Санкт-Петербургский государственный университет

Друмов Максим Андреевич

Выпускная квалификационная работа

Расширение функциональности отладчика в IDE для языков C/C++ на платформе Eclipse

Уровень образования: бакалавриат

Направление *09.03.04 «Программная инженерия»*

Основная образовательная программа *СВ.5080.2021 «Программная инженерия»*

Научный руководитель:
доцент кафедры системного программирования, к.ф.-м.н., Луцив Д. В.

Консультант:
Старший программист-разработчик ПО I категории, ООО «Софтком», Бабанов П. А.

Рецензент:
Исполнительный директор, ООО «Софтком», Оносовский В. В.

Санкт-Петербург
2025

Saint Petersburg State University

Drumov Maxim

Bachelor's Thesis

Extending Debugger Functionality in IDE for C/C++ Languages on Eclipse Platform

Education level: bachelor

Speciality *09.03.04 "Software Engineering"*

Programme *CB.5080.2021 "Software Engineering"*

Scientific supervisor:
C.Sc., docent D. V. Luciv

Consultant:
Senior Software Engineer at "Softcom" P. A. Babanov

Reviewer:
COO at "Softcom" V. V. Onosovsky

Saint Petersburg
2025

Оглавление

Введение

Отладка кода является важнейшим этапом разработки программного обеспечения, поскольку позволяет своевременно выявлять и устранять ошибки в работе программы. Без неё процесс обнаружения и исправления неисправностей значительно осложняется, что может привести к снижению производительности и возникновению сбоев. Отсутствие отладки делает невозможным надёжное функционирование сложных программных решений и увеличивает затраты на их дальнейшую поддержку и исправление ошибок на этапе эксплуатации [12].

Многие интегрированные среды разработки (*IDE*), такие как *Visual Studio* [16], *CLion* [1], *Eclipse CDT* [6] и др., предоставляют мощные инструменты для отладки, что позволяет разработчикам эффективно выявлять и исправлять ошибки в коде. Компания ООО «Софтком» разрабатывает *IDE* на платформе *Eclipse* [8], на основе инфраструктуры *LLVM* [11] для работы с кодом на языках *C/C++*, ориентируясь на потребности заказчика, который активно использует различные фреймворки и аппаратные платформы. Эта *IDE* разрабатывается с целью объединить все необходимые инструменты в едином решении, обеспечивая удобную работу с кодом и устранение фрагментации процесса разработки, присущей использованию нескольких *IDE*.

Имеющаяся реализация отладчика была изначально разработана для ограниченного сценария — взаимодействия с эмулятором архитектуры *MIPS64*. Возможности удалённой отладки отсутствуют, что затрудняет работу с внешними устройствами и целевыми платформами. Кроме того, текущая архитектура не поддерживает гибкое расширение под различные форматы исполняемых файлов и отладочной информации, что ограничивает применимость инструмента в сложных сценариях. Эти недостатки препятствовали интеграции инструмента в полноценный процесс разработки и сопровождаются дополнительными временными и ресурсными затратами.

В данной работе для устранения указанных ограничений необходимо доработать архитектуру отладчика в *IDE* с целью обеспечить под-

держку различных архитектур, расширяемость под разные форматы отладочной информации, возможность работы не только с эмулятором *MIPS64*, но и с произвольными целевыми платформами, включая поддержку удалённой отладки.

1. Постановка задачи

Целью работы является расширение функциональности отладчика в *IDE* для языков *C/C++* на платформе *Eclipse*. Для её выполнения были поставлены следующие задачи.

1. Провести обзор текущей реализации отладчика в разрабатываемой IDE и существующих решений для выявления требований.
2. Доработать архитектуру отладчика с учётом требований расширяемости и поддержки удалённой отладки.
3. Выполнить реализацию следующих функциональных возможностей:
   - обработку форматов исполняемых файлов и отладочной информации (ELF/DWARF);
   - отображение регистров;
   - отображение стека вызовов;
   - программные точки останова;
   - пошаговое выполнение (с заходом, выходом и обходом);
   - отображение переменных;
   - промежуточный отладочный формат.
4. Провести тестирование.

2. Обзор

2.1. Текущая реализация отладчика

В этом разделе рассматривается текущая реализация отладчика. Для сборки применяются инструменты *LLVM* и *Clang* [2], которые способствуют оптимизации, интерпретации и компиляции кода. *LLVM (Low Level Virtual Machine)* — это набор компиляторов и инструментов, разработанных для упрощения создания программного обеспечения. *Clang* функционирует как фронтенд для языков *C, C++* и *Objective-C*.

В процессе разработки функциональности отладчика были рассмотрены различные подходы. Отладка может быть как локальной, так и удаленной, при этом оба подхода имеют свои особенности реализации. Ранее предпринимались попытки установить связь с *LLDB* [10] через стандартные потоки ввода-вывода (*stdin/stdout*) на локальной машине. Однако такой метод оказался сложным в реализации. Рассматривая локальную отладку как частный случай удаленной, где удаленный сервер заменяется локальным соединением через *localhost*, можно унифицировать процесс и взаимодействовать с *LLDB* (*lldb-server*) по единому подходу. Описанный выше способ является универсальным и подходит для нашей работы, так как удовлетворяет всем требованиям.

2.2. Существующие решения

Поскольку технологии *LLVM* и *LLDB* были выбраны ранее в проекте, то в обзоре существующих решений будет уделено внимание интегрированным средам разработки, которые используют эти инструменты. Однако стоит отметить, что таких *IDE* не так много. Рассмотрим несколько из них:

1. *Visual Studio*

   Visual Studio в основном использует свой собственный отладчик и не поддерживает *LLDB* по умолчанию. Однако в версии *Visual*

Studio 2022 для *macOS* можно активировать поддержку *LLDB*. Для других операционных систем, таких как *Windows*, поддержка *LLDB* не предусмотрена.

Что касается удаленной отладки, *Visual Studio* поддерживает эту функцию через свой встроенный отладчик, позволяющий подключаться к удаленным машинам. Однако это не связано с *LLDB*.

В *VS* удалённая отладка организована через подключение к удалённому компьютеру, на котором запущен *Remote Debugger*. Чтобы начать отладку, на удалённой машине сначала запускают *Remote Debugging Monitor*, который может работать в разных режимах подключения, таких как *Windows Authentication* или *No Authentication*. Важно, чтобы версии *Visual Studio* и удалённого отладчика совпадали.

Процесс выглядит так:

- настройка удалённого отладчика: На удалённой машине запускают *msvsmon.exe*, что позволяет *Visual Studio* установить соединение;
- настройка проекта в *Visual Studio*: На локальном компьютере в настройках проекта указывается тип отладки как "*Remote Windows Debugger*". Затем настраиваются параметры, такие как удалённый *IP*-адрес и путь к исполняемому файлу на удалённой машине;
- подключение и запуск отладки: После настройки можно запустить процесс отладки, и *Visual Studio* подключится к удалённой машине, чтобы управлять отладкой. Это позволяет задавать точки останова, пошагово выполнять код и просматривать состояние программы удалённо.

Visual Studio Remote Debugger больше ориентирован на *Windows* и требует специфической версии отладчика на удалённой машине. В отличие от *Visual Studio Remote Debugger*, который требует установки и настройки дополнительного ПО на удалённой системе,

lldb-server можно запускать без сложной установки или конфигурации, просто указав адрес для подключения.

2. *Visual Studio Code* [17]

   Visual Studio Code (VS Code) — кроссплатформенный редактор с расширяемой архитектурой. Поддержка отладки реализована через *Debug Adapter Protocol (DAP)* — протокол обмена *JSON*-сообщениями между пользовательским интерфейсом и отладчиком [19].

   Расширение *CodeLLDB* подключает *LLDB* через *DAP*-адаптер, написанный на *Rust* [3]. Оно поддерживает локальную и удалённую отладку, управление точками останова, переменными и регистрами, пошаговое выполнение и т.д.

   DAP получил широкое распространение благодаря универсальности и возможности повторного использования одного и того же отладочного *backend*'а в разных средах. Он особенно полезен в редакторах, не имеющих собственной модели отладки, таких как *VS Code*. В *Eclipse* же уже существует интегрированная отладочная модель — *Eclipse Debug Platform*.

   Хотя *DAP* применяется и в *Eclipse* (через проекты вроде *DAP4E*), его использование добавляет промежуточную прослойку между моделью и протоколом. В контексте прямого взаимодействия с *lldb-server* через *RSP* такая архитектура избыточна и усложняет реализацию без значимых преимуществ.

3. *Xcode* [18]

   Xcode — это интегрированная среда разработки, разработанная *Apple*, и она предназначена для разработки приложений под платформы *Apple*, такие как *iOS, macOS, watchOS и tvOS*. Встроенный отладчик *LLDB* был добавлен в *Xcode* начиная с версии 4, что позволило разработчикам использовать расширенные функции, такие как символьные точки останова, которые останавливают выполнение при вызове определённых символов (методов или

функций). Также поддерживается команда *expression*, позволяющая изменять значения переменных во время выполнения или выполнять сложные выражения прямо в отладчике. Визуальный интерфейс *Xcode* предоставляет инструменты для управления потоками, анализа структуры *UI*, инспектирования свойств объектов, что важно при разработке приложений. Кроме того, командный интерфейс *LLDB* в *Xcode* позволяет автоматизировать различные отладочные задачи с помощью пользовательских скриптов.

Xcode также предлагает консоль, позволяющую использовать команды *LLDB* напрямую. Команды, такие как *po* (*print object*), позволяют разработчикам выводить значения объектов, а команды для анализа стека и переменных дают детальное представление о текущем состоянии программы.

Благодаря *RSP* необходимость в консоли отпадает, так как в клиентской части отладчика возможно разместить все в специально отведенных для этого представлениях. Например, в *Variables* будут находиться значения переменных. Так же есть: *Registers View, Memory View, Breakpoints, Expressions* и др.

Поскольку *Xcode* доступен только на *macOS*, это ограничивает нас в разработке и снижает гибкость среды. Предпочтительно использовать решение, которое поддерживает кроссплатформенность, так как оно позволило бы разработчикам использовать одни и те же инструменты независимо от операционной системы;

4. *Eclipse CDT*

   Проект *CDT* (*C/C++ Development Tools*) направлен на создание полнофункциональной интегрированной среды разработки (*IDE*) на языках *C* и *C++* для платформы *Eclipse*. *CDT* полностью открыт и реализован исключительно на языке *java* в виде набора плагинов для платформы *Eclipse*. Отладчик *CDT* по умолчанию использует *GDB* (*GNU Debugger*), но есть экспериментальная поддержка *LLDB*. С использованием *LLDB* можно работать на *Linux*

дистрибутивах и на *MacOS*, а для *Windows* еще ведется разработка.

В настоящее время существует довольно много ограничений, но интеграция с *LLDB* очень новая и развивающаяся. Вот некоторые из них:

- не реализована удаленная отладка;
- не работают точки наблюдения(*watchpoints*);
- нельзя редактировать переменные (*View Variables*);
- нельзя редактировать память (*View Memory*);
- не реализованы действия *Move to Line*, *Resume at Line* (но *Run to Line* работает).

Помимо наличия множества недоработок, интеграция с *LLDB* не развивалась в течение нескольких лет, что ограничивает возможности использования этого отладчика;

5. *CLion*

   CLion использует *LLDB* для отладки приложений на языках *C* и *C++* и имеет интеграцию для удаленной отладки с использованием *lldb-server*. Данная функциональность особенно полезна, если сборка на целевой машине затруднительна или невозможна, так как *CLion* позволяет подключиться к отладочному серверу удаленно, сохраняя возможности локальной отладки. Для этого *CLion* предлагает две основные конфигурации:

   (a) *Remote Debug* — подходит для любых форматов проектов и систем сборки. С этим подходом отладочный сервер (*lldb-server*) настраивается вручную, и *CLion* подключается к нему для выполнения отладки.

   (b) *Remote GDB Server* — эта конфигурация используется, если проект создан с помощью *CMake*. *CLion* автоматически загружает скомпилированный бинарный файл на удаленную

машину и запускает его под *lldb-server*, упрощая настройку процесса.

CLion поддерживает функции отладки, включая точки останова, переменные, просмотр памяти и *watchpoints*. С помощью *SSH* подключаются к удалённой машине через графический интерфейс *CLion* по пути *Settings -> Build, Execution, Deployment*. По такому подобию в *Debug Configuration* будет реализован дополнительный тип запуска отладки со своими настройками и параметрами.

2.3. Вывод

Проведённый обзор показал, что интеграция инструментов отладки, таких как *LLDB* и *LLVM*, активно используется в современных *IDE* для разработки на *C* и *C++*. Среди рассмотренных решений каждый из инструментов имеет свои сильные и слабые стороны.

Опираясь на результаты анализа, можно выделить две группы решений.

Решения, ограничения которых планируется преодолеть:

1. ***Visual Studio*** — встроенный отладчик обладает высокой функциональностью, однако интеграция с *LLDB* отсутствует. Удалённая отладка реализована через собственный *Remote Debugger*, что затрудняет использование LLDB в качестве основного инструмента.

2. ***Eclipse CDT*** — предоставляет базовую интеграцию с *LLDB*, но не поддерживает удалённую отладку и ограничена в ряде ключевых функций: отсутствует поддержка *watchpoints*, невозможны редактирование переменных и памяти во время отладки, а также выполнение операций *Move to Line* и *Resume at Line*.

3. ***Visual Studio Code*** — поддерживает LLDB через расширение *CodeLLDB* и протокол *DAP*. Однако использование *Debug Adapter Protocol* добавляет промежуточную прослойку, которая в контек-

сте платформы Eclipse приводит к дублированию архитектуры и усложнению реализации при отсутствии значимых преимуществ.

Решения, на которые стоит опираться при проектировании:

1. ***Xcode*** — демонстрирует глубокую интеграцию с *LLDB*, включая поддержку командного интерфейса (например, *expression*) и расширенные функции отладки. Однако замкнутость на платформу *macOS* ограничивает применимость этого подхода.

2. ***CLion*** — обеспечивает наиболее полную интеграцию с *LLDB*, включая полноценную поддержку удалённой отладки через *lldb-server*. Универсальность и расширяемость делают данное решение эталонным примером удачной реализации отладчика на базе *LLDB*.

Таким образом, проектируемое решение будет стремиться сочетать архитектурные достоинства *CLion* и *Xcode*, с одновременным устранением недостатков *Visual Studio*, *Eclipse CDT* и *VS Code*. В рамках данной работы предпочтение отдано подходу на основе *Eclipse Debug Platform*, взаимодействующего напрямую с *lldb-server* по протоколу *RSP*, без использования *DAP*, так как это позволяет упростить архитектуру и обеспечить гибкую, расширяемую интеграцию отладчика в инфраструктуру Eclipse.

3. Архитектура

В данной главе описываются следующие ключевые характеристики отладчика: обзор базовой функциональности и технологий реализации, структура, интеграция в целевое *IDE*.

3.1. Обзор функциональности

В данной подсекции описывается набор ключевых функций, которые отладчик должен выполнять в процессе работы с программой. Основной задачей является управление процессом отладки через взаимодействие с отладчиком *lldb-server*, а также обеспечение интеграции с пользовательским интерфейсом IDE для отображения и управления различными аспектами отладки.

Отладчик должен поддерживать следующие ключевые функции:

1. установка и управление точками останова,
2. пошаговая отладка (с заходом, с выходом, с обходом),
3. отображение значений переменных и регистров, потоков и стека вызовов.

Эти функции обеспечиваются с помощью протокола *RSP*, а также встроенных возможностей *Eclipse Debug Platform* для обработки и отображения отладочной информации.

3.2. Технологии реализации

Отладчик реализован с использованием следующих ключевых технологий, выбранных исходя из требований расширяемости, поддержки *LLVM/LLDB*, а также интеграции с существующей архитектурой *Eclipse IDE*.

1. ***Eclipse Debug Platform*** — платформа предоставляет абстракции для организации отладочной сессии в рамках *Eclipse IDE*.

Она включает базовые интерфейсы и расширения для управления потоками исполнения, процессами, точками останова, переменными, регистрами и другими элементами, связанными с отладкой программ. Использование этой платформы позволяет встраивать собственный отладчик в стандартные представления *IDE*, сокращая трудозатраты на реализацию пользовательского интерфейса и обеспечивая совместимость с другими инструментами.

2. ***lldb-server и Remote Serial Protocol (RSP)*** — для взаимодействия с отлаживаемым процессом используется компонент *lldb-server*, входящий в состав *LLDB*. Он запускается на целевой (локальной или удалённой) машине и предоставляет *API* через текстовый протокол *RSP*. Протокол основан на отправке компактных текстовых пакетов с контрольной суммой и используется для обмена командами и данными: установки и удаления точек останова, чтения регистров, пошагового выполнения, чтения памяти и пр.

3.3. Структура отладчика

Для реализации отладочной подсистемы использована существующая модель отладчика, предоставляемая *Eclipse Debug Platform*. Основной точкой входа в эту модель является Цель отладки (*DebugTarget*) (Рис. 2) [7]. Она состоит из одного процесса и может содержать от одного до многих потоков. Использование данной модели позволяет интегрироваться в инфраструктуру отладки *Eclipse* и переиспользовать существующие механизмы управления потоками, переменными и событиями.

Взаимодействие между отладчиком и моделью обрабатывается нашим компонентом *EventDispatchJob*, далее именуемым *EventDispatcher* (Рис. 1). Он принимает входящие события с помощью метода *addEvent()* и доставляет их через *handleEvent()*. Сами события могут быть двух типов:

1. *IModelRequests* — запросы, инициированные со стороны пользова-

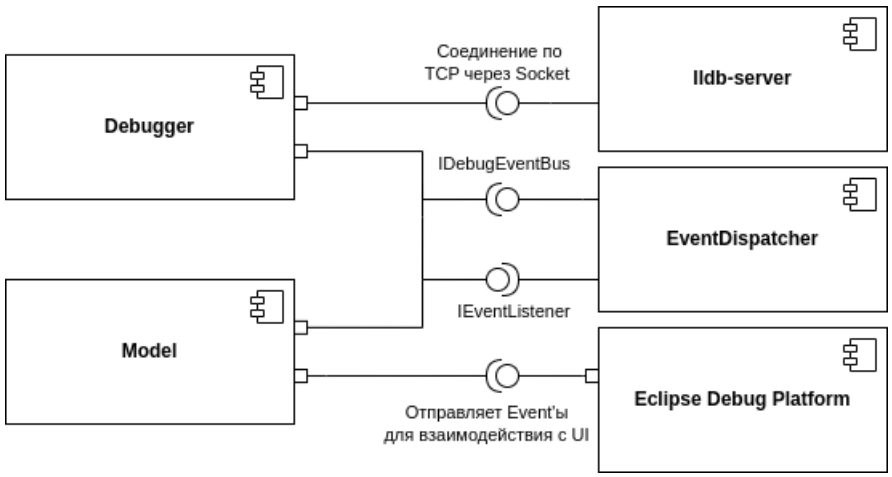

Рис. 1: Диаграмма отладчика

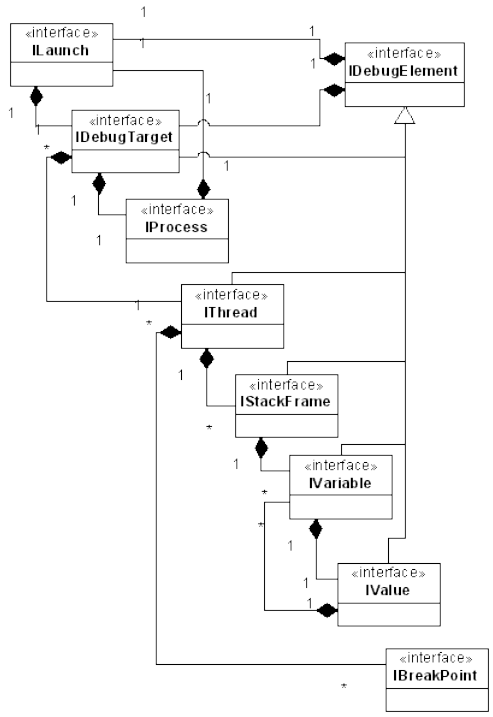

Рис. 2: Диаграмма отладочной модели из Eclipse Debug Platform

тельского интерфейса *Eclipse* (например, установка точки останова или выполнение шага).

2. *IDebuggerEvents* — события, поступающие от отладчика (например, достижение точки останова, завершение программы или изменение значения переменной).

Компонент *Debugger* реализует клиентскую часть взаимодействия с *lldb-server*. Связь с сервером осуществляется через *TCP*-соединение, устанавливаемое с помощью класса *Socket* из стандартной библиотеки *java.net*:

```
socket.connect(new InetSocketAddress(host, port);
outputStream = new DataOutputStream(socket.getOutputStream());
inputStream = new DataInputStream(socket.getInputStream());
```

Обмен сообщениями с *lldb-server* реализован на базе протокола *Remote Serial Protocol (RSP)* [14], который передаёт команды и ответы в виде компактных текстовых пакетов с контрольной суммой (Рис. 2). Это позволяет управлять выполнением программы, устанавливать точки останова, выполнять пошаговую отладку, читать и записывать память, регистры и т.д.

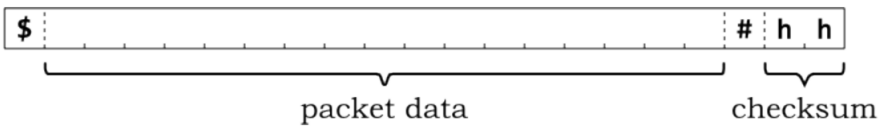

Рис. 3: Структура пакета в Remote Serial Protocol

Для отображения переменных, регистров и стека вызовов в *IDE* используется отладочная информация в формате *DWARF*. Парсинг *DWARF*-структур реализован вручную на основе анализа секций исполняемого файла *ELF*. Это позволяет получать имена переменных, типы, адреса, области видимости и связывать их с текущим состоянием процесса. Поскольку *IDE* ориентирована на языки *C* и *C++*, отладчику требуется лишь ограниченное подмножество информации из формата *DWARF*. Значительная часть тегов и конструкций *DWARF* предназначена для поддержки других языков программирования и в данной задаче не используется. В связи с этим использование полнофункциональных сторонних библиотек для чтения *DWARF* было бы избыточным.

Таким образом, отладчик реализует связку между низкоуровневым механизмом управления процессом через *lldb-server* и высокоуровневым пользовательским интерфейсом *IDE*, используя как собственную логику обработки команд, так и встроенные возможности *Eclipse Debug Platform*.

3.4. Интеграция в IDE на платформе Eclipse

Интеграция отладчика в *IDE* осуществляется через механизмы расширения *Eclipse Debug Platform* [9]. Используются следующие ключевые точки расширения:

1. `org.eclipse.debug.core` — предоставляет основу для поддержки различных отладочных архитектур. Включает менеджеры запуска, точек останова и выражений, а также события отладки. Позволяет регистрировать и управлять запусками приложений, сохранять конфигурации запуска и контролировать отладочную сессию. Используя ее, можно реализовать свою отладочную модель и подключить её к платформе через расширяемые точки (*launch delegates, launch modes, process factories*).

2. `org.eclipse.debug.core.model` — используется для реализации модели отладки в *Eclipse*. Пакет определяет основные интерфейсы, описывающие структуру и поведение отладочной сессии: *IDebugTarget* (отлаживаемый процесс), *IThread* (потоки), *IStackFrame* (кадры стека), а также переменные, значения, регистры, память, точки останова и выражения. Эти интерфейсы позволяют интегрировать собственный отладчик в инфраструктуру *Eclipse* без необходимости создания собственного механизма отображения данных.

3. `org.eclipse.debug.ui` — используется для интеграции пользовательского интерфейса отладки в *Eclipse*. Обеспечивает отображение стандартных представлений (переменных, регистров, памяти,

точек останова и выражений), а также поддержку управления выполнением (пауза, продолжение, шаги). Через механизм расширений предоставляет возможность кастомизации внешнего вида и поведения элементов отладочной модели (иконки, подписи, редакторы, подробности значений).

Механизм запуска реализуется через собственную реализацию `ILaunchDelegate`, которая инициирует подключение к *lldb-server*, создает экземпляры *DebugTarget* и *LLDBClient* и запускает *EventDispatcher*. Полученные от отладчика события транслируются в модель *Eclipse*, что позволяет отображать поток выполнения, стек вызовов и значения переменных в стандартных окнах отладки.

Обновление интерфейса происходит асинхронно, по мере получения событий от *lldb-server*. Благодаря использованию существующей отладочной модели, отладчик легко встраивается в рабочий процесс *IDE* и работает аналогично встроенным отладчикам *Eclipse CDT* или *JDT*.

4. Реализация

В данном разделе будет представлена реализованная функциональность отладчика в *IDE* для языков *C/C++* на платформе *Eclipse*.

4.1. Обработка форматов исполняемых файлов и отладочной информации (*ELF/DWARF*)

Доработан парсер *ELF*-файлов. А также чтение *DWARF*-информации, необходимой для отладки [5] [4]. Ранее была реализация, которая считывала и обрабатывала только малую часть секций и в следствии было сильно переработано чтение *DWARF* формата. В процессе обработки учитывается, что *ELF*-файлы содержат множество секций, не связанных с отладкой, поэтому считываются только те, которые необходимы для работы отладчика.

На первом этапе парсер анализирует заголовок *ELF*-файла, чтобы определить смещение таблицы заголовков секций. Затем извлекаются секции, начинающиеся с *.debug_ **, поскольку именно они содержат *DWARF*-данные для отладки. В случае отсутствия секции *.debug_frame* также обрабатывается секция *.eh_frame*, которая может содержать аналогичную информацию.

4.2. Отображение регистров

После каждой остановки потока, будь то точка останова или выполнение шага — *lldb-server* отправляет пакет *T Reply* [15], который содержит информацию о приостановленном потоке (Рис. 4). Этот пакет может включать адрес текущей инструкции, значения регистров, код сигнала и другие данные.

Когда выполнение останавливается на определённом адресе, в *DebugTarget* отправляется событие *SuspendedEvent*, которое включает информацию о значениях регистров и, позднее, будет содержать данные о текущем стеке вызовов. Для корректного отображения этих данных необходимо реализовать класс *RegisterGroup* из *Eclipse* платфор-

мы, который позволит сгруппировать регистры по их назначению. Этот класс будет интегрирован в *Register View*, обеспечивая удобное отображение информации.

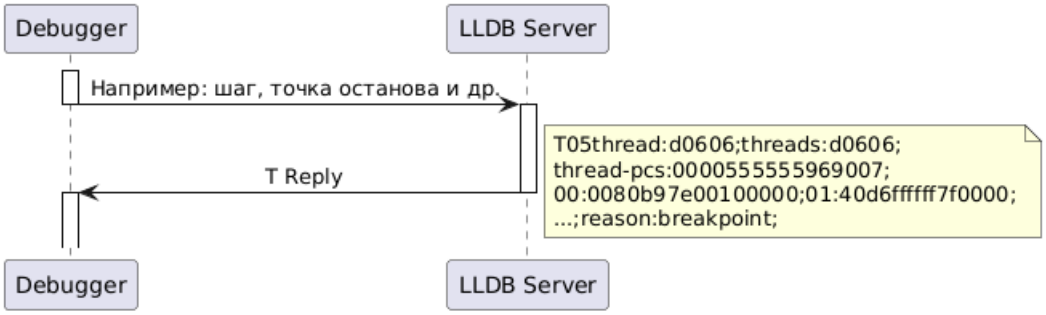

Рис. 4: T пакет с сервера

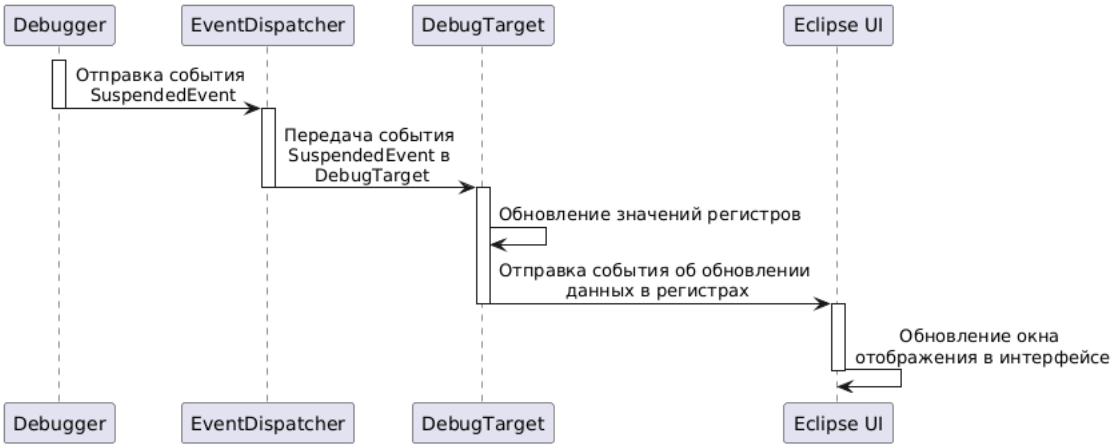

Рис. 5: Диаграмма последовательности отображения регистров

После получения новых данных о регистрах, их значения обновляются, и отправляется событие для *Eclipse*, чтобы сигнализировать об изменении контента (Рис. 5). Это заставит интерфейс обновить окно отображения, обеспечивая актуальность информации для пользователя (Рис. 6).

4.3. Стек вызовов

Помимо регистров, было реализовано отображение стека вызовов. Для этого перед отправкой происходит раскрутка стека с использовани-

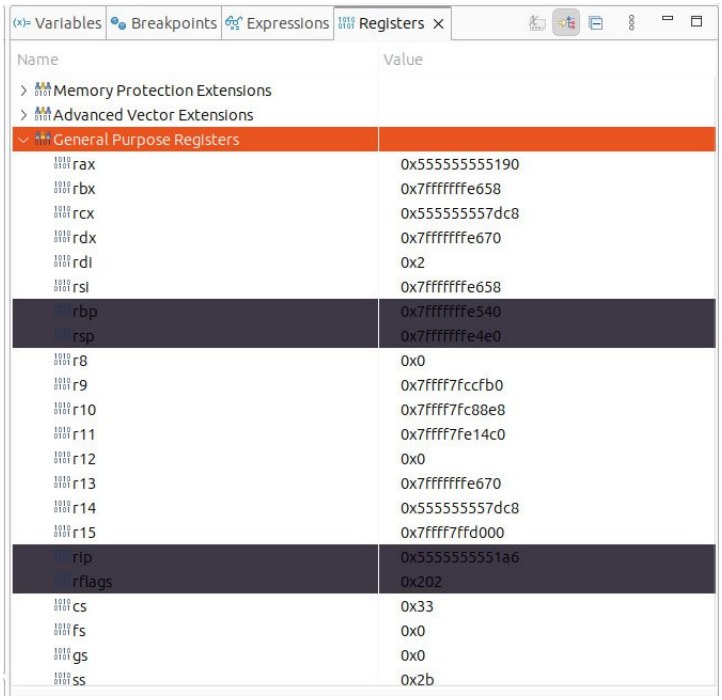

Рис. 6: Отображение регистров

ем отладочной информации для кадра из определенного исполняемого файла.

Начинается процесс с получения информации о текущем адресе инструкции, а также значениях регистров, которые необходимы для дальнейшей раскрутки стека.

Для раскрутки стека используется указатель стека (SP), который хранится в одном из регистров. С его помощью извлекаются данные из памяти, начиная с указанного адреса, и продолжается процесс чтения данных, пока не будет извлечена вся необходимая информация.

Для каждого кадра стека ищется соответствующая информация в таблицах отладочной информации, таких как *.eh_frame* или *.debug_frame*. Эти данные содержат информацию о структуре стека, что позволяет понять, как был изменен стек при переходах между функциями.

На основе информации из отладочных данных восстанавливаются значения регистров для каждого кадра стека. Для каждого регистра применяется соответствующее правило, которое указывает на смещение в стеке, и регистр восстанавливается с учетом этого смещения. Это

позволяет восстановить значения регистров на каждом уровне стека.

После восстановления значений регистров для текущего кадра, процесс продолжается, переходя к предыдущему кадру. Для этого используется адрес возврата, который извлекается из регистра, и если для этого адрес есть отладочная информация, то раскрутка стека продолжается с использованием этого адреса для поиска следующего кадра.

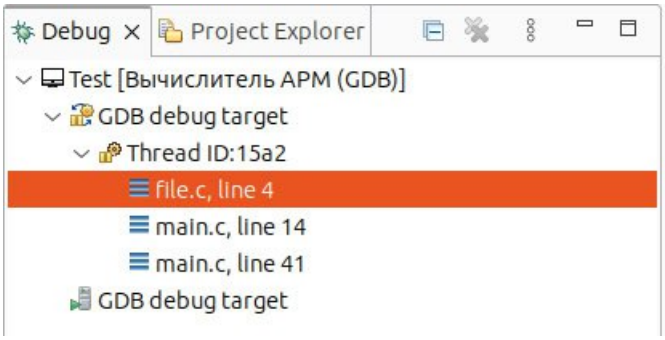

Рис. 7: Отображение стековых кадров

В процессе раскрутки стека, значения регистров и описание каждого кадра стека передаются в *DebugTarget*, который обновляет отображение информации для пользователя. Это позволяет наблюдать текущие значения регистров и стек вызовов на каждом этапе выполнения программы (Рис. 7).

4.4. Точки останова

В рамках работы была реализована возможность установки и удаления точки останова. Через пользовательский интерфейс передается команда с указанием файла и строки, где должна быть установлена точка останова.

DebugTarget инициирует событие *BreakpointEvent*, которое передается в *Debugger*. В свою очередь, *Debugger* использует отладочную информацию для определения адреса, соответствующего указанной строке в файле (Рис. 8).

После этого в *lldb-server* отправляется пакет с данными о точке останова. В зависимости от типа операции передается команда на установку

(*Z0*) или удаление (*z0*) программной точки останова по адресу *addr* типа *kind*.

‘z0,addr,kind’

‘Z0,addr,kind[;cond_list...][;cmds:persist,cmd_list...]’

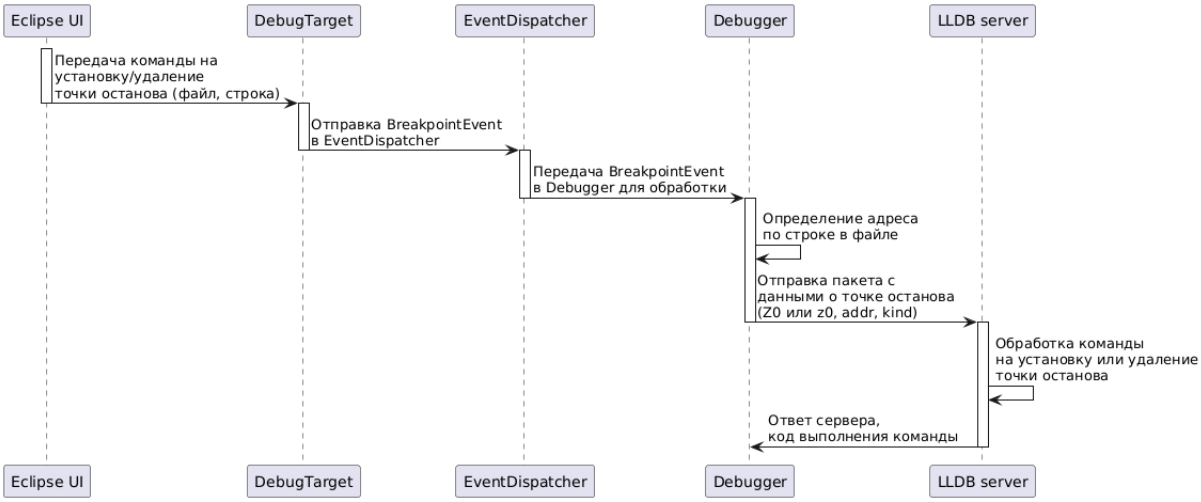

Рис. 8: Диаграмма последовательности управления точками останова

Точки останова можно установить или удалить, используя двойное нажатие или выбрав соответствующую опцию в всплывающем меню боковой линейки (Рис. 9).

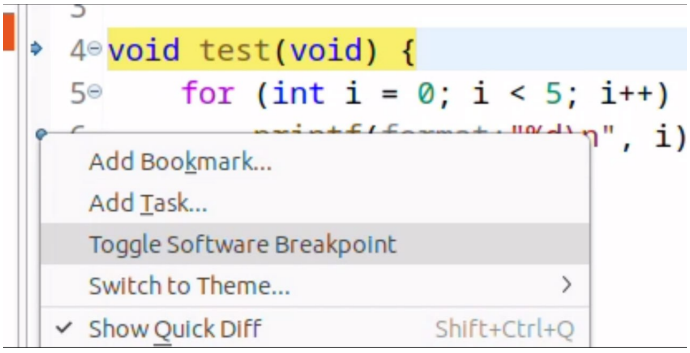

Рис. 9: Добавление и удаление точек останова через всплывающее меню

4.5. Шаги

В проекте используется интерфейс *IStep* из пакета *org.eclipse.debug.core.model*. Он предоставляет методы для выполнения различных шагов отладки: шаг с заходом, шаг с обходом и шаг с выходом. Этот интерфейс реализован в ключевых классах, таких как *DebugTarget*, *StackFrame* и *DebugThread*.

В процессе выполнения шагов отправляется событие *ResumeEvent* в *Debugger*. Оно содержит информацию о текущем состоянии выполнения, включая тип шага (шаг с заходом, шаг с обходом или шаг с выходом). На основании полученного состояния в *Debugger*'е выполняется соответствующая логика для реализации заданного шага.

4.5.1. Шаг с выходом

Первым делом находится отладочная информация для актуального адреса инструкции. Затем определяется, какая из секций *.eh_frame* или *.debug_frame* присутствует, и извлекается соответствующая информация. Эти секции содержат необходимую информацию для размотки стека, что позволяет найти адрес возврата. В таблице секции ищем строку, соответствующую данному адресу, и, используя данные из регистров и правила из предыдущей секции, приводим значения регистров к виду предыдущего стекового кадра.

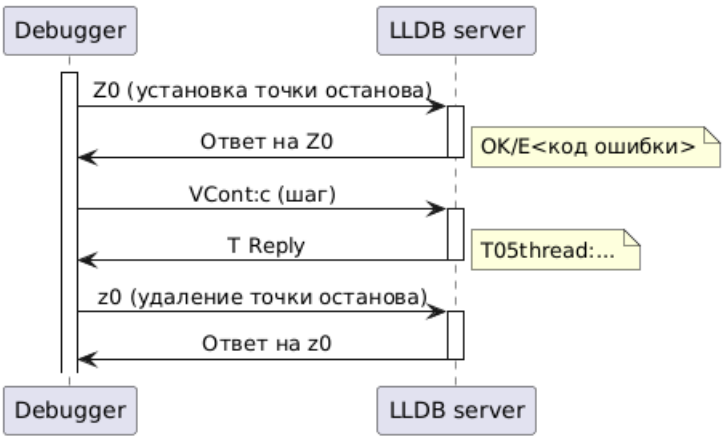

Рис. 10: *Continue* до заданного адреса

Из *CallFrameInformation* извлекается номер регистра, в котором хранится адрес возврата. Из списка изменённых регистров берётся значение регистра с адресом возврата, после чего отправляется последовательность пакетов на сервер (Рис. 10):

1. *Z0* — ставим брейкпоинт.

2. *VCont:c* — продолжаем выполнение программы.

3. *z0* — убираем точку останова.

После завершения выполнения команд с сервером отправляется *SuspendedEvent*.

4.5.2. Шаг с заходом

Имея адрес из *T Reply* пакета, на котором был остановлен поток, мы получаем актуальный исполняемый файл и соответствующую отладочную информацию. По этому адресу определяется, в какой функции мы находимся в данный момент. В текущем подходе используется метод отслеживания переходов между функциями с помощью указателя на стек. После каждого шага мы проверяем, изменился ли указатель стека, что позволяет определить, был ли произведён переход в новую функцию или завершено выполнение предыдущей. Этот метод работает, поскольку при вызове функции в стеке выделяется новый кадр, а при возврате из функции стек восстанавливается до предыдущего состояния. Таким образом, сравнив текущий указатель стека с предыдущим, можно точно определить, произошло ли изменение, связанное с входом или выходом из функции.

Если остановка произошла из-за точки останова, отправляется последовательность команд серверу для обработки точки останова и продолжения работы. Эта последовательность включает в себя пакеты: *z0, VCont:s, Z0* [13], которые позволяют пройти через адрес, на котором установлен брейкпоинт. Если же остановка была вызвана другим событием, отправляется только команда *VCont:s* для продолжения работы (Рис. 11).

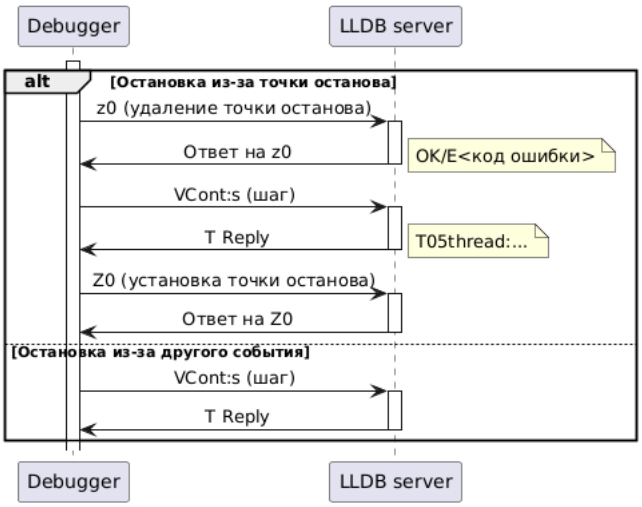

Рис. 11: Шаг на одну инструкцию

'vCont[;action[:thread-id]]...'

Возобновляет работу потока, указывая различные действия для него. Поддерживаются следующие действия:

1. '*c*' — Продолжить.
2. '*C sig*' — Продолжить с сигналом *sig*. Сигнал *sig* должен состоять из двух шестнадцатеричных цифр.
3. '*s*' — шаг.
4. '*S sig*' — шаг с сигналом *sig*. Сигнал *sig* должен состоять из двух шестнадцатеричных цифр.
5. '*t*' — остановить.
6. '*r start,end*'.

После того как остановились на каком-то адресе, проверяем исполняемый файл и функцию, в которой находится этот адрес. Возможные варианты действий:

1. Проверяется, подходит ли этот адрес для остановки шага. В секции *.debug_line* формата *DWARF* содержится информация об адресах, помеченных как *is_stmt* (свойство указывающее, что те-

кущая инструкция является рекомендованным местом для точки останова):

(a) Если адрес подходит, метод *step* останавливается, и отправляется событие *SuspendedEvent* в *DebugTarget*.

(b) Иначе повторяем процедуру отправки пакетов на сервер.

2. Не нашлась для адреса отладочная информация, то ставится точка останова на адрес возврата из секции *.*_frame* как и при нахождении стека вызовов, а затем отправляется пакет *VCont:c*. Как только остановились на брейкпоинте, убираем его пакетом *z0* (Рис. 10).

4.5.3. Шаг с обходом

Этот процесс похож на последовательность действий при шаге с заходом в функцию. Однако, если после выполнения шага адрес принадлежит другой функции, то необходимо найти адрес возврата и повторить шаги из пункта 2.

4.6. Переменные

Для определения переменных, относящихся к текущей функции, мы анализируем отладочную информацию для заданного адреса. На основе информации о функции и её тегах извлекаем переменные. Если переменная имеет базовый тип, то её значение извлекается непосредственно из памяти. В случае более сложных типов сохраняем только её адрес, по которому можно найти значение. В момент когда поток остановился и в *DebugTarget* пришло событие *SuspendedEvent*, обновляются кадры стека, регистры и отсылается событие для *Eclipse*, чтобы обновить данные все. В этот момент запрашивается для стекового кадра информация для переменных из *Debugger*'а. В *Debugger*'е из отладочной информации находим актуальные переменные и отправляем в *DebugTarget*.

И когда значения переменных пришли, отправляется также обновляющее событие для *Eclipse*, чтобы обновить отображение в *UI*.

Отображение переменных структур, классов и объединений работает аналогично отображению переменных с простыми типами — в отладчик отправляется информация о переменной и её типе. Чтобы просмотреть содержимое такой переменной, то есть её поля, необходимо в *Variables View* нажать на стрелку для раскрытия. Это инициирует запрос к *Debugger*'у на получение данных о содержимом переменной на заданную глубину. После этого *DebugTarget* получает событие и отображает обновлённую информацию.

4.7. Промежуточный отладочный формат

Был спроектирован промежуточный отладочный формат *IDF (Intermediate Debug Format)*, чтобы упростить поддержку различных форматов отладочной информации.

Ключевым классом в структуре *IDF* является *DebugInfo*. Он содержит информацию обо всех функциях программы, а также объект *LinesInfo*, представляющий сведения о строках исходного кода. Класс *LinesInfo* хранит данные о соответствиях между адресами и строками, включая номер строки и имя исходного файла, в котором она расположена.

4.7.1. Функции

В иерархии классов, связанных с представлением функций, интерфейс *IFunction* объединяет несколько аспектов: он наследует *IDeclared*, *IDebugInfoElement* и *IAddressableElement* (Рис. 12). Наследование от *IDeclared* означает, что каждая функция может содержать информацию о своём расположении в исходном коде, включая путь к файлу и номер строки. Через *IDebugInfoElement* функция включается в общую систему элементов отладочной информации, а *IAddressableElement* позволяет ассоциировать её с определённым диапазоном адресов в исполняемом коде.

Базовая реализация *IFunction* представлена абстрактным классом *AbstractFunction*, от которого наследуются конкретные классы:

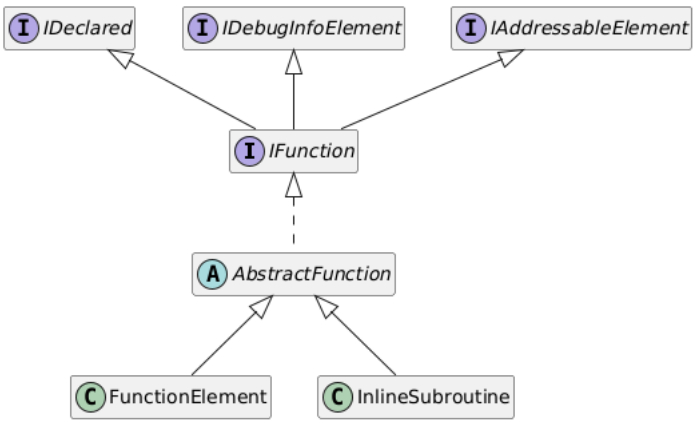

Рис. 12: Диаграмма классов функций *IDF*

FunctionElement и *InlineSubroutine*. Первый представляет обычную функцию, а второй, встроенную подпрограмму, возникающую, например, при инлайнинге во время компиляции.

4.7.2. Переменные

Следующим элементом отладочной информации являются переменные. Все они реализуют интерфейс *IDebugInfoVariable*, который, как и в случае с функциями, наследует *IDebugInfoElement* и *IDeclared* (Рис. 13). Это означает, что переменные включаются в общую систему отладочной информации и содержат сведения о своём расположении в исходном коде.

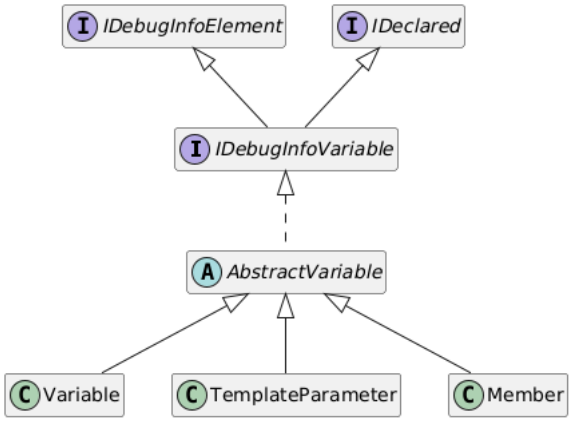

Рис. 13: Диаграмма классов переменных *IDF*

Абстрактный класс *AbstractVariable* служит основой для различных видов переменных. Класс *Variable* описывает локальные и глобальные переменные, а также формальные параметры функции, *TemplateParameter* представляет параметры шаблонов, а *Member* используется для описания членов классов или структур. Такое разграничение позволяет учитывать контекст переменной при её обработке, при этом сохраняя единый подход к работе с отладочной информацией.

4.7.3. Типы

Типы данных в *IDF* моделируются через иерархию, построенную вокруг интерфейса *IType*, который наследует *IDeclared* и *IDebugInfoElement* (Рис. 13). Формат охватывает все типы данных, встречающиеся в языках *C* и *C++*, включая как простые, так и составные типы. Простые типы, такие как целые числа или символы, описываются с помощью класса *BaseType*.

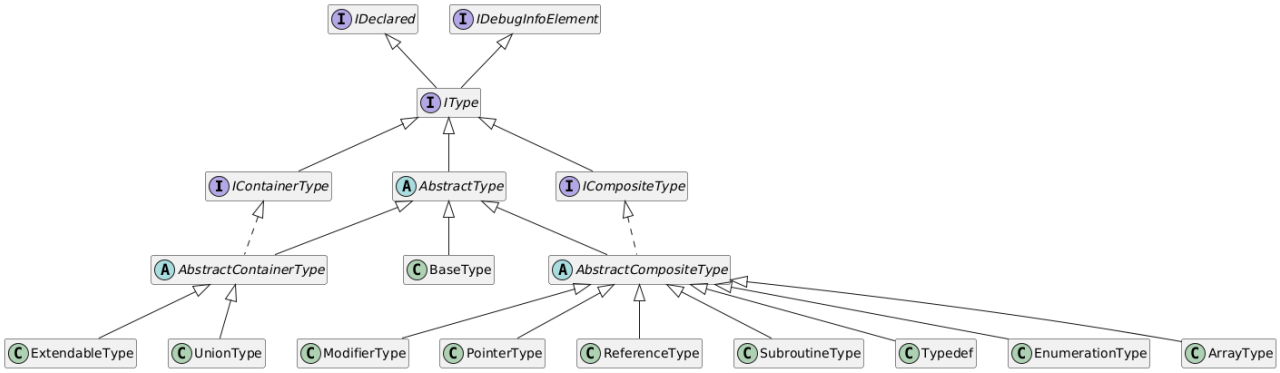

Рис. 14: Диаграмма классов типов *IDF*

Более сложные случаи обрабатываются через специализированные интерфейсы и абстрактные классы. Интерфейс *ICompositeType* описывает производные типы, содержащие другие типы. К примеру, *ArrayType* моделирует массив элементов заданного типа, а *PointerType* — указатель на другой тип. Все такие типы реализуют логику вложенности и позволяют рекурсивно восстанавливать структуру сложных выражений.

Для представления составных типов, содержащих набор членов разного типа, таких как структуры, классы и объединения, используется интерфейс *IContainerType*. Его реализации, такие как *UnionType* и *ExtendableType*, позволяют хранить коллекцию полей и методов.

Дополнительно система типов поддерживает перечисления (*EnumerationType*), модификаторы типов (*ModifierType*), ссылки (*ReferenceType*), подпрограммы (*SubroutineType*) и псевдонимы типов (*Typedef*). В совокупности эта иерархия позволяет описывать любую комбинацию типов, встречающуюся в программах на *C/C++*.

4.7.4. Дополнение

Также в *IDF* реализованы и другие важные классы, которые также реализуют интерфейс *IDebugInfoElement*. Эти классы расширяют возможности отладчика, позволяя ему работать с различными аспектами программы, такими как блоки кода, вызовы функций и другие элементы, которые могут быть полезны при анализе поведения программы.

CodeBlock — класс для хранения информации о блоках кода, таких как циклы, блоки *try*, *catch* и другие структуры управления. Это позволяет отладчику распознавать и анализировать выполнение программы в этих блоках.

Enumerator — этот класс представляет элементы перечислений. Он хранит информацию о значениях перечисляемых типов и помогает отладчику правильно отображать эти значения во время работы программы.

FunctionCallInfo — класс, который хранит информацию о вызовах функций, включая местоположение вызова, передаваемые аргументы и возвращаемые значения. Это помогает отслеживать выполнение программы, анализируя, какие функции были вызваны и с какими параметрами.

FunctionCallInfoParameter — класс для хранения информации о параметрах, передаваемых в вызовы функций. Он позволяет отладчику точнее отображать значения аргументов, что особенно важно при отслеживании данных, передаваемых между функциями.

Namespace — класс для работы с пространствами имён. Он помогает отладчику правильно интерпретировать элементы, такие как переменные и функции, которые могут быть разделены по разным пространствам имён, особенно в больших проектах.

UnspecifiedParameters — класс, который используется для обработки параметров, информация о которых недоступна или не определена. Это позволяет отладчику продолжать работу, даже если часть данных отсутствует.

4.8. Точка расширения для промежуточного формата

Для добавления поддержки нового формата достаточно реализовать фабрику преобразования в промежуточный формат и после этого отладчик сможет работать с ним без изменения остальной логики. Эта фабрика регистрируется в системе расширений *Eclipse* через определённый *extension point*, объявленный в плагине отладчика.

В рамках данной работы написана фабрика преобразования из формата *DWARF* в *IDF (Intermediate Debug Format)*. Реализованная фабрика включает основную функциональность по парсингу *DWARF*-данных, а также модули для преобразования информации в структуры и классы *IDF*.

Например, для регистрации новой фабрики используется расширение следующего вида:

```
<extension
    point="su.softcom.cldt.debug.core.info">
      <readerFactory
            factoryName="DwarfReaderfactory"
            name="DwarfReaderFactory"
            readerFactory="su.softcom.cldt.internal.debug.core...DwarfReaderFactory">
      </readerFactory>
   </extension>
```

Инфраструктура отладчика в момент инициализации загружает все реализации, зарегистрированные через данный *extension point*, и деле-

гирует выбор подходящей фабрике в зависимости от структуры загружаемого файла. Это позволяет подключать поддержку форматов по мере необходимости, не изменяя основную архитектуру отладчика.

5. Тестирование

В данной главе описывается тестирование реализованной функциональности, а именно фабрики промежуточного отладочного формата для Dwarf и клиент отладки.

5.1. Промежуточный отладочный формат

Для проверки корректности преобразования отладочной информации из формата DWARF во внутреннее представление были реализованы модульные тесты с использованием *JUnit 5* и *Mockito*. Эталонные данные для тестов формировались с помощью *Python*-скриптов на основе библиотеки *pyelftools*. Скрипты извлекали из *ELF*-файлов адреса функций, переменные и *DWARF*-выражения, после чего сохраняли данные в формате *YAML*.

Тесты параметризованы и автоматически считывают *YAML*-файлы, содержащие путь к *ELF*-файлу, адресные таблицы, функции, переменные и бинарные представления выражений размещения. После инициализации фабрика отладочной информации извлекает соответствующие структуры, которые сравниваются с эталонными значениями. Для *DWARF*-выражений реализован собственный интерпретатор, позволяющий проверить семантическую корректность вычислений на основе начального окружения регистров и логики *DWARF*.

В тестах также проверяется соответствие количества переменных, правильность диапазонов, корректность адресов и соответствие вычисленного значения размещения переменной. Общий уровень покрытия кода по результатам тестирования составил около 85%.

5.2. Клиент отладки

Клиент в *Debugger* реализует взаимодействие с серверной частью отладчика по протоколу *RSP*, обрабатывая команды, ответы и связанные с ними события. Для тестирования также использовались *JUnit 5* и *Mockito*. Подключение к реальному серверу не выполнялось. Общение

с сервером эмулировалась с помощью мокированного сокета и потоков ввода-вывода, что позволило точно контролировать отправку команд и получение ответов.

В качестве входных данных использовались *YAML*-файлы, содержащие команды и ожидаемые ответы от сервера. Эти файлы формировались автоматически с помощью специально разработанного скрипта на *Python*, который позволяет быстро добавлять новые тестовые сценарии. Такой подход упрощает расширение тестового покрытия, для добавления поддержки новой команды достаточно лишь добавить соответствующую запись в *YAML*.

В тестах проверялась корректность формирования и передачи команд в сокет, построение *RSP*-пакетов, а также интерпретация и разбор ответов сервера. Особое внимание уделялось событиям, генерируемым клиентом: добавлению и удалению точек останова, завершению работы, запуску отладки и другим действиям. Генерация событий отслеживалась через мокированный *EventDispatcher*.

Покрытие тестами составляет более 80% кода, включая ключевую логику работы с протоколом *RSP* и внутренними событиями отладки.

Заключение

В рамках данной работы были выполнены следующие задачи.

1. Проведен обзор текущей реализации, выявлены ее недостатки и рассмотрен вариант их устранения. На основе проанализированных существующих решений выявлены требования к отладчику.

2. Доработана архитектура отладчика с учётом требований расширяемости и поддержки удаленной отладки.

3. Выполнена реализация:

   - доработана обработка форматов исполняемых файлов и отладочной информации (ELF/DWARF);
   - реализовано отображение регистров в Register View и стека вызовов для потоков;
   - программные точки останова ставятся через выпадающего меню обзорной линейки, а также по двойному нажатию;
   - поддержано пошаговое выполнение (с заходом, выходом и обходом);
   - реализовано отображение в Variables View переменных примитивов, структур, классов и т.д.;
   - разработан промежуточный отладочный формат;
   - реализовал Eclipse Extension Point для промежуточного отладочного формата.

4. Проведено модульное тестирование фабрики отладочного формата и клиента протокола RSP. Покрытие составляет более 80%.

Исходный код закрыт.

Список литературы

[1] CLion. — URL: https://www.jetbrains.com/clion/ (дата обращения: 27 мая 2025 г.).

[2] Clang. — URL: https://clang.llvm.org/ (дата обращения: 27 мая 2025 г.).

[3] CodeLLDB. — URL: https://github.com/vadimcn/codelldb (дата обращения: 27 мая 2025 г.).

[4] DWARF. — URL: https://dwarfstd.org/doc/DWARF5.pdf (дата обращения: 27 мая 2025 г.).

[5] ELF. — URL: https://refspecs.linuxbase.org/elf/elf.pdf (дата обращения: 27 мая 2025 г.).

[6] Eclipse CDT (C/C++ Development Tooling). — URL: https://projects.eclipse.org/projects/tools.cdt (дата обращения: 27 мая 2025 г.).

[7] Eclipse Debug Model. — URL: https://www.eclipse.org/articles/Article-Debugger/how-to.html (дата обращения: 27 мая 2025 г.).

[8] Eclipse Platform. — URL: https://projects.eclipse.org/projects/eclipse.platform (дата обращения: 27 мая 2025 г.).

[9] Eclipse Platform API Specification. — URL: https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Findex.html (дата обращения: 27 мая 2025 г.).

[10] LLDB. — URL: https://lldb.llvm.org/ (дата обращения: 27 мая 2025 г.).

[11] LLVM. — URL: https://llvm.org/ (дата обращения: 27 мая 2025 г.).

[12] McConnell Steve. Code Complete: A Practical Handbook of Software Construction. — 2 edition. — Microsoft Press, 2004.

[13] Packets. — URL: https://sourceware.org/gdb/current/onlinedocs/gdb.html/Packets.html#Packets (дата обращения: 27 мая 2025 г.).

[14] Remote Serial Protocol. — URL: https://sourceware.org/gdb/current/onlinedocs/gdb.html/Remote-Protocol.html (дата обращения: 27 мая 2025 г.).

[15] Stop Reply Packets. — URL: https://sourceware.org/gdb/current/onlinedocs/gdb.html/Stop-Reply-Packets.html#Stop-Reply-Packets (дата обращения: 27 мая 2025 г.).

[16] Visual Studio. — URL: https://visualstudio.microsoft.com/ru/ (дата обращения: 27 мая 2025 г.).

[17] Visual Studio Code. — URL: https://code.visualstudio.com/ (дата обращения: 27 мая 2025 г.).

[18] XCode. — URL: https://developer.apple.com/xcode/ (дата обращения: 27 мая 2025 г.).

[19] https://microsoft.github.io/debug-adapter-protocol//. — URL: https://code.visualstudio.com/ (дата обращения: 27 мая 2025 г.).