Санкт-Петербургский Государственный Университет

Кафедра компьютерных технологий и систем

Меньщиков Максим Александрович

Выпускная квалификационная работа бакалавра

Нахождение условий гонки в коде на C методом статического анализа.

Направление 010300

Фундаментальная информатика и информационные технологии

Научный руководитель:
к.ф. - м.н., доцент Лепихин Т.А.

Санкт-Петербург
2016 г.

Содержание

Введение

Компьютерные технологии определяют вектор развития науки и техники. Программы для компьютеров — неотъемлемая их часть. В то время, когда сложность разработки прикладного ПО всё время падает вследствие повышения уровня абстракции и применения специализированных языков программирования, сложность же системного ПО для поддержки всего "парка" техники неминуемо возрастает. Это происходит из-за увеличения числа поддерживаемого аппаратного обеспечения, аппаратных архитектур и постоянно изменяющихся потребностей разработчиков.

Описываемые проблемы выражаются в нескольких аспектах: рост фактической сложности кода (измеряемой, например, методом Т.Дж.Маккейба [1]), разветвление кода для поддержки "необычного" железа, увеличение числа пользователей отдельных функций. Соответственно, происходит ухудшение качества программного продукта и снижение надежности системы.

Одним из важных достижений компьютерных технологий является параллелизм — возможность решать несколько задач одновременно — причем как в рамках одной программы, так и в пределах ОС и компьютера в целом. Такое логичное и правильное изобретение, однако, вносит свою лепту в описанный выше процесс.

Если открыть официальный сайт Linux, а именно журнал изменений за последние года, можно убедиться, что каждый год решаются *тысячи* проблем параллелизма [2]. В частности, следует упомянуть состояния гонки (*race conditions*).

Определение. Состояние гонки — это ошибка проектирования программы, при существовании которой корректность кода зависит от порядка выполнения его участков.

Сложность в том, что эта проблема является плавающей, т.е. она может возникать в абсолютно случайные моменты времени, не проявляясь во время тестирования или проявляясь незаметно для тестировщика. Более того, зачастую ошибка *накапливается*, и заказчики сталкиваются с тем,

что сертифицированная и много раз проверенная программа становится нестабильной.

Задача выявления состояний гонки выглядит ещё более непростой на фоне того разнообразия средств межпроцессного и межпотокового взаимодействия, которые предоставляют современные операционные системы. К примеру, в Linux для одной лишь блокировки использования переменных в разных потоках служит более трех базовых примитивов (spinlock, rwlock, semaphore). Часто в системном ПО выключают обработку прерываний, что на некоторых системах полностью блокирует системный планировщик задач. Также существуют различные варианты атомарных функций, которые по определению обеспечивают атомарность, но в сочетании с другими инструкциями вполне могут вызывать тяжело диагностируемые логические ошибки. На вопрос, как всё это многообразие будет между собой взаимодействовать, зачастую не может ответить даже опытный человек.

Всё, что способен определить "на глаз" человек, вполне реализуемо в рамках программного продукта. Однако это не так просто. Достаточно проблематично систематизировать все возможные ошибки, приводящие к состояниям гонки. Более того, семантика используемых функций серьезно зависит от операционной системы и может отличаться даже в сравнении с достаточно близкими версиями одного ядра.

Перед реализацией любой подобной системы необходимо чётко определить требуемый баланс между качеством и скоростью. Нахождение ошибок параллельности является NP-сложной задачей [3] и потому эффективный алгоритм определить достаточно сложно. Можно проверить весь код до малейших деталей, но при этом обнаружить, что на полную проверку программы уйдёт время, сравнимое с временем жизни Вселенной. А можно запустить *пере*оптимизированный код и понять, что не самый тривиальный ошибочный случай просто останется необнаруженным, и тогда встанет вопрос, есть ли вообще смысл от такого анализа.

Иногда в среде начинающих программистов можно заметить пренебрежение в отношении подобных проблем. Казалось бы, проблема возникает настолько редко, что замечается в одном случае на миллион. И одним из аргументов подобных авторов является производительность,

которая при отсутствии блокировок будет действительно выше, чем в ином случае [4]. С этим сложно поспорить, но если, например, незначительная ошибка при выведении графического объекта в одном кадре в час будет едва ли заметна, то вряд ли гипотетический пациент обрадуется малозаметной ошибке в работе медицинского прибора, которая, в каких-то случаях, может стоить ему жизни [5].

На эти доводы можно возразить: ведь не только состояния гонки приносят проблемы. И это верно. Но гонки особенно коварны — их незаметность и зависимость от внешних факторов заставляют быть особенно внимательными при проектировании систем. Изменения "железной" части программно-аппаратного продукта могут привести к задержкам при обработке, и проблема в некорректно написанном коде встанет в полный рост. Увеличит пользователь нагрузку на дисковую подсистему — и все принятые в программе предположения (*assumptions*) о времени обработки IO-операции окажутся неверными. Примеры можно приводить бесконечно.

В случае с гонками язык программирования как таковой роли не играет: они могут присутствовать во всех программах, язык которых не предполагает строгости формализации параллельного выполнения. Чистый C, рассматриваемый в данной работе, далеко не в первых рядах по популярности [6], но всё же стоит отметить, что большинство инфраструктурных open-source проектов написаны на C. Под термином "инфраструктурные проекты" здесь скрывается системное и, зачастую, свободное программное обеспечение, которое обеспечивает функционирование различных подсистем: маршрутизация, DNS, NTP, VPN, криптография, и т.д. Подобное ПО активно используется на серверах, во встраиваемых устройствах, на смартфонах, планшетах, иногда в автомобилях, уж не говоря о клиентских компьютерах, внедрение непроприетарного ПО в которые является одной из важных задач образовательного и государственного секторов многих стран мира.

Часто в новостных сводках можно услышать про "очередную найденную уязвимость в проекте X". И, между тем, весомый процент уязвимостей в системах безопасности — в тех самых базовых программах. К примеру, весьма простая ошибка обработки Heartbeat-запросов

в OpenSSL привела к возможности раскрытия пользовательской информации на более чем 17% веб-серверов в мире (уязвимость Heartbleed [7]). И хотя подобные ошибки не являются предметом рассмотрения, упомянутое “коварство” гонок позволяет предположить наличие сотен не менее значительных проблем в компьютерах на базе свободного ПО.

Существует множество программных продуктов, реализующих поиск ошибок в исходных кодах других программ. Они различаются по принципам работы: обычно за основу берутся методы статического или динамического анализа.

Динамический анализ основывается на внедрении сбора статистики в программу на этапе её компиляции. Естественными требованиями является необходимость модификации компиляторов (или установки “ловушек” в существующие), а также в целом возможность компиляции исходного кода. Если первый процесс является достаточно трудозатратным, то второе требование не всегда допустимо. Например, если возникает необходимость проверки лишь небольшого участка кода, то потребуется скомпилировать весь проект. Более того, помимо временных затрат на сборку, результат может оказаться не вполне ожидаемым. Ведь добавленные процедуры сбора статистики увеличивают время выполнения программы, тем самым уменьшая вероятность обнаружения гонки. Ну и наиболее важным недостатком является отсутствие полноценных проверок редко вызываемых частей программ.

Статический анализ лишен подобных проблем, но имеет свои собственные. В то время как динамический анализатор встраивается в уже рабочую программу, и его инструкции выполняются напрямую процессором, статический вариант анализатора вынужден сам быть некоторого рода интерпретатором. Это даёт большую гибкость, но значительно увеличивает сложность разработки. К примеру, банальное определение справедливости условий перехода по инструкции `if` уже не представляется такой тривиальной задачей [8].

Подводя итог вышеизложенному, можно сделать вывод, что работа в направлении статического анализа на состояния гонки в программах на языке C является перспективной задачей. Теоретически, такой проект способен выполнять точную, полную и достаточно быструю проверку на

подобный тип ошибок. И что особенно ценно, исправление найденных таким проектом ошибок позволит улучшить стабильность и надежность практически любого компьютера на основе Linux.

В рамках данной квалификационной работы будет разработан подобный статический анализатор кода на C.

1 Постановка задачи

1.1 Проблемы поиска состояний гонки

На любом языке программирования и на любой процессорной архитектуре *последовательное выполнение операций* происходит в тех случаях, когда точно известно, что за оператором S_i следует оператор S_{i+1}. Последовательное выполнение операторов в общем смысле не подразумевает возможности параллельного исполнения инструкций.

Рассмотрим случай простой последовательной программы. Такая программа обладает свойством *детерминированности*.

Определение 1.1. Детерминированный алгоритм — алгоритм, результат которого и последовательность шагов к достижению которого однозначно определяются входными параметрами.

Даже однопоточная и на первый взгляд сторого последовательная программа обычно опирается на API — набор системных вызовов и каких-либо библиотек — и поэтому *по-настоящему* детерминированными такие программы назвать сложно. Недетерминированность, помимо всего прочего, приводит к нарушениям порядка доступа к переменным, и вследствие этого возникают различные ошибки параллельности, например, гонки.

Определение 1.2. Гонка (*состояние гонки*) — набор конфликтующих обращений к общему ресурсу, для которых порядок доступа не определен, т.е. они выполняются в произвольном порядке или могут происходить одновременно.

Таким образом, наша задача заключается в обнаружении “конфликтующих наборов”. Однако, как определить конфликты обращений к переменным, имея только исходный код программы? Для начала его надо прочитать.

На языке C есть достаточно активно используемый разработчиками программ препроцессор.

Определение 1.3. Препроцессор языка C — программа, выполняющая обработку директив препроцессора в исходном коде.

До обработки препроцессором код может быть не в полной мере читаемым. После — мы получим возможность считать синтаксическое дерево кода.

Основная проблема этого этапа заключается в том, что необходимо поддерживать всевозможные конструкции языка. Несмотря на нахождение стандартов C в широком доступе, написание собственного парсера — задача для отдельной работы. Поэтому мы вынуждены адаптировать какой-либо парсер для нашей задачи.

После получения синтаксического дерева мы должны выделить в нём наиболее важные для понимания соответствующего кода детали. Этот этап называется *семантическим* анализом.

Разобрав семантику, мы имеем достаточно сведений, чтобы начать непосредственно процедуру анализа на состояния гонки.

Однако возникают резонные вопросы.

1. Какие алгоритмы применить?

2. Что является необходимым условием существования конфликта доступа?

После обнаружения гонок нам необходимо их показать пользователю. Для этого требуется графический интерфейс. Какой SDK (*Software Development Kit*) применить с целью сохранения кроссплатформенности?

Иногда анализ может занимать существенное время. Возможно ли его проведение в облаке? Насколько в таком случае ускорится выполнение?

Ответы на эти вопросы я постараюсь дать в данной работе.

1.2 Задача

Опираясь на описанную проблематику, задачу квалификационной работы можно охарактеризовать следующим образом: *разработать программный комплекс, осуществляющий нахождение состояний гонки в программах на C методом статического анализа.*

Также, чтобы удостовериться в применимости написанного на практике, в работе будет рассмотрено создание кроссплатформенного графического интерфейса пользователя и "облачной" части, позволяющей значительно ускорить выполнение анализа.

В работе можно выделить несколько этапов.

1. Обзор существующих решений и алгоритмов.
2. Определение технических характеристик проекта.
3. Препроцессинг исходного кода на языке C.
4. Парсинг кода и составление синтаксического дерева.
5. Составление семантической модели программы.
6. Анализ семантической модели на "гонки".
7. Создание графического интерфейса пользователя.
8. Создание облачного сервиса.
9. Тестирование.
10. Подведение итогов.

1.3 Требования

Можно выделить следующие требования к проекту.

1. *Достаточно полная поддержка языка C.*
2. *Сравнимость.* Результат нахождения гонок необязательно должен быть полным. В работе применяются алгоритмы, обеспечивающие

целенаправленный поиск некоторых типов ошибок. Поэтому основное качество результата предлагается оценивать по критерию сходства с другими проектами.

3. *Кроссплатформенность*. Программа не должна быть привязана к конкретному аппаратному обеспечению или операционной системе.

4. *Применимость*. Проект должен быть полезным в различных условиях, например, в облаке, в билд-системах и т.д.

2 Обзор литературы

2.1 Алгоритмы

Одной из первых работ по верификации программ считается статья T.Hoare [9]. В ней вводится аксиоматический подход к доказательству *частичной корректности* простых последовательных программ. Z.Manna и A.Pnuelli в работе [10] развивают его идеи для доказательства *полной* корректности программ. Множество статей распространяют логику Хоара в том числе и до многопоточных программ [11, 12]. Однако следует отметить, что логика Хоара хоть и является достаточно точным методом определения правильности программ (в т.ч. и многопоточных), но описываемый метод едва ли применим в реальности в силу своей сложности и трудоемкости.

L.Lamport в работе [13] представляет более применимый в реальных условиях алгоритм “Happens-before” для установления порядка различных событий в сложных системах. Он же лёг в основу многих динамических анализаторов кода, но имеет существенный недостаток: не всегда информация об очередности имеется в исходном коде. К тому же, он едва ли подходит для частичной проверки проектов, когда информации меньше по определению.

Eraser — динамический анализатор кода, разработанный S.Savage, M.Burrows, G.Nelson, P.Sobalvarro и T.Anderson, впервые представивший алгоритм `Lockset` [14]. В данной работе этот алгоритм будет применен в

статическом варианте. Более подробное описание алгоритма приведено в 3.6.4.

В работе нам понадобится алгоритм анализа псевдонимов. Проблема возникает в основном при использовании указателей: один и тот же указатель может указывать на один или несколько разных объектов. Один из алгоритмов был предложен L.O.Andersen в его тезисе [15] (идея была развита в дальнейшем B.Steensgaard в работе [16]). Он получил название "Points-to analysis". Идея его в том, что для каждого объекта указательного типа мы определяем набор возможных объектов, на которых объект *может* указывать. Предложенный механизм является нечувствительным к потоку исполнения (*flow-insensitive*), что несколько снижает качество его результатов, но сильно упрощает весь процесс. Алгоритм будет впоследствии использован в данной работе (3.6.8).

2.2 Современные проекты

Если говорить о современных проектах, то невозможно не отметить некоторые из них.

RacerX — это проект Dawson Engler и Ken Ashcraft из Стенфордского университета [17]. Основная идея проекта — в статическом обнаружении состояний гонки и *deadlock*. Информация о методе парсинга не раскрывается, однако в статье описывается подход к обнаружению гонок. Заключение о наличии гонки требует ответа на несколько вопросов:

1. Корректен ли Lockset?
2. Может ли код, содержащий незащищенный доступ к данным, конкурировать с другим потоком? (является ли код многопоточным?)
3. Должен ли код быть защищен?

В процессе сбора ответов на данные вопросы, RacerX использует ранговую систему, позволяющую выделять наиболее интересные нарушения потокобезопасности. Корректность Lockset проверяется во многом аналогично 3.6.4. Вопрос о многопоточности метода решается путем проверки вызываемых им функций. Ответ на необходимость защиты

переменной даётся методом “programmer beliefs” — оценивается, считает ли программист некоторый фрагмент заслуживающим блокировки (например, даже некорректно блокируемая последовательность всё-таки заблокирована им — а значит он *верит*, что это необходимо). Помимо этого, RacerX использует различные эвристические методы для сокращения числа ложных результатов. Полная проверка 1.8 миллионов строк кода за 2-14 минут предполагает очень хорошую оптимизацию. По своей идее именно RacerX является более развитым аналогом проекта, рассматриваемого в данной работе, но следует отметить, что его исходный код недоступен общественности, как и бинарные файлы, вследствие чего о проекте можно рассуждать только основываясь на заверениях авторов.

Другим известным проектом по статическому анализу является Coverity. Используемые им алгоритмы неизвестны, тем не менее, в числе авторов [2] есть и авторы RacerX, что предполагает наличие общих черт между двумя этими проектами. Для использования доступен бесплатный облачный сервис Coverity Code Scan. Авторы заявляют о тысячах проблем, исправленных в open-source проектах.

Helgrind, подпроект известного динамического анализатора кода **Valgrind** [18], реализует алгоритм *Happens-before* [13]. К нему применимы все преимущества и недостатки оного. Проблемой **Helgrind** является то, что он проверяет только проблемы, связанные с использованием библиотеки `pthread`, что означает его неприменимость, к примеру, для проверки операционных систем, их драйверов и т.д.

2.3 Семантика операционных систем

Для повышения точности анализа нам потребуется знать принципы различных операционных систем (3.6.1). Классическим является труд Э.Таненбаума [19]. В нём описываются смыслы системных вызовов Linux и Windows, приводится подробное описание работы потоков в них, принципы планировки задач. Довольно обстоятельно раскрываются проблемы блокировок и условий гонки.

Книга Таненбаума фундаментальна, но, к сожалению, иногда упускает некоторые детали о работе конкретных операционных систем.

Авторы D.P.Bovet, M.Cesati [20], J.Corbet, A.Rubini, G.Kroah-Hartman [21] предоставляют подробный взгляд на логику операционной системы Linux, драйверов, механизма прерываний и т.д.

О внутренних аспектах Windows подробно рассказывает книга M.Russinovich, D.A.Solomon и A.Ionescu “Windows Internals” [22, 23].

Информация из этих книг была использована при подготовке наборов семантики приведенных ОС и позволила серьезно увеличить точность анализа.

3 Разработка программного продукта

3.1 Технические аспекты

Невозможно начать проект, не разобравшись с его характеристиками.

3.1.1 Выбор языка программирования

Язык программирования играет определяющую роль в проекте. Он влияет на его структуру, а также самым прямым образом влияет на реализацию. Для языка программирования был определен следующий список требований: *простота*, *функциональность*, *объектно-ориентированность*, *кроссплатформенность*, *наличие простых в использовании методов параллелизации и удобного синтаксиса для операций выбора, фильтрации и поиска элементов в массиве.*

На выбор технологического стека повлияли не только указанные характеристики: также были учтены существующие на платформах решения для задачи синтаксического парсера (3.3).

По совокупности характеристик был выбран язык C#. Если первый пункт весьма субъективен (хотя C-подобный синтаксис и считается простым для восприятия), то второй пункт обеспечивается развитыми стандартными библиотеками .NET. Объектно-ориентированность у C# присутствует по определению. Кроссплатформенность среди ОС семейства Windows обеспечивается за счёт .NET Framework, а совместимость с Linux и другими системами — благодаря open-source проекту Mono.

Технология LINQ [24], поддерживаемая начиная с .NET Framework 4 — важный “кирпичик” в фундаменте проекта, так как наличие данной опции сильно упрощает работу с синтаксическими деревьями и позволяет писать “однострочники” вместо сложных функций.

Обычно код на первых этапах пишется без учета многопоточности, и потому важным аспектом было наличие простого способа распараллелить вычисления. В частности, `System.Threading.Tasks` добавляет метод `Parallel.For` [25], переход на использование которого занимает не больше нескольких строк кода.

Аналогичным по характеристикам языком является Java, но решающий выбор был сделан в пользу решения от Microsoft из-за удобного синтаксического парсера, предоставляемого для работы с .NET-платформами компанией.

3.1.2 Выбор среды разработки

Написание проекта под технологию .NET/Mono существенно ограничивает нас в выборе среды.

Microsoft Visual Studio — среда по умолчанию для .NET-проектов. Она обладает широкими возможностями по проектированию, написанию, отладке любых программ. Особо следует отметить стабильность среды. Именно данная IDE (*Integrated Development Environment*) стала основной для проекта.

MonoDevelop и его форк **Xamarin Studio** тоже обладают полным функционалом для построения .NET-приложений. Однако важной функцией данных сред оказалось наличие визуального редактора графических интерфейсов GTK# (Glade). Подробнее разработка интерфейса рассматривается в главе 3.7. Их минусом является низкая стабильность, плохая подсветка ошибок.

Таким образом, основной код писался в Microsoft Visual Studio, а некоторые части графического интерфейса — в Xamarin Studio.

3.1.3 Выбор системы контроля версий

Системы контроля версий необходимы для правильного и удобного менеджмента исходного кода, трекинга ошибок и истории изменений. Выбор встал между системами Git и SVN.

Git [26] — основная система учёта ревизий в Linux. Легко устанавливаема, проста в использовании. Позволяет создавать отдельные ветки для удобства разработки отдельных функций и впоследствии объединять их. Также обладает удобной системой фильтров исходного кода, что немаловажно для решений (*solutions*) Visual Studio, так как последняя генерирует большое число временных файлов.

SVN [27] — более старая система. Обладает удобным набором утилит для Windows, тоже проста в использовании, но не поддерживает ветки и содержит худшую систему фильтрации файлов.

Выбор был сделан в пользу Git, не в последнюю очередь благодаря его нативной интеграции в Visual Studio.

3.2 Препроцессинг исходного кода

Предварительная обработка кода нужна для выполнения директив препроцессора. Обычно препроцессор является одной из частей набора для сборки приложения, в которую входят компиляторы, компоновщики и другие компоненты.

В ходе изучения вопроса было принято решение реализовать собственный препроцессор. Причин для этого было несколько. Во-первых, большая гибкость собственного препроцессора по сравнению с аналогами. Некоторая информация из препроцессинга необходима для дальнейших этапов. Во-вторых, для анализа не требуется реализовывать полный функционал: например, статическому анализатору нет никакого дела до парсинга `#pragma`, зато важно уметь правильно раскрывать макросы. В-третьих, так как одной из целей проекта является анализ фрагментов исходного кода, то препроцессор должен уметь работать с неполной информацией. Всё это невозможно в сторонних разработках.

Были рассмотрены различные варианты реализации, но предпочтение было дано следующему: предварительная обработка

осуществляется простым детерминированным конечным автоматом, или иными словами — построчно, в цикле, с сохранением истории.

3.2.1 Оператор #if

Препроцессор должен уметь обрабатывать оператор `#if`. По стандарту, в качестве операнда здесь может выступать любое арифметическое или логическое выражение. Например, в Linux часто встречается макросы `KERNEL_VERSION` и `LINUX_VERSION_CODE`, и, соответственно, директивы:

```
#if KERNEL_VERSION > LINUX_VERSION_CODE(2, 6, 32)
...
#endif
```

Естественно, результат операции должен быть посчитан на этапе препроцессинга. Для реализации этого был выбран проект `NCalc` [28], который обладает способностью вычислять значения любых заданных программистом операций, правильно расставлять приоритеты операторов, учитывать скобки, а также имеет задаваемую пользователем процедуру получения значения переменных по названию, что идеально подходит для препроцессинга.

Оценка значения выражения производится следующим образом. Переопределяется метод `EvaluateParameter`, который запрашивает значения переменных. Они берутся, во-первых, из числа аргументов макроса, а во-вторых — из числа констант и прочих определений препроцессора, которые уже хранятся в истории. Метод `EvaluateFunction` же вводит новую псевдо-функцию `defined`, задача которой — выяснить, существует ли макрос в истории препроцессора. Другими обрабатываемыми процедурами являются макрофункции. Для них происходит формирование нового набора аргументов и рекурсивный запуск оценщика.

Препроцессор занимается учётом блоков *условной компиляции*. Для этого в стек записываются пары значений `ExecutedOnce` (выполнилось ли условие хотя бы единожды) и `LastDecision` (результат оценки последнего условия) для каждой `#if/#elif/#else/#endif`-конструкции, которые обновляются вплоть до `#endif`, а затем удаляются из стека.

Вопрос, нужно ли включать строку в итоговый код, решается так: *все* `LastDecision`*, содержащиеся в стеке, должны быть равны* `true`*, т.е. все последние условия должны выполняться.*

3.3 Парсинг кода и составление синтаксического дерева

Предварительно обработанный код — код без директив препроцессора — полученный на предыдущем этапе, на данном этапе является одним из параметров процедуры парсинга кода. Наша цель — получить синтаксическое дерево кода, т.е. преобразовать конструкции на языке C в дерево из объектов, характеризующих эти конструкции.

3.3.1 Выбор парсера

Реализация парсера языка — сложная задача, а написание собственной процедуры разбора языка вполне может являться темой отдельной работы. Существует большое число различных парсеров, и в ходе работы над проектом я тестировал несколько из них.

Я обозначил следующие требования к парсеру:

1. Поддержка C-подобного языка.
2. .NET-платформа: отсутствие необходимости использовать маршалинг.
3. Либеральная лицензия.
4. Продуманность синтаксического маппинга.
5. Перспективы развития.
6. Отсутствие сильной “заточки” под конкретный язык программирования — т.е. возможность в разумные сроки подключить другой язык программирования.

Под эти требования идеально вписался проект Microsoft Roslyn. Он поддерживает C#, Visual Basic.NET; создан авторами .NET-платформы;

распространяется под лицензией Apache 2.0, разрешающей коммерческое использование и позволяющей распространять проект под любой лицензией. Roslyn активно развивается корпорацией Microsoft и внедряется в IDE Visual Studio как основной синтаксический парсер.

Для всех поддерживаемых языков используются обобщенные классы для синтаксических структур.

3.3.2 Совместимость C и C#

Безусловно, C напрямую не поддерживается в Roslyn в настоящий момент, но между C и C# существуют определенные сходства. В некотором роде C является подмножеством C# за исключением некоторых моментов. В таблице приведен список наиболее значимых отличий, выявленных мною в процессе разработки.

Конструкция C	**Конструкция C#**	**Комментарий**
`#include`	—	В C# сильно упрощенный препроцессор и поэтому директивы `#include` не существует.
`sizeof`	`sizeof`	В C в этом операторе можно использовать как названия типов, так и названия переменных/массивов. В C# — только названия типов. Ближайший аналог для массивов — Marshal.SizeOf(x), однако для целей анализа корректное нахождение размеров объктов не требуется.

Конструкция C	**Конструкция C#**	**Комментарий**
—	`params`	В C не существует такого модификатора. Однако Roslyn обрабатывает его как ключевое слово и поэтому, если в коде используется название переменной `params`, то парсинг происходит некорректно.
`union`	`[FieldOffset]`	В C# типы данных — управляемые — и поэтому использование `union`, в котором запись в одну переменную может перезаписывать другую, не одобряется даже авторами языка. Однако, для наших целей, пусть и с потерей некоторой точности, можно банально заменить `union` на `struct`.
`struct`, `union`, `enum` перед названиями типов	—	В C перед названиями структур, объединений или перечислений необходимо использовать соответствующее ключевое слово. Например: `enummsg_typet;` Зачастую для избавления от лишнего слова используют конструкцию `typedef`. Для наших же целей достаточно удалить подобное слово при предварительной обработке кода.

Конструкция C	**Конструкция C#**	**Комментарий**
Строки	Строки	C#, в отличие от C, не проводит конкатенацию смежных строк. С целью обеспечить разбор средствами Roslyn достаточно добавить оператор `+` между двумя (или больше) смежными строками.

Список не является всеобъемлющим, однако он включает наиболее показательные отличия C от C#.

Несмотря на приведенные различия языков, на большинство проблем можно найти решения, не меняющие семантический смысл программы, поэтому использование Roslyn выглядит разумным с учетом всех его достоинств.

3.3.3 Архитектура Roslyn

Верхним элементом иерархии Roslyn является класс `SyntaxTree`, который инкапсулирует в себе всю информацию об анализируемом исходном коде, *однозначно* идентифицирующую этот исходный код, и наоборот. Главным же элементом можно назвать `SyntaxNode`. Это базовый класс для всех синтаксических объектов. Он обеспечивает доступ ко всем `SyntaxTrivia` и `SyntaxToken`, ассоциированным с ветвью, а также позволяет обращаться к другим связанным ветвям (например, потомкам или родителям).

В то время, как `SyntaxTrivia` имеет незначительную роль — они отвечают за комментарии, пробелы и т.д., `SyntaxToken` представляют собой терминальные символы языковых грамматик — литералы, ключевые слова, идентификаторы и т.д.

Важное качество `SyntaxTree` и `SyntaxNode` — они по своей реализации *неизменяемые*. Следовательно любая попытка внесения

изменений в такие объекты приведет к созданию новых соответствующих объектов. Это вносит свою лепту в процесс внесения корректировок в код [29], так как последовательное множественное использование метода `ReplaceNode` затруднено (два разных синтаксических дерева не имеют связи друг с другом без применения специальных средств трекинга ветвей).

3.4 Связь препроцессора и парсера

Для достижения поставленной задачи раздельных препроцессора и парсера недостаточно, и на это есть множество причин. Одной из наиболее важных проблем является обработка макросов. Помимо них существуют дополнительные директивы-операторы, требующие нашего вмешательства. К тому же, синтаксис может оказывать большое влияние на процесс анализа кода. Рассмотрим все эти случаи подробнее.

3.4.1 Простые макросы

Простыми макросами можно назвать подстановки, не изменяющие синтаксической сути исходного кода программы. *Константы*, *функции* — важнейшие их типы.

Пример 3.1. Макросы-константы выглядят следующим образом.

```
#define NUM_THREADS 5
```

В этом примере всё просто: все `IdentifierNameSyntax` можно заменить на соответствующий литерал.

Пример 3.2. Макросы-функции требуют больших усилий с нашей стороны.

```
#define KERNEL_VERSION(a,b,c)
                      (((a) << 16) + ((b) << 8) + (c))
```

Для выполнения данной подстановки потребуется найти все `InvocationSyntax` с `Expression` равным `KERNEL_VERSION`

(IdentifierNameSyntax), вставить ArgumentSyntax вместо IdentifierNameSyntax, и "вклеить" выражение в исходный текст.

Возможны различные комбинации данных примеров, и все они должны быть обработаны. Например, подстановки ExpressionSyntax вместо StatementSyntax (и наоборот) невозможны, поэтому такие случаи тоже учитываются в процессе работы. Также макросы могут подменять определения функции, но и это обрабатывается в коде препроцессора-парсера.

3.4.2 Макросы — вариативные функции

В языке C синтаксис вариативных функций используется достаточно часто — например, в printf-подобных алгоритмах. Он может использоваться и в макросах.

Пример 3.3. `#define ERROR(_args...) printk(KERN_INFO _args)`

Основной проблемой является "пересадка" аргументов ERROR в вызов printk с учётом всех SyntaxTrivia, SyntaxToken и различий C/C#. В частности, в приведенном примере происходит неявная конкатенация строк (KERN_INFO — строка, состоящая из одного символа и обозначающая тип сообщения в журнале событий).

3.4.3 Операторы Stringification и Token-Pasting

Иногда в макросах возникает необходимость обернуть параметр в кавычки или "вклеить" параметр в уже имеющий идентификатор. Тривиальных способов для этого в языке не существует, и поэтому авторами языка C были созданы специальные операторы # и ##.

Пример 3.4. Оператор Stringification

```
#define varname(name) #name
int i;
printf("Variable is called %s", varname(i));
```

Реализация оператора приведена в Приложении 1.

Пример 3.5. Оператор Token-Pasting

```
#define type(name) ##name _t
type(uint32) i; // аналогично "uint32_t i;"
```

Реализация функции во многом аналогична реализации оператора Stringification.

3.4.4 Общие корректировки

Результат работы синтаксического разбора — это прямое переложение исходного кода в набор синтаксических объектов, к которому обычно может быть применена обратная процедура, т.е. синтаксическое дерево *однозначно* идентифицирует некоторый исходный код, и наоборот. Основной задачей этапа предварительной обработки и чтения синтаксиса является упрощение представления кода для того, чтобы дальнейший анализ имел дело с меньшим числом сущностей. Применив какие-либо преобразования, мы, несомненно, изменим *синтаксическую* структуру кода, но мы имеем право применить такие преобразования, в результате которых код не изменит *семантическую* структуру.

Пример 3.6. Незначительное преобразование из

```
if (foo) bar();
```

в

```
if (foo) { bar(); }
```

изменяет синтаксическую структуру `IfStatementSyntax`: `Expression` меняет тип с `ExpressionStatementSyntax` на `BlockSyntax`.

Объективно, семантических изменений в приведенном выше примере нет, но присутствие `BlockSyntax` позволяет рассматривать вызов `bar()` как последовательность, состоящую из одного элемента. Данное преобразование выполняется для всех конструкций, где фигурные скобки опущены. Помимо очевидного выигрыша в количестве кода (один единственный случай вместо двух), это позволяет упростить алгоритм фрагментации кода (3.6.5).

3.4.5 Платформо-зависимые корректировки

Другим примером полезного исправления синтаксиса является добавление пустой строки после вызовов функций разблокировки блокировочных примитивов. Это — платформо-зависимая корректировка, так как смысл функций блокировки зависит от используемой операционной системы.

Пример 3.7. Следующий случай иллюстрирует добавление пустых строк.

```
spin_lock(&sl);
... // фрагмент 1
spin_unlock(&sl);
; // Добавленный фрагмент.
spin_lock(&sl);
... // фрагмент 2
spin_unlock(&sl);
```

Это полезно для упрощения логики фрагментации, в частности, позволяет воспринимать пространство между фрагментами как самостоятельный фрагмент, состоящий из одного элемента `EmptyStatementSyntax`. Более подробно назначение этой операции будет разъяснено в пункте 3.6.5, а реализация приведена в Приложении 2.

3.5 Семантический анализ

Разбор смысла кода в общем смысле — основная задача работы. Однако мы сделаем разделение базовой семантики и семантики, специфичной для анализа на состояния гонок, и в данном разделе рассмотрим базовую часть.

Здесь и далее мы полагаем, что синтаксическое дерево кода уже получено путем использования препроцессора и парсера из предыдущих разделов.

3.5.1 Основные семантические понятия

Рассмотрим термины, используемые в проекте. Ключевым классом является `Resource`.

Определение 3.8. Ресурс — основная семантическая единица, обозначающая объект, имеющий некоторое название, тип, а также область значения.

Примечание. Иными словами, любая переменная или процедура — это ресурс.

Также приведем формальное определение ресурса.

Определение 3.9. Ресурс — это набор $r = (n, t, E, p)$, единственным образом ассоциируемый с каждой переменной и процедурой из исходного кода проверяемой программы, где n — название переменной или процедуры; t — тип ресурса — любая осмысленная комбинация флагов, характеризующих объект; E — набор выражений, вычисление значений которых и последующее объединение результатов дают область значений; p — родительный ресурс.

Выделяется несколько типов ресурсов.

1. Static

   Модификатор `static` служит для нескольких целей: глобальные объекты внутри модуля, приобретая модификатор `static`, становятся невидимыми для других модулей; статические локальные переменные — являются в некотором роде глобальными.

2. Local

   Обозначает локальные переменные, по умолчанию перекрывающие любые глобальные с тем же именем.

3. Parameter

   Подтип локальной переменной — параметр метода.

4. StructureMember

   Член структуры.

5. Method

   Обозначает любой метод.

6. NotExistent

   Данный флаг назначается всем объектам, не присутствующим в рассматриваемом коде, но к которым имеются обращения.

Область значения ресурсов формируется из *выражений* путем объединения результатов вычисления их значений.

Определение 3.10. Выражение — любое унарное, бинарное или тернарное действие над операндами-ресурсами или литералами.

В описании алгоритмов нам потребуется ещё несколько определений.

Определение 3.11. Конечные ресурсы выражения — прямо или косвенно задействованные в выражении ресурсы.

Определение 3.12. Родителем называется ресурс, **потомки** которого не могут существовать независимо от родителя.

Примечание. В качестве примера связи “родитель-потомок” можно привести объекты структурных типов и их структурные члены.

3.5.2 Алгебра диапазонов значений

В любых программах мы неизбежно сталкиваемся с какими-либо выражениями. Выражения-утверждения обычно формируют область значений ресурсов, а выражения-условия используются для выбора ветви в *переходах*.

Так или иначе, нам необходимо как-то формализовать выражения. Поэтому была разработана так называемая **алгебра диапазонов значений**.

Наименьшей единицей в алгебре диапазонов является класс `Value`. Он содержит в себе некоторое числовое значение. В качестве исключения в нем также может храниться строковый литерал — для удобства их

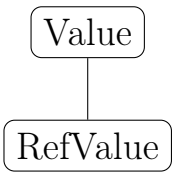

Рис. 1: Схема наследования классов Value

учёта. Также важным является *уровень косвенности* (Indirection level): количество указателей на пути к значению переменной. Значение самого указателя — адрес, хранящийся в нём — по естественным причинам не сохраняется.

Класс `RefValue` — это фактически ссылка на другой объект Value, что бывает полезно с целью упростить повторное использование одного и того же объекта.

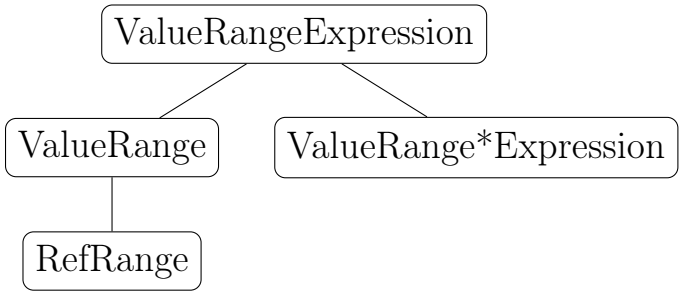

Рис. 2: Схема наследования классов Expression

Следующим по значимости является класс `ValueRangeExpression`, предоставляющий абстрактный метод `Evaluate`, возвращающий набор результатов вычисления выражения. Самым простым наследником является `ValueRange` — хранилище списка значений, для которого переопределены все арифметические и логические операции.

Схемы наследования данных классов приведены на рис.1 и 2.

Основная идея алгебры диапазонов значений такова: так как алгоритмы в нашем проекте не чувствительны к потоку выполнения, то теоретически любая арифметическая или логическая операция может выполняться к любой паре значений из диапазонов операндов.

Пример 3.13. $[5, 6, 7] + [3, 4] = [5 + 3, 5 + 4, 6 + 3, 6 + 4, 7 + 3, 7 + 4] = [8, 9, 9, 10, 10, 11] = [8, 9, 10, 11]$

Следовательно, все переопределенные операторы строят новый

`ValueRange`, в который включаются результаты применения оператора к каждой паре.

Частными случаями `ValueRangeExpression` являются `ValueRangeBinaryExpression` (бинарные выражения: `a+b`), `ValueRangePrefixUnaryExpression` (префиксно-унарные выражения: `++i` или `&x`), `ValueRangePostfixUnaryExpression` (постфиксно-унарные выражения: `i++`) и `ValueRangeTernaryExpression` (тернарные выражения: `x?a:b`). Все они переопределяют абстрактный метод `Evaluate` и выполняют заданные им при вызове конструктора операции.

3.5.3 Обходчик дерева

Как уже было сказано ранее, в нашем распоряжении имеется синтаксическое дерево рассматриваемого исходного кода. Для получения семантических данных необходимо каким-либо образом обработать синтаксические данные. Этой цели служит **обходчик** дерева, который мы будем называть `SyntaxRunner`, чтобы избежать дублирования названия `SyntaxWalker` из пространства имен Microsoft Roslyn.

Обход дерева осуществляется *поиском в глубину* [30] по синтаксическому дереву. При этом нет большой необходимости избегать рекурсивности: любой исходный код заведомо конечен. Потому обходчик отличается предельно простой структурой.

Важной задачей обходчика является упрощение синтаксических конструкций. Дело в том, что дальнейшим этапам анализа не особенно важно, какие именно синтаксические структуры описывают некоторую семантику. Эта задача выполняется путем обобщения синтаксических конструкций. Ниже приведен их список.

1. **VariableDeclaration** — определение переменной
2. **MethodDeclaration** — определение метода.
3. **BlockTraverse** — начало или завершение блока, обычно обозначаемого фигурными скобками.
4. **Invocation** — вызов функции.

5. **Expression** — выражение-утверждение (например, `x=a+b`).

6. **Return** — ключевое слово `return` и связанное с ним выражение.

7. **For** и **While** — соответствующие циклы.

8. **ConditionCheck** — проверка какого-либо условия.

9. **Goto** и **GotoLabel** — ключевое слово `goto` и метки, в которые возможен переход.

Другим важным свойством `SyntaxRunner` является *расширяемость*. Следующим этапам анализа требуется получать некоторую *свертку* выделенной из синтаксиса семантики. Свойство реализуется путем использования механизма *событий* [31]. Помимо этого, каждый класс, желающий получать информацию от SyntaxRunner, должен быть наследован от `ISyntaxRunner` и зарегистрирован при помощи метода `Attach`. После того, как такой класс подпишется на события, соответствующие приведенным выше синтаксико-семантическим группам, он становится полноправным семантическим анализатором-компаньоном.

Пример реализации приведён в Приложении 3.

3.5.4 Ресурсный обходчик

Первым зарегистрированным обходчиком синтаксического дерева является **ресурсный обходчик** (ResourceRunner). Его задачи приведены в таблице.

Событие	**Действие**
VariableDeclaration	Ресурсный обходчик создает новый ресурс с типом `Local`, если текущий уровень выше нулевого, с модификатором `Static`, если соответствующее ключевое слово имеется в коде декларации, и типом `Parameter`, если переменная объявляется внутри списка параметров метода.

Событие	**Действие**
MethodDeclaration	Обходчик добавляет ресурс `Method` с модификатором `Static` (если требуется).
BlockTraverse	Управляет стеком, хранящим текущую последовательность ветвей-блоков (BlockSyntax). Помимо этого, добавляет или удаляет новые списки в стек `CurrentResources`.
Expression	Преобразовывает синтаксические выражения в их семантические аналоги и ассоциирует их с ресурсами, на которые происходит воздействие.
Return	Обновляет список выражений, составляющих область значения ресурсов-методов.
Invocation	Создаёт несуществующие (`NotExistent`) ресурсы-методы в случае, если вызываемый метод не присутствует в коде. Обновляет список возможных значений параметров методов.
For, While	Декларируют новые локальные переменные, действительные только для блока цикла, генерирует события `Expression` для инициализаторов переменных и отсекает заведомо невыполнимые блоки путем использования функции `Evaluate` на условии цикла.
ConditionCheck	Добавляет новое выражение-условие в соответствующий список для всех ресурсов, участвующих в выражении.

3.5.5 Механизм “обещаний”

Отдельного упоминания стоит случай отсутствия *полного* знания об обрабатываемой программе в момент отработки какой-нибудь синтаксической ветви.

Пример 3.14. Выражение `x=a()` встречается по коду раньше, чем происходит определение функции `a`.

В приведенном примере получается так, что обходчик *обязан* добавить выражение `a()` в область значения `x`. Однако в этот момент область значения `a()` ещё неизвестна. Для решения проблемы используется класс `RefRange`, который является ссылкой на `ValueRange`.

При обнаружении несуществующего объекта создаётся `RefRange` со значением переменной `Promise` равной `true` и пустой ссылкой на `Resource`. Впоследствии при добавлении какого-либо ресурса происходит обновление всех выполненных обещаний — они преобретают ссылку на `Resource` и, т.к. `RefRange` по своей архитектуре указывает на внутренние переменные ресурса, то этот ссылочный класс практически становится неотличим от настоящего `ValueRange`.

3.6 Анализ кода на состояния гонки

В предыдущих разделах мы провели синтаксический разбор и получили важнейшие семантические сведения из исходного кода, в частности — для каждого ресурса мы получили список выражений, составляющих область его значений. Теперь наша задача — использовать эти сведения для получения данных о состояниях гонки. Для выполнения задачи нам необходимо ввести несколько понятий.

3.6.1 Семантика системных вызовов

В процессе анализа кода мы зачастую сталкиваемся с вызовами внешних функций. Идея автоматического анализа всех таких методов выглядит интересно, но на практике — проверять код всей операционной системы и, возможно, других библиотек несколько накладно по

вычислительным ресурсам, по времени. Предварительное задание семантики определенного числа системных вызовов является одним из лучших решений.

Подобный подход прослеживается во множестве проектов. Обычно составление *аннотаций* к системным методам доверяется пользователю.

В случае с поиском состояний гонки мы вполне имеем право описать избранные смысловые аспекты функций. Одним из наиболее важных является *контекст выполнения*.

Определение 3.15. Контекст выполнения метода — состояние операционной системы в момент вызова данной функции, влияющее на семантику системных вызовов.

Метод, явно или неявно принимающий на вход указатель на функцию, так или иначе вызывает эту функцию в каком-либо контексте. Этот контекст будем называть *контекстом системного вызова*.

Пример 3.16. `request_irq(91, handle_irq, ...)`

Данный фрагмент кода зарегистрирует `handle_irq` как обработчик прерывания под номером 91. `handle_irq` же в конечном счете будет выполнена в контексте аппаратного прерывания. Поэтому контекстом системного вызова `request_irq` назовём `KHardIrq`.

Приведем значимые отличия введенных нами контекстов.

Контекст	**Системный вызов**	**Комментарий**
`Generic`, `KGeneric`	–	Контекст по умолчанию (KGeneric — по умолчанию для ядра).

Контекст	**Системный вызов**	**Комментарий**
`Thread`	`pthread_create` (Linux), `CreateThread` (Windows)	Контекст для функций, используемых в потоках. Не имеет весомых отличий от `Generic`, но в некоторых случаях не позволяет использовать API пользовательского интерфейса.
`Signal`	`signal`, `sigaction` (Linux)	Обработчики сигналов (Signal) в Linux. Выполнение каких-либо функций за исключением атомарных нежелательно. В момент вызова обработчика сигнала другие потоки приостанавливаются, что может порождать состояния гонки и прочие ошибки в случае неправильного использования. Два обработчика сигналов не могут конфликтовать между собой.
`KHardIrq`	`HookInterrupt` (Windows), `IWDFDevice3::CreateInterrupt` (Windows), `setup_irq` (Linux), `request_irq` (Linux)	Обычно обработчики аппаратных прерываний должны выполняться *очень* быстро –устанавливая какую-либо задачу в очередь и завершаясь. Основное ограничение — использование только неблокирующих системных вызовов.

Контекст	**Системный вызов**	**Комментарий**
`KTasklet`	Linux	Контекст программного прерывания. Может прерывать любую операцию, может прерываться аппаратным прерыванием. Разные тасклеты могут выполняться одновременно, однако один тасклет выполняется только на одном процессоре в любой момент времени.
`KTimer`	Linux, Windows	Контекст программного прерывания. Может прерывать любую операцию, может прерываться аппаратным прерыванием. Выполняет работу через определенные промежутки времени.
`KThread`	Linux	Запускаются в контексте процессов. Схожи с `Thread`, но существуют только в ядре.

Приведенный список не полон, но отражает основные контексты в ОС Linux и Windows.

3.6.2 Контексты пользовательских функций

Помимо системных вызовов, мы также можем рассматривать понятие контекста по отношению к функциям в проверямом исходном коде.

Определение 3.17. Контекст функции называется **внешним**, если он определен посредством анализа семантики использующих указатель на данную функцию методов.

Определение 3.18. Контекст функции называется **внутренним**, если он определяется путем проверки системных вызовов, используемых внутри данного метода.

Следовательно, уже упомянутые контексты системных вызовов по факту влияют на *внешний* контекст пользовательской функции.

Определение внутренних характеристик процедур не представляет особых затруднений, но сильно зависит от семантики операционной системы.

Одним из наиболее удобных методов является нахождение `InvocationExpressionSyntax` и `IdentifierNameSyntax` по определенным правилам. К примеру, в ОС Linux методы аллокатора памяти в ядре имеют названия, начинающиеся с буквы "k" — например, `kmalloc`, `kzalloc`. Их использование — верный признак контекста `Kernel` [21].

Использование такого макроса, как `IRQ_RETVAL`, фактически обозначает контекст прерывания, в то время как использование `delay`, `sleep`, наоборот, в общем случае показывает обратное [21].

Существуют и другие признаки, однако, приведенные здесь примеры отражают основной их принцип.

Определение 3.19. Внешний контекст допускает внешнее использование функции (или, иначе, её вызов за пределами исследуемой программы), если он принадлежит любым классам за исключением `Unknown`, `Generic` и `KGeneric`.

Объяснение этому определению простое. Описанные нами в 3.6.1 классы контекстов системных методов приводят к вызову отправляемых в регистрирующую процедуру указателей на функции из системы. Соответственно, внешний вызов здесь фактически гарантирован.

3.6.3 Блокировки

Поиск состояний гонки невозможен без учета *захваченных блокировок*.

Определение 3.20. Блокировка — механизм ограничения параллельного доступа к каким-либо объектам.

Блокировки обычно реализуются на основе *атомарных операций* — в частности, `compare-and-swap`, `test-and-set` и других — но встречаются и безблокировочные алгоритмы [20]. “Получение разрешения” на использование связанных с блокировочным примитивом переменных обычно обозначается термином *захват*.

Наиболее распространенными блокировками являются **мьютексы**, **семафоры**, **критические секции**, **спин-локи**.

Определение 3.21. Мьютекс (mutual exclusion — взаимное исключение) — блокировочный примитив, захват которого одновременно возможен только одним потоком, тогда как остальные потоки остаются в состоянии ожидания высвобождения объекта.

Критические секции аналогичны мьютексам, однако на практике они используются только в пределах одного процесса.

Определение 3.22. Семафор — блокировочный примитив, захват которого возможен некоторым заранее заданным количеством потоков, тогда как остальные потоки остаются в состоянии ожидания.

Определение 3.23. Спинлок — блокировочный примитив, аналогичный мьютексу по идее, но не вызывающий принудительную планировку процессов и потоков.

Спинлоки удобны в тех случаях, когда временные затраты на планировку процессов сильно превышают среднее время ожидания перехода примитива в “свободное” состояние.

Не приведенные выше методы блокировки обычно (за редкими исключениями) являются производными от этих трёх, поэтому они не представляют особого интереса.

Часто используемым методом обеспечения корректного совместного доступа является *запрет прерываний*. Запрет производится путём вызова

соответствующей системной функции, которую можно считать некоторого рода блокировочным примитивом. Прерывания в таком случае обычного блокируются для текущего процессора.

Более крупное объединение данных понятий называется *Lockset*.

Определение 3.24. Lockset (от англ. *набор блокировок*) — совокупность блокировок и их состояний.

Для удобства, здесь и далее в работе мы будем под Lockset понимать набор *захваченных* объектов. Объектами в нашей терминологии выступают *ресурсы*.

Итоговая архитектура классов, отвечающих за блокировки, изображена на рис.3, 4 и 5.

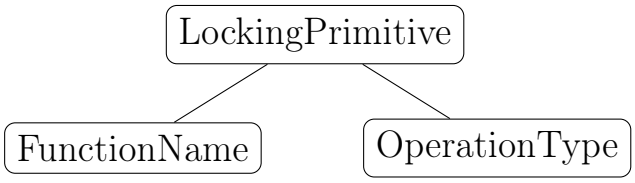

Рис. 3: Схема LockingPrimitive

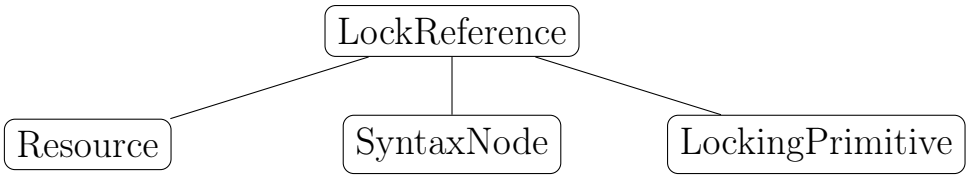

Рис. 4: Схема LockReference

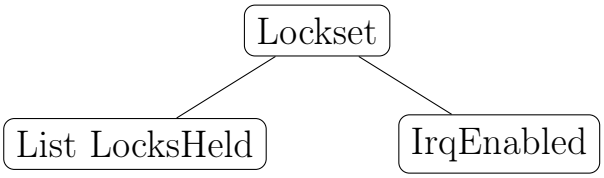

Рис. 5: Схема Lockset

3.6.4 Пересечение Lockset

Операция пересечения является наиболее важной функциональной частью класса Lockset. Суть операции — в поиске блокировочных объектов,

обеспечивающих взаимоисключение защищенных такими Lockset'ами областей.

Предположим, что у нас есть два потока, не ограниченных никакими блокировками. Тогда доступ к одной и той же переменной из этих потоков будет небезопасен. Получается, что операция пересечения вернёт пустое множество.

Обернув код обработчика одного из потоков в мьютекс, мы не изменим ровным счётом ничего: параллельная запись всё равно будет возможна. Пересечение — пустое множество.

Добавив тот же мьютекс во второй поток, параллельное изменение переменных будет невозможно. Пересечение — добавленный мьютекс. Однако если мы бы добавили *другой* мьютекс, то итоговое множество всё равно осталось бы пустым.

Некоторые типы блокировок имеют уровни разрешенного доступа — чтение или запись. В таком случае безопасность будет придавать наличие *хотя бы одного* запрета записи. Это вполне согласуется с условиями Бернстайна.

Теорема 3.25. ***Условия Бернстайна*** *определяют следующие правила безопасного чтения/записи в ячейки памяти:*

Если процесс P_i записывает ячейку M_i, тогда процесс P_j не может её читать.

Если процесс P_i читает информацию из ячейки M_i, то процесс P_j не может в неё записывать.

Если процесс P_i записывает информацию в ячейку M_i, то процесс P_j не может в неё писать.

Примечание. Иными словами, одновременно можно только читать из ячейки. Никакие операции не должны происходить одновременно с единовременной записью.

Семафоры, как уже было сказано в 3.6.3, обладают способностью пускать внутрь "запретной" секции n потоков. Однако, ставя цель поиска

состояний гонки, мы вполне можем считать, что семафор пропускает либо 1 поток, либо 2 — редукция большего числа потоков до 2 вполне оправдана, так как при проведении статического анализа мы не можем быть абсолютно уверены, что часть слотов уже не занята.

Мы можем сформулировать правила пересечения Lockset следующим образом.

1. Выделить общие объекты двух множеств.
2. Если список пуст — вернуть пустое множество.
3. Сравнить общие объекты.
   (a) Если хотя бы один объект предоставляет доступ на запись — перейти к следующей паре.
   (b) Иначе:
      i. Если объект — семафор, принимающий больше двух потоков, то удалить объект из множества.
      ii. Если объект — не семафор — удалить объект из множества.
4. Вернуть оставшиеся элементы множества.

Пример реализации пересечения Lockset приведен в Приложении 4.

3.6.5 Фрагментация исходного кода

Проверка каждой строки на предмет наличия в ней состязательных ситуаций — долгая задача. Возможным её упрощением может быть группировка строк кода по некоторым признакам, что позволило бы уменьшить количество проверяемых областей программ. Рассмотрим один из подходов к задаче фрагментации кода.

Определение 3.26. Фрагмент исходного кода — некоторая конечная последовательность строк исходного кода, обладающая определенными общими свойствами.

Под данное определение подпадает почти любой набор кода, однако мы будем считать именно Lockset таким “общим свойством”. Рассмотрим примеры.

Пример 3.27. Здесь фрагмент один, поскольку набор блокировок пуст и не меняется вовсе.

```
void test()
{
    i = 5;
    j = 4;
}
```

Пример 3.28. Здесь два фрагмента: с одной стороны, заблокированный фрагмент с $i = 5$, с другой — $j = 4$ без всяких блокировок.

```
void test()
{
    spin_lock(&lock);   // 1
    i = 5;              // 1
    spin_unlock(&lock); // 1
    j = 4;              // 2
}
```

Пример 3.29. Здесь мы сталкиваемся с новыми обстоятельствами. Во-первых, случай, когда `lock` захвачен, а во-вторых, когда не захвачен. Более того, мы не можем быть уверены, что значение x не поменяется за время выполнения строк 3, 4 и 5, следовательно, строка 8 может быть как заблокирована, так и не заблокирована.

```
1: void test()
2: {
3:  if (x == 6)
4:      spin_lock(&lock);
5:  i = 5;
6:  if (x == 6)
7:      spin_unlock(&lock);
8:  j = 4;
9: }
```

Приведенные выше примеры заставляют нас сформулировать некоторые правила фрагментации.

Утверждение 3.30. Фрагмент не включает в себя строки разных уровней вложенности.

Утверждение 3.31. Фрагмент завершается либо по окончании блока, либо при изменении Lockset, в частности, при добавлении нового блокировочного объекта в набор или удалении оного.

За внешней простотой этих правил скрывается один важный аспект. Фрагмент какого-либо уровня вложенности либо запускается безусловно (уровень тела функции), либо имеет некоторое, возможно пустое, условие (конструкции `if`, `while`, `for` обязательно включают условия). Условия входа в блок напрямую влияют на характер состязательных ситуаций. Это обуславливает наш интерес не смешивать разные уровни вложенности или разные блоки.

Переходя непосредственно к алгоритму фрагментации, покажем, что данная операция возможна путем использования `SyntaxRunner`.

1. Вход в блок или выход из него характеризуются событием `BlockTraverse`.

2. Использование примитивов синхронизации, меняющих Lockset, вызывает событие `Invocation`.

3. Обход синтаксического дерева ведётся в глубину, что обеспечивает полноту осмотра дерева на предмет блокирующих конструкций.

Обходчик дерева, выделяющий фрагменты, назовём `FragmentRunner`. Сформулируем алгоритм его работы.

Пусть `CurrentFragments` — список действующих на данный момент фрагментов. Пусть каждый фрагмент обладает свойствами `Since` и `Until` — указатели на синтаксические ветви начала и конца фрагмента.

1. При входе в блок

(a) Создаем столько новых фрагментов, сколько уже существует в списке `CurrentFragments`, при этом в качестве `Since` устанавливаем первую подветвь блока, а `Lockset` копируем из каждого фрагмента.

(b) Если `CurrentFragments` пуст, то создаем новый фрагмент с пустым `Lockset`. `Since` — начало блока.

2. При выходе из блока

(a) Удаляем все фрагменты, `Since` которых принадлежит законченному блоку.

(b) В случае, если фрагмент — открывающий, то для каждого фрагмента из закрытого блока создаем новый фрагмент с объединением Lockset фрагментов.

Определение 3.32. Фрагмент называется ***открывающим***, если хотя бы один блокировочный объект им захватывается, но не выпускается.

Определение 3.33. Фрагмент называется ***закрывающим***, если хотя бы один блокировочный объект им только выпускается.

Определение 3.34. Фрагмент, не являющийся открывающим или закрывающим, называется ***нейтральным***.

Вызовы функций обрабатываются `FragmentRunner` несколько сложнее.

Ранее в 3.4.5 мы приводили алгоритм вставки пустых строк в код. Объясним подробнее назначение этой операции. Дело в том, что между двумя нейтральными фрагментами существует “дыра”: сразу после разблокировки, и, соответственно, выхода из первого фрагмента, операционная система может принять решение о планировке процессов, по завершении которой сразу же стартует следующий фрагмент. Но планировка может заставить процессор перейти к выполнению другого потока, что теоретически может вызывать состязательные ситуации. Поэтому необходимо этот момент как-нибудь зафиксировать. Но дело в том, что по принятому соглашению фрагмент начинается

со строки x и заканчивается строкой $x + n$, и в данном случае получается, что у нас нет ни начальной, ни финальной строки. Вставка `EmptyStatementSyntax` выглядит логичным решением проблемы, тем более её можно замаскировать путем удаления `SyntaxTrivia`.

Перейдём к алгоритму разбора вызовов функций.

1. Проверяем, является ли вызванная функция частью примитива синхронизации.

2. Проверяем, захватывает ли она блокировку.

   (a) Для каждого текущего фрагмента, принадлежащего исследуемому блоку, создаем новый фрагмент с `Since` равным ветви и клонированным Lockset, объединенным с `LockReference`, указывающим на захватываемую в данной ветви блокировку.

3. Проверяем, высвобождает ли она блокировку.

   (a) Для каждого текущего фрагмента, принадлежащего исследуемому блоку, создаем новый фрагмент с `Since` равным *следующей* ветви и клонированным `Lockset`, но без высвобождаемого объекта.

Помимо всего прочего, в целях оптимизации дальнейшей проверки, события `Expression` и `ConditionCheck` сохраняют выражения в соответствующие им фрагменты. Это реализовано посредством взаимодействия `FragmentRunner` с `ResourceRunner`.

3.6.6 Цепочки фрагментов

Выделенные в коде фрагменты сами по себе не представляют никакой ценности. Дело в том, что мы почти никогда не можем рассматривать методы по отдельности: они вызывают друг друга, формируя длинный путь от первоначального вызова до итогового, рассматриваемого фрагмента. Представление Lockset тоже вырастает, но зачастую несущественно, так как обычно количество объектов синхронизации в коде не столь велико, как количество методов.

Логичным в данной ситуации выглядит введение понятия **цепочки фрагментов**.

Определение 3.35. Цепочка фрагментов — последовательность фрагментов, каждый из которых, кроме последнего, либо является предшественником следующего фрагмента в смысле его выполнения процессором, либо содержит вызов метода, ассоциированного с последующим элементом.

Цепочка фрагментов однозначно определяет путь выполнения кода процессором и выделяет единственный локально возможный набор блокировок.

Цепочки фрагментов в общем случае не могут начинаться со статических функций. Однако это правило нарушается вследствие наличия косвенных вызовов функций через указатели, например, зарегистрированных как обработчики прерываний.

3.6.7 “Накачка” цепочек фрагментов

При наличии списка фрагментов логичной представляется задача нахождения набора их цепочек. Приведем возможный алгоритм.

1. Пройдём по списку методов и выделим те, которые не являются статическими и не обладают внешне-вызываемым контекстом.
2. Для всех фрагментов верхнего уровня вложенности, принадлежающих найденным методам, добавляем в итоговый список по одной новой цепочке фрагментов из одного элемента.
3. Начнём “накачку” цепочек.
   (a) Рассмотрим каждую цепочку по отдельности.
   (b) Выделим последний фрагмент цепочки.
   (c) Для всех выражений, принадлежащих фрагменту, определим конечные ресурсы-методы.

(d) Для каждого фрагмента верхнего уровня, принадлежащего определенному ресурсу-методу, склонируем рассматриваемую цепочку и добавим в неё данный фрагмент, если он ещё не принадлежит цепи.

(e) Рассмотрим дочерние элементы фрагментов и аналогично добавим их в скопированные цепочки.

(f) Повторим процесс до тех пор, пока количество новых элементов не снизится до нуля.

Конечность процесса очевидна: количество фрагментов в исходном коде ограничено и рано или поздно каждый потенциальный фрагмент уже окажется в цепочке. Производительность же процесса сильно зависит от сложности проверяемого кода и может быть достаточно низкой. Однако следует отметить, что процесс прекрасно поддается параллелизации и отправке на кластер, так как деятельность цикла по разбору отдельных цепочек не зависит от разбора остальных.

В результате работы алгоритма мы получаем почти все возможные и невозможные пути выполнения кода процессором. “Почти все” — потому что полный разбор рекурсивных методов при данном подходе невозможен: методы не могут повторяться в цепочке. Это несколько уменьшает наши возможности по последующему анализу, но, с другой стороны, написание кода, который создаст проблемы (в виде ложных результатов) в дальнейшем — чрезвычайно редкая и не всегда возможная практика. Задача же построения абсолютно полного списка *реально возможных* путей выполнения является едва ли выполнимой задачей, требующей полной эмуляции всей среды выполнения, что невозможно при существующих мощностях компьютеров.

3.6.8 Анализ псевдонимов

В любом программном коде в самых разных методах встречаются указатели, являющиеся ссылками на одни и те же объекты. Поиск небезопасных обращений к общим переменным сильно затрудняется без знания данных *псевдонимов*.

Определение 3.36. Псевдонимом переменной называется связанная с ней переменная, изменение любой из которых приводит к изменению другой.

В языке C псевдонимы могут быть реализованы только при помощи указателей.

В 3.5.4 приведены задачи ресурсного обходчика. При внимательном рассмотрении можно заметить, что оный собирает все выражения для каждой переменной. Пересмотрев определение 3.11, нельзя не отметить, что таким образом для каждой переменной собираются все *конечные ресурсы*, т.е. все объекты, на которые ссылается данная переменная тем или иным способом.

Подход имеет недостаток: параметры экспортируемых методов зачастую не ссылаются ни на один объект, если мы рассматриваем только сам модуль, а не его пользователей. Следовательно, необходимо проводить дополнительное исследование на псевдонимы на параметрах методов. В нём предлагается проверять типы переменных. Сформулируем важную теорему для *основанного на типах* поиска псевдонимов.

Теорема 3.37. *Ошибочное приравнивание объектов двух указательных сложных типов в ходе анализа псевдонимов может привести лишь к появлению ложно-положительных результатов анализа на гонки (ложно-отрицательные результаты не будут присутствовать)*

Доказательство. Истинные результаты мы не рассматриваем в силу их правильности по определению.

Предположим, что мы ошибочно приравняли два объекта, и в дальнейшем анализе сгенерировался ложно-отрицательный результат, следовательно, гонка не обнаружена, хотя она имеется. По определению, гонки существуют только при возможности параллельного изменения данных некоторой переменной, но объекты разные из-за ошибочности приравнивания. Возникает противоречие. Утверждение доказано. □

Пример 3.38. Проиллюстрируем ситуацию.

```
void f1()
{
    struct hcd *hc1 = malloc(sizeof(struct hcd));
    hc1->x = 5;
    free(hc1);
}

void f2(struct hcd *hc2)
{
    hc2->x = 7;
    ...
}
```

В примере `hc1` и `hc2` — структурно-указательные псевдонимы, но при этом возможный диагноз о наличии гонки в строках `hc1->x=5` и `hc2->x=7` является ложно-положительным, так как объект `hc1` уничтожается до момента его передачи куда-либо.

Объяснение этому правилу следующее: структурные типы обычно являются “хранилищем” контекстной информации для различных методов и потому особо важны для анализа.

Передача структурных (сложных) типов по значению является плохой и крайне редкой практикой, так как и размер стека, и размер регистров большинства архитектур недостаточно велики, чтобы хранить большое количество крупных структур. Более того, структуры, переданные по значению, — по определению копии информации, поэтому проблемы с гонками в них возникать не могут (за исключением случаев, когда в составе структуры находится указатель, и доступ производится к его членам — но это уже другой объект). Именно поэтому в утверждении мы говорим об указателях.

Как показывает Horwitz в работе [32], данная задача является NP-сложной. Поэтому мы не стремимся к идеальной точности исследования. Наличие же ложно-положительных результатов вполне удовлетворяет задаче работы.

3.6.9 Анализ на состояния гонки

Объединим информацию из всех предыдущих глав в итоговом анализе.

Оговоримся, что данный анализ не даёт полной картины о гонках, связанных с последовательностью выполнения действий, но способен предложить список *возможных* ошибок параллельности определенных классов, в частности — связанных с *дисциплиной блокирования* (*locking discipline*) [14].

Рассмотрим все пары цепочек фрагментов, для `Lockset` которых метод `Intersect` (3.6.4) возвращает пустое множество. Каждую такую пару мы можем рассматривать в параллельном режиме. Одну из цепочек для интуитивности сравнения назовём *левой*, а другую — *правой*.

Для начала выделим все выражения левой цепочки, не являющиеся условиями или *чистыми вызовами*. Для каждого такого выражения выделим записываемую переменную, которая для присваиваний является левой частью утверждения, а для унарных выражений — операндом. Отфильтруем локальные переменные и параметры — их изменение как таковое не влияет на состояния гонки. Далее, если записываемая переменная — член структуры, то следует уточнить, является ли сама структура локальной, и если да, то необходимо пропустить такую переменную.

На данном этапе мы имеем кандидата в синтаксическом смысле: переменная *теоретически* может быть изменена где-нибудь в другом месте.

Теперь проверим условия *правой* стороны. Если среди конечных ресурсов выражений-условий правой стороны имеется псевдоним (3.6.8) рассматриваемой переменной левой стороны, то проведём следующий простой анализ.

1. Подсчитаем пересечение `Lockset` (3.6.4) левой и правой сторон.
2. Если оно пустое (т.е. нет защищающих блокировок) и синтаксические ветви сторон различаются то гонка обнаружена.

Таким образом мы проверим условия Бернстайна (3.25) на предмет *записи* и *чтения*.

После проверки условий обратимся к присваивающим утверждениям. Возьмём изменяемые переменные и сравним их с проверяемой левосторонней переменной. Если они псевдонимы, то проведём аналогичный анализ `Lockset` и сделаем заключение о наличии гонки. Если изменяемая переменная — параметр, то дополнительно проверим конечные ресурсы, и тоже добавим новую гонку в итоговый список.

Проверим также следующую ситуацию. В цепи имеется доступ к одной и той же переменной, но под разными блокировками. Это может привести к проблеме, когда первый поток записывает переменную, используя один `Lockset`, а второй поток — другой, и это в пределах одной цепи. Для решения проблемы просканируем список фрагментов цепочки и попытаемся убедиться, что начиная с первого использования переменной и до последнего фрагмента у нас всегда имеется непустое пересечение `Lockset`. Если оно пусто — необходимо проинформировать пользователя об ошибке.

Данный алгоритм обладает некоторыми недостатками.

1. В основном алгоритм направлен на нахождение фрагментов кода, не соблюдающих *необходимых* условий потокобезопасности. По отдельности записи данных в отдельные переменные могут быть безопасными, но в совокупности могут вести к ошибкам.

2. Алгоритм не проверяет серийность. В частности, если выполнение двух цепочек жестко разделено по времени, то алгоритм всё равно выдаст ошибки. Это несколько ограничивает возможности алгоритма при работе с некоторыми типами программ (решение подобной проблемы затрагивается в алгоритмах Happens-Before [13]).

3.7 Пользовательский интерфейс

3.7.1 Выбор SDK

Для конечных пользователей программы довольно важное значение имеет интерфейс. С целью выбора лучшего кроссплатформенного SDK для разработки UI (User Interface) было протестировано множество вариантов. К SDK были предъявлены следующие требования.

1. Как уже было отмечено, *кроссплатформенность*. Поддержка как можно большего числа платформ без значительных изменений в коде (желательно без них вовсе).

2. Наличие гибких интерфейсных примитивов для комфортного написания интерфейса.

3. Простой API.

4. Наличие удобной среды разработки, по возможности — со встроенным графическим дизайнером.

Самым простым и очевидным вариантом был API **Windows Forms**, стандартный для платформы .NET. Несмотря на то, что он поддерживается в Mono, были выявлены значительные недостатки его реализации в этой среде, и приложение работало по-разному в Linux и Windows, что неприемлемо. Остальные требования выполнялись, однако плохая работоспособность полностью перекрывала все преимущества.

XWT (Xamarin Window Toolkit) соответствует всем требованиям, кроме наличия удобного редактора, что не совсем удобно для быстрой разработки программного обеспечения. Однако он основан на GTK, C#-версия которого оказалась лучшим решением.

GTK# (GIMP Toolkit для C#) полностью кросс-платформенный: заявлена поддержка Mono (соответственно, большинства современных дистрибутивов Linux), Windows, а также OS X. API действительно прост и удобен, что позволяет легко добавлять новые элементы управления даже из кода. Наконец, в составе MonoDevelop/Xamarin Studio присутствует удобный визуальный редактор. Все эти факторы в совокупности позволили сделать выбор в пользу GTK#.

3.7.2 Архитектура решения

Для программы был выбран вкладочный интерфейс.

Общим элементом интерфейса является `ComboBox` с выбором платформы (Linux, Windows) и кнопки запуска непосредственно анализа, включения подчеркивания ошибок и редактирования директив препроцессора.

Для каждого файла исходного кода выделяется отдельная вкладка, содержащая два текстовых поля, список ошибок и трассировку стека (Stack Trace — фактически, последовательность вызова функций). За это отвечает класс `Tab`. Метод `CreateTab` создает новую вкладку в окне интерфейса и связывает редакторы с соответствующими моделями представления. `Analyze` инициализирует платформенную семантику в зависимости от выбора операционной системы, включает счетчик времени, и запускает полный анализ кода. Время его выполнения отображается в интерфейсе.

В качестве элемента управления текстового редактора используется редактор из MonoDevelop — кросс-платформенный и основанный на GTK#. Добавление подчеркивания ошибок прозводится путём добавления маркеров (Markers) к инстансу `Document` из объектной структуры редактора. Менеджмент маркеров осуществляется через их учёт в `List` и методы `AddMarker` и `ClearMarkers`.

Два текстовых поля в окне приложения служат цели правильного восприятия информации пользователем. Как было описано в 3.6.9, анализ осуществляется попарно, следовательно "виноватой" в конкретной ошибке всегда является пара цепочек фрагментов, и один редактор показывает проблемную *запись*, тогда как другой показывает конфликтующую строчку.

Снимок интерфейса приведен в Приложении 5.

3.8 Облачный сервис Azure

Для демонстрации возможности применения облачных технологий в задаче ускорения анализа был выбран сервис **Microsoft Azure**. Его использование выглядит логичным на фоне того факта, что проект основан на .NET Framework. Это обеспечивает возможность использовать библиотеки проекта в облаке без единой правки.

Сервис разделен на две роли, стандартные для шаблона "Cloud Service" [33]: `WebRole` и `WorkerRole`. `WebRole` представляет собой веб-интерфейс, позволяющий пользователю ввести код проверяемой программы и сохраняющий его в пользовательский сессии, а также предоставляющий возможность отправить задачу в `WorkerRole`,

обладающий существенно большей производительностью, и показывающий результаты выполнения.

3.8.1 Архитектура WorkerRole

WorkerRole представляет собой простейшее облачное приложение, основной класс которого наследуется от `RoleEntryPoint` [33].

В методе `OnStart` происходит создание очереди (`ServiceBus`) для сообщений с исходным кодом. К очереди создаётся *клиент* (`QueueClient`), позволяющий непосредственно получать сообщения.

В методе `Run` происходит асинхронное чтение очереди сообщений посредством использования `QueueClient.ReceiveAsync()`. Далее сообщение десериализуется стандартными средствами .NET Framework и проверяемый исходный код подвергается всем стадиям анализа. Результат выполнения простейшим образом сериализуется и отправляется в очередь сообщений, указанную в теле `BrokeredMessage` веб-частью.

Также `WorkerRole` регистрируется как хаб в SDK `SignalR`, что позволяет упростить процесс передачи данных между `WebRole` и `WorkerRole`, используя методы, вызываемые веб-частью по названию через обозначенный хаб. Метод `Analyze`, открытый для доступа извне, создаёт очередь сообщений со случайным именем (сгенерированного как строковое представление `GUID`) и заставляет основной цикл роли выполнить запрос.

3.8.2 Архитектура WebRole

WebRole является простейшим ASP.NET-приложением, основанным на MVC (Model — View — Controller) [34]. Внешний вид генерируется с применением библиотек **jQuery** и **Bootstrap**.

Основные представления следующие:

1. Index — главная страница.

2. Editor — редактор кода. Сохранение выполняется вовнутрь `Session` через использование соответствующего глобального `Controller`.

3. Analyze — отправка задания в `WorkerRole` и загрузка страницы с результатами. Код отправки исследуемой программы приведен в

Приложении 6.

4. Results — тривиальный список результатов.

3.8.3 Настройка Azure

Для веб-части используется виртуальная машина конфигурации `Standard_A1`, обладающая самой низкой производительностью из всех возможных в Azure. Рабочая часть сервиса в режиме ожидания базируется на `Standard_D1v2`, но средствами облачного провайдера автоматически масштабируется до необходимого уровня при поступлении запросов, основываясь на данных загрузки центрального процессора и памяти. Виртуальные машины `Standard_DXv2` обладают более быстрыми процессорами [35], что уменьшает время на проведение анализа.

3.8.4 Производительность

Базовые виртуальные машины вроде `Standard_D1` не сильно отличаются по производительности от обычных пользовательских компьютеров (а скорость и не требуется в режиме ожидания), но `Standard_D3` и далее уже превосходят обычные компьютеры. Так как алгоритмы из 3.6.6 и 3.6.9 предполагают хорошее распараллеливание и небольшое число межпоточных взаимодействий, то увеличение числа ядер почти линейно влияет на производительность анализа, а большое количество памяти позволяет рассматривать проекты с очень большим количеством фрагментов (3.6.5).

Запросы от пользователей успешно разносятся на разные виртуальные машины с применением автоматического увеличения числа инстансов роли через AutoScale от Azure.

3.8.5 Ограничения и перспективы проекта

Выполненный облачный сервис соответствует поставленным нами задачам, однако не является полной заменой графического интерфейса. Это не проблема Azure или ASP.NET, как и не ограничение проекта:

основной идеей создания облачной части было желание протестировать применимость библиотек как таковую.

Перспективными направлениями развития проекта могло бы быть улучшение редактора и более подробная визуализация ошибок, что в сочетании с дополнительным (и достаточно существенным) развитием основной платформы в сторону обнаружения обыкновенных ошибок программирования позволило бы приблизиться по удобству и функциональности к известным коммерческим онлайн-проектам вроде **Coverity Code Scan**.

3.9 Тестирование

Для раннего обнаружения ошибок и регрессий в коде применяются юнит-тесты. Отдельная программа-тестер выполняет проверку сложных случаев взаимодействия различных этапов анализа, выявленных в процессе разработки.

3.9.1 Препроцессор

Большое внимание уделяется тестированию правильной работы препроцессора. В то время как простые макросы не требуют специальной обработки, макросы, добавляющие фрагменты синтаксических конструкций, представляют особый интерес.

Множество тестов проверяют, как работают макросы вроде `module_init` [21], макросы со специальной обработкой параметров (3.4.3).

Основной метод тестирования выглядит следующим образом.

1. Задание исходного кода и директив препроцессора.
2. Запуск препроцессора на заданном фрагменте кода.
3. Сравнение обработанного кода с эталоном.

Также проверяется обработка таких ключевых слов C# как `params`, `lock`, `sizeof` и т.д. (3.3.2), в частности, правильность их замены на аналогичные конструкции/слова языка C.

В общей сложности было сгенерировано более 50 тестов, проверяющих функционал данного этапа, большинство из них — ручным способом.

3.9.2 Парсинг

Проверка правильности парсинга отдана на откуп команде разработчиков Roslyn, выполняющей все виды тестирования при каждом релизе.

3.9.3 Различные виды анализа

К сожалению, проведение всех тестов в значительной степени не представляется возможным из-за синтаксической "свободы" C-подобных языков. Поэтому для тестирования были отобраны сложные случаи, обнаруженные во время разработки. К примеру, модуль анализа псевдонимов (3.6.8) проверяется на случае рекурсии, модуль фрагментации (3.6.5) — на странных с точки зрения практики применениях примитивов блокировки (например, двойная разблокировка мьютексов), а модуль "накачки цепочек" (3.6.6) — на проверке, создаются ли цепочки, начинающиеся с чисто статических методов, вызывающихся извне методов и т.д.

3.9.4 Реальное тестирование

Программа была протестирована в реальных условиях в одной из коммерческих компаний Санкт-Петербурга. Анализатор помог исправить несколько сложных и запутанных случаев гонок.

Пример 3.39. В одном из проприетарных драйверов для Linux существовала следующая проблема: в таймере (который, как было показано в 3.6.1, работает в контексте SoftIrq) существовало обращение к ARP-таблице с использованием Neighbour API. Это вызывало достаточно редкие гонки с другим фрагментом кода, также работающим с ARP-таблицей и имевшим различные виды блокировок. Однако этого

было недостаточно, так как ядро компилировалось без опций `Preemption` (вытесняющая многозадачность) и `SMP` и, соответственно, обычные блокировки не имели никакого эффекта. Трассировка стека показывала только проблемы в таймере, но сам таймер — `Bottom Half` в терминах Linux, и, следовательно, прервать его выполнение могли лишь "железные" прерывания, которые в этом случае не использовались. Логично, что где-то существовал фрагмент кода, не блокировавший `Bottom Halves`.

Для решения проблемы пришлось переконфигурировать семантику системных вызовов, в частности — добавить новую платформу `Linux_NonSMP` с игнорированием обычных блокировок. Проблема была обнаружена анализатором по обращению к `arp_tbl` и исправлена включением `spin_lock_bh` и `spin_unlock_bh` в код драйвера.

Пример 3.40. Существовали серьезные проблемы в проприетарном USB Host Controller Driver. Гонка была неочевидной, т.к. задевала Netlink-сообщения и некоторым образом проявлялась в модуле WiFi, и `KernelPanic` (сообщение об ошибке уровня ядра) показывал его участок кода в качестве проблемного. Так как проблемы возникали именно при использовании USB, соответствующий драйвер был изучен анализатором и были обнаружены гонки при завершении URB (USB Request Block), приводящие в последствии к Use-After-Free (использованию памяти после её высвобождения) и повреждению структур данных.

Ручной разбор случая занял несколько недель и не увенчался успехом, тогда как использование одной из первых версий анализатора позволило обнаружить проблемный участок кода в течение нескольких минут.

3.9.5 Сравнение с другими проектами

К сожалению, большинство программных продуктов, выполняющих анализ на состояния гонки, недоступны широкой общественности. Для

тестирования были выбраны Coverity Code Scan и Helgrind (Valgrind).

Для сравнения с Helgrind было выбрано несколько проприетарных программ, использующих библиотеку `pthread`. Для Coverity Code Scan были выбраны open-source проекты старых версий, такие как **openswan**, **DNSMasq** и др. В среднем, альтернативные анализаторы обнаружили на 20% больше гонок.

Valgrind не обнаружил проблем в редко достижимых фрагментах кода, что с успехом получилось у нашего проекта. Разработанная программа сгенерировала значительно больше псевдо-ложных ошибок. Они справедливы для обнаруженного пути исполнения кода, но по факту получалось, что данные пути вообще никогда не вызывались или вызывающей програмой производились дополнительные блокировки, минимизирующие потенциальный вред от гонки.

Coverity Code Scan обнаружил несколько больше сложных гонок, связанных с системными вызовами.

По результатам тестирования были внесены некоторые изменения в препроцессор, семантику системных вызовов. Алгоритм накачки цепочек стал более распараллелен и стал лучше учитывать случаи использования статических методов извне средствами операционной системы или сторонними программами.

4 Выводы

В целом, о проекте можно сделать несколько выводов.

1. В нашем проекте не хватает эвристических правил: некоторые инструкции безопасны, но, тем не менее, считаются опасными программой. Примером может служить использование метода `join` для потоков, которое не учитывалось.

2. Иногда возникали проблемы с предварительной трансляцией кода в C#. Макросы определенных видов создавали проблемы с подстановкой оных в код: не учитывалось различие типов синтаксических конструкций.

3. Некоторые семантические правила были упущены из виду.

4. Скорость анализа у сторонних программных продуктов несколько выше. Это связано с несильно эффективным алгоритмом накачки цепочек фрагментов.

5. Графический интерфейс пользователя достаточно удобен для анализа исходных кодов.

Главные выводы таковы:

1. Проект вполне может конкурировать с известными альтернативами даже сейчас.

2. Некоторая полировка программы необходима, чтобы учитывать больше случаев в исходном коде проверяемых программ.

4.1 Возможные улучшения

В процессе разработки и тестирования были выявлены некоторые проблемы, требующие внесения изменений в различные уровни ПО. Отчасти модификации были осуществлены в ходе выполнения квалификационной работы, но не были завершены из-за трудоёмкости.

По части семантики предлагается ввести новую систему тегирования системных методов. `ResourceRunner` (или отдельный `TagRunner`) добавлял бы теги к методам прямо в процессе его работы. Это позволило бы проще вводить новую семантику, что сейчас не поддерживается в полной мере.

Препроцессор следовало бы обновить и по мере возможности “отцепить” его от парсера. Создание препроцессора сложная и трудоемкая задача: его написание отняло бы много времени. В нашем проекте был принят компромиссный вариант: использование парсера уже на этапе препроцессинга, что позволило быстрее закончить данный этап и вывести продукт на рабочий уровень значительно раньше. Проблемой этого варианта является сложность внедрения синтаксических ветвей в код в нетривиальных случаях использования макросов.

Ресурсный обходчик требует консолидации ресурсов в “мега-ресурсы”. В таком случае сложный и порой неэффективный алгоритм проверки эквивалентности ресурсов можно было бы исключить.

Сейчас применяется малопроизводительный метод “накачки” цепочек фрагментов. Одно из направления улучшения — генерация цепочек фрагментов во время работы фрагментного обходчика, что позволило бы сильно увеличить производительность при чтении сложных исходных кодов.

Непосредственно алгоритм нахождения гонок работает исправно, но, вероятно, есть смысл добавить проверку более сложных путей выполнения. Это было бы возможно с учётом другого метода поиска цепочек фрагментов. Другой проблемой является группировка ошибок и их ранжирование. В настоящий момент сходные ошибки, даже находящиеся в соседных строках, не объединяются, что порождает огромный объём лишних данных, который с большой вероятностью будет пропущен программистом при разборе результатов. Ранжирование позволило бы поднять вверх по списку наиболее значимые ошибки.

В процессе использования в коммерческих структурах было отмечено, что платформа, разработанная для проекта, отлично подходит и для других типов ошибок, которые сейчас не обрабатываются. Дописать другие типы проверок, используемые в обыкновенных статических анализаторах кода, имея такой большой набор семантических данных, не представляет особого труда. Это позволило бы объявить продукт полноценным инструментом статического анализа и радикально поднять его ценность.

Важным направления развития является расширение поддержки языков программирования. К счастью, нам в этом помогает выбор парсера: даже в настоящий момент достаточно легко добавить поддержку C# и Visual Basic.NET. Несомненно, корпорацией Microsoft будут постепенно добавляться и другие языки, а поддержка существующих будет улучшаться. Таким образом почти без дополнительных вложений возможно улучшить синтаксическую часть анализатора.

5 Заключение

В ходе работы были выполнены все поставленные задачи. Был разработана программа, которая обладает следующими качествами.

1. Реализует препроцессинг кода на C собственными средствами.
2. Транслирует его в некоторый аналог C#.
3. Читает код и получает его синтаксическое дерево при помощи Microsoft Roslyn.
4. Разбирает семантику исследуемой программы.
5. Анализирует псевдонимы, проводит разбивку на фрагменты, “накачивает” цепочки и всячески расширяет известную семантическую информацию.
6. Выполняет поиск состояний гонки в разработанной семантике.
7. Имеет графический интерфейс пользователя на основе GTK#.
8. Применима в облаке Azure и способна серьезно масштабироваться.
9. Работает на всех платформах, поддерживающих .NET или Mono: Windows, Linux, теоретически OS X.

Мы не ставили цели поддерживать определенный уровень точности и полноты исследования на гонки. Тем не менее, точность такова, что позволяет проекту конкурировать с известными аналогами.

Программный комплекс прошёл тестирование на нескольких open-source проектах и показал свою применимость в коммерческой среде, успешно устранив несколько избранных проблем, встретившихся в одной из Санкт-Петербургских компаний-разработчиков программного обеспечения.

Список литературы

[1] McCabe T. J. A complexity measure // Proceedings of the 2Nd International Conference on Software Engineering. — ICSE ’76. — Los Alamitos, CA, USA : IEEE Computer Society Press, 1976. — P. 407.

[2] Bessey A., Block K., Chelf B. et al. A few billion lines of code later: Using static analysis to find bugs in the real world // Commun. ACM. — Vol. 53, no. 2.

[3] Netzer R. H. B., Miller B. P. What are race conditions? some issues and formalizations // ACM Lett. Program. Lang. Syst. — Vol. 1, no. 1.

[4] Multi-threaded programming: efficiency of locking. — URL: `https://attractivechaos.wordpress.com/2011/10/06/multi-threaded-programming-efficiency-of-locking` (дата обращения: 05.05.2016).

[5] Leveson N. G., Turner C. S. An investigation of the therac-25 accidents // Computer. — Vol. 26, no. 7.

[6] GitHut - Programming Languages and GitHub. — URL: `http://githut.info` (дата обращения: 05.05.2016).

[7] Durumeric Z., Kasten J., Adrian D. et al. The matter of heartbleed // Proceedings of the 2014 Conference on Internet Measurement Conference. — IMC '14. — 2014.

[8] Ramalingam G. Context-sensitive synchronization-sensitive analysis is undecidable // ACM Trans. Program. Lang. Syst. — Vol. 22, no. 2.

[9] Hoare C. A. R. An axiomatic basis for computer programming // Commun. ACM. — Oct. 1969. — Oct. — Vol. 12, no. 10. — P. 576–580.

[10] Manna Z., Pnueli A. Axiomatic approach to total correctness of programs // j-ACTA-INFO. — Vol. 3, no. 3. — P. 243–263.

[11] Pellarini D., Lenisa M. Hoare logic for multiprocessing // Proceedings of the 13th Italian Conference on Theoretical Computer Science. — 2012.

[12] Stirling C. A generalization of owicki-gries's hoare logic for a concurrent while language // Theoretical Computer Science. — 1988. — Vol. 58, no. 1. — P. 347 – 359.

[13] Lamport L. Time, clocks, and the ordering of events in a distributed system // Commun. ACM. — Vol. 21, no. 7.

[14] Savage S., Burrows M., Nelson G. et al. Eraser: A dynamic data race detector for multithreaded programs // ACM Trans. Comput. Syst. — Vol. 15, no. 4.

[15] Andersen L. O. Program Analysis and Specialization for the C Programming Language : Ph. D. thesis / L. O. Andersen ; DIKU, University of Copenhagen. — 1994.

[16] Steensgaard B. Points-to analysis in almost linear time // Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — POPL '96. — 1996.

[17] Engler D., Ashcraft K. RacerX: Effective, Static Detection of Race Conditions and Deadlocks // SIGOPS Oper. Syst. Rev. — Vol. 37, no. 5.

[18] Nethercote N., Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation // SIGPLAN Not. — Vol. 42, no. 6.

[19] Tanenbaum A. S., Bos H. Modern Operating Systems. — 2014.

[20] Bovet D., Cesati M. Understanding The Linux Kernel. — 2005.

[21] Corbet J., Rubini A., Kroah-Hartman G. Linux Device Drivers, 3rd Edition. — 2005.

[22] Russinovich M., Solomon D. A., Ionescu A. Windows Internals, part 1. — Redmond, Wash : Microsoft Press, 2012.

[23] Russinovich M., Solomon D. A., Ionescu A. Windows Internals, part 2. — Redmond, Wash. Farnham : Microsoft O'Reilly distributor, 2012.

[24] Marguerie F., Eichert S., Wooley J. Linq in Action. — 2008.

[25] Method Parallel.For. — URL: https://msdn.microsoft.com/library/system.threading.tasks.parallel.for.aspx (дата обращения: 05.05.2016).

[26] Chacon S. Pro Git. — 2009.

[27] Pilato M. Version Control With Subversion. — 2004.

[28] NCalc - Mathematical Expressions Evaluator for .NET. — URL: https://ncalc.codeplex.com (дата обращения: 05.05.2016).

[29] Getting Started: C# Syntax Transformation. — URL: https://github.com/dotnet/roslyn/wiki/Getting-Started-C%23-Syntax-Transformation (дата обращения: 05.05.2016).

[30] Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. Introduction to Algorithms, Third Edition. — 2009.

[31] C# Programming: Delegates and Events. — URL: https://en.wikibooks.org/wiki/C_Sharp_Programming/Delegates_and_Events (дата обращения: 05.05.2016).

[32] Horwitz S. Precise flow-insensitive may-alias analysis is np-hard // ACM Trans. Program. Lang. Syst. — Vol. 19, no. 1.

[33] Krishnan S. Programming Windows Azure. — O'Reilly Media, 2010.

[34] Esposito D. Programming Microsoft ASP.NET. — 2003.

[35] Pricing - Virtual Machines (VMs) // Microsoft Azure. — URL: https://azure.microsoft.com/pricing/details/virtual-machines (дата обращения: 05.05.2016).

Приложение

1 Stringification Operator

В примере `fullstr` — вся строка макроса, `name` — название параметра макроса, `replacement` — внедряемое выражение в текстовой форме.

```
public static string
Replace(string fullstr, string name, string replacement)
{
    string s = fullstr;
    while (true)
    {
        // Ищем место использования параметра
        int ind = s.IndexOf("#" + name);
        if (ind == -1)
            break;
        // Выполняем замену
        s = s.Remove(ind, 1 + name.Length);
        s = s.Insert(ind, "\"" + replacement + "\"");

        // Удалим лишние пробелы после строки
        int right = ind + replacement.Length + 2;
        while (right < s.Length &&
               String.IsNullOrWhiteSpace(s[right].ToString()))
            right++;
        s = s.Remove(ind + replacement.Length + 1,
                     right - ind - replacement.Length - 2);

        // Удалим лишние пробелы до строки
        int left = ind - 1;
        while (left >= 0 &&
               String.IsNullOrWhiteSpace(s[left].ToString()))
            left--;

        if (left >= 0 && left != (ind - 1))
            s = s.Remove(left + 1, ind - 1 - left);
    }
    return s;
}
```

2 Вставка пустых строк

В данном фрагменте кода рекурсивность используется исходя из допущения, что любое синтаксическое дерево конечно.

```
public static
SyntaxNode FindWhereInsert(IPlatformSemantics semantics,
                           SyntaxNode root)
{
    foreach (var itm in root.ChildNodes())
    {
        // Получаем примитив, вызываемый в данном фрагменте
        // кода
        var lp = Lockset.GetPrimitive(semantics, itm);
        if (lp != null &&
            lp.OperationType.HasFlag(OperationType.Unlock))
        {
            // Если следующей ветви кода не существует
            // возвращаем найденную.
            var nextNode = itm.NextNode();
            if (nextNode == null)
                return itm;

            // Если следующая ветвь - уже пустое выражение
            // то пропускаем ветвь (задача уже выполнена)
            if (nextNode is EmptyStatementSyntax)
                continue;

            // Если следующая ветвь не вызывает блокировку,
            // то пропускаем ветвь (пустое выражение не требуется)
            var nlp = Lockset.GetPrimitive(semantics, nextNode);
            if (nlp == null)
                continue;

            return itm;
        }

        // Рекурсивно ищем нужную ветвь среди потомков
        // текущей.
        var r = FindWhereInsert(semantics, itm);
        if (r != null)
            return r;
    }
```

```
    return null;
}

public static SyntaxTree
FixSyntax(IPlatformSemantics semantics,
          SyntaxTree tree)
{
    var root = tree.GetRoot();
    while (true)
    {
        var node = FindWhereInsert(semantics, root);
        if (node == null)
            break;

        // Вставляем пустое выражение сразу за
        // найденным.
        root = root.InsertNodesAfter(node,
            new List<SyntaxNode>() {
                SyntaxFactory.EmptyStatement()
            .WithLeadingTrivia(node.GetLeadingTrivia())
            .WithTrailingTrivia(new List<SyntaxTrivia>() {
                SyntaxFactory.SyntaxTrivia(
                    SyntaxKind.EndOfLineTrivia,
                    "\n")
            })
            }
        );
    }
    return SyntaxFactory.SyntaxTree(root);
}
```

3 SyntaxRunner

Базовая реализация синтаксического обходчика. Для краткости некоторые незначительные моменты заменены на многоточия.

```
public void Go(SyntaxNode node)
{
    var childNodes = node.ChildNodes();
    BlockTraverseEvent?.Invoke(...);
    if (node.Parent is LabeledStatementSyntax)
```

```
{
    GotoLabelEvent?.Invoke(...);
}
if (node is MethodDeclarationSyntax)
{
    var method = node as MethodDeclarationSyntax;
    foreach (var param in
      method.ParameterList.Parameters)
    {
        VariableDeclarationEvent?.Invoke(...);
        if (param.Default != null)
            ExpressionEvent?.Invoke(...);
    }
}

foreach (var childNode in childNodes)
{
    if (childNode is GotoStatementSyntax)
    {
        GotoEvent?.Invoke(...);
    }
    if (childNode is ForStatementSyntax)
    {
        ForEvent?.Invoke(...);
    }
    if (childNode is IfStatementSyntax)
    {
        var ifs = childNode as IfStatementSyntax;
        ConditionCheckEvent?.Invoke(...);
        Go(ifs.Statement);

        if (ifs.Else != null)
            Go(ifs.Else.Statement);
        continue;
    }
    if (childNode is SwitchStatementSyntax)
    {
        var sss = childNode as SwitchStatementSyntax;
        ConditionCheckEvent?.Invoke(...);
        foreach (var section in sss.Sections)
        {
            Go(section.Statements.FirstOrDefault());
        }
```

```
    continue;
}
if (childNode is InvocationExpressionSyntax &&
 InvocationEvent != null)
{
    InvocationEvent.Invoke(...);
}
if (childNode is MethodDeclarationSyntax &&
 MethodDeclarationEvent != null)
{
    MethodDeclarationEvent.Invoke(...);
}
if (childNode is ReturnStatementSyntax &&
 ReturnEvent != null)
{
    ReturnEvent.Invoke(...);
}
if (childNode is ExpressionStatementSyntax &&
 ExpressionEvent != null)
{
    var expr = childNode as
     ExpressionStatementSyntax;
    if (expr.Expression is
     AssignmentExpressionSyntax)
    {
        var asexp = expr.Expression as
         AssignmentExpressionSyntax;
        ExpressionEvent.Invoke(...);
    }
    else
    {
        ExpressionEvent.Invoke(...);
    }
}
if (VariableDeclarationEvent != null &&
   (childNode is FieldDeclarationSyntax ||
    childNode is LocalDeclarationStatementSyntax))
{
    VariableDeclarationSyntax decl;
    if (childNode is FieldDeclarationSyntax)
        decl = (childNode as
         FieldDeclarationSyntax)
        .Declaration;
```

```
            else
                decl = (childNode as
                 LocalDeclarationStatementSyntax)
                .Declaration;

            foreach (var decl1 in decl.Variables)
            {
                VariableDeclarationEvent.Invoke(...);
                if (decl1.Initializer != null)
                    ExpressionEvent?.Invoke(...);
            }
        }
        Go(childNode);
    }
    BlockTraverseEvent?.Invoke(...);
}
```

4 Пересечение Lockset

Ниже представлен фрагмент кода, выполняющий операцию `Intersect` для наборов блокировок.

```
// Поиск общих объектов
// Метод статический, так как работает с
// набором списков блокировок.
public static List<LockReference>
    Intersect(List<List<LockReference>> lst)
{
var res = new List<LockReference>();
    if (lst.Count > 0)
{
        foreach (var l in lst.First())
        {
            if (lst.Where((x) =>
             x.Contains(l)).Count() == lst.Count)
                res.Add(l);
        }
    }
    return res;
}
```

```
// Поиск пересечений двух Lockset (this и ls),
// с учётом условий Бернстайна, а также правил
// для семафоров.
public List<LockReference> Intersect(Lockset ls)
{
    var is1 = LocksHeld.Intersect(ls.LocksHeld).ToList();
    for (int i = is1.Count - 1; i >= 0; i--)
    {
        var lr = is1[i];
        var obj1 = this.LocksHeld.Find((x) => x.Equals(lr));
        var obj2 = ls.LocksHeld.Find((x) => x.Equals(lr));
        if (obj1.Lock.OperationType
         .HasFlag(OperationType.ReadLock) &&
            obj2.Lock.OperationType
             .HasFlag(OperationType.ReadLock))
        {
            if (obj1.Lock.OperationType
             .HasFlag(OperationType.Semaphore) &&
                obj2.Lock.OperationType
                 .HasFlag(OperationType.Semaphore))
            {
                if (obj1.Resource
                 .SemaphoreCount.Count() == 0 ||
                    obj1.Resource
                     .SemaphoreCount.Select((x) =>
                     x.Val).Max() >= 2)
                    is1.RemoveAt(i);
            }
            else
            {
                is1.RemoveAt(i);
            }
        }
    }
    return is1;
}
```

5 Внешний вид пользовательского интерфейса

Приведен пример (рис. 6) внешнего вида пользовательского интерфейса в среде Microsoft Windows. В Linux приложение выглядит аналогично.

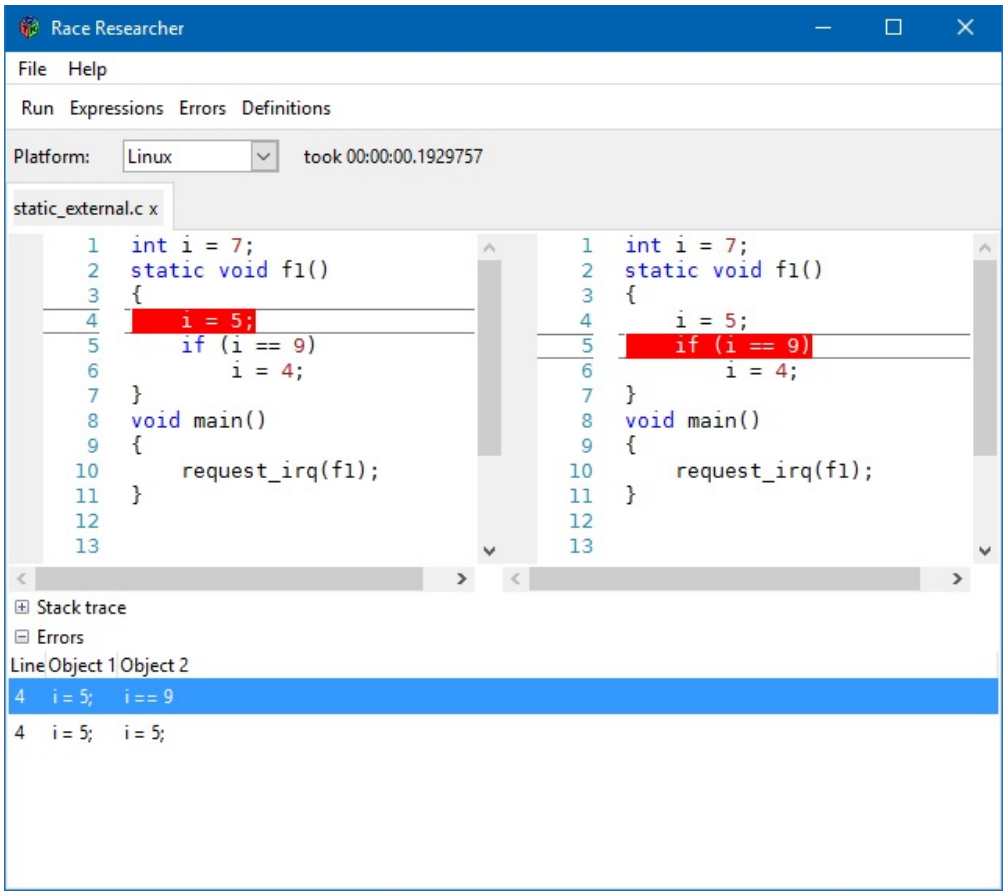

Рис. 6: Интерфейс в среде Microsoft Windows

6 Отправка задачи в WorkerRole

Здесь приведен упрощенный код отправки задачи в облачное приложение без проверки ошибок.

```
// Определение конечной точки WorkerRole
var protocol = "http";
var ipAddress = RoleEnvironment.Roles["RaceCloudWorker"]
.Instances.First()
    .InstanceEndpoints.ToArray()
    .Where(ep => ep.Value.Protocol == protocol)
    .First()
```

```
    .Value.IPEndpoint.ToString();
var stringEndpoint = string.Format("{0}://{1}", protocol,
ipAddress.ToString());

// Открытие SignalR-прокси для хаба RaceHub
HubConnection connection = new HubConnection(stringEndpoint);
IHubProxy proxy = connection.CreateHubProxy("RaceHub");
connection.Start().Wait();

// Выполнение метода Analyze из хаба RaceHub
var task = proxy.Invoke<string>("Analyze", Model.Code);
task.Wait();

// Сохранение и вывод результатов.
Session["LastResult"] = task.Result;
Response.Redirect("~/Home/Results");
```