Автономная некоммерческая организация высшего образования
«Университет Иннополис»
(АНО ВО «Университет Иннополис»)

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
(МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ)
по направлению подготовки
09.04.01 – «Информатика и вычислительная техника»

GRADUATION THESIS
(MASTER GRADUATE THESIS)
Field of Study
09.04.01 – «Computer Science»

Направленность (профиль) образовательной программы
«Управление разработкой программного обеспечения»
Area of specialization / academic program title:
«Software Engineering»

Тема / Topic	**Портирование и переработка кода систем мягкого реального времени с использованием контрактного программирования на примере видеоигры Doom / Porting and refactoring soft real-time systems using Design by Contract: DOOM video game example**

Работу выполнил / Thesis is executed by	**Мустафин Ильгиз Айварович / Mustafin Ilgiz Aivarovich**	подпись / signature
Руководитель выпускной квалификационной работы / Graduation Thesis Supervisor	**Маццара Мануэль / Manuel Mazzara**	подпись / signature

Иннополис, Innopolis, 2021

Оглавление

1 Введение **5**

2 Обзор литературы **8**

2.1 Проблемы портирования ПО 8

2.2 Портирование и языковые конверсии 8

2.2.1 Цели языковой конверсии 9

2.2.2 Автоматическая языковая конверсия 10

2.2.3 Смена парадигмы 10

2.2.4 Инструменты для языковых конверсий 12

2.3 Контрактное программирование 12

2.4 Языковые конверсии и отладка 13

2.5 Контрактное программирование и переработка кода 14

3 Методология **16**

3.1 Процесс портирования 16

3.2 Массивы и адресная арифметика 17

3.3 Модули 18

3.4 Порядок выполнения 20

3.5 Безопасость пустых ссылок 20

3.6 Внешние библиотеки 21

4 Реализация **22**

4.1 Подготовка среды разработки 22

4.2 Применение методологии портирования 23

4.3 Основные типы 25

4.4 Ресурсы игры и порядок байтов 25

4.5 Операторы порядка выполнения 26

4.6 Кластеры 27

5 Выводы **29**

5.1 Производительность 29

5.2 Размер испольняемого файла 32

5.3 Размер кода 33

5.4 Отладка 34

5.4.1 Неопределённое поведение для указателей на элемент перед первым элементом массива 35

5.4.2 Чтение за пределами массива 35

5.4.3 Разыменовывание нулевого указателя 36

Аннотация

В этой работе обсуждается проблема портирования программного обеспечения с одного языка на другой. Описывается процесс интеграции контрактного программирования (Design by Contract) и безопасности пустых ссылок (Void Safety) в процесс портирования. Результаты показывают, что портирование програмы мягкого реального времени, такой как Doom видеоигра (id Software, 1993) из Си в Эйфель возможна. Производительности порта хватает для бесперебойной работы игры с необходимой производительностью. Интеграция контрактного программирования и безопасности нулевых ссылок выявили ошибки в исходной системе, которые были исправлены в порте.

Получившиеся исходные коды опубликованы в интернете и являются ПО с открытым исходным кодом. Порт самой игры называется *Brie Doom* и доступен по ссылке. `https://github.com/imustafin/brie_doom`. Расширенная версия библиотеки *wrap_sdl* доступна по адресу `https://github.com/imustafin/wrap_sdl` и может быть переиспользована в других программах на Эйфеле, где требуется использование библиотеки SDL.

Глава 1

Введение

Эта работа о портировании программного обеспечения (ПО) с одного языка программирования на другой и связанных процессах для повышения корректности программ в процессе портирования.

Портируемость — это мера того, насколько легко перенести программу. из одной среды в другую [1]. Потребность в использовании ПО в другой среде может уже после разработки системы, что потребует портирования программы для работы в другой среде. Однако, существуют различные препятствия в портировании программного обеспечения, которые определены Таненбаумом, Клинтом и Бомом в [2].

Иногда перевод программы на другой язык программирования (языковая конверсия) используется как способ портирования ПО. Задача языковой конверсии — переписать систему с одного языка на другой, сохраняя внешнее поведение. Популярным примером является концепция языка Джава «Напиши один раз, запускай где угодно» (Write once, run anywhere). Перенос программы, например, с Си на Джаву может позволить программе работать в более широком наборе сред.

Перевод программ имеет свой набор проблем, которые могут возникнуть во время процесса. Идеальное и автоматическое преобразование не всегда возможно и может быть дорогостоящим в реализации, и при этом всё равно дать неверную программу на выходе [3].

В этой диссертации мы исследуем применимость контрактного программирования [4] в качестве инструмента устранения ошибок во время языковой конверсии. Контрактное программирование — это принцип, согласно которому обязательства между единицы программного обеспечения указаны в виде предусловий, постусловий и инвариантов.

Одним из параметров процесса преобразования языка является цель преобразования. Требуется ли, чтобы код «просто компилировался» и был совместим с целевыми средами выполнения или код должен быть в добавок легко изменяемым, чтобы позволить дальнейшую разработку на новом языке? Здесь нужно найти компромисс. Можно использовать автоматическое перевод с одного языка на другой, однако получившийся код может быть сложно изменять.

Цель этой диссертации состоит в том, чтобы ответить на исследовательский вопрос “применим ли Эйфель для переноса вычислительно-интенсивной системы мягкого реального времени с языка Си?”. Результаты показывают, что Эйфель применим. Однако при использовании контрактного программирования возникают проблемы с производительностью системыы в среде разработки.

В качестве эксперимента мы портируем видеоигру Doom разработанную id Software в 1993 году [5] из Си в Эйфель. Мы выбрали эту программу в качестве примера по нескольким причинам. Основные причины: наличие исходного кода; размер программы, подходящий для времени, вы-

деленного на дипломную работу; жесткие требования к эффективности и вычислительной интенсивности программы, которую можно использовать в качестве эталона.

Эйфель [6] — это объектно-ориентированный язык программирования, поддерживающий контрактное программирование. Цель этого эксперимента — сделать версию игры, работающую на современных системах, компилирующуюся современными компиляторами и легко модифицирующуюся. Последнее требование означает, что во время перевода мы будем выполнять переработку частей кода для повышения модфицируемости программы.

Структура диссертации следующая: обзор существующей литературы по темам портирования ПО, языковых конверсий, переработки кода с контрактным программированием представлен в глав. 2, описание предложенной методологии для выполнения языковой конверсии представлено в глав. 3, особенности применения процесса для портирования игры Doom представлены в глав. 4, анализ результатов языковой конверсии представлен и общие результаты и выводы исследования представлены в глав. 5.

Глава 2

Обзор литературы

В этой главе мы рассмотрим существующую литературу по теме портирования ПО, переработке кода, контрактного программирования и безопасности пустых ссылок. Также здесь обозначаются проблемы языковых конверсий и их возможные решения.

2.1 Проблемы портирования ПО

Процесс портирования ПО имеет ряд сложностей, которые могут сделать процесс портирования сложнее или вообще невозможным [2], [7]. Сложности могут различаться по важности в зависимости от характеристик исходных сред, целевых сред и особенностей самой программы. Эти сложности могут быть разделены на несколько типов.

2.2 Портирование и языковые конверсии

Большинство сложностей в портировании можно представить как особенности различных *интерфейсов* [8]. Такие интерфейсы могут предо-

ставлять доступ к процессору, операционной системе, Such interfaces may give access to the processor, operating system, библиотек, устройств ввода-вывода и др. Если программа использует эти интерфейсы для доступа к соответствующим ресурсам и эти интерфейсы выглядят идентично снаружи в разных средах, то можно сказать, что достигнута портируемость в этих средах. Таким образом, перевод программы с одного языка на другой может быть использован как способ портирования.

Высокоуровнеые языки программирования могут абстрагировать сложности программирования в своих интерфейсах и других абстракциях. Для разработки портируемого ПО следует выбирать подходящие языки [8]. Некоторые особенности языков становятся более важными, когда есть требования к портируемости ПО. Хорошо стандартизованные языки предоставляют более высокий уровень портируемости [2], [8].

2.2.1 Цели языковой конверсии

В зависимости от целей перевода программы на другой язык, особое внимание может быть уделено на некоторые характеристики результирующего кода.

Количество работы необходимой для разработки инструментов для автоматического перевода программ на другой язык часто недооценивается [3], [9].

Особое внимание стоит уделить на следующие параметры [3]: должна ли система быть функционально идентична исходной и планируется ли дальнейшая поддержка системы? Также стоит учесть, что производительность системы должна быть достаточной и во время выполнения, и во время компиляции. Если планируется дальнейшая поддержка системы, то

размер переведённой программы не должен быть сильно больше размера исходной.

В зависимости от языка, используемого для разработки исходной системы, некоторые структуры языка могут быть более или менее трудоёмкими для перевода. Однако не все программы используют все структуры языка, поэтому инструменты для перевода могут не поддерживать часть структур исходного языка [10].

Возможно, что целевой язык содержит больше возможностей, чем исходный. Напимер, Джава содержит более богатый инструментарий для работы со строками чем Си [10]. В таких случаях, языковая конверсия может упростить код путём использования расширенного инструментария целевого языка.

2.2.2 Автоматическая языковая конверсия

Существующие исследования показывают, что автоматическая языковая конверсия в общем случае нетривиальна [3]. Так как программы часто не используют все конструкции исходного языка, извлечения подмножества языка для перевода может уменьшить количество необходимой работы для автоматической языковой конверсии [10].

2.2.3 Смена парадигмы

Во время портирвания ПО с языка с одной парадигмой на язык с другой (например, процедурная парадигма в Си и объектно-ориентированная парадигма в Джаве) происходит смена парадигмы.

Параллельно могут происходить несколько смен парадигм. Напри-

мер, гибкая и нестрогая работа с памятью в Си с использованием указателей и строгая модель памяти в Джаве с использованием ссылок [11].

Извлечение классов

Процедурный код Си не имеет классов, поэтому перевод программ на объектно-ориентированный язык подразумевает извлечение классов.

Существует несколько подходов к определнию возможных классов и их членов. Одни подходы основываются на определении структур данных и функций, которые с ними работают [12]—[14].

Другие основываются на документации потоков данных и использовании диаграмм структур для извлечения объектно-ориентированной архитектуры исходной системы [15], [16].

От указателей к ссылкам

Одна из смен парадигм, которая происходит при переводе программ с Си на Джаву, — это переход от процедурного стиля к объектно-ориентированному. Это связано с тем, что в Джаве нет указателей и применяется более строгая модель управления памятью.

Демайн [17] предлагает общий метод для преобразования указателей языка Си в ссылки языка Джава. Метод основывется на модели *блоков* для представления участков памяти. Утверждается, что предложенная модель покрывает почти все возможные структуры использования ссылок. Однако, в меру наших знаний, реализация этого метода ещё не была закончена.

2.2.4 Инструменты для языковых конверсий

Существует несколько инструментов, которые предоставляют возможность для автоматического перевода программ с разных языков и поддерживающие разные наборы инструкций [11], [18]—[22].

Так как цель этого исследования заключается в портировании процедурного Си кода в объектно-ориентированный код на Эйфеле, отдельно стоит отметить инструменты Ephedra [11], C2J [20] и C2Eif [22], которые работают в этой сфере.

2.3 Контрактное программирование

Контрактное программирование [4] — это приницп разработки ПО, по которому взаимные обязанности между единицами программы должны быть чётко определены. Поддержка контрактного программирования встроена в Эйфель, а для других языков существуют инструменты, которые добавляют такую поддержку [23]—[28].

Для определения обязанностей между единицами ПО контрактное программирование использует контракты, которые выражаются в виде предусловий, постусловий и инвариантов классов.

Если первая единица ПО использует вторую, то первая должна удовлетворить предусловиям второй, а вторая должна удовлетворить своим постусловиям.

Инварианты классов описывают состояние объектов после создания и после выполнения каждой экспортированной процедуры, если инвариант выполнялся на входе экспортированной процедуры.

Эти правила можно записать используя нотацию из рис. 2.1.

$$\{P\}\, A\, \{Q\}$$

Рис. 2.1: Правило предусловия и постусловия

Где P и Q — утверждения, и A — последовательность инструкций. Эта формула обозначает, что если A выполняется в состоянии, где удовлетворяется утверждение P, то получившееся состояние будет удовлетворять Q.

Правило для процедур создания описано на рис. 2.2.

$$\{P\}\, C\, \{I\}$$

Рис. 2.2: Правило процедур создания

Где P — предусловие, C — тело процедуры создания, а I — инвариант класса .

Правило для экспортированных процедур представлено на рис. 2.3.

$$\{P \wedge I\}\, R\, \{Q \wedge I\}$$

Рис. 2.3: Правило экспортированных процедур

Где P — предусловие экспортированной процедуры, I инвариант класса, R — тело процедуры, а Q — постусловие процедуры.

Когда процедура переопределяется в наследовании, предусловие может быть ослаблено, а постусловие может быть усиленно.

2.4 Языковые конверсии и отладка

Перевод программы на другой язык может помочь в отладке. Например, язык Си позволяет использовать значение неинициализированных

переменных или деаллоцированных участков памяти, что невозможно в Джаве. Джава запрещает чтение из ссылок, которые содержат значение `null`. Перевод программы с Си в Джаву описанный в [11] показал, что такое свойство помогло найти ошибку в исходной программе.

Во время перевода программ на Эйфель возможно добавление простых контрактов, которые так же помогают найти ошибки.

2.5 Контрактное программирование и переработка кода

Контрактное программирование помогает в разработке по принципам экстремального программирования (Extreme Programming, XP), а конкретно в юнит-тестировании и переработке кода [29].

Контрактное программирование на заменяет юнит-тестирование, а дополняет его. Простые юнит-тесты, которые проверяют правильность результатов функций могут быть заменены контрактами, что уменьшает количество маленьких юнит-тестов, которые проверяют тривиальные свойства. При таком подходе юнит-тестирование может использоваться для проверки менее тривиальных свойств.

Во время изменения кода, вместе с ним должны и обновляться тесты. Такие подходы, как разработка через тестирование (Test Driven Development, TDD) [30] и разработка через поведение (Behaviour Driven Development, BDD) [31] определяют, что сначала должны обновляться тесты и только потом сам код. Похожий подход может использоваться и в контрактном программировании: сначала должны обновляться контракты, а потом сам код [29].

Также существуют инструменты, которые помогают в обновлении контрактов при автоматической переработке кода: Crepe [32] для Джавы, EiffelStudio [33] для Эйфеля.

Глава 3

Методология

В этой главе обсуждается общий процесс для портирования программ из Си в Эйфель.

Этот процесс основан на подходе C2Eif и AutoOO [14]. Однако мы производим перевод вручную чтобы найти возможные улучшения для процесса и улучшения качества получившегося кода.

3.1 Процесс портирования

В этом эксперименте мы портируем только часть оригинальной системы. Мы выбрали самые вычислительно-нагруженные части чтобы оценить производительность порта.

На высоком уровне требования к порту звучат так: первый уровень игры должен быть проходим, что включает в себя 3D движок игры, звуковые эффекты и музыку, ИИ для врагов, взаимодействие с игровым миром.

Чтобы определить части системы, которые должны быть портированны, используется анализ графа вызовов. Основывясь на высокоуровневых требованиях мы определяем список более низкоуровневых требований как

«когда игра включается, должно быть показано главное меню», «когда игра включается, должна играть музыка главного меню», «когда начинается новая игра из главного меню, должен включиться первый уровень». Во время портирования, требования уточняются и перерабатываются в конкретные требования к конкретным функциям.

Если во время портирования одной функции встречаются другие ещё не портированные функции, то они создаются с пустым телом. Если эта функция необходима для выполнения требования, то она портируется.

3.2 Массивы и адресная арифметика

В C2Eif массивы переводятся с использованиям специального обобщённого класса `CE_ARRAY [G]`. Чтобы получить более идиоматичный код на Эйфеле мы переводим массивы в класс Эйфеля `ARRAY [G]`.

Отдельное внимание уделяется байтовым массивам. В зависимости от того как они используются, они переводятся разными способами.

Если байтовый массив просто читается и передаётся обратно в Си, то мы используем класс Эйфеля `MANAGED_POINTER` чтобы хранить указатель на участок памяти этого массива.

Если же массив изменяется в коде, то он переводится в настоящий массив `ARRAY [NATURAL_8]`. Это позволяет нам переиспользовать код и контракты класса `ARRAY`.

Код оригинальной системы использует адресную арифметику чтобы создавать указатели на определённые участки массивов. Такое поведение воссоздаётся в коде на Эйфеле с использованием двух специальных классов `MANAGED_POINTER_WITH_OFFSET` и `INDEX_IN_ARRAY`. Оба класса хранят

в себе отступ от начала соответствующего контейнера. Операция доступа к элементу добавляет отступ к запрашиваемому индексу, повторяя поведение оператора индексации `[]`.

Адресная арифметика реализуется через операторы сложения и вычитания для создания новых контейнеров с отступами как показано в рис. 3.1.

Этот подход похож на `CE_POINTER` из C2Eif, но он более специализирован для работы с массивами, улучшая читаемость кода.

В отличие от C2Eif, размерности массива проверяются инвариантами содержащего класса, а так же переиспользуются контракты из класса `ARRAY`.

3.3 Модули

Код оригинальной системы разбит на несколько `.c` и `.h` файлов. Мы называем `.c` файл и соответствующий `.h` файл *модулем*.

В начале каждый модуль переводится в класс, что похоже на пакетные классы (bundle classes) в AutoOO. В отличие от подхода AutoOO, классы модулей содержат нестатические члены. Такой подход позволяет использовать инварианты классов для проверки свойств глобальных переменных этого класса.

Это значит, что ссылки на объекты классов модулей должны быть доступны модулям, которые их используют в оригинальном коде. Мы решаем эту проблему тем, что ссылки на объекты классов модулей хранятся в корневом классе системы.

```
class
  INDEX_IN_ARRAY [G]

create
  make

feature

  index: INTEGER

feature

  make (a_index: INTEGER; a_array: ARRAY [G])
    do
      index := a_index
      array := a_array
    end

feature {NONE}

  array: ARRAY [G]

feature

  plus alias "+" (num: INTEGER): like Current
    do
      create Result.make (index + num, array)
    end

  item alias "[]" (num: INTEGER): G assign put
    require
      valid_index (num)
    do
      Result := array [num + index]
    end

  put (v: G; num: INTEGER)
    require
      valid_index (num)
    do
      array [num + index] := v
    end

end
```

Рис. 3.1: Класс INDEX_IN_ARRAY для реализации адресной арифметики

3.4 Порядок выполнения

Последовательное выполнение, циклы и условные переходы переводятся в стандартные инструкции языка Эйфель. Инструкции, которые ломают стандартный порядок выполнения (`return`, `break`, `continue`, `goto`) переводятся в зависимости от того, как они используются.

В отличие от C2Eif, мы не используем циклы для перевода оператора `goto` в случаях, когда прыжок выполняется вперёд. Вместо этого мы добавляем вспомогательную переменную, такую же как и для операторов `return` и `break`.

3.5 Безопасость пустых ссылок

Безопасность пустых ссылок (Void Safety) [34], [35] — это свойство системы, которое означает, что такая безопасная система никогда не будет производить разыменование ссылки, которая содержит значение `Void`. Механизмы для обеспечения безопасности пустых ссылок включены в язык Эйфель.

Это свойство лишь частично обрабатывается в подходе C2Eif, где оно выражается в виде предусловий, которые проверяют то, что ссылки, переданные в аргументах, в самом деле привязаны к какому-то значению. Привязанность глобальных переменных не рассматривается явно.

Во время портирования мы изначально считаем, что все глобальные переменные должны быть всегда привязанны к объекту (`attached` в Эйфеле), а потом расслабляем это ограничение если находятся подтверждения тому, что переменная в самом деле может быть отвязана от какого либо объекта (`detachable` в Эйфеле).

```
void f(t* x) {
    if (x) {
        g(x->y);
    }
}
```

Рис. 3.2: Пример подтверждения того, что аргумент может быть NULL

Примером такого подтверждения является проверка на присутствие значения в ссылке как показано в рис. 3.2.

3.6 Внешние библиотеки

Doom использует внешнюю библиотеку для ввода-вывода аудио и видео. В подходе C2Eif, такие использования были бы переведены с использованием вспомогательных классов для доступа к соответствующей библиотеке. Наш подход похож на C2Eif в этом плане, но мы используем другой инструмент для его реализации.

Мы будем использовать инструмент WrapC [36] (прежде известный как EWG — Eiffel Wrapper Generator [37]). Это позволит нам создать переиспользуемый пакет на Эйфеле, доступный для включения в другие проекты.

Глава 4

Реализация

В этой главе мы описываем особенности применения разработанной методологии для портирования игры Doom.

4.1 Подготовка среды разработки

Процесс портирования начинается с получения исходного кода оригинальной системы. Мы будем использовать официально опубликованный исходный код игры DOOM [38]. В репозитории находится *linuxdoom-1.10*, которая работает в среде Linux.

Опубликованная версия использует X Window System (X11) и поддерживает только старый 256-цветный видео режим. Чтобы сделать порт рабочим на современных машинах, мы заменим X11 на (Simple DirectMedia Layer) [39]. Поддержка SDL2 в порте основывается на Chocolate Doom [40] — современной версии игры Doom, которая использует SDL2 и работает на современных машинах.

```
int
main
( int     argc,
  char**  argv )
{
  myargc = argc;
  myargv = argv;

  D_DoomMain ();

  return 0;
}
```

Рис. 4.1: Главная функция оригинальной системы

4.2 Применение методологии портирования

После того, как удалось запустить оригинальную программу в среде разработки, мы можем начать процесс портирования. Мы начинаем с главной функции. Её реализация находится в файле `i_main.c` и приведена на рис. 4.1.

Главной функции программ на Си соответствует корневая процедура в программах на Эйфеле. Перевод функции `main` приведён на рис. 4.2

Следуя методологии, мы добавляем ссылки на модули, используемые в переводимой функции, в корневой класс, а именно `m_argv: M_ARGV` и `d_main: D_MAIN`. Большинство модулей реализуются как `once`-функции, которые создают объект класса модуля, что является шаблоном проектирования «Одиночка» (Singleton). Также, при создании объектов классов модулей, в процедуру создания может передаваться ссылка на корневой класс `I_MAIN` для того, чтобы модули могли получить доступ к другим модулям через `I_MAIN`.

```
class
  I_MAIN

inherit

  ARGUMENTS_32

create
  main

feature -- Globals

  m_argv: M_ARGV

  d_main: D_MAIN
    once
      create Result.make (Current)
    end

  -- other modules omitted

feature {NONE} -- Initialization

  main
      -- Run application.
    do
      create m_argv.make (argument_array)
      d_main.D_DoomMain
    end

end
```

Рис. 4.2: Перевод главной функции на Эйфель

4.3 Основные типы

Подобно подходу C2Eif, мы переводим основные типы Си в соответствующие классы Эйфеля как показано в табл. I. Стоит отметить, что тип `float` не используется в оригинальной системе, а тип `double` используется только в одной функции.

Вместо использования чисел с плавающей запятой, оригинальная система использует числа с фиксированной запятой. 16 бит выделяется на целую часть и 16 бит выделяется на дробную часть. Числа с фиксированной запятой в коде определяются как `typedef int fixed_t` и `typedef unsigned angle_t`. Эти два типа переводятся в классы `expanded class FIXED_T inherit INTEGER_32_REF` и `expanded class ANGLE_T inherit NATURAL_32_REF`. Также отметим, что Doom вызывает функцию `abs` с аргументом `angle_t`, расчитывая на то, что беззнаковый аргумент будет приведён к знаковому типу. Мы добавили функцию `abs` в класс `ANGLE_T`, который повторяет такое поведение, хотя в классе `NATURAL_32_REF` нет такой функции.

Doom определяет тип `byte` как `typedef unsigned char byte`. Этот тип переводится в `NATURAL_8` потому что тип `byte` никогда не используется как символ, а всегда используется как число.

В отличие от подхода C2Eif, указатели на символы `char*`, которые представляют строку, переводятся напрямую в класс `STRING`.

4.4 Ресурсы игры и порядок байтов

Ресурсы игры хранятся в `.WAD` файлах. Структура этих файлов отображается в коде в виде структур (`struct`). Оригинальный код читает сырые

C type	Eiffel class
short int	INTEGER_16
int	INTEGER_32
long	INTEGER_32
long long	INTEGER_64
unsigned short	NATURAL_16
unsigned	NATURAL_32
double	REAL_64
char	CHARACTER_8
char*	STRING
boolean	BOOLEAN
fixed_t	FIXED_T
angle_t	ANGLE_T
byte	NATURAL_8

Таблица I: Перевод типов из Си в Эйфель

байты и приводит их к соответствующим структурам. Такое прямое приведение байтов в структуру невозможно в Эйфеле и для реализации такого поведения в структуры добавлены процедуры создания, которые принимают массивы байтов. Эта переработка кода позволила уменьшит повторения кода и улучшила читаемость.

Числа в `.WAD` файлах хранятся в порядке от младшего к старшему. Мы использовали функции класса `MANAGED_POINTER` для чтения чисел с учётом этого порядка байтов.

4.5 Операторы порядка выполнения

Как мы раньше упомянали в разд. 3.4, операторы, нарушающие прямой порядок выполнения как `return`, `continue`, `break`, `goto` требуют специального перевода, так как Эйфель не имеет таких операторов.

Мы переводим оператор `goto` без использования циклов, когда пры-

жок выполняется вперёд. Пример такого перевода приведён на рис. 4.3.

```
    if (x) {
        f(x);
        goto dest;
    }

    g(y);
    return;

dest:
    h(z);
```

Входной код на Си

```
if x then
    f(x)
    goto_dest := True
end
if not goto_dest then
    g(y)
    returned := True
end
if not returned then
    h(z)
end
```

Выходной код на Эйфеле

Рис. 4.3: Пример перевода инструкции `goto` без использования циклов

Такой подход улучшает читаемость кода.

4.6 Кластеры

Кластеры используются для структурирования исходного кода программ на Эйфеле. Одна из особенностей кластеров заключается в том, что можно включать и выключать контракты для отдельных кластеров. Позже мы покажем важность этой возможности в разд. 5.1.

Основные выделенные кластеры:

1. *sound* для кода, обеспечивающего воспроизведение звуковых эффектов и музыки

2. *status_bar* для кода, рисующего строку состояния внизу экрана

3. *math* включает в себя классы `ANGLE_T`, `FIXED_T` и `M_FIXED`, которые отвечают за арифметику с числами с фиксированной запятой

4. *render* сам 3D движок игры

5. *pointers* включает в себя классы `MANAGED_POINTER_WITH_OFFSET` и `INDEX_IN_ARRAY` для реализации адресной арифметики

Кластеры представляют собой деление всей системы на подсистемы, которые можно тестировать по отдельности. Например, когда мы уверены, что код в кластере *math* не содержит ошибок, мы можем выключить проверку контрактов в этом кластере и увеличить производительность системы. Это позволит эффективнее протестировать другие подсистемы.

Глава 5

Выводы

В этой главе мы анализируем программу, получившуюся в результате портирования, представляем результаты, которые включают в себя найденные ошибки в оригинальном коде.

Мы считаем эксперимент по портированию успешным в том смысле, что главное требование «первый уровень должен быть проходимым» выполняется. Порт содержит в себе главное меню, 3D движок и основные элементы игры.

Снимки экрана работающей игры представлены на рис. 5.1

5.1 Производительность

Мы собрали количественные метрики для нескольких типов исполняемых файлов, которые создал компилятор EiffelStudio. В EiffelStudio используется технология «тающего льда» (Melting Ice Technology) для ускорения перекомпиляции порграммы посредством создания нескольких видом исполняемых файлов. *Замороженные* (frozen) файлы могут отлаживаться инструментами EiffelStudio и включают в себя проверку контрактов.

Главное меню

Начало первого уровня

Сражение с монстрами

Статистика пройденного уровня

Рис. 5.1: Снимки экрана работы порта

Финализированные (finalized) файлы не могут отлаживаться инструментами EiffelStudio и не проверяют контракты, но в отличие от замороженных исполняемых, во время компиляции проводятся дополнительные оптимизации.

Разбиение программы на кластеры описано в разд. 4.6.

- *Финал.* — финализированные исполняемые файлы
- *Основн.* — замороженные исполняемые файлы, где все проверки контрактов отключены для главного кластера, проверяются только предусловия стандартной библиотеки Эйфеля
- *Главн.* — то же, что и *Основн.*, но проверяются контракты для классов, которые не находятся в подкластере главного кластера
- *Главн. без инв.* — то же, что и *Главн.*, но отключены проверки инвариантов классов
- *Все* — то же, что и *Основн.*, но все контракты включены для главного кластера и всех подкластеров.

Одно из главных свойств игры — это скорость выполнения. Обычно она считается по количеству кадров в секунду (Frames Per Second, FPS), которые рисуются игрой. Doom разрабатывался для работы в 35 FPS, поэтому мы берём это значение за эталон. В этом эксперименте ограничение кадров не было включено в код, поэтому значение FPS может превышать 35. Данные по значениям FPS представлены в табл. II.

В этой таблице столбец $P_{5\%}$ означает пятый процентиль, то есть значение, которое разделяет наименьшие 5% значений от 95% наибольших.

	Среднее	Мин.	$P_{5\%}$	$P_{95\%}$	Макс.
Финал.	106,8	37,	38,2	420,8	461,0
Основн.	8,1	5,0	5,0	20,9	38,0
Главн.	4,6	0,0	2,0	6,0	39,0
Главн. без инв.	7,3	0,0	4,25	15,25	38,0
Все	0,8	0,0	0,0	1,0	31,0

Таблица II: Частота кадров порта

Данные показывают, что финализированная версия имеет достаточную производительность (Мин. = 37). Кроме того, есть возможность добавления нереализованных функций (Среднее = 106,8, $P_{5\%} = 38,2$), не опуская частоту кадров ниже необходимых 35 кадров в секунду.

Однако, все замороженные версии показывают значительное уменьшение производительности. Проверка только предусловий стандартной библиотеки оказывается сильно менее производительной, но всё ещё достаточной для проведения ручного тестирования. Мы считаем, что для этого достаточно всего трёх кадров в секунду. Проверка всех контрактов опускает частоту кадров ниже одного в секунду, что делает проведение ручного тестирования почти невозможным.

Разбиение программы на кластеры для отдельного тестирования разных подсистем позволяет добиться минимальной необходимой производительности.

5.2 Размер испольняемого файла

Размер исполняемого файла так же является важной метрикой. Скомпилированная оригинальная система занимает 1,4МБ, а финализированная версия порта занимает 5,5МБ.

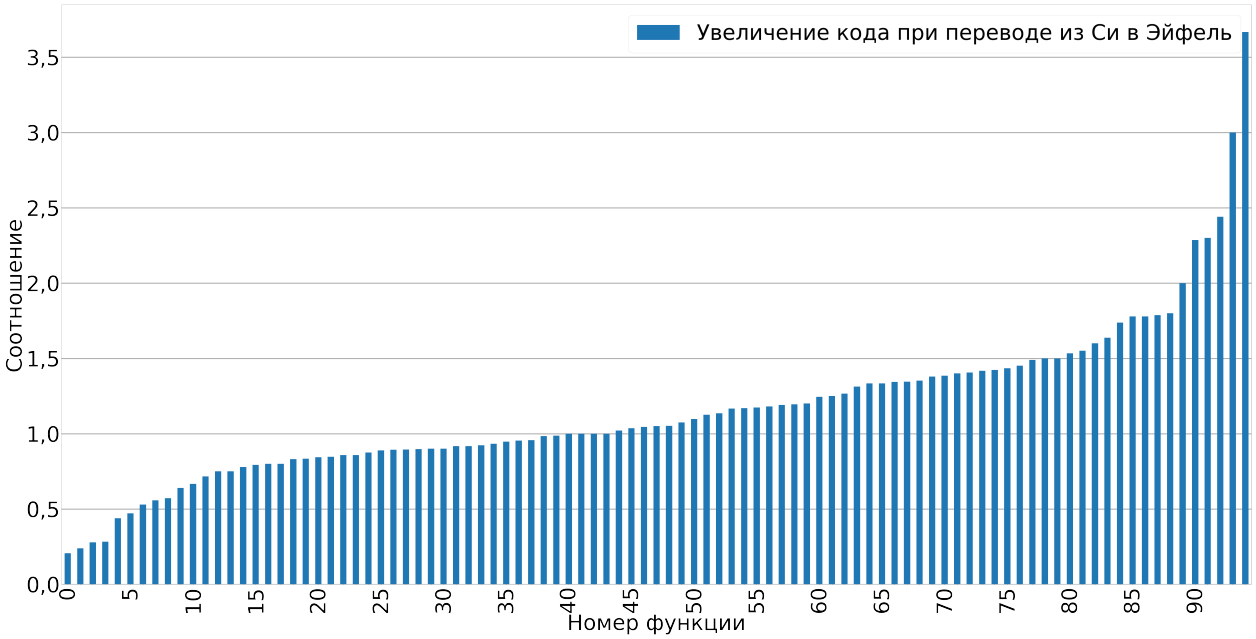

Рис. 5.2: Распределение увеличения кода при переводе из Си в Эйфель

Мы считаем такое увеличение размера исполняемого файла незначительным.

5.3 Размер кода

Размер оригинальной системы составляет 33,4 тыс. строк кода (KLOC), а размер частичного порта составляет 23,2 KLOC. Для подсчёта размера кода программы на Си использовался инструмент CCCC [41], а для Эйфеля использовались инструменты EiffelStudio.

CCCC определил 301 функцию в оригинальной программе, 95 из которых были портированы, таким образом 31% всех функций были портированны. Мы можем оценить, что порт всей системы может содержать около 74,8 KLOC, что примерно в два раза больше оригинальной системы.

На рис. 5.2 мы показываем распределение увеличение размера кода при портировании из Си в Эйфель. Часть данных представлена в табл. III.

Номер	Название	Си LOC	Эйфель LOC	Увеличение
0	`ptr_shoottraverse`	73	15	0,21
1	`p_loadlinedefs`	67	16	0,24
2	`xlatekey`	54	15	0,28
3	`ptr_aimtraverse`	53	15	0,28
4	`i_getevent`	73	32	0,44
90	`wi_drawstats`	21	48	2,29
91	`st_updatewidgets`	30	69	2,30
92	`wi_drawshownextloc`	25	61	2,44
93	`st_refreshbackground`	10	30	3,00
94	`st_drawwidgets`	21	77	3,67

Таблица III: C and Eiffel function LOC increase data

Данные показывают, что некоторые функции становятся значительно длиннее, большинство остаются примерно такими же и некоторые становятся короче.

Функции могут удлинняться из-за многословности конструкций на Эйфеле. Функции могут укорачиваться из-за проведённой переработки кода.

5.4 Отладка

Использование контрактного программирования и внедрение безопасности пустых ссылок помогло нам найти несколько ошибок в оригинальной системе. Эти ошибки были исправлены в порте.

5.4.1 Неопределённое поведение для указателей на элемент перед первым элементом массива

Стандарт языка Си [42] говорит, что создание указателя, указывающего на элемент перед первым элементом структуры, вызывает неопределённое поведение.

Такое случается в функции `R_SortVisSprites`, приведённой на рис. 5.3.

```
for (ds=vissprites ; ds<vissprite_p ; ds++)
{
  ds->next = ds+1;
  ds->prev = ds-1;
}

vissprites[0].prev = &unsorted;
```

Рис. 5.3: Пример вызова неопределённого поведения при создании указателя на элемент перед первым элементом массива

В этом примере `vissprites` — массив определённый как `vissprite_t vissprites[MAXVISSPRITES]` и `ds` имеет тип `vissprite_t*`.

На первой итерации цикла `ds == vissprites` и вычисление значения выражения `ds-1` вызывает неопределённое поведение.

Эта ошибка была найдена благодаря контрактам для адресной арифметики, описанным в разд. 3.2.

5.4.2 Чтение за пределами массива

Функция `R_DrawColumn` при некоторых обстоятельствах может обращаться к элементам за пределами массива. Часть функции приведена на рис. 5.4.

```
// Determine scaling,
//  which is the only mapping to be done.
fracstep = dc_iscale;
frac = dc_texturemid + (dc_yl-centery)*fracstep;

// Inner loop that does the actual texture mapping,
//  e.g. a DDA-lile scaling.
// This is as fast as it gets.
do
{
  // Re-map color indices from wall texture column
  //  using a lighting/special effects LUT.
  *dest = dc_colormap[dc_source[(frac>>FRACBITS)&127]];

  dest += SCREENWIDTH;
  frac += fracstep;

} while (count--);
```

Рис. 5.4: Пример чтения за пределами массива

В этом примере переменные с префиксом `dc_` — глобальные переменные, которые используются как аргументы функции. Ошибка происходит в выражении `dc_source[(frac>>FRACBITS)&127]`. При особой конфигурации входных параметров может происходить чтение за пределами массива.

5.4.3 Разыменовывание нулевого указателя

Функция `P_GroupLines` иногда производит разыменовывание нулевого указателя. Соответствующая часть кода приведена на рис. 5.5.

Эта ошибка происходит в строчке `li->frontsector->linecount++`. В некоторых случаях `li->frontsector` может быть нулевым указателем. Разыменовывание нулевого указателя является неопределённым поведением в Си.

Эта ошибка была найдена в процессе вывода прикреплённости, опи-

```
// count number of lines in each sector
li = lines;
total = 0;
for (i=0 ; i<numlines ; i++, li++)
{
  total++;
  li->frontsector->linecount++;

  if (li->backsector && li->backsector != li->frontsector)
  {
      li->backsector->linecount++;
      total++;
  }
}
```

Рис. 5.5: Пример разыменовывания нулевого указателя

санной в разд. 3.5.

Результаты показывают, что Эйфель может быть использован для разработки систем мягкого реального времени, по уровню соответствующим уровню игры Doom. Контрактное программирование и механизмы безопасности пустых ссылок помогают в отладке программ и улучшения их качества. Примеры неопределённого поведения оригинальной программы были найдены благодаря портированию программы на Эйфель.

Однако, производительность программы ухудшается при включении проверки контрактов. Это делает проблематичным проверку всей системы сразу. Разделение системы на подсистемы позволяет проверять подсистемы по отдельности.

Портирование оригинальной системы показывает, что код на Эйфеле получается в два или три раза длиннее. Так же увеличивается размер исполняемого файла, но незначительно.

Финализированная версия системы на Эйфеле имеет достаточную производительность по сравнению с кодом на Си.

Получившиеся исходные коды опубликованы в интернете и являются ПО с открытым исходным кодом. Порт самой игры называется *Brie Doom* и доступен по ссылке. `https://github.com/imustafin/brie_doom`. Расширенная версия библиотеки *wrap_sdl* доступна по адресу `https://github.com/imustafin/wrap_sdl` и может быть переиспользована в других программах на Эйфеле, где требуется использование библиотеки SDL.

В целом, ответ на исследовательский вопрос «применим ли Эйфель для переноса вычислительно-интенсивной системы мягкого реального времени с языка Си?» — да. Мы считаем, что портирование программ с Си на Эйфель — это жизнеспособный вариант для портирования программ. Используя процесс, разработанный в этой работе, надёжность систем может быть улучшена во время портирования. Существующие инструменты для автоматического портирования могут быть улучшены путём включения процессов из этой работы.

Список литературы

[1] P. Poole и W. M. Waite, «Portability and adaptability,» в *Software Engineering, An Advanced Course, Reprint of the First Edition [February 21-March 3, 1972]*, 1975, с. 183—277.

[2] A. S. Tanenbaum, P. Klint и W. Bohm, «Guidelines for software portability,» *Software: Practice and Experience*, т. 8, № 6, с. 681—698, 1978.

[3] A. A. Terekhov и C. Verhoef, «The realities of language conversions,» *IEEE Software*, т. 17, № 6, с. 111—124, 2000.

[4] B. Meyer, «Applying 'design by contract',» *Computer*, т. 25, № 10, с. 40—51, 1992.

[5] id Software, *Doom*, 1993.

[6] B. Meyer, «Eiffel: A language and environment for software engineering,» *Journal of Systems and Software*, т. 8, № 3, с. 199—246, 1988.

[7] T. Tanaka, M. Hakuta, N. Iwata и M. Ohminami, «Approaches to making software porting more productive,» в *Proceedings of the 12th TRON Project international Symposium*, IEEE, 1995, с. 73—85.

[8] J. D. Mooney, «Bringing portability to the software process,» *Dept. of Statistics and Comp. Sci., West Virginia Univ., Morgantown WV*, 1997.

[9] M. Boyd, «Computing Calamities: Lessons Learned from Products, Projects, and Companies That Failed,» *Software Quality Professional*, т. 4, № 4, с. 45, 2002.

[10] D. Boulychev, D. Koznov и A. A. Terekhov, «On project-specific languages and their application in reengineering,» в *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, IEEE, 2002, с. 177—185.

[11] J. Martin и H. A. Muller, «C to java migration experiences,» в *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, IEEE, 2002, с. 143—153.

[12] G. Canfora, A. Cimitile и M. Munro, «An improved algorithm for identifying objects in code,» *Software: Practice and Experience*, т. 26, № 1, с. 25—48, 1996.

[13] S.-S. Liu и N. Wilde, «Identifying objects in a conventional procedural language: An example of data design recovery,» в *Proceedings. Conference on Software Maintenance 1990*, IEEE, 1990, с. 266—271.

[14] M. Trudel, C. A. Furia, M. Nordio и B. Meyer, «Really automatic scalable object-oriented reengineering,» в *European Conference on Object-Oriented Programming*, Springer, 2013, с. 477—501.

[15] J. George и B. D. Carter, «A strategy for mapping from function-oriented software models to object-oriented software models,» *ACM SIGSOFT Software Engineering Notes*, т. 21, № 2, с. 56—63, 1996.

[16] H. Gall и R. Klosch, «Finding objects in procedural programs: an alternative approach,» в *Proceedings of 2nd Working Conference on Reverse Engineering*, IEEE, 1995, с. 208—216.

[17] E. D. Demaine, «C to Java: converting pointers into references,» *Concurrency: Practice and Experience*, т. 10, № 11-13, с. 851—861, 1998.

[18] K. Kontogiannis, J. Martin, K. Wong, R. Gregory, H. Müller и J. Mylopoulos, «Code migration through transformations: An experience report,» в *CASCON First Decade High Impact Papers*, 2010, с. 201—213.

[19] K. Yasumatsu и N. Doi, «SPiCE: a system for translating Smalltalk programs into a C environment,» *IEEE Transactions on Software Engineering*, т. 21, № 11, с. 902—912, 1995.

[20] Novosoft, *C2J — C to Java translator*, `http://www.novosoft-us.com/solutions/product_c2j.shtml`, 2001.

[21] T. Waddington, *Java backend for GCC*, 2000.

[22] M. Trudel, C. A. Furia, M. Nordio, B. Meyer и M. Oriol, «C to OO translation: Beyond the easy stuff,» в *2012 19th Working Conference on Reverse Engineering*, IEEE, 2012, с. 19—28.

[23] G. T. Leavens и Y. Cheon, *Design by Contract with JML*, 2006.

[24] R. Kramer, «iContract-the Java/sup TM/design by Contract/sup TM/tool,» в *Proceedings. Technology of Object-Oriented Languages. TOOLS 26 (Cat. No. 98EX176)*, IEEE, 1998, с. 295—307.

[25] M. Karaorman, U. Hölzle и J. Bruno, «jContractor: A reflective Java library to support design by contract,» в *International Conference on Metalevel Architectures and Reflection*, Springer, 1999, с. 175—196.

[26] R. Henne-Wu, W. Mitchell и C. Zhang, «Support for Design by Contract TM in the C Programming Language,» дис. ... док., California State University, Sacramento, 2003.

[27] R. Plosch, «Design by contract for Python,» в *Proceedings of Joint 4th International Computer Science Conference and 4th Asia Pacific Software Engineering Conference*, IEEE, 1997, с. 213—219.

[28] P. Guerreiro, «Simple support for design by contract in C++,» в *Proceedings 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems. TOOLS 39*, IEEE, 2001, с. 24—34.

[29] Y. A. Feldman, «Extreme design by contract,» в *International Conference on Extreme Programming and Agile Processes in Software Engineering*, Springer, 2003, с. 261—270.

[30] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[31] J. Smart, *BDD in Action: Behavior-driven development for the whole software lifecycle*. Simon и Schuster, 2014.

[32] M. Goldstein, Y. A. Feldman и S. Tyszberowicz, *Refactoring with contracts*. IEEE, 2006.

[33] B. Meyer, «EiffelStudio: A Guided Tour,» Eiffel Software Technical Report TR-EI-68/GT. http://docs. eiffel. com ..., тех. отч., 2001.

[34] B. Meyer, A. Kogtenkov и E. Stapf, «Avoid a void: The eradication of null dereferencing,» в *Reflections on the Work of CAR Hoare*, Springer, 2010, с. 189—211.

[35] A. Kogtenkov, «Void safety,» дис. ... док., ETH Zurich, 2017.

[36] J. Velilla и L. Rix, «WrapC,» *https://github.com/eiffel-wrap-c/WrapC*, 2020.

[37] A. Leitner, «Eiffel Wrapper Generator,» *http://ewg.sourceforge.net/*, 2003.

[38] id Software, *DOOM*, `https://github.com/id-Software/DOOM/`, 2012.

[39] SDL Community, «Simple directmedia layer 2,» *http://www.libsdl.org/*, 2013.

[40] S. Howard и др., *Chocolate Doom*, `https://www.chocolate-doom.org/`, 2005.

[41] T. Littlefair, «An investigation into the use of software code metrics in the industrial software development environment,» 2001.

[42] ISO, *ISO/IEC 9899:2018 Information technology — Programming languages — C*. 2018.