные части: интерфейс набора инструкций и интерфейс графа инструкций.

5.3.1 Интерфейс графа инструкций

Данный интерфейс является оберткой над интерфейсом компилятора. Как уже было сказано выше, данный интерфейс работает с графом инструкцйи в `SSA` форме. Остановимся на этотм моменте более подробно. `SSA` (`Single Static Assignment`) - вид промежуточного представления кода функции. основной особенностью `SSA` формы является то, что присвоение в каждую переменную в ней происходит лишь один раз, что значитеотно упрощает анализ потока данных. Еще дним важным свойством, присущим всем разновидностям SSA, вклю- чая приведенное выше определение, является ссылочная прозрачность: поскольку для каждой переменной в тексте программы существует только одно определение, значение переменной не зависит от ее положения в программе.

Граф инструкций в `SSA` форме состоит из базовых блоков, которые образуют граф потока управления. Базовый блок - это линейный участок инструкций, в который можно попасть только через самую первую инструкцию, и не имеющий инструкций условного перехода, кроме самой последней Наличие понятия базового блока заметно упрощает анализ поттока управления функции.

Как видно из определения, данный граф не имеет привязки к какому-либо конкретному набору инструкций, поэтому интерфейс графа инструкций весьма прост:

- Объявляет сущности `lib_Core_Graph`, `lib_Core_BasicBlock`, `lib_Core_Inst`
- Предоставляет функции для работы ними. Интерфейс для `lib_Core_Instruc` не содержит языковой специфики, состоит только из функций для ин-

спекции и манипуляции входами (`inputs`) и пользователями (`users`) инстукций. По тем же причинам не включает в себя интерфейс для создания экземпляров инструкций, поскольку данные функции являются зависимыми от используемого набора инструкций.

5.3.2 Интерфейс набора инструкций

Для определения возможных наборов инструкций был выделен отдельный интерфейс для работы с ними. Данный интерфейс определяет набор инструкций как список функций-конструкторов, а также функций-геттеров и -сеттеров для получившихся экземпляров `lib_Core_Instruction`. Подобное разделение позволяет предоставить более понятный пользовательский интерфейс, а также изолировать проверки на то, что добавляемая инструкция соответствует набору инструкций, использующихся в графе в едином месте в имплементации.

Каждый из наборов инстукций в виду их специфики может представлять лишь ограниченное число языков, в связи с чем за каждым из интерфейсов для набора инструкций закреплен определенный и фиксированный список `Target`-ов, для которых он применим. Так, например, набор инструкций виртуальной машины, имплементирующей спецификацию `ECMA-262`, очевидно может быть применен для кодирования тел функций языков `JavaScript` и `TypeScript`. Чтобы сделать данную логику прозрачной для пользователя, в библиотеку были внесены функции, позволяющие опрашивать граф, каким набором инструкций он закодирован.

```
1  /* include/graph_core.h */
2
3  enum lib_Core_ISA {
4      lib_Core_ISA_ISA1,
5      lib_Core_ISA_ISA2,
6      /* ... */
7  };
8
9  struct lib_Core_GraphApi {
10     /* ... */
```

```
11      GraphGetTarget, // returns lib_Core_Target
12      GraphGetIsaType,   // returns lib_Core_ISA
13      /* ... */
14  };
```

Данный функционал позволяет пользователям писать общий код для работы с графом, используя `Target`-специфичный интерфейс только по необходимости.

Таким образом, заголовочные файлы интерфейса для работы с набором инструкций выглядят примерно так:

```
1   /* include/isa/isa_1.h */
2
3   enum lib_Core_ISA1_Opcode {
4       lib_Core_ISA1_Opcode_XXX,
5       lib_Core_ISA1_Opcode_YYY,
6       /* ... */
7   };
8
9   struct lib_Core_ISA1_Api {
10      IsaGetIsaType, // returns lib_Core_ISA
11
12      InstCreateXXX,
13      InstCreateYYY,
14      /* ... */
15
16      InstGetClass,
17      InstGetString,
18      /* ... */
19
20      instSetClass,
21      InstSetString,
22      /* ... */
23  };
24
25  /* include/isa/isa_2.h */
26
27  enum lib_Core_ISA2_Opcode {
28      lib_Core_ISA2_Opcode_AAA,
29      lib_Core_ISA2_Opcode_BBB,
```

```
30      /* ... */
31  };
32
33  struct lib_Core_ISA2_Api {
34      IsaGetIsaType, // returns lib_Core_ISA
35
36      InstCreateAAA,
37      InstCreateBBB,
38      /* ... */
39
40      InstGetClass,
41      InstGetField,
42      /* ... */
43
44      instSetClass,
45      InstSetField,
46      /* ... */
47  };
```

Заметим, что доступные наборы функций-геттеров и -сеттеров определяются спецификой конкретной виртуальной машины и ее набора инструкций.

5.4 Примеры использования

Для того чтобы показать практическую применимость разработанной библиотеки, рассмотрим, как некоторые из производственных задач могут быть решены с ее помощью.

5.4.1 Имплементация логгирующего аспекта

Постановка задачи: распространенным сценарием в промышленной `Java` разработке является добавление прологов или эпилогов в существующие функции, поэтому разрабатанная библиотека должна позволять пользователям делать это. Проиллюстрируем это на примере добавления логгирующих прологов и эпилогов к функциям. Пусть имеется следующий исходный код:

```
1   class MyClass {
```

```
2       foo() {
3           /* business logic */
4       }
5   }
```

Ожидаемый результат: Требуется, имея бинарное представление исходного кода, преобразовать его таким образом, чтобы его исполнение имело эффект, аналогичный коду ниже.

```
1   class MyClass {
2       foo() {
3           console.log("File: /path/to/file")
4           console.log("Function: foo")
5           const start = new Date().getTime();
6           /* business logic */
7           const end = new Date().getTime();
8           console.log("Elaplsed time: ", end - start);
9       }
10  }
```

Имплементация при помощи библиотеки: В рамках описанной библиотеки данная логика выражается последовательностью инструкций, представленной ниже. В данном примере будет подробно рассмотрен процесс инспекции метаданных, вплоть до непосредственного изменения инструкций метода. Процесс изменения графа будет рассмотрен в следующем примере.

```
1   static int main() {
2       /* Open file */
3       lib_Core_File *file = api->OpenBinary(input);
4       /* Make transformations */
5       api_core->FileEnumerateModules(file, void, ModuleEnumerator);
6       /* Write output */
7       api->WriteBinary(file, output);
8   }
9
10  static bool ModuleEnumerator(lib_Core_Module* m, void* data) {
11      api_core->ModuleEnumerateClasses(m, data, ClassEnumerator);
12      api_core->ModuleEnumerateTopLevelFunctions(m, data, FunctionEnumerator)
13      return true;
14  }
```

```
15
16  static bool ClassEnumerator(lib_Core_Class* c, void* data) {
17      api_core->ClassEnumerateMethods(c, data, FunctionEnumerator);
18      return true;
19  }
20
21  static bool FunctionEnumerator(lib_Core_Function* f, void* data) {
22      if (NeedToLogMethod(f)) {
23          api->TransformMethod(f, void, AddLogToMethod);
24      }
25      return true;
26  }
27
28  static void AddLogToMethod(lib_Core_ModificationContext *ctxM, lib_Core_Function *f,
        void *data) {
29      auto oldCode = api_core->MethodGetCode(method);
30      lib_Core_Graph *g = api->codeToGraph(ctxI, oldCode);
31      auto isa = api_g->GraphGetIsaType(g);
32
33      /* Work with an exact ISA will be demonstrated in next example */
34      if (api_isa1->IsaGetIsaType() == isa) {
35          /* ... */
36      } else if (/* process other isas */) {
37          /* ... */
38      } else {
39          assert(false);
40      }
41
42      lib_Code *newCode = api->graphToCode(g);
43      api_core->MethodSetCode(ctxM, method, newCode);
44  }
```

5.4.2 Удаление условного кода

Постановка задачи: Еще одним распространенным сценарием является изменение логики исполнения кода или проведение некоторых пользовательских оптимизаций исходя из внешних параметров. Пусть в проекте есть конфигурационный класс `Config`, содержащий статические поля с настройками проекта. Требуется найти все места в коде, где есть какая-либо

логика, исполняющаяся под условием `Config.isDebug`, как показано ниже.

```
function foo() {
    /* business logic */
    if (Config.isDebug) {
        /* debug logic */
    } else {
        /* release logic */
    }
    /* business logic */
}
```

Ожидаемый результат: Требуется, имея бинарное представление исходного кода, преобразовать его таким образом, чтобы убрать все `true` ветки таких условных операторов:

```
function foo() {
    /* business logic */
    /* release logic */
    /* business logic */
}
```

Имплементация при помощи библиотеки: В рамках описанной библиотеки данная логика выражается последовательностью инструкций, представленной ниже. В данном примере будет подробно описан процесс манипуляции графом. Процесс нахождения нужных функций, а также процесс получения графа были подробно расписаны в прошлом примере.

```
/* The code to obtain graph is pretty much the same as before */

static void TransformISA1(lib_Core_ModificationContext *ctxM, lib_Core_Function* f,
    lib_Core_Graph* g) {
    /*
     * Suppose that ISA1 is for representing dynamic languages like JavaScript.
     * Considering this, FIXME:.
     */

    /* FIXME: Insert needed instructions from ISA1 */
}

static void TransformISA2(lib_Core_ModificationContext *ctxM, lib_Core_Function* f,
    lib_Core_Graph* g) {
```

```
13        /*
14         * Suppose that ISA2 is for representing static languages like Java.
15         * Considering this, FIXME:.
16         */
17        auto file = api_core->FunctionGetFile(f);
18
19        /* FIXME: Insert needed instructions from ISA2 */
20    }
```

5.5 Сравнение с аналогами

FIXME:

6 Заключение

В результате проделанной работы была разработана библиотека для манипуляции бинарными файлами многоязыковой платформы. Для достижения этой цели были сделаны следующие шаги:

- Был проведем анализ сущесвтующих решений для других платформ. Для каждого из них были обозначены преимущества, недостатки, и полезные для данной работы аспекты.
- Был спроектирован и имплементирован интерфейс для работы с языковыми метаданными бинарного файла. Имплементированный интрефейс ползволяет пользователям писать как обобщенный код для работы со всеми бинарными файлами платформы, так и языко-специфичный код, работающий только с бинарными файлами, полученными из конкретных фронтендов.
- В рамках интерфейса для для работы с языковыми метаданными бинарного файла был спроектирован интерфейс для анализа и добавления внешних зависимостей с учетом возможности интероперабельности.
- Благодаря выбору абстракций уровня исходного языка в рамках интерфейса для для работы с языковыми метаданными удалось скрыть

детали имплементации бинарного фала платформы и его фронтендов.

- Был спроектирован и имплементирован интерфейс для работы с телами функций. Имплементированный интерфейс предоставляет возможность писать сложные анализы, поскольку предоставляет код функций в `SSA` форме. Благодаря введенной системе `Target`-ов, данный интерфейс получилось сделать безопасным для использования, отделив все возможные наборы инструкций платформы от структуры графа, и привязав каждый набор инструкций к конкретным `Target`-ам.

На данный момент описанная библиотека еще находится на стадии разработки, и на данный момент имплементирован функционал, необходимый для написания пользовательских сценариев. В дальнейшем планируется продолжать имплементировать и развивать библиотеку. Первоочередной задачей является имплементация всех не реализованных на данный момент функций. После этого библиотека будет развиваться путем добавления новых языковых расширений оп мере развития платформы, а также путем поддержки уже существующих расширений. Еще одной целью является создание оберток для библиотеки на языках высокого уровня. Для увеличения производительности библиотеки планируется поддержать режим работы, основанный на шаблоне "посетитель аналогично фреймворку `ASM`. Это уменьшит накладные расходы на создание оберток библиотеки, что позволит встроить данную библиотеку в платформу для динамической манипуляции бинарными файлами.

Список литературы

[1] *Keller, Ralph.* Binary Component Adaptation / Ralph Keller, Urs Hölzle // *E. Jul (Ed.): ECOOP'98, LNCS 1445.* — 1998. — Pp. 307–329.

[2] Altering Java Semantics via Bytecode Manipulation / Eric Tanter, Marc Segura-Devillechaise, Jacques Noye, Jose Piquer // *D. Batory, C. Consel, and W. Taha (Eds.): GPCE 2002, LNCS 2487.* — 2002. — Pp. 283–298.

[3] *Malone, Todd.* Interoperability in Programming Languages / Todd Malone // *UMM CSci Senior Seminar Conference.* — 2014.

[4] *Wegner, Peter.* Interopability / Peter Wegner // *ACM Computing Surveys, Vol. 28, No. 1.* — 1996.

[5] *Brockschmidt, Kraig.* Inside OLE / Kraig Brockschmidt. — 1995.

[6] Aspect-Oriented Programming / Gregor Kiczales, John Lamping, Anurag Mendhekar et al. // *Springer-Verlag LNCS 1241.* — 1997.

[7] *С.В., Жемжицкий.* Введение в аспекто-ориентированное программирование / Жемжицкий С.В.

[8] A Model Driven Reverse Engineering Framework for Extracting Business Rules out of a Java Application / Valerio Cosentino, Jordi Cabot, Patrick Albert et al. // *RuleML.* — 2012.

[9] Lesson: Annotations (The JavaTM Tutorials > Learning the Java Language). — 2024. — March. `https://docs.oracle.com/javase/tutorial/java/annotations/index.html`.

[10] *Li, Yue.* Understanding and Analyzing Java Reflection / Yue Li, Tian Tan, Jingling Xue. — 2017.

[11] *Maes, Pattie.* Computional reflection / Pattie Maes // *Artificial intelligence laboratory, Vrije Universiteit, Brussels, Belgium.* — 1987.

[12] Description of package java.lang.classfile. — 2024. — March. https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/lang/classfile/package-summary.html.

[13] *Bruneton, Eric.* ASM: a code manipulation tool to implement adaptable systems / Eric Bruneton, Romain Lenglet, Thierry Coupaye // *France Télécom R&D, DTL/ASR.* — January 2002. — Vol. 28.

[14] *Lee, Han Bok.* BIT: A Tool for Instrumenting Java Bytecodes / Han Bok Lee, Benjamin G. Zorn // *Proceedings of the USENIX Symposium on Internet Technologies and Systems Monterey, California.* — 1997.

[15] *Welch, Ian.* Kava - Using Byte code Rewriting to add Behavioural Reflection to Java / Ian Welch, Robert J. Stroud. — 2001.

[16] *Welch, Ian S.* Kava – A Powerful and Portable Reflective Java / Ian S. Welch, Robert J. Stroud // *OOPSLA 2000 Companion.* — 2000.

[17] An Overview of AspectJ / Gregor Kiczales, Erik Hilsdale, Jim Hugunin et al.

[18] Description of the JNI - Java Native Interface. — 2024. — March. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html.

[19] Description of the N-API - Node API. — 2024. — March. https://nodejs.org/api/n-api.html.

[20] *Binder, Walter.* Advanced Java Bytecode Instrumentation / Walter Binder, Jarle Hulaas, Philippe Moret // *PPPJ.* — 2007.

[21] *Chiba, Shigeru.* Load-time Structural Reflection in Java / Shigeru Chiba // *Springer Verlag.* — 2000. — Pp. 313–336.

[22] *Chiba, Shigeru.* Javassist. A Reflection-based Programming Wizard for Java / Shigeru Chiba.

[23] *Aarniala, Jari.* Instrumenting Java Bytecode / Jari Aarniala // *Department of Computer Science University of Helsinki, Finland.* — 2005.