Санкт-Петербургский государственный университет

Диденко Андрей Антонович

Выпускная квалификационная работа

Импорт QMake-проектов в среду разработки на базе платформы IDE Eclipse: автоматизация преобразования и интеграция CMake

Уровень образования: бакалавриат

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

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

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

Рецензент:
Разработчик ПО, ООО «Софтком», Низамов Р. Ф.

Санкт-Петербург
2025

Saint Petersburg State University

Andrey Didenko

Bachelor's Thesis

Importing QMake projects into the Eclipse IDE environment: automation of conversion and CMake integration

Education level: bachelor

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

Programme *CB.5162.2021 "Programming Technologies"*

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

Reviewer:
Software Developer, "Softcom" R.F. Nizamov

Saint Petersburg
2025

Оглавление

Введение

Разработка программного обеспечения на языках C и C++ связана с необходимостью управления сложными процессами компиляции, линковки и конфигурации проектов. Системы сборки автоматизируют эти задачи, организуя зависимости и обеспечивая воспроизводимость сборки. Такие инструменты позволяют разработчикам сосредоточиться на создании функциональности, минимизируя рутинные операции, что особенно важно для крупных проектов с множеством компонентов.

Для C/C++ проектов активно применяются системы сборки, такие как CMake, в то время как QMake и Make использовались ранее, но сейчас их популярность снизилась. Make, появившийся в 1970-х годах, обеспечивает детальный контроль над процессом сборки, но требует написания сложных сценариев [13]. QMake, созданный для фреймворка Qt, облегчает разработку Qt-приложений благодаря интеграции с Qt Creator, но его возможности ограничены вне экосистемы Qt [17]. С выходом Qt6 разработчики Qt отказались от поддержки QMake, перейдя на CMake, который поддерживает кроссплатформенность и генерирует сборочные файлы для различных компиляторов и платформ [16]. CMake упрощает управление проектами, абстрагируя низкоуровневые детали [18].

Legacy-проекты, использующие Make или QMake, сталкиваются с трудностями сопровождения из-за ограниченной совместимости с современными инструментами [7]. Проекты на старых версиях Qt, основанные на QMake, плохо интегрируются с новыми IDE, а сложность настройки Makefile увеличивает затраты на поддержку и вероятность ошибок при внесении изменений. Переход на CMake позволяет модернизировать такие проекты, обеспечивая совместимость с интегрированными средами разработки, такими как Visual Studio, CLion и Eclipse, благодаря поддержке множества платформ и упрощённой конфигурации сборки. Однако это требует переработки конфигурационных файлов сборки.

Компания ООО «Софтком» разрабатывает программные решения

для корпоративных заказчиков, многие из которых используют ранее созданные проекты на старых версиях Qt с QMake или Make. Эти проекты, разработанные для специфических задач, усложняют сопровождение из-за устаревших инструментов сборки. Переход на CMake позволит унифицировать процессы сборки и упростить поддержку, что отвечает потребностям заказчиков и повышает эффективность работы компании.

Ключевым продуктом ООО «Софтком» является IDE-плагин CLDT, разработанный на базе Eclipse. CLDT предоставляет удобную среду для разработки приложений на C/C++, включая управление зависимостями и настройку сборки. Интеграция инструмента для конвертации файлов сборки в CMake в CLDT упростит обновление проектов для разработчиков и заказчиков, сохраняя привычную рабочую среду.

Конвертация файлов сборки из Make и QMake в CMake необходима для упрощения перехода на новую систему сборки. Автоматизированный инструмент для этой задачи сократит трудозатраты и снизит вероятность ошибок, обеспечивая совместимость с инфраструктурой компании и требованиями заказчиков.

Цель данной работы — подготовка инструмента для автоматизированной конвертации проектов из систем сборки Make и QMake в CMake с интеграцией в IDE-плагин CLDT, разрабатываемый в ООО «Софтком». Решение обеспечит эффективное обновление Qt-проектов заказчиков, упростит их сопровождение и повысит соответствие современным требованиям разработки.

1. Постановка задачи

Целью данной работы является интеграция инструмента для автоматической миграции QMake проектов в CMake в плагин CLDT для среды разработки Eclipse.

Для достижения цели были выделены следующие основные **задачи**:

- проанализировать существующие инструменты миграции;
- адаптировать найденный инструмент для соответствия функциональным требованиям;
- разработать подход к интеграции конвертера в состав CLDT;
- реализовать GUI с вызовом инструмента миграции и настройкой параметров сборки;
- провести тестирование и апробацию.

2. Обзор

2.1. Обзор существующих решений

В процессе подготовки к работе были проанализированы инструменты для автоматизации конвертации систем сборки из Makefile, qmake, Autotools, SCons и Visual Studio в CMake по следующим критериям:

Критерии обзора

Для оценки инструментов использовались следующие критерии:

- Актуальность поддерживаемых систем сборки (Make, qmake и т.д.).
- Расширяемость — возможность адаптации под новые системы сборки.
- Качество и гибкость API для интеграции с другими инструментами.
- Поддержка сложных проектов с условной компиляцией и динамической генерацией файлов.
- Соответствие функциональным требованиям для конвертации qmake в CMake (см. ниже).

Функциональные требования для конвертации qmake в CMake

- **Основные переменные проекта**: TEMPLATE, TARGET, CONFIG
- **Исходные файлы и артефакты**: SOURCES, HEADERS, FORMS, RESOURCES
- **Настройки компиляции**: QMAKE_CXX, QMAKE_CC, QMAKE_CXXFLAGS, QMAKE_CFLAGS, DEFINES, INCLUDEPATH

- **Настройки линковки**: LIBS, QMAKE_LFLAGS
- **Qt-специфичные переменные**: QT, MOC_DIR, UI_DIR, RCC_DIR
- **Подпроекты**: SUBDIRS
- **Условные конструкции**: CONFIG(debug, debug|release), Вложенные условные конструкции
- **Дополнительные настройки**: QMAKE_CXXFLAGS += -std=c++11 | -std=c++17 | -std=c++20, DESTDIR, DEPENDPATH, PRECOMPILED_HEADER, win32 | unix | macx, QMAKE_CXXFLAGS_RELEASE | QMAKE_CXXFLAGS_DEBUG, DISTFILES, VERSION, QMAKE_EXTRA_COMPILERS, PRE_TARGETDEPS, QMAKE_POST_TARGETDEPS, RC_FILE, INSTALLS, include, depends, QMAKE_EXTRA_TARGETS, commands

Особое внимание уделено реализации конвертации Makefile в CMake, выполненной студентом Лоскутовым Владиславом Евгеньевичем в рамках учебного проекта.

2.1.1. make2cmake

[11] **Как работает конвертация:**

1. Считывание и парсинг Makefile: Инструмент, реализованный на Python, читает Makefile как текстовый файл. Лексический анализатор разбивает файл на токены, идентифицируя переменные (`CC`, `CFLAGS`), цели (`targets`), правила (`rules`) и команды сборки (`recipes`). Например, строка `CC = gcc` распознаётся как переменная компилятора, а `all: main.o` — как цель.
2. Анализ структуры и зависимостей: Извлекаются зависимости между целями (например, `main.o: main.c utils.h`).

Команды компиляции (например, `gcc -c main.c -o main.o`) анализируются для определения компилятора, флагов (`-c`, `-O2`) и исходных файлов. Переменные, такие как `$(CC)`, разрешаются в их значения.

3. Преобразование в CMake: Переменные преобразуются в команды CMake: `CC` → `set(CMAKE_C_COMPILER ...)`, `CFLAGS` → `target_compile_options(...)`. Цели становятся `add_executable()` или `add_library()`. Зависимости добавляются через `target_link_libraries()`.

4. Генерация CMakeLists.txt: Формируется файл с командами CMake, включая пути к исходным файлам (`target_sources(...)`), флаги компиляции и зависимости. Инструмент создаёт минималистичный `CMakeLists.txt`, ориентированный на простые проекты.

Пример преобразования:

```
CC = gcc
CFLAGS = -O2 -Wall
all: main
main: main.o utils.o
    $(CC) main.o utils.o -o main
main.o: main.c utils.h
    $(CC) $(CFLAGS) -c main.c
utils.o: utils.c utils.h
    $(CC) $(CFLAGS) -c utils.c
```

CMakeLists.txt (сгенерированный):

```
cmake_minimum_required(VERSION 3.10)
project(MyProject)
set(CMAKE_C_COMPILER gcc)
add_executable(main main.c utils.c)
target_compile_options(main PRIVATE -O2 -Wall)
```

Особенности: Инструмент разработан как учебный проект и ориентирован на простые Makefile для C/C++ проектов. Он не поддерживает сложные конструкции, такие как условная компиляция (`ifdef`), макросы или динамические зависимости. Поддержка qmake отсутствует, так как инструмент не предназначен для работы с Qt-проектами.

Преимущества:

- Простота реализации, подходит для небольших проектов.
- Открытый исходный код, доступный для доработки.

Недостатки:

- Ограниченная поддержка сложных Makefile с нестандартными конструкциями.
- Отсутствие обработки qmake и других систем сборки.
- Минимальная документация и отсутствие API для интеграции.

Поддержка функциональных требований qmake:

- **Не поддерживается:** Инструмент не работает с qmake, поэтому требования, такие как TEMPLATE, TARGET, QT, FORMS, SUBDIRS, не реализованы.
- **Частичная поддержка:** Для Makefile поддерживаются аналоги SOURCES (`target_sources(...)`), QMAKE_CXXFLAGS (`target_compile_options(...)`), LIBS (`target_link_libraries(...)`). Условные конструкции и Qt-специфичные переменные не обрабатываются.

2.1.2. CMake Converter

[10] **Как работает конвертация:**

1. Считывание файлов Visual Studio: Инструмент анализирует файлы решения (`*.sln`) и проектов (`*.vcxproj`), используя XML-парсер для извлечения структуры: список проектов, зависимости, целевые платформы (Win32, x64) и конфигурации (Debug/Release). Например, `<ProjectReference>` определяет зависимости между проектами.

2. Анализ настроек компиляции: Извлекаются параметры: компиляторы (MSVC, Clang), флаги (`/O2`, `/W3`), пути к библиотекам (`/LIBPATH`) и включаемые файлы (`/I`). Учитываются платформозависимые настройки, такие как `<PlatformToolset>`.

3. Сопоставление с CMake: Каждый проект преобразуется в `add_executable()` или `add_library()`. Флаги компиляции добавляются через `target_compile_options()`, а зависимости — через `target_link_libraries()`. Платформозависимые настройки преобразуются в условные блоки (`if(WIN32)...`).

4. Генерация CMakeLists.txt: Создаётся файл, включающий команды для сборки, пути к исходным файлам, настройки платформы и зависимости. Инструмент генерирует иерархию `CMakeLists.txt` для решений с несколькими проектами.

Пример преобразования:

```
<ItemGroup>
    <ClCompile Include="main.cpp" />
    <ClCompile Include="utils.cpp" />
</ItemGroup>
<PropertyGroup>
    <AdditionalIncludeDirectories>include</AdditionalIncludeDirectories>
    <AdditionalOptions>/O2 /W3</AdditionalOptions>
</PropertyGroup>
<Link>
    <AdditionalDependencies>user32.lib</AdditionalDependencies>
</Link>
```

CMakeLists.txt (сгенерированный):

```
cmake_minimum_required(VERSION 3.10)
project(MyProject)
add_executable(my_project main.cpp utils.cpp)
target_include_directories(my_project PRIVATE include)
target_compile_options(my_project PRIVATE /O2 /W3)
target_link_libraries(my_project PRIVATE user32)
```

Особенности: Инструмент полностью автоматизирован для Visual Studio, поддерживает C/C++ и Fortran. Не работает с qmake, Makefile или другими системами сборки. Подходит для миграции Windows-проектов на кроссплатформенные системы.

Преимущества:

- Эффективная обработка Visual Studio проектов.
- Поддержка многоплатформенных конфигураций.

Недостатки:

- Отсутствие поддержки qmake и Autotools, ограниченная поддержка Makefile.

Поддержка функциональных требований qmake:

- **Не поддерживается:** Нет поддержки qmake, исключая QT, FORMS, SUBDIRS и другие Qt-специфичные конструкции.
- **Частичная поддержка:** Аналоги SOURCES, HEADERS (`target_sources(...)`), QMAKE_CXXFLAGS (`target_compile_options(...)`), LIBS (`target_link_libraries(...)`) для Visual Studio проектов.

2.1.3. Autotools-to-CMake

[2] **Как работает конвертация:**

1. Считывание `configure.ac` и `Makefile.am`: Инструмент парсит `configure.ac` для извлечения макросов (`AC_INIT`, `AC_PROG_CC`) и переменных конфигурации (например, `CFLAGS`). `Makefile.am` анализируется для определения целей (`bin_PROGRAMS`), исходных файлов и зависимостей.

2. Анализ конфигурации: Макросы преобразуются в команды CMake: `AC_PROG_CC` → `set(CMAKE_C_COMPILER ...)`, `AC_OUTPUT` игнорируется или адаптируется. Переменные, такие как `CFLAGS`, преобразуются в `target_compile_options()`.

3. Сопоставление зависимостей: Зависимости между файлами и библиотеками извлекаются из `Makefile.am` и преобразуются в `target_link_libraries()`. Поддерживаются сложные проекты с несколькими `Makefile.am`.

4. Генерация CMakeLists.txt: Создаются команды `add_library()` или `add_executable()`, формируется структура директорий для больших проектов с вложенными `Makefile.am`.

Пример преобразования:

```
bin_PROGRAMS = my_program
my_program_SOURCES = main.c utils.c
my_program_CFLAGS = -O2 -Wall
my_program_LDADD = -lm
```

CMakeLists.txt (сгенерированный):

```
cmake_minimum_required(VERSION 3.10)
project(MyProject)
add_executable(my_program main.c utils.c)
target_compile_options(my_program PRIVATE -O2 -Wall)
target_link_libraries(my_program PRIVATE m)
```

Особенности: Инструмент оптимизирован для проектов на Autotools, особенно старых, поддерживает сложные проекты с явно

указанными зависимостями, но может требовать ручной доработки для динамических или нестандартных зависимостей. Подходит для миграции legacy-проектов на CMake.

Преимущества:

- Специализация на Autotools.
- Автоматизация миграции старых проектов.

Недостатки:

- Отсутствие поддержки qmake и прямой поддержки Makefile (за исключением Makefile, сгенерированных Autotools).
- Требуется ручная корректировка для сложных проектов.

Поддержка функциональных требований qmake:

- **Не поддерживается:** Нет поддержки qmake, исключая QT, FORMS, SUBDIRS.
- **Частичная поддержка:** Аналоги SOURCES, HEADERS, QMAKE_CXXFLAGS, LIBS для Autotools.

2.1.4. Make2CMake

[3] **Как работает конвертация:**

1. Парсинг Makefile: Инструмент читает Makefile, идентифицируя переменные (`CC`, `CFLAGS`), цели, правила и команды. Используется синтаксический анализ для выделения компиляторов (например, `g++`), флагов (`-O2`) и зависимостей.
2. Анализ зависимостей: Правила (например, `main.o: main.c`) преобразуются в списки исходных файлов и зависимости. Переменные, такие как `$(CC)`, разрешаются и преобразуются в команды CMake.

3. Сопоставление с CMake: Команды компиляции преобразуются в `target_compile_options()`, а цели — в `add_executable()` или `add_library()`. Зависимости добавляются через `target_link_libraries()`.

4. Генерация CMakeLists.txt: Создаётся файл с командами для сборки, включая пути к файлам, флаги и зависимости. Инструмент поддерживает базовые Makefile, но не сложные конструкции.

Пример преобразования:

```
CC = g++
CFLAGS = -O2 -Wall
main: main.o
    $(CC) main.o -o main
main.o: main.cpp
    $(CC) $(CFLAGS) -c main.cpp
```

CMakeLists.txt (сгенерированный):

```
cmake_minimum_required(VERSION 3.10)
project(MyProject)
set(CMAKE_CXX_COMPILER g++)
add_executable(main main.cpp)
target_compile_options(main PRIVATE -O2 -Wall)
```

Особенности: Инструмент ориентирован на Makefile, но не поддерживает условную компиляцию (`ifdef`). Ограниченная поддержка сложных Makefile с нестандартными правилами или динамическими зависимостями, несмотря на настраиваемые параметры. Не работает с qmake, так как предназначен только для Make.

Преимущества:

- Удобство для простых Makefile.
- Простота интеграции.

Недостатки:

- Ограниченная поддержка сложных Makefile.
- Отсутствие поддержки qmake.

Поддержка функциональных требований qmake:

- **Не поддерживается:** Нет поддержки qmake, исключая QT, FORMS, SUBDIRS.
- **Частичная поддержка:** Аналоги SOURCES, HEADERS, QMAKE_CXXFLAGS, LIBS.

2.1.5. CMakeify

[4] **Как работает конвертация:**

1. Анализ Makefile: Парсятся цели, зависимости и команды с помощью лексического анализатора. Извлекаются исходные файлы, флаги компиляции (`CFLAGS`, `CXXFLAGS`) и библиотеки (`LIBS`).
2. Сопоставление с CMake: Переменные и команды преобразуются в `add_executable()`, `add_library()`, `target_compile_options()`, `target_link_libraries()`. Зависимости и пути включаются через `target_include_directories()`.
3. Генерация CMakeLists.txt: Создаётся файл, сохраняющий структуру проекта, включая пути к файлам и базовые настройки. Инструмент ориентирован на простые и средние по сложности Makefile.

Пример преобразования:

```
# Makefile
my_program: main.c utils.c
gcc -O2 -Wall -o my_program main.c utils.c -lm
```

CMakeLists.txt (сгенерированный):

```
^^Icmake_minimum_required(VERSION 3.10)
^^Iproject(MyProject)
^^Iadd_executable(my_program main.c utils.c)
^^Itarget_compile_options(my_program PRIVATE -O2 -Wall)
^^Itarget_link_libraries(my_program PRIVATE m)
```

Особенности: Поддерживает только Makefile, ориентирован на автоматизацию базовых проектов. Не работает с SCons (вопреки некоторым предположениям) или qmake, так как не обрабатывает Python-скрипты или Qt-специфичные конструкции.

Преимущества:

- Автоматизация для простых и средних Makefile.
- Генерация минималистичных `CMakeLists.txt`.

Недостатки:

- Отсутствие поддержки qmake.
- Ограниченная обработка сложных Makefile с нестандартными правилами.
- Минимальная документация.

Поддержка функциональных требований qmake:

- **Не поддерживается:** QT, FORMS, RESOURCES, SUBDIRS, QMAKE_EXTRA_COMPILERS, условные конструкции (например, `CONFIG(debug, debug|release)`).
- **Частичная поддержка:** Аналоги SOURCES, HEADERS, QMAKE_CXXFLAGS, LIBS могут быть обработаны, если qmake-проект экспортирован в Makefile, но без Qt-специфики.

2.1.6. BuildMigrator

[8] **Как работает конвертация:**

1. Запуск Make с логированием: Инструмент выполняет `make` с флагами `--trace` или `--debug`, генерируя подробные логи сборки. Логи содержат команды компиляции (например, `gcc -c main.c`), зависимости (`main.o: main.c utils.h`) и флаги (`-O2`, `-I`).

2. Анализ логов: Парсер извлекает команды, зависимости и флаги. Например, строка `gcc -O2 -Wall -Iinclude -c main.c -o main.o` разбивается на компилятор (`gcc`), флаги (`-O2 -Wall -Iinclude`) и файлы (`main.c`, `main.o`).

3. Сопоставление с CMake: Команды преобразуются в `add_executable()` или `add_library()`. Флаги добавляются через `target_compile_options()`, а зависимости — через `target_link_libraries()`. Условные конструкции (`ifdef`) частично поддерживаются, если они отражены в логах.

4. Генерация CMakeLists.txt: Создаётся файл с командами для сборки, включая исходные файлы, флаги, зависимости и структуру проекта.

Пример преобразования:

```
gcc -O2 -Wall -Iinclude -c main.c -o main.o
gcc -O2 -Wall -Iinclude -c utils.c -o utils.o
gcc main.o utils.o -o my_program
```

CMakeLists.txt (сгенерированный):

```
cmake_minimum_required(VERSION 3.10)
project(MyProject)
add_executable(my_program main.c utils.c)
target_compile_options(my_program PRIVATE -O2 -Wall)
target_include_directories(my_program PRIVATE include)
```

Особенности: Использование логов делает инструмент гибким для сложных Makefile, включая проекты с динамическими зависимостями. Требуется выполнение сборки для генерации логов. Не поддерживает qmake, так как ориентирован на Make.

Преимущества:

- Гибкость для сложных Makefile.
- Поддержка динамических зависимостей.

Недостатки:

- Зависимость от логов сборки.
- Отсутствие поддержки qmake.

Поддержка функциональных требований qmake:

- **Не поддерживается:** Нет поддержки qmake, исключая QT, FORMS, SUBDIRS.
- **Частичная поддержка:** Аналоги SOURCES, HEADERS, QMAKE_CXXFLAGS, LIBS через анализ логов.

2.1.7. qmake2cmake

[15] **Как работает конвертация:**

1. Парсинг `*.pro` файлов: Инструмент, реализованный на Python, читает qmake файлы, используя синтаксический анализатор для извлечения переменных (`SOURCES`, `HEADERS`, `QT`, `FORMS`), настроек компиляции (`QMAKE_CXXFLAGS`) и зависимостей (`LIBS`). Поддерживаются сложные конструкции, такие как `SUBDIRS`.
2. Анализ Qt-специфичных конструкций: Файлы `.ui` преобразуются в `qt5_wrap_ui()`, `.qrc` — в `qt5_add_resources()`. Модули Qt (`QT += core gui`) преобразуются в `find_package(Qt5 ...)`. Условные конструкции (`CONFIG(debug, debug|release)`) обрабатываются частично.

3. Сопоставление с CMake: Переменные преобразуются: `TEMPLATE` → `add_executable()`/`add_library()`, `TARGET` → `set_target_properties(... OUTPUT_NAME ...)`. Флаги и зависимости добавляются через `target_compile_options()` и `target_link_libraries()`.

4. Генерация CMakeLists.txt: Создаётся файл, включающий Qt-модули, исходные файлы, зависимости и подпроекты. Инструмент генерирует иерархию `CMakeLists.txt` для проектов с `SUBDIRS`.

Пример преобразования:
CMakeLists.txt (сгенерированный):

```
cmake_minimum_required(VERSION 3.10)
project(MyApp)
set(CMAKE_BUILD_TYPE Debug)
find_package(Qt5 COMPONENTS Core Gui REQUIRED)
add_executable(MyApp main.cpp widget.cpp)
target_sources(MyApp PRIVATE widget.h)
qt5_wrap_ui(UI_HEADERS widget.ui)
target_sources(MyApp PRIVATE ${UI_HEADERS})
target_link_libraries(MyApp PRIVATE Qt5::Core Qt5::Gui)
```

Особенности: Инструмент специализирован на qmake и Qt-проектах, поддерживает большинство Qt-специфичных конструкций. Ограниченная поддержка сложных условных конструкций и специфичных настроек, таких как `PRECOMPILED_HEADER`.

Преимущества:

- Полная поддержка Qt-проектов.
- Простота интеграции с Qt.

Недостатки:

- Ограниченная поддержка сложных qmake файлов.
- Отсутствие поддержки других систем сборки.

Поддержка функциональных требований qmake:

- **Полная поддержка:** TEMPLATE, TARGET, CONFIG, SOURCES, HEADERS, FORMS, RESOURCES, QT, SUBDIRS, QMAKE_CXX, QMAKE_CXXFLAGS, LIBS.
- **Частичная поддержка:** CONFIG(debug, debug|release), win32 | unix | macx, MOC_DIR, UI_DIR, RCC_DIR.
- **Не поддерживается:** PRECOMPILED_HEADER, DISTFILES, VERSION.

2.1.8. pro2cmake.py

[14] **Как работает конвертация:**

1. Парсинг `*.pro` файлов: Python-скрипт читает qmake файлы, используя регулярные выражения для извлечения переменных (`SOURCES`, `QT`, `FORMS`, `TEMPLATE`, `TARGET`). Поддерживаются базовые конструкции, но сложные, такие как `win32 | unix | macx`, часто игнорируются.
2. Анализ Qt-конструкций: Файлы `.ui` преобразуются в `qt5_wrap_ui()`, `.qrc` — в `qt5_add_resources()`. Модули Qt (`QT += core gui`) преобразуются в `find_package(Qt5 ...)`.
3. Сопоставление с CMake: Переменные преобразуются в команды CMake: `SOURCES` → `target_sources(...)`, `LIBS` → `target_link_libraries(...)`. Условные конструкции обрабатываются ограниченно.
4. Генерация CMakeLists.txt: Создаётся файл с базовыми командами для Qt-проектов, ориентированный на небольшие проекты.

Пример преобразования:
CMakeLists.txt (сгенерированный):

```
cmake_minimum_required(VERSION 3.10)
project(MyApp)
find_package(Qt5 COMPONENTS Core Gui REQUIRED)
add_executable(MyApp main.cpp)
qt5_wrap_ui(UI_HEADERS main.ui)
target_sources(MyApp PRIVATE ${UI_HEADERS})
target_link_libraries(MyApp PRIVATE Qt5::Core Qt5::Gui)
```

Особенности: Лёгкий инструмент для небольших Qt-проектов, но не поддерживает сложные qmake файлы или нестандартные конструкции. Подходит для простых приложений.

Преимущества:

- Простота и лёгкость.
- Поддержка базовых Qt-проектов.

Недостатки:

- Ограниченная поддержка сложных qmake конструкций.
- Отсутствие поддержки Makefile или других систем.

Поддержка функциональных требований qmake:

- **Полная поддержка:** TEMPLATE, TARGET, SOURCES, HEADERS, FORMS, RESOURCES, QT, QMAKE_CXXFLAGS, LIBS.
- **Частичная поддержка:** CONFIG, SUBDIRS.
- **Не поддерживается:** MOC_DIR, UI_DIR, RCC_DIR, PRECOMPILED_HEADER, DISTFILES, VERSION, win32 | unix | macx.

2.1.9. QMake2CMake (davidtazy)

[19] **Как работает конвертация:**

1. Парсинг `*.pro` файлов: Инструмент использует продвинутый парсер для анализа qmake файлов, извлекая переменные (`SOURCES`, `QT`, `FORMS`), настройки и зависимости. Поддерживаются сложные конструкции, такие как `SUBDIRS` и `CONFIG(debug, debug|release)`.

2. Анализ Qt-конструкций: Файлы `.ui`, `.qrc` и модули Qt преобразуются в `qt5_wrap_ui()`, `qt5_add_resources()` и `find_package(Qt5 ...)`. Условные конструкции обрабатываются с использованием `if(...)` в CMake.

3. Сопоставление с CMake: Переменные преобразуются в команды CMake: `TEMPLATE` → `add_executable()`/`add_library()`, `SUBDIRS` → `add_subdirectory()`. Флаги и зависимости добавляются через `target_compile_options()` и `target_link_libraries()`.

4. Генерация CMakeLists.txt: Создаётся файл с полной поддержкой Qt, включая подпроекты и сложные конфигурации.

Пример преобразования:
CMakeLists.txt (сгенерированный):

```
cmake_minimum_required(VERSION 3.10)
project(MyApp)
set(CMAKE_BUILD_TYPE Debug)
find_package(Qt5 COMPONENTS Core Gui REQUIRED)
add_executable(MyApp main.cpp widget.cpp)
qt5_wrap_ui(UI_HEADERS widget.ui)
target_sources(MyApp PRIVATE ${UI_HEADERS})
target_link_libraries(MyApp PRIVATE Qt5::Core Qt5::Gui)
if(CMAKE_BUILD_TYPE MATCHES Debug)
    target_compile_definitions(MyApp PRIVATE DEBUG_MODE)
endif()
```

Особенности: Активно развивается, поддерживает сложные qmake проекты и Qt. Подходит для крупных Qt-приложений, но документация ограничена.

Преимущества:

- Полная поддержка Qt и сложных qmake файлов.
- Активное развитие.

Недостатки:

- Ограниченная документация.
- Отсутствие поддержки других систем сборки.

Поддержка функциональных требований qmake:

- **Полная поддержка:** TEMPLATE, TARGET, CONFIG, SOURCES, HEADERS, FORMS, RESOURCES, QT, SUBDIRS, QMAKE_CXX, QMAKE_CXXFLAGS, LIBS.
- **Частичная поддержка:** CONFIG(debug, debug|release), win32 | unix | macx, MOC_DIR, UI_DIR, RCC_DIR.
- **Не поддерживается:** PRECOMPILED_HEADER, DISTFILES, VERSION.

2.2. Сводная таблица сравнения инструментов

Ниже представлена сводная таблица, сравнивающая рассмотренные инструменты по критериям обзора. Для функциональных требований qmake указаны оценки поддержки каждого требования (+: полная поддержка, ±: частичная поддержка, -: отсутствие поддержки).

Критерий	qmake2cmake	pro2cmake.py	QMake2CMake	CMake Converter	Autotools-to-CMake	Make2CMake	make2cmake	CMakeify	BuildMigrator
Качество архитектуры и автоматизация	• Архитектура: маппинг конструкций qmake, поддержка Qt-специфичных переменных. • Высокая автоматизация, читаемый CMakeLists.txt.	• Архитектура: маппинг .pro файлов для Qt-репозиториев. • Хорошая автоматизация, но ограничена Qt.	• Архитектура: маппинг .pro файлов, JSON-конфигурация. • Хорошая автоматизация, читаемый CMakeLists.txt.	• Архитектура: парсинг Visual Studio проектов. • Высокая автоматизация для Visual Studio.	• Архитектура: парсинг Autotools файлов. • Хорошая автоматизация для Autotools.	• Архитектура: парсинг Makefile. • Низкая автоматизация для Qt.	• Архитектура: парсинг Makefile, учебный проект. • Низкая автоматизация, только простые проекты.	• Архитектура: парсинг Makefile. • Хорошая автоматизация для Makefile.	• Архитектура: миграция различных систем. • Низкая автоматизация, требует доработки.
Расширяемость	Ограниченная (открытый код).	Ограниченная (открытый код).	Поддержка через JSON.	Ограничена Visual Studio.	Умеренная, ограничена Autotools.	Ограничена Makefile.	Ограничена Makefile.	Умеренная для Makefile.	Через конфигурации и плагины.
Документация и поддержка	Хорошая: README, баг-трекер.	Отсутствует: только исходный код.	Ограниченная: примеры, редкие обновления.	Хорошая: документация для Visual Studio.	Хорошая: документация для Autotools.	Минимальная: для Makefile.	Минимальная: README.	Минимальная: README.	Ограниченная: документация, неактивная поддержка.
Поддерживаемые системы сборки	qmake (.pro, .pri, Qt 6+).	qmake (.pro, Qt-репозитории).	qmake (.pro, Qt5).	Visual Studio (.sln, .vcxproj).	Autotools (configure.ac, Makefile.am).	Makefile.	Makefile.	Makefile.	Makefile, ninja, MSBuild, strace.
Недостатки	• Не поддерживает Qt 5. • Проблемы с многоуровневыми SUBDIRS. • Ограниченная поддержка сложных условий.	• Ориентирован на Qt-репозитории. • Устарел, заменён qmake2cmake. • Баги с .qmake.conf.	• Проблемы с многоуровневыми SUBDIRS. • Ошибки в сложных условиях. • Требуется JSON для нестандартных случаев.	• Не поддерживает qmake и Qt.	• Не поддерживает qmake и Qt.	• Теряет Qt-данные. • Не поддерживает .ui, .qrc, SUBDIRS.	• Не поддерживает qmake и Qt. • Ограничена простыми Makefile. • Неактивная разработка.	• Не поддерживает qmake и Qt.	• Нет поддержки qmake и Qt. • Требуется ручная доработка.
Основные переменные проекта	+	+	+	-	-	-	-	-	-
Исходные файлы и артефакты	+	+	+	-	-	±	±	±	±
Настройки компиляции	+	+	+	-	-	±	±	±	±
Настройки линковки	+	+	+	-	-	±	±	±	±
Qt-специфичные переменные	+	+	+	-	-	-	-	-	-
Подпроекты	±	±	±	-	-	-	-	-	-
Условные конструкции	±	±	±	-	-	-	-	-	-

Рис. 1: Сводная таблица

2.3. Обзор используемых технологий

Make [6] – это одна из самых старых и широко используемых систем сборки, которая была создана для автоматизации компиляции программ. Она особенно популярна в проектах на языках C и C++, хотя может использоваться и для других языков программирования. Основное назначение Make — автоматизация процесса сборки исходного кода в исполняемые файлы, обеспечивая последовательность выполнения команд, таких как компиляция, линковка и генерация библиотек.

qmake [17] — это система сборки, разработанная в рамках проекта Qt и предназначенная для упрощения процесса компиляции и сборки приложений, использующих фреймворк Qt. Она позволяет автоматически генерировать Makefile-файлы на основе описания проекта в ‘.pro‘-файлах, где задаются исходные файлы, зависимости, параметры компиляции и линковки. qmake особенно удобен для кросс-платформенной разработки, так как абстрагирует детали конкретной платформы и позволяет собирать один и тот же проект на разных операционных системах. Благодаря тесной интеграции с Qt и простому

синтаксису, qmake широко используется для быстрой настройки и сборки небольших и средних проектов, особенно в контексте разработки графических приложений.

CMake [9] — это ключевой инструмент для конечной сборки проектов после их конвертации. В рамках работы CMake используется как целевая система сборки, обеспечивающая возможность управления процессом сборки, конфигурации и генерации исполняемых файлов. Он предоставляет инструменты для получения абстрактного синтаксического дерева (AST) сборки проекта, что позволяет анализировать структуру и конфигурацию проекта, упрощая процесс миграции и интеграции.

Eclipse [5] — это мощная и гибкая интегрированная среда разработки (IDE), широко используемая для разработки программного обеспечения на различных языках программирования, таких как Java, C, C++, Python и других. Платформа Eclipse предоставляет богатую функциональность, включая поддержку различных систем сборки, таких как CMake, а также интеграцию с системами управления версиями, дебаггерами и профайлерами. Важным преимуществом Eclipse является его расширяемость через плагины, что позволяет настроить среду под специфические нужды разработчиков. Благодаря поддержке многочисленных инструментов и плагинов, Eclipse идеально подходит для работы над большими проектами и разработки на различных платформах, обеспечивая удобную работу с кодом, тестированием и отладкой.

2.4. Выводы

На основе анализа инструментов для конвертации систем сборки в CMake сделаны следующие выводы. Для Makefile хорошим вариантом является BuildMigrator, который обеспечивает гибкость и поддержку сложных проектов. Для qmake оптимальны qmake2cmake и QMake2CMake (davidtazy), предоставляющие полную поддержку

Qt-специфичных конструкций и сложных проектов. Однако, учитывая основное требование компании — возможность расширяемой поддержки как Makefile, так и qmake в рамках единого инструмента, был выбран BuildMigrator, так как он демонстрирует потенциал для дальнейшей адаптации под обе системы сборки.

3. Ход работы

3.1. Архитектура BuildMigrator

BuildMigrator — расширяемый инструмент, разработанный для автоматизации миграции проектов с систем сборки, основанных на Make (или аналогичных, таких как Ninja), на CMake. Его ключевая особенность — использование логов сборки для анализа и построения абстрактной модели проекта (Build Object Model, BOM), которая затем преобразуется в файл `CMakeLists.txt`. Ниже описан полный pipeline работы инструмента, включая конфигурацию, парсеры, пресеты и внутренние механизмы.

Шаг 1: Конфигурация и запуск системы сборки

BuildMigrator начинает работу с запуска системы сборки (например, `make` или `ninja`) с включённым логированием. Для этого используются флаги, такие как `--trace` или `--debug` в случае Make, которые обеспечивают вывод подробной информации о каждом этапе сборки.

Команда запуска:

```
/BuildMigrator/bin/build_migrator --commands build \
    --source_dir "path_to_source_dir" \
    --out_dir "path_to_out_dir" \
    --build_command "make --trace -C path_to_makefile" 2>&1\
    --log_provider CONSOLE
```

Флаг `--trace` записывает в лог все команды компиляции, зависимости и параметры, включая вызовы компилятора (`gcc`, `g++`), флаги (`-O2`, `-Iinclude`) и файлы ввода-вывода.

Логи сохраняются в файл (например, `build.log`) или передаются через стандартный ввод. Поддерживаются сложные сценарии, где сборка выполняется в несколько этапов.

Шаг 2: Парсинг логов

`BuildMigrator` поддерживает пресеты — конфигурации, определяющие тип логов и набор активных парсеров. Пресет задаётся через JSON-файл или аргументы командной строки, такие как `--log_type`.

Пользователь может задать пресеты через конфигурационный файл, чтобы указать:

- тип платформы (`linux`, `windows`);
- тип системы сборки (`Make`, `Ninja`);
- путь к компилятору или специфические настройки (например, кросс-компиляция);
- игнорируемые флаги (например, -m32 (архитектура), -O2 (оптимизация));
- дополнительные флаги для логирования (например, `--debug=v`).

```
{
    "platform": "linux",
    "parsers": [
    "clang_gcc",
    "gnu_ar",
    "gnu_cp_ln_mv",
    "libtool",
    "objcopy"
    ],
    "delete_flags": [
    "^-m(32|64)$",
    "^-O[123sg]$",
    "^-DNDEBUG$",
    "^--sysroot",
    "^-x c"
    ]

}
```

Парсер BuildMigrator считывает логи построчно, используя кроссплатформенное чтение с `io.open(newline=None)` для обработки логов с любыми окончаниями строк. Каждая строка передаётся набору парсеров, которые обновляют глобальную переменную `target` — словарь, содержащий данные о текущей цели сборки (например, исполняемый файл или библиотека).

Структура `target` включает поля:

- `type` — тип цели (`executable`, `library`, `file`);
- `sources` — список исходных файлов (например, `["main.cpp", "utils.cpp"]`);
- `output` — выходной файл (например, `"build/my_program"`);
- `compile_flags` — флаги компиляции (например, `["-Wall", "-O2"]`);
- `dependencies` — зависимости (например, библиотеки или файлы);
- `working_dir` — рабочая директория для цели.

BuildMigrator использует модульную систему парсеров, где каждый парсер имеет приоритет (целое число, меньшее значение означает более ранний вызов). Парсеры обрабатывают строки логов последовательно, передавая `target` от одного к другому. Пресет определяет, какие парсеры активны, но `build_log_parser` (приоритет 0) включается всегда.

Строка лога считывается и передаётся в `build_log_parser` (приоритет 0).

- нормализация путей (`normalize_path`) с учётом `source_dir` и `build_dirs`;
- управление `target` через `current_target`, включая регистрацию целей (`register_target`);

- обработка аргументов, таких как `--targets`, `--capture_sources`, для фильтрации целей.

Если строка не обработана, она передаётся парсерам с более высокими приоритетами (например, `clang_gcc` с приоритетом 7).

Парсер `clang_gcc` автоматически обрабатывает строки компиляции или линковки, соответствующие `gcc`, `g++`, `clang`, или `clang++`. Он использует `ArgumentParserEx` для токенизации строк, извлекая:

- исходные файлы (`.c`, `.cpp`, `.s`);
- флаги компиляции (`-Wall`, `-O2`, `-I`);
- флаги линковки (`-L`, `-l`);
- выходной файл (`-o`);
- библиотеки и зависимости.

Каждый парсер может модифицировать `target` или вернуть новый список целей.

Пример последовательности для строки `g++ -Wall -O2 -c main.cpp -o main.o`:

- `build_log_parser` инициализирует `target` с минимальной структурой, содержащей строку лога `{"line": "g++ -Wall -O2 -c main.cpp -o main.o"}`;
- `clang_gcc` распознаёт строку, добавляя `sources=["main.cpp"]`, `compile_flags=["-Wall", "-O2"]`, `output="main.o"`.

После обработки всех строк логов `target` для каждой цели включается в BOM — список целей, хранящийся в `context.targets`. BOM содержит полные данные о целях, включая `type`, `sources`, `output`, `compile_flags`, `dependencies`.

BOM сохраняется в файл `bom.pickle` и может быть экспортирован для повторных запусков или отладки.

Шаг 3: Оптимизация Build Object Model (BOM)

После построения **Build Object Model** (BOM) **BuildMigrator** выполняет оптимизацию модели, устраняя избыточные и дублирующиеся элементы. Этот процесс направлен на минимизацию сложности графа сборки, повышение точности и подготовку BOM к преобразованию в команды CMake.

Результат оптимизации: BOM преобразуется в упрощённый граф, где каждый узел (файл, цель, команда) представлен единожды, а рёбра (зависимости) минимизированы. Например, вместо двух узлов для `main.o` с флагами `-Iinclude -Wall` и `-Iinclude -O2`, создаётся один узел с объединёнными уникальными флагами `-Iinclude -Wall -O2`.

Шаг 4: Преобразование BOM в CMake-команды

На основе BOM **BuildMigrator** генерирует эквивалентные CMake-команды, преобразуя элементы сборки в соответствующие конструкции.

Маппинг объектов:

- исходные файлы преобразуются в списки для `add_executable()` или `add_library()`;
- флаги компиляции мапятся на `target_compile_options()` или `target_compile_definitions()`;
- зависимости преобразуются в `target_link_libraries()` или `target_include_directories()`.

Обработка сложных случаев:

- *динамические файлы:* генерируемые файлы (например, через `bison`) добавляются через `add_custom_command()`;
- *множественные цели:* создаются отдельные `add_executable()` и `add_library()` с зависимостями.

Шаг 5: Генерация CMakeLists.txt

После преобразования BOM в CMake-команды инструмент создаёт файл `CMakeLists.txt`, который включает минимальную версию CMake, имя проекта, списки файлов, целей, флагов и зависимостей.

Подготовка данных: перед генерацией `CMakeLists.txt` выполняется фильтрация целей, исключая избыточные или помеченные атрибутом `skip` (например, ненужные директории), для обеспечения корректности выходного файла.

Итеративная генерация: повторный запуск с новыми логами или пресетами уточняет результат (например, с флагом `--always-make`).

Технические детали

Производительность: Парсер оптимизирован для больших логов благодаря потоковому анализу.

Поддерживаемые платформы: Linux, macOS, Windows.

Зависимости: Требуются CMake и система сборки (Make или Ninja).

Особенность: В отличие от других инструментов, представленных в обзоре, требует предварительного запуска системы сборки.

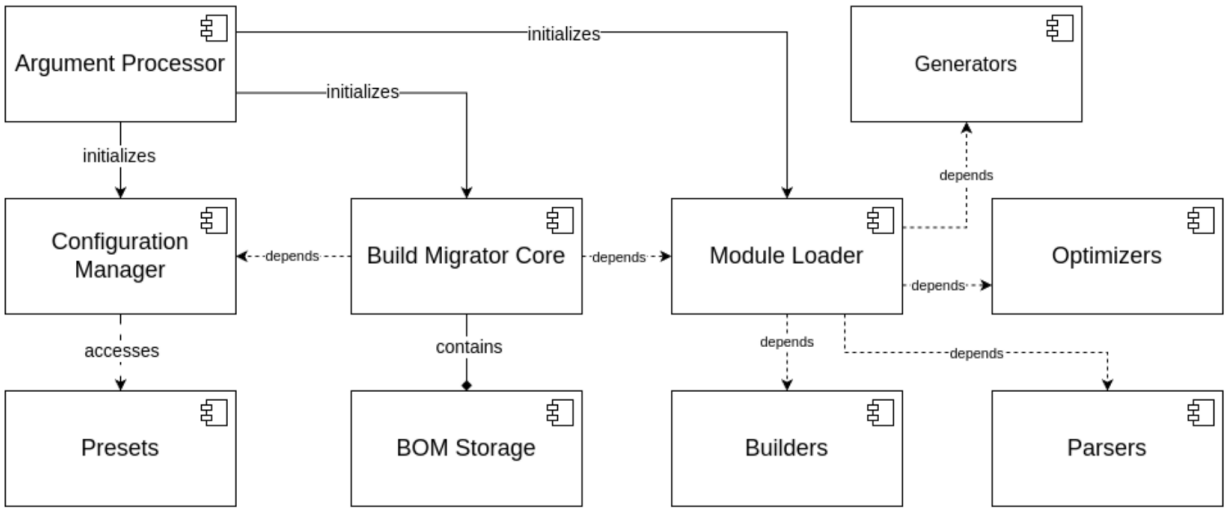

Рис. 2: Диаграмма компонентов BuildMigrator

Argument Processor (`main.py`):

- отвечает за обработку аргументов командной строки (`--commands`, `--out_dir`, `--presets`), задающих параметры работы инструмента;
- инициализирует `ModuleLoader` для загрузки модулей и `SettingsLoader` для управления настройками;
- зависимости: `Configuration Manager` (для загрузки пресетов из JSON-файлов) и `BuildMigrator` (для выполнения указанных команд).

Configuration Manager (`settings.py: SettingsLoader`):

- загружает и объединяет конфигурационные пресеты из внешних JSON-файлов, содержащих настройки, такие как `--log_type` и `--commands`;
- предоставляет конфигурационные параметры для всех этапов конвертации: сборка (`build`), парсинг (`parse`), оптимизация (`optimize`) и генерация (`generate`);
- зависимости: внешние JSON-файлы с пресетами.

Module Loader (`modules.py: ModuleLoader`):

- обеспечивает динамическую загрузку модулей, реализующих функциональность сборки (`builders`), парсинга (`parsers`), оптимизации (`optimizers`) и генерации (`generators`);
- создаёт экземпляры модулей через внутренний механизм `_Initializer`, основанный на их интерфейсах (`Builder`, `Parser`, `Optimizer`, `Generator`, `EntryPoint`).

Build Migrator Core (`core.py: BuildMigrator`):

- центральный компонент, координирующий этапы конвертации: сборка исходного проекта, парсинг логов, оптимизация модели и генерация файлов `CMakeLists.txt`;
- использует модули, загруженные через `ModuleLoader`, и настройки, предоставленные `SettingsLoader`;
- зависимости: `Module Loader`, `Configuration Manager`, `BOM Storage`.

BOM Storage (`core.py: BuildMigrator.save/load_build_object_model`

- хранит Build Object Model (BOM) — промежуточное представление структуры проекта — в файле `bom.pickle`;
- обеспечивает сохранение и загрузку BOM для последующего использования на этапах конвертации;
- зависимости: `Build Migrator Core`.

Модули: Builders, Parsers, Optimizers, Generators (`modules.py`):

- реализуют интерфейсы `Builder`, `Parser`, `Optimizer`, `Generator` и `EntryPoint`, выполняя специализированные задачи на каждом этапе конвертации;
- пример: модули `Parsers` (включая `build_log_parser`, `Clang_Gcc`) анализируют логи сборки, формируя BOM на основе данных о компиляции;
- управляются `Module Loader` и используются `Build Migrator Core`.

3.2. Поддержка qmake в BuildMigrator

BuildMigrator не поддерживает прямую конвертацию файлов qmake, используемых в проектах на основе Qt. Это ограничение потребовало выбора подхода для обеспечения совместимости BuildMigrator с qmake-based проектами. Были рассмотрены два варианта: генерация Makefile с помощью qmake с последующей передачей в BuildMigrator и разработка собственного парсера логов qmake.

1. Генерация Makefile из qmake Первый подход предполагает использование qmake для генерации Makefile, который затем обрабатывается BuildMigrator для создания `CMakeLists.txt`.

Преимущества:

- упрощённая реализация: использование встроенного механизма qmake для генерации Makefile минимизирует необходимость дополнительной разработки;
- совместимость с BuildMigrator: Makefile предоставляет стандартизированные данные о сборке, которые инструмент может обработать без модификаций.

Недостатки:

- ограниченная гибкость: поддержка различных конфигураций сборки (например, debug/release) требует дополнительных настроек qmake или ручной корректировки Makefile;
- потеря Qt-специфичных данных: конструкции, такие как `FORMS` или `SUBDIRS`, могут быть некорректно преобразованы в Makefile, что снижает точность конвертации;
- вспомогательный характер подхода: метод подходит только для простых проектов и требует ручной доработки для сложных Qt-приложений, что ограничивает его универсальность.

2. Разработка парсера логов qmake Второй подход заключается в создании собственного парсера для анализа логов, генерируемых qmake, с последующей интеграцией в процесс BuildMigrator.

Преимущества:

- соответствие архитектуре BuildMigrator: парсер логов qmake работает аналогично встроенным парсерам для Make, обеспечивая единообразный процесс;
- полная поддержка qmake: возможность обработки Qt-специфичных конструкций (`FORMS`, `SUBDIRS`, `RESOURCES`);
- парсер можно адаптировать под различные версии qmake и нестандартные проекты.

Недостатки:

- временные затраты: реализация и тестирование парсера займут больше времени, чем использование Makefile;
- высокие трудозатраты: требуется разработка логики для анализа логов и их преобразования в BOM;
- риски ошибок: неправильная интерпретация логов может привести к некорректной генерации CMake.

Сравнение и выбор подхода Сравнение подходов проводилось с учётом целей работы, требований заказчика и архитектурных особенностей BuildMigrator. Генерация Makefile через qmake является более простым и быстрым решением, одобренным заказчиком для базовых сценариев. Однако этот подход ограничивает возможности обработки сложных Qt-проектов.

Разработка собственного парсера, несмотря на высокие трудозатраты, имеет ряд преимуществ:

- **интеграция с BuildMigrator:** парсер логов qmake полностью соответствует модульной архитектуре инструмента, позволяя расширить его функциональность без изменения ядра;

- **полнота данных:** логи qmake содержат структурированную информацию о проекте, включая Qt-специфичные элементы, что обеспечивает точное преобразование в CMake;
- **гибкость:** парсер можно дорабатывать для поддержки дополнительных систем сборки или специфичных требований заказчика.

После обсуждения с заказчиком и научным руководителем было принято решение выбрать разработку парсера как основной подход. Это решение позволяет не только решить задачу конвертации qmake-проектов, но и внести вклад в развитие BuildMigrator, обеспечив его универсальность и расширяемость.

3.3. Разработка QmakeLogParser

3.3.1. Входные данные парсера

Парсер `QmakeLogParser` принимает на вход логи, генерируемые `qmake` с флагом отладки `-d`. Эти логи содержат детальную информацию о конфигурации проекта, включая пути к файлам, переменные и их значения. Формат логов хорошо структурирован и легко читаем, так как каждая строка включает уровень отладки (`DEBUG 1`), путь к файлу проекта (`.pro` или `.prf`), номер строки, название переменной и её значение. Это упрощает извлечение данных с помощью регулярных выражений.

Пример логов, генерируемых `qmake -d`:

```
DEBUG 1: /home/k0lba/Documents/make5/qmake.pro:4: TARGET := test_app
DEBUG 1: /home/k0lba/Documents/make5/qmake.pro:5: TEMPLATE := app
DEBUG 1: /home/k0lba/Documents/make5/qmake.pro:7: SOURCES := main.cpp
↪ mainwindow.cpp
DEBUG 1: /home/k0lba/Documents/make5/qmake.pro:8: HEADERS :=
↪ mainwindow.h
DEBUG 1: /home/k0lba/Documents/make5/qmake.pro:9: FORMS := mainwindow.ui
DEBUG 1: /home/k0lba/Documents/make5/qmake.pro:10: RESOURCES :=
↪ resources.qrc
DEBUG 1: /home/k0lba/Documents/make5/qmake.pro:11: DISTFILES :=
↪ sample.txt
DEBUG 1: /home/k0lba/Documents/make5/qmake.pro:14: version_h.input :=
↪ version_script.sh
DEBUG 1: /home/k0lba/Documents/make5/qmake.pro:15: version_h.output :=
↪ version.h
DEBUG 1: /home/k0lba/Documents/make5/qmake.pro:16: version_h.commands :=
↪ bash /home/k0lba/Documents/make5/version_script.sh
DEBUG 1: /home/k0lba/Documents/make5/qmake.pro:17: version_h.variable :=
↪ HEADERS
```

3.3.2. Принципы работы парсера

Парсер `QmakeLogParser` реализован как часть проекта `buildmigrator` и наследуется от базового класса `Parser`, что

обеспечивает единообразную интеграцию с другими компонентами системы. Он реализует функциональные требования по обработке логов `qmake`, включая извлечение информации о исходных файлах, заголовках, библиотеках, флагах компиляции и других параметрах проекта.

Основной принцип работы заключается в использовании регулярных выражений для анализа строк логов. Каждое регулярное выражение соответствует определённой переменной (например, `SOURCES`, `HEADERS`, `LIBS`) и извлекает её значение. Парсер поддерживает обработку сложных конструкций, таких как условные конфигурации (`CONFIG(debug, debug|release)`), платформенные настройки (`win32`, `unix`) и пользовательские компиляторы.

3.3.3. Интеграция с buildmigrator

Для интеграции с `buildmigrator` был добавлен файл конфигурации `preset.json`, который определяет тип логов (`qmake`) и список парсеров, включая `QmakeLogParser`. Этот файл вызывается через модуль `settings.py`, который обеспечивает запуск парсера в рамках общей цепочки обработки логов. Пример содержимого `preset.json`:

```
{
    "log_type": "qmake",
    "parsers": [
        "build_log_parser",
        "qmake"
    ]
}
```

Класс `QmakeLogParser` имеет приоритет `priority = 1`, что определяет порядок его выполнения в цепочке парсеров. Парсер инициализирует контекст проекта, хранит глобальные настройки (такие как пути, флаги, библиотеки) и формирует модель сборки (`build_object_model`).

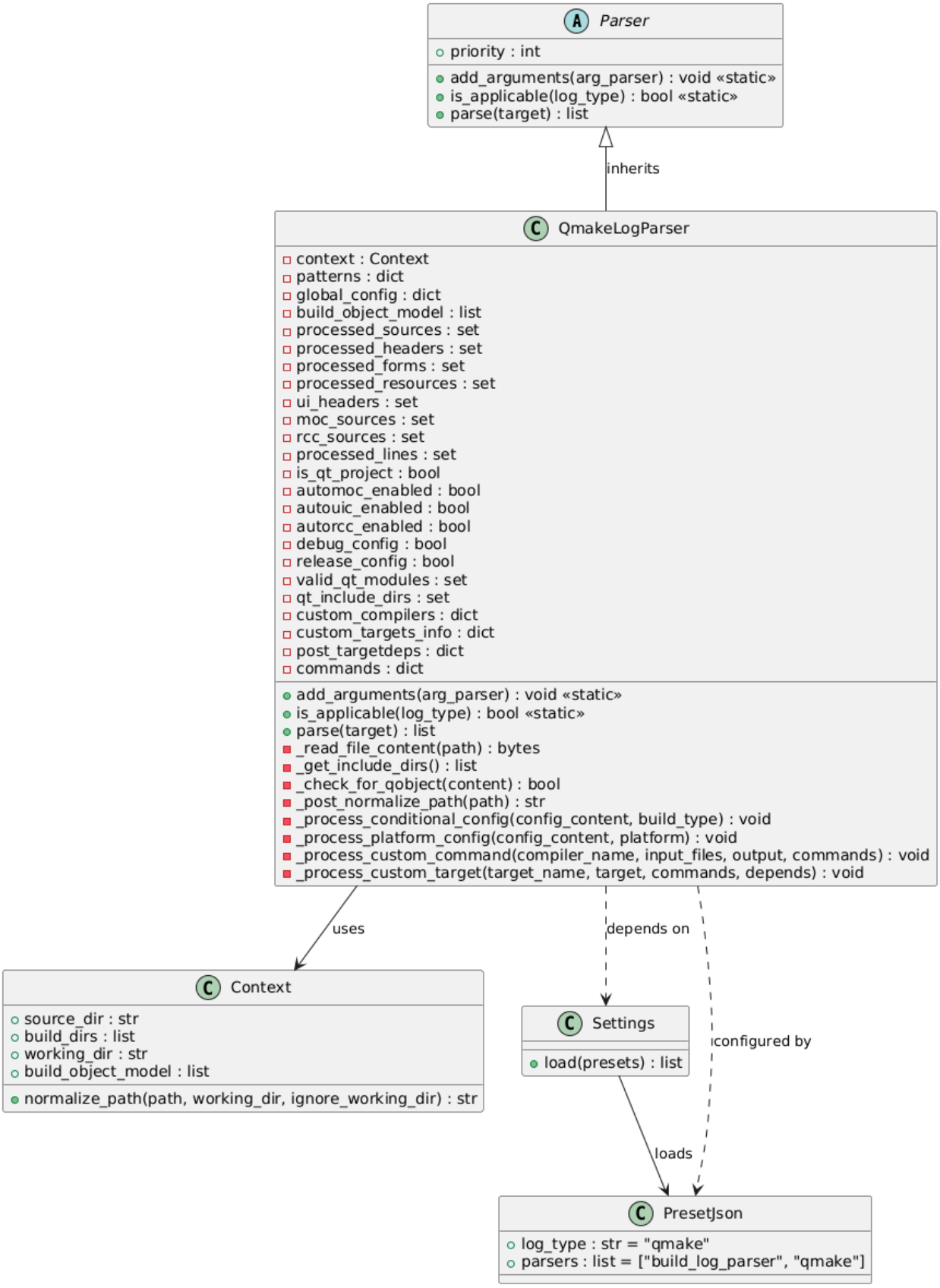

Рис. 3: Диаграмма классов

3.3.4. Формирование build_object_model

Результатом работы `QmakeLogParser` является `build_object_model` — список словарей, представляющих структуру сборки проекта. Каждый словарь описывает объект сборки с полями:

- `type`: Тип объекта (`module` или `file`).
- `module_type`: Тип модуля (`object_lib`, `executable`, `shared_library`, `test_executable`, `subdirs`).
- `name`: Имя модуля.
- `output`: Путь к выходному файлу.
- `sources`: Список словарей с информацией об исходных файлах:
  - `path`: Путь к файлу.
  - `language`: Язык программирования (`C++` или `None`).
  - `compile_flags`: Флаги компиляции (например, `[-std=c++17, -Wall]`).
  - `include_dirs`: Список директорий для включения.
- `objects`: Список объектных файлов (для исполняемых целей).
- `dependencies`: Список зависимостей.
- `compile_flags`: Общие флаги компиляции.
- `link_flags`: Флаги линковки (например, `[-Wl,-O1]`).
- `libs`: Список библиотек (например, `[fmt, Qt5Core]`).
- `include_dirs`: Директории для включения.
- `msvc_import_lib`: Библиотека импорта для MSVC (`None`, если не применимо).
- `version`: Версия модуля.

- `compatibility_version`: Версия совместимости.
- `content`: Содержимое файла (если доступно, иначе `None`).

Модель `build_object_model` формируется динамически по мере обработки логов. Например, для исходных файлов (`SOURCES`) создаются записи с типом `file`, а для конечного модуля — запись с типом `module` и полным списком зависимостей, включая исходные файлы, заголовки, формы и ресурсы.

Ограничения реализации

Проблема с условной компиляцией

В процессе разработки `QMakeLogParser` выяснилось, что логи QMake не содержат данных о ветках условной компиляции, которые не были выполнены. Например, в `.pro`-файлах могут присутствовать конструкции вида `CONFIG(debug, debug|release)`, однако логи отражают только ту ветку, которая соответствует текущей конфигурации, например `debug` или `release`. Это ограничение препятствует корректному анализу всех возможных вариантов условной компиляции, что делает невозможной реализацию функциональных требований, связанных с полной поддержкой таких конструкций в BuildMigrator. Для преодоления данной проблемы был создан дополнительный парсер `.pro`-файлов. В отличие от `QMakeLogParser`, который обрабатывает логи, новый парсер непосредственно анализирует `.pro`-файл и извлекает полную структуру условных конструкций, включая как выполненные, так и невыполненные ветки. Такой подход обеспечил доступ к полной информации об условиях компиляции и позволил корректно формировать выходные данные для `CMakeLists.txt`. Заказчик рассмотрел предложенное решение и утвердил его.

Проблема с подпроектами

Другое ограничение связано с поддержкой подпроектов, расположенных в поддиректориях. Особенности запуска BuildMigrator изначально не предусматривали автоматическую обработку

подпроектов, находящихся в разных папках. Это создавало трудности при генерации корректных `CMakeLists.txt` для проектов со сложной структурой директорий, что является важным функциональным требованием. Для частичного решения этой проблемы в BuildMigrator была добавлена обработка поддиректорий. Была реализована логика парсинга строк, связанных с поддиректориями, а также добавлена возможность записи соответствующих путей в генераторе `CMakeLists.txt`. Однако полноценная поддержка подпроектов была достигнута в среде Eclipse CLDT. Подпроекты заранее идентифицируются, и BuildMigrator запускается отдельно для каждого найденного `.pro`-файла. Такой подход обеспечил полную обработку подпроектов и корректное формирование `CMakeLists.txt` для всей структуры проекта. Заказчик утвердил данное решение, признав его оптимальным для текущих задач и инфраструктуры.

3.4. Модификация генератора CMakeLists.txt

Данная часть работы была направлена на устранение выявленных проблем в сгенерированных файлах и улучшение их совместимости с IDE, а также на обеспечение корректной обработки Qt-специфичных компонентов.

3.4.1. Проблемы исходного генератора

Анализ сгенерированных файлов `CMakeLists.txt` выявил следующие недостатки:

Избыточные записи для исходных файлов: Для каждого исходного файла в `CMakeLists.txt` явно указывались индивидуальные параметры компиляции, такие как флаги и директории включения. Это усложняло чтение файла и его поддержку в IDE.

Сложные пути: Использование переменной `${CMAKE_CURRENT_LIST_DIR}` в путях к файлам затрудняло восприятие структуры проекта в IDE и приводило к избыточным деталям.

Некорректная обработка Qt-библиотек: Qt-библиотеки указывались в виде абсолютных путей (например, `/usr/lib/x86_64-linux-gnu/libQt5Widgets.so`), что снижало переносимость и не соответствовало современным практикам использования `CMake`.

Отсутствие динамической обработки Qt: Настройки для Qt (например, `find_package(Qt5)` и `CMAKE_AUTOMOC`) включались независимо от наличия Qt-компонентов, что приводило к избыточным зависимостям.

3.4.2. Реализованные решения

Для устранения выявленных проблем генератор `CMakeLists.txt` был существенно модернизирован. Основные изменения включали оптимизацию структуры файла, улучшение обработки Qt-компонентов и упрощение путей. Ниже описаны ключевые улучшения.

Оптимизация флагов и директорий Для устранения избыточных записей был реализован механизм группировки флагов компиляции и директорий включения. Общие флаги и директории, применимые ко всем исходным файлам модуля, извлекаются и задаются на уровне цели через команды `target_compile_options` и `target_include_directories`. Уникальные флаги и директории для отдельных файлов сохраняются только при необходимости.

Аналогичный подход применён для директорий включения, что позволило сократить объем сгенерированного кода и улучшить читаемость `CMakeLists.txt`.

Автоматическая обработка Qt-компонентов Для корректной работы с Qt-проектами были реализованы следующие улучшения:

Обработка MOC и FORM: Генератор автоматически распознаёт файлы, содержащие макрос `Q_OBJECT`, и активирует `CMAKE_AUTOMOC` для генерации `moc_*.cpp`. Файлы с расширением `.ui` добавляются в список источников и обрабатываются с помощью `CMAKE_AUTOUIC`.

Поддержка ресурсов: Файлы `.qrc` включаются в список источников, а `CMAKE_AUTORCC` активируется для автоматической обработки ресурсов.

Исключение сгенерированных файлов: Файлы `moc_*.cpp` и `ui_*.h` исключаются из явного списка источников, так как их генерация обрабатывается автоматически.

Упрощение путей Для упрощения путей в `CMakeLists.txt` переменная `${CMAKE_CURRENT_LIST_DIR}` была заменена на относительные пути, нормализованные относительно директорий проекта. Это улучшило читаемость и совместимость с IDE. IDE анализирует CMakeLists.txt для определения структуры проекта, включая файлы исходного кода, зависимости и настройки сборки, поэтому относительные пути упрощают индексацию и навигацию по проекту.

Модернизация обработки библиотек Qt-библиотеки, ранее указанные как абсолютные пути, были заменены на современные `CMake`-цели, такие как `Qt5::Widgets`, `Qt5::Gui`, `Qt5::Core`. Аналогично, библиотека `GL` заменена на `OpenGL::GL`, а `pthread` — на `Threads::Threads`.

Динамическое включение Qt Для минимизации избыточных зависимостей настройки Qt (например, `find_package(Qt5)`) включаются только при наличии `.ui`, `.qrc` или Qt-библиотек в проекте. Это реализовано через проверку списка источников и библиотек.

Обработка условной компиляции Для обработки условных конструкций в BuildMigrator реализована логика извлечения условий (`condition`) из Build_Object_Model. На основе этих данных формируются условные конструкции в CMakeLists.txt, обеспечивающие

корректное воспроизведение логики компиляции, заданной в исходных .pro-файлах.

3.4.3. Результаты модернизации

Модернизация генератора `CMakeLists.txt` позволила достичь следующих результатов: **Упрощение структуры**: Сгенерированные файлы стали компактнее за счёт группировки общих флагов и директорий, а также исключения избыточных записей.

Чистота структуры проекта: Сгенерированные Qt-файлы теперь корректно размещаются в сборочной директории благодаря `CMAKE_AUTOMOC`, `CMAKE_AUTOUIC` и `CMAKE_AUTORCC`.

Переносимость: Замена абсолютных путей библиотек на `CMake`-цели повысила кроссплатформенность сгенерированных файлов.

Гибкость: Динамическое включение Qt-зависимостей и поддержка пользовательских команд обеспечили адаптивность генератора к различным типам проектов.

```
project(PROJECT CXX)

list(APPEND CMAKE_MODULE_PATH ${CMAKE_CURRENT_LIST_DIR})
include(extensions)

set(SOURCE_DIR "${CMAKE_CURRENT_LIST_DIR}/source" CACHE PATH "")
configure_file(${CMAKE_CURRENT_LIST_DIR}/prebuilt/_build/ui_mainwindow.h ${CMAKE_CURRENT_BINARY_DIR}/_build/ui_mainwindow.h COPYONLY)
configure_file(${CMAKE_CURRENT_LIST_DIR}/prebuilt/_build/moc_mainwindow.cpp ${CMAKE_CURRENT_BINARY_DIR}/_build/moc_mainwindow.cpp COPYONLY)

set_source_files_properties(${SOURCE_DIR}/main.cpp PROPERTIES COMPILE_OPTIONS
    "-O2;-Wall;-Wextra;-D_REENTRANT;-fPIC;-DQT_NO_DEBUG;-DQT_WIDGETS_LIB;-DQT_GUI_LIB;-DQT_CORE_LIB"
)
set_source_files_properties(${SOURCE_DIR}/main.cpp PROPERTIES INCLUDE_DIRECTORIES
    "${SOURCE_DIR};${CMAKE_CURRENT_BINARY_DIR}/_build;/usr/include/x86_64-linux-gnu/qt5;/usr/include/x86_64-linux-gnu/qt5/QtWidgets;/usr/include/x86_64-linux-gnu/qt5/QtGui;/usr/include/x86_64-linux-gnu/qt5/QtCore;/usr/lib/x86_64-linux-gnu/qt5/mkspecs/linux-g++"
)
set_source_files_properties(${SOURCE_DIR}/mainwindow.cpp PROPERTIES COMPILE_OPTIONS
    "-O2;-Wall;-Wextra;-D_REENTRANT;-fPIC;-DQT_NO_DEBUG;-DQT_WIDGETS_LIB;-DQT_GUI_LIB;-DQT_CORE_LIB"
)
set_source_files_properties(${SOURCE_DIR}/mainwindow.cpp PROPERTIES INCLUDE_DIRECTORIES
    "${SOURCE_DIR};${CMAKE_CURRENT_BINARY_DIR}/_build;/usr/include/x86_64-linux-gnu/qt5;/usr/include/x86_64-linux-gnu/qt5/QtWidgets;/usr/include/x86_64-linux-gnu/qt5/QtGui;/usr/include/x86_64-linux-gnu/qt5/QtCore;/usr/lib/x86_64-linux-gnu/qt5/mkspecs/linux-g++"
)
set_source_files_properties(${CMAKE_CURRENT_BINARY_DIR}/_build/moc_mainwindow.cpp PROPERTIES COMPILE_OPTIONS
    "-O2;-Wall;-Wextra;-D_REENTRANT;-fPIC;-DQT_NO_DEBUG;-DQT_WIDGETS_LIB;-DQT_GUI_LIB;-DQT_CORE_LIB"
)
set_source_files_properties(${CMAKE_CURRENT_BINARY_DIR}/_build/moc_mainwindow.cpp PROPERTIES INCLUDE_DIRECTORIES
    "${SOURCE_DIR};${CMAKE_CURRENT_BINARY_DIR}/_build;/usr/include/x86_64-linux-gnu/qt5;/usr/include/x86_64-linux-gnu/qt5/QtWidgets;/usr/include/x86_64-linux-gnu/qt5/QtGui;/usr/include/x86_64-linux-gnu/qt5/QtCore;/usr/lib/x86_64-linux-gnu/qt5/mkspecs/linux-g++"
)
add_executable(03-GUI-Programming
    ${SOURCE_DIR}/main.cpp
    ${SOURCE_DIR}/mainwindow.cpp
    ${CMAKE_CURRENT_BINARY_DIR}/_build/moc_mainwindow.cpp
)
target_link_options(03-GUI-Programming PRIVATE -Wl,-O1)
target_link_libraries(03-GUI-Programming PRIVATE
    /usr/lib/x86_64-linux-gnu/libQt5Widgets.so
    /usr/lib/x86_64-linux-gnu/libQt5Gui.so
    /usr/lib/x86_64-linux-gnu/libQt5Core.so
    GL
    Threads::Threads
)
set_target_output_subdir(03-GUI-Programming RUNTIME_OUTPUT_DIRECTORY _build)
```

Рис. 4: CMakeLists.txt до модернизации

```
cmake_minimum_required(VERSION 3.13)

project(PROJECT CXX)

list(APPEND CMAKE_MODULE_PATH ${CMAKE_CURRENT_LIST_DIR})
include(extensions)

set(CMAKE_AUTOMOC ON)
set(CMAKE_AUTOUIC ON)
set(CMAKE_AUTORCC ON)

find_package(Qt5 COMPONENTS Core Gui Widgets REQUIRED)
find_package(OpenGL REQUIRED)
find_package(Threads REQUIRED)

add_executable(03-GUI-Programming main.cpp mainwindow.cpp mainwindow.ui)
target_link_options(03-GUI-Programming PRIVATE -Wl,-O1)
target_link_libraries(03-GUI-Programming PRIVATE Qt5::Core Qt5::Gui Qt5::Widgets OpenGL::GL Threads::Threads)
target_compile_options(03-GUI-Programming PRIVATE -D_REENTRANT -Wall -pipe -O2 -Wextra)
```

Рис. 5: CMakeLists.txt после модернизации

3.5. Интеграция BuildMigrator в IDE Eclipse CLDT

Для обеспечения автоматизации преобразования Qt-проектов, основанных на `.pro`-файлах, в `CMake`-проекты в рамках IDE Eclipse CLDT, была реализована интеграция инструмента `BuildMigrator`. Данная интеграция позволила упростить процесс миграции проектов, обеспечив интерфейс для пользователей и автоматическую настройку проекта в рабочей области Eclipse.

3.5.1. Функциональные требования

Интеграция `BuildMigrator` в Eclipse CLDT была выполнена с учётом следующих функциональных требований:

Окно для ввода данных пользователем:

- открытие окна при запуске процесса импорта проекта в Eclipse;
- возможность указания пути к директории проекта;
- возможность указания пути к `.pro`-файлу проекта.

Консоль для вывода логов `BuildMigrator`: Обеспечение отображения логов выполнения `BuildMigrator` для диагностики и отладки.

Автоматическое преобразование в `CMake`-проект:

- генерация файла `CMakeLists.txt` на основе `.pro`-файла;
- обработка всех зависимостей проекта, включая исходные файлы, библиотеки и Qt-компоненты.

Интеграция в Eclipse Workspace:

- добавление преобразованного проекта в рабочую область Eclipse;
- автоматическая настройка всех зависимостей и файлов проекта для корректной работы в IDE.

3.5.2. Подготовка среды

Перед началом интеграции была выполнена подготовка среды разработки:

- загружен исходный код CLDT в Eclipse;
- настроен репозиторий проекта, добавлены необходимые зависимости (например, Qt, CMake, SWT);
- выполнена успешная компиляция CLDT для проверки работоспособности окружения.

Эти шаги обеспечили наличие стабильной платформы для интеграции `BuildMigrator`.

3.5.3. Подключение BuildMigrator как автономного инструмента

Для интеграции `BuildMigrator` в CLDT было решено использовать его как автономное приложение (standalone), чтобы избежать необходимости установки на локальную машину разработчика или пользователя. Это позволило минимизировать зависимости и упростить процесс преобразования `.pro`-файлов.

Интеграция исходных файлов BuildMigrator Для включения `BuildMigrator` в структуру проекта CLDT использовался подход с **git submodule**, который позволяет добавлять внешний репозиторий как часть основного проекта. Процесс включал:

Добавление подмодуля:

```
git submodule add https://github.com/softcom-su/BuildMigrator
```

Инициализация и обновление подмодуля:

```
git submodule init
git submodule update
```

Этот подход обеспечил доступ к исходным файлам `BuildMigrator` и их синхронизацию с актуальной версией репозитория.

Вызов BuildMigrator в процессе импорта Во время импорта Qt-проекта пользователь указывает путь к директории проекта и `.pro`-файлу через графический интерфейс. `BuildMigrator` вызывается как автономное приложение через системный вызов с передачей параметров командной строки. Результатом работы являются файлы `CMakeLists.txt` и `extensions.cmake`, которые затем используются для настройки проекта в Eclipse. Пример вызова:

```
    buildmigrator --commands build \
    --source_dir "/path/to/source" \
    --out_dir "/path/to/output" \
    --build_command "qmake -d /path/to/source/project.pro" \
    --log_provider CONSOLE
```

3.5.4. Разработка мастера импорта (Wizard)

Для обеспечения взаимодействия пользователя с `BuildMigrator` в Eclipse CLDT был разработан мастер импорта (wizard), реализованный с использованием библиотеки **Standard Widget Toolkit (SWT)**.

Wizard представляет собой графический интерфейс, который пошагово проводит пользователя через процесс импорта Qt-проекта, включая выбор `.pro`-файла, преобразование в `CMake` и интеграцию в рабочую область Eclipse.

Структура мастера Мастер состоит из двух основных компонентов:

Page: Компонент, представляющий страницу мастера, где пользователь вводит данные (например, путь к проекту и `.pro`-файлу).

Wizard: Основной компонент, управляющий страницами, обрабатывающий введённые данные и выполняющий логику преобразования и импорта.

Реализация Frontend (Page) Frontend мастера был реализован как класс, описывающий форму ввода данных, созданный с использованием SWT. Основные элементы интерфейса:

Текстовые поля: Для указания пути к директории проекта и `.pro`-файлу.

Кнопки: Для выбора файлов/папок через диалоговое окно и навигации по мастеру.

Метки (labels): Для отображения подсказок и состояния ввода.

Обработчики ввода: Валидация данных, включающая:

- проверку существования указанных файлов и директорий;
- проверку уникальности имени проекта для предотвращения конфликтов в workspace;
- проверку прав доступа к файлам и папкам.

Для локализации интерфейса использовался файл `messages.properties`, содержащий строковые ресурсы (метки, подсказки), что позволяет адаптировать мастер для разных языков. Пример структуры `messages.properties`:

```
Wizard.Title=Import Qt Project
Wizard.ProjectPath.Label=Project Directory:
Wizard.ProFile.Label=Qt .pro File:
Wizard.Error.InvalidPath=Invalid path or file does not exist
```

Реализация Backend Backend мастера отвечает за обработку данных, введённых пользователем, и выполнение логики импорта. Основные шаги:

Вызов BuildMigrator: После валидации данных `BuildMigrator` запускается как автономное приложение с передачей пути к `.pro`-файлу и параметров для генерации `CMakeLists.txt` и `extensions.cmake`.

Создание Eclipse-проекта: Используется API Eclipse для создания проекта (`IProject`) с добавлением следующих спецификаций:

- `CMakeNature`: обеспечивает поддержку `CMake`-проектов в IDE;
- `QTNature`: активирует инструменты для работы с Qt-проектами.

Описание проекта: Создаётся `Project Description`, содержащее метаданные о проекте (название, путь, настройки).

Импорт в Workspace: В рабочую область добавляются только необходимые файлы (`CMakeLists.txt`, `extensions.cmake`, исходные файлы). Временные файлы, созданные `BuildMigrator`, удаляются.

Логи выполнения `BuildMigrator` отображаются в консоли Eclipse, что позволяет пользователю отслеживать процесс и диагностировать ошибки.

Интеграция с Maven (plugin.xml) Для включения мастера в проект CLDT была добавлена зависимость в файл `plugin.xml`, обеспечивающая доступ к библиотекам SWT и API Eclipse. В этом файле используются точки расширения (*extension points*), которые представляют собой механизм Eclipse, позволяющий плагинам расширять функциональность платформы или других плагинов. Точка расширения, объявленная в `plugin.xml`, определяет место,

где другие плагины могут подключать свои компоненты, такие как представления, команды или мастера, через соответствующие расширения (*extensions*). В данном случае точка расширения `org.eclipse.ui.views` использовалась для регистрации представления мастера. Пример зависимости и расширения:

```
<extension point="org.eclipse.ui.views">
  <view
    id="su.softcom.cldt.qt.ui.internal.views.QtImportConsoleView"
    name="QtImport Console"
    class="su.softcom.cldt.qt.ui.internal.views.QtImportConsoleView"
    category="su.softcom.cldt.qt.ui.views">
  </view>
  <category
    id="su.softcom.cldt.qt.ui.views"
    name="Qt Tools">
  </category>
</extension>
```

Это позволило интегрировать мастер как часть плагина Eclipse, обеспечивая его доступность через меню импорта.

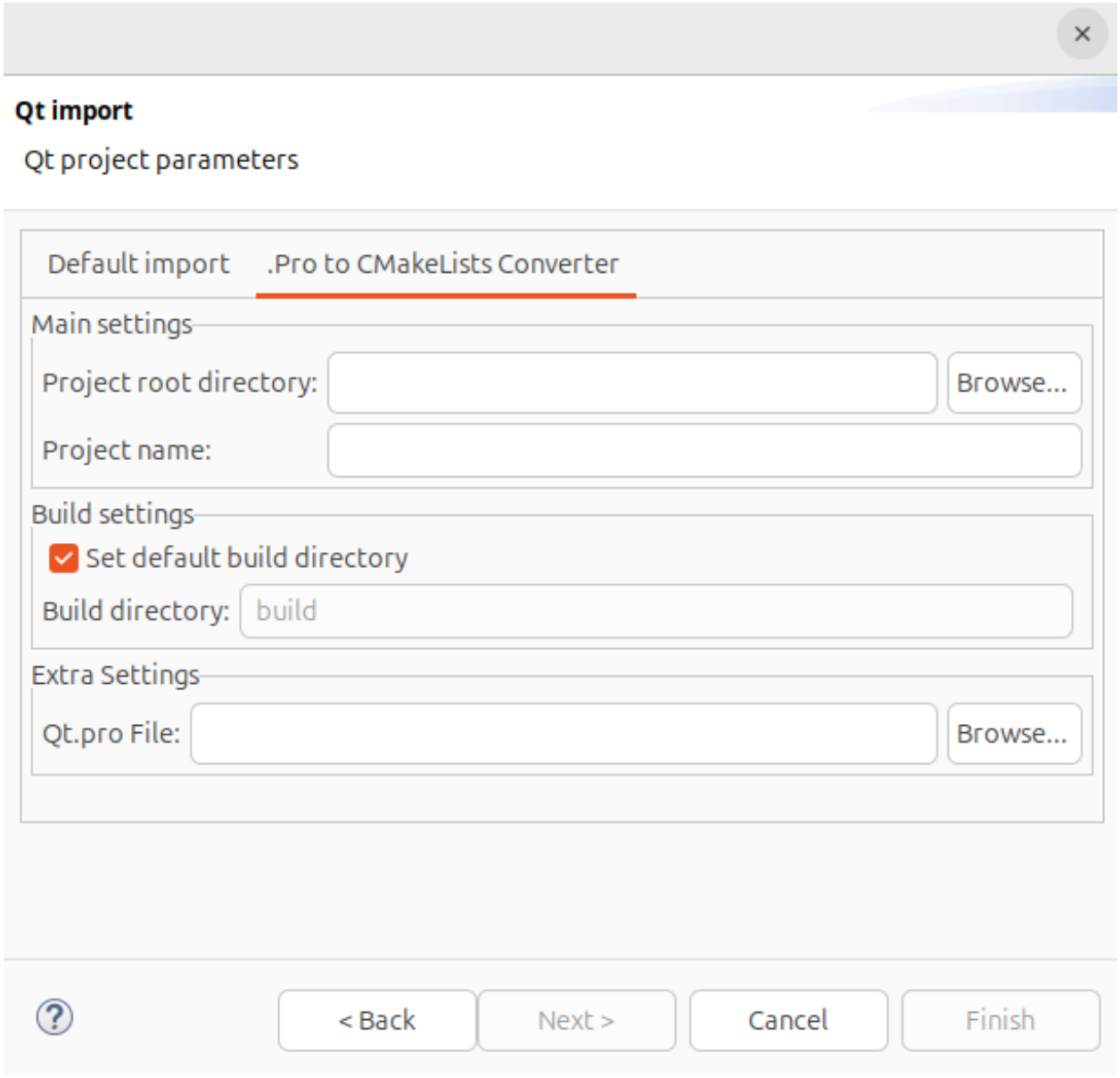

Рис. 6: Интерфейс мастера импорта Qt-проекта в Eclipse

Апробация и тестирование разработанного решения

Апробация

Апробация разработки проводилась в рамках зимней защиты и предзащит выпускной квалификационной работы. На этих мероприятиях демонстрировались основные возможности модифицированного BuildMigrator, включая генерацию файлов

`CMakeLists.txt` из логов QMake. Полученные результаты были обсуждены с научным руководителем и членами комиссии, что позволило выявить направления для дальнейших улучшений. Тесное взаимодействие с компанией, выступающей заказчиком, обеспечивало регулярное получение обратной связи о проделанной работе и внесённых изменениях. Эта обратная связь учитывалась при доработке функциональности, такой как упрощение путей в `CMakeLists.txt` и обработка условной компиляции. Для демонстрации результатов был организован визит в офис компании, где разработка была представлена сотрудникам. В ходе презентации продемонстрированы ключевые аспекты работы, включая интеграцию с Eclipse CLDT. Сотрудники компании провели ревью, предоставили замечания и положительный отзыв, подтвердив соответствие разработки поставленным требованиям. Результаты тестирования консультантом на закрытых проектах заказчиков подтверждают применимость кода.

Тестирование

Тестирование процесса миграции проектов с QMake на CMake проводилось с использованием файлов `compile_commands.json`, сгенерированных инструментом `bear` [12]. Этот подход выбран, поскольку `bear` позволяет автоматически фиксировать команды компиляции, вызываемые при сборке проекта, обеспечивая точное отображение параметров компилятора, флагов, путей включения и обрабатываемых файлов. Такой метод упрощает анализ и сравнение процессов сборки QMake и CMake, выявляя различия в конфигурации и их влияние на результирующие бинарные файлы. Несмотря на возможные различия, такие как порядок флагов или пути включения, `compile_commands.json` даёт возможность проследить правильность выполнения компиляции и сравнить набор исходных файлов, используемых в обоих случаях, что подтверждает функциональную эквивалентность сборок.

Тестирование проводилось на нескольких проектах, полученных из

открытых репозиториев GitHub. Проекты отбирались по следующим критериям:

- наличие `.pro` файла, обеспечивающего сборку с использованием QMake;
- отсутствие необходимости в загрузке дополнительных утилит, библиотек или сложных зависимостей, что упрощало воспроизведение сборки.

Пример compile_commands.json

Для анализа использовались файлы `compile_commands.json`, сгенерированные для сборок QMake и CMake. Ниже приведены фрагменты для компиляции `main.cpp` из проекта `SASM` [1]. SASM (SimpleASM) - простая кроссплатформенная среда разработки для языков ассемблера NASM, MASM, GAS, FASM с подсветкой синтаксиса и отладчиком.

QMake:

```
{
    "arguments": [
    "/usr/bin/g++", "-c", "-pipe", "-O2", "-Wall", "-Wextra",
    ↪ "-D_REENTRANT",
    "-fPIC", "-DQT_NO_DEBUG", "-DQT_WIDGETS_LIB", "-DQT_GUI_LIB",
    ↪ "-DQT_CORE_LIB",
    "-I.", "-I/usr/include/x86_64-linux-gnu/qt5",
    "-I/usr/include/x86_64-linux-gnu/qt5/QtWidgets",
    "-I/usr/include/x86_64-linux-gnu/qt5/QtGui",
    "-I/usr/include/x86_64-linux-gnu/qt5/QtCore",
    "-I.", "-I.", "-I/usr/lib/x86_64-linux-gnu/qt5/mkspecs/linux-g++",
    "-o", "main.o", "main.cpp"
    ],
  "directory": "/home/k0lba/Documents/SASM",
  "file": "/home/k0lba/Documents/SASM/main.cpp",
  "output": "/home/k0lba/Documents/SASM/main.o"
}
```

CMake:

```
{
    "arguments": [
    "/usr/bin/g++", "-DQT_CORE_LIB", "-DQT_GUI_LIB", "-DQT_NO_DEBUG",
    "-DQT_WIDGETS_LIB",
    "-I/home/k0lba/Documents/SASM/build_results/SASM_autogen/include",
    "-isystem", "/usr/include/x86_64-linux-gnu/qt5",
    "-isystem", "/usr/include/x86_64-linux-gnu/qt5/QtCore",
    "-isystem", "/usr/lib/x86_64-linux-gnu/qt5/mkspecs/linux-g++",
    "-isystem", "/usr/include/x86_64-linux-gnu/qt5/QtGui",
    "-isystem", "/usr/include/x86_64-linux-gnu/qt5",
    "-pipe", "-O2", "-Wall", "-Wextra", "-D_REENTRANT", "-fPIC",
    "-DQT_NO_DEBUG", "-DQT_WIDGETS_LIB", "-DQT_GUI_LIB",
    ↪  "-DQT_CORE_LIB",
    "-c", "-o",
    "CMakeFiles/SASM.dir/main.cpp.o",
    "/home/k0lba/Documents/SASM/main.cpp"
    ],
  "directory": "/home/k0lba/Documents/SASM/build_results",
  "file": "/home/k0lba/Documents/SASM/main.cpp",
  "output": "CMakeFiles/SASM.dir/main.cpp.o"

}
```

Компилятор: в обоих случаях используется `/usr/bin/g++`.

Основные флаги: `-pipe`, `-O2`, `-Wall`, `-Wextra`, `-D_REENTRANT`, `-fPIC`, `-DQT_NO_DEBUG`, `-DQT_WIDGETS_LIB`, `-DQT_GUI_LIB`, `-DQT_CORE_LIB`.

Исходные файлы: обрабатываются `main.cpp`, `mainwindow.cpp` и файлы, связанные с Qt (`moc_mainwindow.cpp` в QMake, `mocs_compilation.cpp` в CMake).

Пути Qt: включены стандартные пути `/usr/include/x86_64-linux-gnu/qt5/*`.

Обработка Qt: QMake явно генерирует `moc_mainwindow.cpp`, CMake использует `AUTOMOC`, создавая `mocs_compilation.cpp`. Это влияет на структуру, но не на функциональность.

Пути включения: QMake использует `-I.`, CMake — `-isystem` и `-I.../SASM_autogen/include`. Различия в путях встраиваются в

метаданные.

Флаги: Порядок флагов отличается, но не влияет на код.

Директория сборки: QMake: /home/k0lba/Documents/SASM, CMake: .../build_results.

Выводы:

Анализ compile_commands.json показывает, что QMake и CMake используют одинаковый компилятор и ключевые флаги, обеспечивая эквивалентную компиляцию. Различия в обработке Qt (AUTOMOC vs. явный moc) и путях включения влияют на метаданные бинарников, но не на их функциональность. Это подтверждает возможность использования bear для проверки корректности миграции, позволяя сравнить набор файлов и параметры компиляции.

Ручное тестирование

Проекты, собранные с использованием QMake и CMake, запускались вручную для проверки наличия ошибок и корректности собранных ресурсов. В частности, для SASM проверялось:

- Запуск приложения без ошибок.
- Корректное отображение графического интерфейса, включая иконки и другие ресурсы.
- Функциональность основных операций (обработка тестового ввода).

Оба бинарника продемонстрировали идентичное поведение, подтверждая эквивалентность сборок, несмотря на различия в метаданных, выявленные в compile_commands.json.

Заключение

В рамках данной работы были получены следующие результаты:

- проведён обзор существующих инструментов для автоматизированной конвертации систем сборки. На основе анализа функциональности и возможностей интеграции выбран инструмент BuildMigrator для дальнейшей доработки и внедрения в среду CLDT;

- адаптирован выбранный инструмент для соответствия функциональным требованиям. Разработан подход к обработке QMake-проектов, включающий реализацию парсера логов QMake. Модифицирован генератор файлов `CMakeLists.txt` для обеспечения совместимости с требованиями миграции, включая корректную обработку зависимостей и конфигураций;

- спроектирован и реализован подход к интеграции конвертера в состав CLDT. BuildMigrator использован как самостоятельное приложение, подключённое через `git submodule`. Выбран мастер импорта (Wizard) для реализации функции добавления QMake-проектов в рабочую область IDE. В `extension point` добавлены окна импорта и консоль для вывода информации о процессе конвертации;

- разработан графический пользовательский интерфейс (GUI). Реализован мастер импорта и консоль для отображения данных, полученных от BuildMigrator, включая диагностические сообщения и логи;

- проведены апробация и тестирование разработанного решения. Тестирование включало анализ `compile_commands.json` для проектов из открытых репозиториев GitHub, а также проверку на реальных проектах заказчиков, выполненную консультантом. На основе полученной обратной связи улучшена реализация.

Исходный код доступен на github репозиториях [1] [2]

[1] https://github.com/softcom-su/BuildMigrator/pull/1

[2] https://github.com/softcom-su/CLDT-extensions/pull/1

Список литературы

[1] D. Manushin. SASM - simple crossplatform IDE for NASM, MASM, GAS and FASM assembly languages. — https://github.com/Dman95/SASM. — Accessed: 2025-05-26.

[2] Deak Ferenc. Autotools to CMake. — https://github.com/fritzone/autocmake. — Accessed: 2025-05-26.

[3] Dmitry Kruglov. Make2CMake. — https://github.com/kruglov-dmitry/make2cmake. — Accessed: 2025-05-26.

[4] Fisher Jomo. CMakeify. — https://github.com/jomof/cmakeify. — Accessed: 2025-05-26.

[5] Foundation Eclipse. Eclipse IDE. — https://www.eclipse.org/. — Accessed: 2025-05-26.

[6] Foundation Free Software. Make: A Build Automation Tool. — https://www.gnu.org/software/make/. — Accessed: 2025-05-26.

[7] Iterative Reengineering of Legacy Systems / Alessandro Bianchi, Danilo Caivano, Vittorio Marengo, Giuseppe Visaggio // IEEE Transactions on Software Engineering. — 2003. — Vol. 29, no. 3. — P. 225–241. — Accessed: 2025-05-26. URL: https://ieeexplore.ieee.org/document/1183932.

[8] KasperskyLab. BuildMigrator Documentation. — https://github.com/KasperskyLab/BuildMigrator. — 2024. — Accessed: 2025-05-26.

[9] Kitware Inc. CMake: Cross-Platform Make System. — https://cmake.org/. — Accessed: 2025-05-26.

[10] Liavonau Pavel. CMake Converter. — https://github.com/pavelliavonau/cmakeconverter. — Accessed: 2025-05-26.

[11] Loskutov V. Automated conversion from build systems to CMake for C/C++ languages. — Accessed: 2025-05-26. URL: https://github.com/lve-gh/make2cmake.

[12] László Nagy. Build EAR. — Accessed: 2025-05-26. URL: https://github.com/rizsotto/Bear.

[13] Mecklenburg Robert. Managing Projects with GNU Make. — O'Reilly Media, 2011. — URL: http://uploads.mitechie.com/books/Managing_Projects_with_GNU_Make_Third_Edition.pdf.

[14] Project Qt. pro2cmake.py: Script for Converting QMake .pro Files to CMakeLists.txt. — https://fossies.org/linux/qt-everywhere/qtbase/util/cmake/pro2cmake.py. — Accessed: 2025-05-26.

[15] Project Qt. qmake2cmake: Tool for Converting QMake Projects to CMake. — https://pypi.org/project/qmake2cmake/. — Accessed: 2025-05-26.

[16] Qt Project. Qt 6.0 Release Notes. — 2021. — Accessed: 2025-05-26. URL: https://doc.qt.io/qt-6/whatsnew60.html.

[17] Qt Project. QMake Manual. — 2023. — Accessed: 2025-05-26. URL: https://doc.qt.io/qt-5/qmake-manual.html.

[18] Schreiner Henry. An Introduction to Modern CMake. — Accessed: 2025-05-26. URL: https://cliutils.gitlab.io/modern-cmake/modern-cmake.pdf.

[19] davidtazy David. QMakeToCMake. — https://github.com/davidtazy/QMake2CMake. — Accessed: 2025-05-26.