Санкт-Петербургский государственный университет

Меньшиков Максим Александрович

Выпускная квалификационная работа

Гибридный языково-независимый статический анализ как элемент комплексной технологии программирования

Уровень образования:
Направление 09.06.01 «Информатика и вычислительная техника»
Основная образовательная программа МК.3019 «Информатика»

Научный руководитель:
Зав.кафедрой системного программирования СПбГУ,
д.ф.-м.н., профессор
Терехов Андрей Николаевич

Рецензент:
доцент НИУ ВШЭ
к.ф.-м.н
Подкопаев Антон Викторович

Санкт-Петербург
2022

Saint Petersburg State University

Maksim Aleksandrovich Menshikov

Graduation thesis

Hybrid language-independent static analysis as an element of complex programming technology

Education level:
09.06.01 “Informatics and Computer Engineering”
MK.3019 “Informatics”

Scientific supervisor:
Head of system programming chair (SPbU),
Doctor of Physics and Mathematics, professor
Andrey Nikolaevich Terekhov

Reviewer:
Associate professor (HSE University),
PhD,
Anton Viktorovich Podkopaev

Saint Petersburg
2022

Оглавление

Стр.

Стр.

Стр.

Стр.

Введение

Стремительное развитие микроэлектроники в 20 веке привело к распространению кремниевых микросхем в различных сферах человеческой деятельности. Долгое время увеличение производительности процессоров происходило через увеличение числа транзисторов по закону Мура [1], но индустрия постепенно стала подходить к физическим пределам плотности их размещения, что потребовало новых подходов к производству транзисторов [2]. Индустрия ответила концепцией *параллельных вычислений*, появились *спекулятивное исполнение*, *внеочередное исполнение* и другие технологии, повышающие производительность. Появились специализированные микросхемы, которые добились большей производительности благодаря специализации вычислений. Первыми такими микросхемами стали видеоускорители. В дальнейшем стали развиваться программируемые микросхемы (FPGA) [3; 4], сетевые процессоры, нейросетевые процессоры, акселераторы глубокого обучения [5], сопроцессоры.

Такие методы решения проблемы производительности привели к значительному увеличению сложности разработки программного обеспечения. Программное обеспечение теперь одновременно разрабатывается для компьютеров, внешних устройств, сопроцессоров, FPGA, требует обучения сопутствующих нейронных сетей. В начале века на рынке процессоров господствовала архитектура x86, и разработчики предпочитали писать программы только под нее. Остальные архитектуры не были массовыми. В настоящее время самой популярной архитектурой можно считать ARM [6] в связи с выпуском большого количества устройств на базе Android [7] (на базе Linux), для стационарных компьютеров — x86_64 (amd64) [8], для маршрутизаторов — MIPS. Переход компании Apple на архитектуру Apple Silicon [9] — процессоров M1 на базе ARM — существенно изменил отношение рынка к процессорам ARM, и теперь сложно представить современную программу для массового потребителя без поддержки данной архитектуры.

Эта тенденция прямо влияет на анализ программ с целью выявления ошибок и несоответствия спецификации. Ранние программы для ЭВМ можно было формально доказать с первой до последней строки в связи с их небольшим размером, но современные программы состоят из множества

модулей, в том числе внешних, написанных на разных языках, для множества операционных систем и для различающегося аппаратного обеспечения. В качестве примера можно привести современные системы быстрого оптического распознавания: низкоуровневое программное обеспечение для камеры и другой периферии написано на C или ассемблере (архитектура может быть различной), прошивка микроконтроллера, использующего камеру, написана на C/C++ (чаще всего для архитектуры ARM), передача потока осуществляется на сервер на базе архитектуры AMD64, нейронная сеть тренируется на нейросетевом процессоре, используются FPGA для увеличения производительности, вывод осуществляется при помощи графического ускорителя. Проверить алгоритмы таких программ очень сложно.

Разнообразие конфигураций порождает проблемы совместимости и масштабируемости. Возникает вопрос: *как производить качественный статический анализ программ на разных языках, для разных процессоров, операционных систем?*

После значительного числа инцидентов [10; 11], связанных с ошибками в программах, исследователи разработали методы защиты от ошибок. Их можно разбить на несколько групп:

– *Установка требований к программному коду, ограничение используемых конструкций*. Организации могут запретить использовать арифметику указателей в C [12], являющуюся значительным источником ошибок. Однако данный метод является организационным, т.к. он не предотвращает возможность допустить ошибку, а скорее делает её затруднительной.
– *Защитные языки программирования*. Их идея сводится к снижению возможности допустить ошибку — в основном, за счет удачного синтаксиса и проверок в компиляторе [13]. Интересным способом достижения корректности является использование сертифицированных компиляторов, таких как CompCert [14].
– *Анализ программ* — это проверка программного кода. Он может осуществляться как *статически*, т.е. без запуска программы и с учетом всех возможных путей исполнения, так и *динамически*, т.е. с запуском программы и с учетом только действующих путей исполнения. Статические проверки кода могут проводиться через *проверку модели*, т.е. сравнение реальной программы с эталоном, заданным

либо полностью формальной моделью, либо через аннотации. Другой вариант — через *символьное исполнение* программы, которое не исключает возможность последующей проверки модели.

– *Методы предотвращения ошибок на уровне среды исполнения*. Один из наиболее очевидных способов — запрет обращения к определенной памяти через проверку допустимости действия во время исполнения, явный запрет разыменования нулевых указателей и т.п.

Комбинирование указанных методов является эффективным способом достижения безопасности программ [15], что подтверждается их использованием в авионике [16] и других сферах, где качество кода критически важно.

Несмотря на разнообразие подходов, в настоящий момент в сфере разработки безопасного программного обеспечения наблюдается некоторая стагнация. Крупные организации все чаще предлагают разработчикам переходить на новые «безопасные» языки программирования, такие как Rust [17], Go [18], а также интерпретируемые языки с JIT [19]: C# [20], Kotlin [21]. Обсуждается даже добавление модулей на языке Rust в Linux [22]. Несмотря на все достоинства безопасных языков, перенос кодовой базы на них является крайне амбициозной задачей. Вероятен сценарий, когда новые ошибки будут добавлены просто вследствие переписывания достаточно хорошо протестированной (но ни в коем случае не безошибочной) кодовой базы на новый язык.

Другая проблема заключается в том, что расширение количества поддерживаемых программным проектом языковых семантик неизбежно влечет необходимость контроля за всеми реализаций как по отдельности, так и в совокупности, что не может не снизить качество программы. При этом статический анализ зачастую достаточно прогрессивен технически, чтобы поддерживать разные языки, но межъязыковое взаимодействие остается нерешенной проблемой. Многоязыковые анализаторы обычно имеют возможность переходить границы языков в рамках одной технологии (JVM [23], .NET [24], биткоды LLVM [25]), но не далее.

Немаловажным является и опыт разработчиков. Переход с одного языка на другой оправдан, если это в небольшой степени влияет на накопленный программистами опыт, не «обнуляет» его. Например, переход с Java на Kotlin не влияет на опыт, если программа пишется в стиле Java, но Kotlin добавляет функциональные вставки, корутины и другие возможности, использование

которых рекомендуется разработчиками языка. При этом «бесшовность» перехода в таком случае является весьма относительной с точки зрения опыта программистов.

Таким образом, переход на безопасные языки программирования является устойчивой тенденцией, при этом процесс происходит медленно: разработчикам и их инструментам необходимо поддерживать взаимодействующие кодовые базы на разных языках.

К настоящему моменту статические анализаторы кода сформировались в отдельные программные продукты, которые обеспечивают две основные сервисные модели: *локальную проверку кода* и *работу на сервере в рамках Continuous Integration*. Фактическое отсутствие других моделей существенно ограничивает взаимодействие анализаторов с внешними программами. Если сделать анализатор менее зависимым от модели использования, то из анализа можно извлечь больше полезных сведений: дополнительные статические инварианты, данные для генерации юнит-тестов, сведения для поддержки межъязыковых контрактов, пути исполнения программ и т.п.

В работе [26] автор рассматривает возможность применения статических анализаторов в рамках нетрадиционных сервисных моделей. Это требует корректировки принципа работы самого анализатора.

Подобные модели требуются для комплексной среды программирования [27; 28], разрабатываемой на кафедре системного программирования Санкт-Петербургского государственного университета под руководством заведующего кафедрой, профессора Андрея Николаевича Терехова. Проект ставит перед собой амбициозную задачу разработки комплексной технологии программирования, подразумевающей полный цикл: языки программирования, компиляторы [29], статические и динамические анализаторы, методы оптимизации программ, виртуальные машины для исполнения [30], кодизайн аппаратного обеспечения [31], тестирование. Важное свойство проекта — взаимодействие с другими языками программирования.

Перед статическими анализаторами в комплексных средах стоит множество вызовов. Во-первых, велика зависимость инструментов анализа от систем чтения кода и их внутренних структур данных. Во-вторых, у анализаторов зачастую отсутствует точная информация о целевой платформе. В-третьих, анализаторы часто не приспособлены к анализу на стадии проектирования, не подразумевают интерактивности и неклассических моделей

использования. В-четвертых, реализация независимых инструментов в таких условиях неминуемо становится сложной.

Таким образом, комплексные среды разработки предполагают, что анализаторы имеют **гибридность** (поддержку различных методов анализа от *контроля синтаксических деревьев* до *проверки модели*, *абстрактной интерпретации*), **языковую независимость** (работоспособность на программах, написанных на разных языках программирования, возможность их анализа в одном контексте), **поддержку различных программно-аппаратных платформ** с адаптацией (маскировкой) под наборы компиляции для уточнения данных о целевой платформе, **простоту интеграции во весь цикл разработки** от проектирования до тестирования и итогового приема, **широкую функциональность** (*обнаружение дефектов*, *проверки инвариантов*, *определение языка проекта*, *описание существующих в коде переменных*, *функций и их назначения*, *навигация по коду*, *выявление параметров сложности кода*, *определение зависимостей*, *помощь в сертификации*). Анализаторам необходимо не только соответствовать всем этим требованиям, но и обеспечивать идентичный опыт разработки на различных технологиях. По отдельности эти задачи решаются, но комплексный подход к созданию данной «экосистемы» по существу не рассматривался и является актуальной задачей программной инженерии.

Для создания подходящего под эти требования статического анализатора необходимо техническое решение, обладающее широким спектром возможностей. **Equid** [32] разрабатывается автором с 2015 г. Проект развился из обычного анализатора шаблонов в анализатор путей исполнения для поиска состояний гонки [33], а затем был развит до полнофункционального анализатора [34]. Проект был успешно применен в различных коммерческих проектах и доказал свою эффективность в решении промышленных задач.

Целью данной работы является разработка методологии проведения статического анализа для комплексной технологии программирования на базе языков C, C++, РуСи с учетом требований о возможности взаимодействия различных частей программного комплекса, полноценной поддержке различных языков программирования и их применения в одном контексте.

Для достижения поставленной цели необходимо решить следующие **задачи**:

1. Разработать метод трансляции абстрактного синтаксического дерева в обобщенное синтаксическое дерево на основе унифицированного языкового базиса (Гл. 3).
2. Подготовить алгоритм трансляции обобщенных синтаксических деревьев в промежуточное представление, обеспечивающее анализ программ на разных языках программирования. (Гл. 4).
3. Исследовать методы композиции семантик различных языков программирования и предложить вариант, позволяющий объединить в одном анализе программы на C, C++, РуСи и других потенциальных языках (п. 4.5).
4. Организовать взаимодействие между методами проверки модели, абстрактной интерпретации и символьного исполнения для проведения качественного и быстрого анализа в условиях многоязыковой среды (Гл. 5)
5. Исследовать архитектуры и способы обеспечения взаимодействия статического анализатора с другими частями комплекса, реализовать библиотеки для обеспечения их бесшовного взаимодействия (Гл. 6).
6. Подготовить архитектурное решение для программного продукта, поддерживающее разработанные методы во всей полноте (Гл. 2).
7. Реализовать разработанные методы и интегрировать их в систему статического анализа.

Научная новизна. Новизной обладает методология обеспечения анализа «в целом», состоящая из следующих частей:

1. Методы трансляции программ в обобщенные синтаксические деревья (Гл. 3) и промежуточное представление (п. 4.3).
2. Язык промежуточного представления Midair, а также соответствующий фреймворк для анализа, состоящий из виртуальных машин и гипервизоров (Гл. 4).
3. Эффективные алгоритмы *гибридного решателя*, подходящие для межъязыкового анализа (Гл. 5).
4. Архитектура и методы обеспечения взаимодействия с компонентами комплексной технологии программирования (Гл. 6).

Теоретическая и практическая значимость. Теоретический результат работы — разработанная методология проведения статического анализа, складывающаяся из сочетания алгоритмов, архитектурных и технических решений, а также опыта решений проблем, и позволяющая проводить анализ программ из разных языковых контекстов.

Практическая значимость определяется фактом разработки средства анализа кода для языков C, C++, РуСи, используемого в промышленных компаниях.

Методология и методы исследования. В работе используются базовые методы информатики и программной инженерии. Статический анализ основывается на методах проверки модели, символьного исполнения и абстрактной интерпретации. Разработанное языково-независимое представление строится на теории компиляторов. Активно используются методы теории для задачи выполнимости формул в теориях (SMT).

Личный вклад. Автор самостоятельно разработал статический анализатор и основные библиотеки для взаимодействия с ним, подготовил и описал методологический базис. Автор консультировал разработчиков дополнения поддержки статического анализа для интегрированного средства разработки комплексной технологии программирования, лично внедрил поддержку анализа в компилятор РуСи, а также вносил правки в другие элементы комплексной технологии программирования.

Глава 1. Обзор литературы

В сфере статического анализа программ существует множество различных методов и технологий, которые будут рассмотрены в данном разделе.

1.1 Анализаторы кода

Рассмотрим анализаторы кода *общего назначения*.

Svace [35; 36] — универсальный анализатор кода, разработанный Институтом Системного программирования РАН. По заявлениям авторов, он является коллекцией анализаторов разных уровней сложности, что позволяет искать дефекты на различных уровнях: абстрактное синтаксическое дерево, промежуточное представление и т.п. Проект поддержан компанией Samsung и применяется в составе Tizen Studio, что предполагает наличие базовых интеграций. **Aegis** [37] является средством обнаружения дефектов в программах на языках C/C++, разработанном в лаборатории Digitek Labs. В нем производится поиск утечек памяти, выходов за границы массивов и т.п., в том числе, и в режиме выделенного сервера. **Borealis** [38; 39] использует метод ограниченной проверки модели для анализа программ на языках C/C++. Он основывается на LLVM/Clang и позволяет проверять соответствие функций аннотациям, заданным на собственном языке описания функций. **PVS Studio** [40] — широко распространенный статический анализатор для C, C++, C#, Java, отличающийся хорошим качеством анализа [41] и большим количеством диагностик. Авторы программы регулярно проводят проверки открытых проектов, демонстрируя высокую степень практической применимости этого инструмента. **AppChecker** [42] может использоваться для поиска уязвимостей в ходе сертификации. Хорошие результаты были достигнуты исследователями при модернизации **BLAST** [43].

Coverity [44] является одним из лидирующих средств анализа, успешно применяемым для проверки крупных промышленных проектов. Количество обнаруживаемых классов дефектов очень велико. Другим известным и распространенным анализатором кода является **cppcheck** [45],

декларируемая цель которого — детектирование действительно существующих ошибок в коде. Анализатором поддерживаются правила MISRA [12]. Также существует проект **CppCheck Premium** [46], по заявлениям авторов выявляющий ещё большее количество ошибок. **SonarQube** [47] — платформа для проведения анализа и непрерывной интеграции. Она обладает дополнениями для различных языков программирования (в том числе C/C++) и анализаторов, позволяет настроить уровни толерантности (*Quality Gate*) для различных дефектов. **semgrep** [48] загружает заданные пользователем правила и обнаруживает соответствующие им дефекты. Он не включает работу с аналитическими представлениями (хотя и позволяет проводить распространение констант), что существенно ограничивает глубину анализа. Аналогично работает **CodeQL** [49] от GitHub, **KICS**[1] от Checkmarx. Среда разработки **CLion** [50] включает инструменты анализа на базе Resharper [51] и *clangd* [52]. По внешним признакам, используемые методы анализа достаточно эффективны, но оценить их в полной мере невозможно из-за закрытости инструмента [53]. Проект **Phasar** [54] принимает на вход промежуточное представление LLVM и проводит на его основе анализ потока данных. Проект чрезвычайно зависим от LLVM, что ограничивает область его применения.

Вышеприведенные проекты успешно применяются для анализа программ. Однако большая часть проектов либо основывается на одной технологии (например, LLVM), либо производит поверхностный анализ кода, но для множества языков. Также возникают сложности с точностью адаптации под целевой компилятор, когда верифицируемый и компилируемый коды не совпадают семантически. Более систематический подход применяется в компиляторах, где анализ явно применяется прямо в процессе компиляции. К таким проектам относятся Clang и Roslyn.

Clang [25], фреймворк для компиляции программ на языке C/C++, поддерживает анализ программ. Он обладает высокой достоверностью результатов [41] и качественным описанием диагностируемого результата с выводом в веб-интерфейс и другие форматы. Его парсер C/C++ используется как элемент описываемой работы. **Roslyn** [55; 56] — интегрированное средство компиляции и анализа программ на различных языках .NET. Это одна из комплексных технологий, где анализ занимает важное место. Несмотря на то,

[1] https://github.com/checkmarx/kics

что фреймворк ещё развивается, он активно применяется в промышленности и является главным текущим компилятором для языков C#, VB.NET, F# и т.п.

Несколько глубже код анализируется в проектах, основывающихся на академических методах.

Astree [57] — научный проект исследователей абстрактной интерпретации [58], нашедший свое применение в аэронавтике и других сферах промышленности [59]. Он обладает высокой точностью и выявляет большое количество дефектов. **Eldarica** [60] предлагает использовать решатели дизъюнктов Хорна [61], что позволяет добиться впечатляющих результатов при анализе циклов и рекурсии [62]. Множество методов анализа реализовано в **Frama-C** [63; 64]: дедуктивная верификация, абстрактная интерпретация и т.п. Аналогично дизъюнкты Хорна применяются в **SeaHorn** [65], автоматизированном фреймворке для анализа кода LLVM. Инструмент **Infer** [66] задействует методы логики разделения и биабдукции для улучшения качества анализа. Согласно многочисленным исследованиям, в том числе по результатам соревнования SV-COMP [67], это позволяет добиться хорошего качества распознавания наиболее общих дефектов. При этом инструмент не включает необычных методов интеграции и не предполагает использования проекта из внешних приложений. Анализатор обладает внутренним языком Infer.AI [68], позволяющим пользователям создавать новые детекторы ошибок.

Среди статических анализаторов можно обнаружить явное разделение на промышленные и академические проекты. Промышленные проекты часто поддерживают множество языков, но не производят глубокий анализ. Академические проекты с высокой точностью находят ошибки или верифицирует их отсутствие, но они менее доступны разработчикам, имеют большой порог входа [69]. Проблема повышения применимости статических анализаторов обсуждалась и в правительственных кругах, что выразилось в инициативе Static Analysis Tools Modernization Project (STAMP) [70], одной из целей которого является унификация подходов к интеграции проектов (SARIF [71], SASP [72]). s

1.2 Представление структуры программного кода

Анализаторы кода применяются к исходному тексту, но его информационное наполнение не позволяет выявлять ошибки, а целевое объектное представление может значительно отличаться от исходного текста по семантике. Для анализа используется некоторое аналитическое представление, от выбора которого зависят как возможность обнаруживать ошибки определенных типов, так и производительность решения.

Абстрактные синтаксические деревья и *ориентированные ациклические графы* [73; 74] — наиболее используемые типы конструкций. Такие представления удобны с точки зрения обнаружения синтаксических проблем. На их основе работают целые классы программ по форматированию кода [75], по выявлению нарушения стиля, проверке использования отдельных конструкций [76] и т.п.

Графы потоков управления, *графы потоков данных* [73; 74] используются не менее часто. Они не всегда сохраняют синтаксическую структуру (хотя технически возможно это реализовать), но сохраняют семантическую информацию о состояниях программы, о переходах состояний, осуществляемых при помощи ветвлений, циклов, вызовов и других конструкций.

Промежуточные представления наиболее удобны для проверки логики. Линейные представления близки по своей структуре к низкоуровневым кодам, с другой стороны — они сохраняют семантические данные. Это позволяет осуществлять эффективный анализ и дальнейшую трансляцию в двоичные коды.

SIMPLE [77] — один из первых вариантов структурированного промежуточного представления, используемого в компиляторе McCAT, включает также представления FIRST и LAST. **GENERIC**, **GIMPLE** [78] и **RTL** (Registry Transfer Language) [79] используются на различных уровнях в GCC. GENERIC представляет функции как деревья. GIMPLE — несколько упрощенное по сравнению с GENERIC представление, заимствующее многое из SIMPLE. В итоге компилятор применяет RTL, который является уже низкоуровневым представлением, зависящим от машинных типов. **SUIF** [80] — промежуточное определение, используемое в первую очередь для оптимизации. В нем низкоуровневые линейные блоки прерываются

высокоуровневыми конструкциями, такими как циклы. Проект **Soot** [81] представляет фреймворк для анализа байт-кода Java. Он состоит из четырех представлений: Baf (простое представление байт-кода), Jumple (3-адресное представление байт-кода), Shimple (форма Single Static Assignment для Jimple), Grimp (неструктурированное представление Java). Также ведутся работы по анализу и других языков в Soot [82], но без сохранения семантики. **Byte Code Engineering Library (BCEL)** [83] — библиотека для манипуляции байт-кодом Java. Она задействуется во множестве проектов, таких как FindBugs [84], AspectJ [85], и т.п.

LLVM IR [86] — язык, используемый в LLVM, максимально повторяющий платформонезависимый ассемблерный код. Он применяется для компиляции, оптимизации, анализа и обфускации. Представление включает такие свойства, как поддержка формы единственного присваивания, машинных типов, метаданных. Оно оказало большое влияние на индустрию за счет привнесения модульности, открытости в этот процесс. Однако LLVM IR имеет некоторые недочеты, которые Латтнер и другие разработчики предложили исправить в отдельном представлении **Multi-Level Intermediate Representation (MLIR)** [87], основывающемся на концепции *диалектов*. Его главное преимущество — возможность более высококачественной оптимизации благодаря расширенному анализу синтаксических структур, неизбежно теряющихся при конверсии в LLVM IR. Проект активно используется в TensorFlow [88] (диалект **tf**).

C intermediate language (CIL) [89] создавался специально для анализа программ. Его отличительной чертой можно назвать модуль чтения, поддерживающий большинство конструкций языка C. В настоящее время почти не развивается, но модифицированная версия лежит в основе Frama-C [63]. **SAIL** [90] — представление для статического анализа предложенное исследователями I. Dillig, T. Dillig и A. Aiken. Оно включает высокоуровневое промежуточное представление, формально близкое к абстрактным синтаксическим деревьям, и низкоуровневое представление, похожее на граф потока управления. Язык **Boogie** [91] был разработан Microsoft Research для нового поколения анализаторов кода. Он основан на языках Racket, Rosette [92]. Поддерживает значительное число теорий и типов данных, позволяя проводить полноценную верификацию. **GraalVM** [93] — виртуальная машина для исполнения приложений на различных основанных на JVM, LLVM (че-

рез проект Sulong [94]) языках программирования (“polyglot applications”, как это называют авторы). Он включает фреймворк для реализации языков Truffle [95], который позволяет создать интерпретаторы, основанные на абстрактных синтаксических деревьях. Также существует представление **REIL** [96] для статического анализа дизассемблированного кода.

Наиболее популярными представлениями являются языки Java [97] и MSIL [98] (для .NET-языков). Они оба базируются на концепции стековой машины.

Важное место занимают низкоуровневые представления на ассемблере, не сохраняющие синтаксис и семантику. В них полностью стираются какие-либо данные, характеризующие исходную программу. К примеру, циклы заменяются на последовательность условных или безусловных переходов, что означает необходимость аналитического обнаружения подобных конструкций. Восстановлению семантической структуры посвящен целый раздел бинарного анализа программ. Самыми известными анализаторами таких структур являются IDA Pro [99], Ghidra [100].

Среди промежуточных представлений можно выделить форму *статического единственного присваивания* [73]. В ней присваивание каждой переменной производится лишь единожды, следовательно, каждое присваивание императивно порождает новую версию переменной. В случае присваивания в поля структуры, изменяется весь объект, а в случае изменения сложных объектов требуется инвалидировать весь объект. Такая форма применяется в некоторых конечных представлениях (например, SMT). Она также удобна для анализа использования переменных. Форма *множественного присваивания* подходит для проведения анализа абстрактных синтаксических деревьев, проверки типов и некоторых других видов анализа.

Исследователи также предпринимали попытки сделать промежуточные представления для многоязычного анализа [101]. Уже упоминался Soot [82] с поддержкой нескольких языков. Создано множество работ, покрывающих отдельные аспекты анализа [102; 103]. Далеко в таком анализе продвинулся проект [104]. Rascal DSL [105] также работает с множеством языков. В работе [106] рассматривается представление для анализа Java и Python, основанное на XML.

1.3 Языки спецификации поведения

В качестве математических формализмов для описания поведения программ используются темпоральные логики LTL (Linear Temporal Logic) и CTL [107] (Computation Tree Logic). LTL позволяет задать темпоральные свойства путей исполнения. CTL — древовидная темпоральная структура, отображающая все возможные пути исполнения программ с ветвлениями и добавляющая темпоральные операторы-кванторы. CTL [108] вводит структуры Крипке [109].

Самым известным языком спецификации поведения является язык PROMELA, используемый в проекте SPIN [110]. Он позволяет специфицировать поведение программы с помощью конечных автоматов с дальнейшей трансляцией в LTL. Предназначается для многопоточных программ с взаимосвязями потоков через каналы и глобальные переменные.

Популярность также набирает язык ACSL [111], используемый во фреймворке для верификации Frama-C [64], основывающийся на языке из фреймворка Caduceus [112]. В нем предлагается записывать спецификации как комментарии в коде, начинающиеся с `/*@` (в отличие от Doxygen [113], где предлагается применять `/**`).

Для специфицирования поведения библиотек служит язык PanLang [114], упрощающий процесс миграции с одной версии библиотеки на другую.

Значительное влияние имеет язык Eiffel [115], в котором *контракты функций* и других сущностей проверяются статически во время компиляции и динамически во время работы программы.

Популярным языком является также и JML [116], на базе которого построены Caduceus и ACSL. Он опирается на идею “Design By Contract” из Eiffel.

Таким образом, большинство языков спецификации поведения для анализаторов общего назначения применяют контракты, в то время как в целях верификации используются более сложные языки задания поведения.

1.4 Методы верификации

Одним из основоположников верификации является Тони Хоар. Он ввел понятие логики Хоара, в центре внимания которой находятся триплеты Хоара:

$$\{P\}C\{Q\}$$

где P представляет собой утверждение-предусловие, C — команду, Q — утверждение-постусловие.

Темпоральные логики, введенные Пнуели [117; 118], позволяют верифицировать реактивные системы. Различаются линейные логики (Linear Temporal Logic, LTL) и логики, основанные на вычислительных деревьях (Computation Tree Logic, CTL). Линейные темпоральные логики применяют пропозициональные переменные AP, логические операторы $\neg$ и lor, и темпоральные модальные операторы X. Логики на вычислительных деревьях рассматривают модель времени как древовидную структуру по различным путям исполнения, добавляя операторы-кванторы A (по всем путям) и E (существует один путь).

У линейной темпоральной логики есть также вариант описания в терминах теории автоматов — автоматы Бюхи [119].

Следующим этапом стала методика *проверки модели* (Model Checking), предложенная Эмерсоном и Кларке [120]. Развитие Model Checking произошло вследствие появившейся необходимости верификации конкурентных систем.

С этим методом связана проблема экспоненциального взрыва пространства состояний, поэтому в практических применениях обычно рассматривается ограниченный вариант метода — ограниченная проверка модели (Bounded Model Checking) [121]. Через ограничение обработки источников экспоненциального взрыва (циклов, рекурсивных вызовов) достигается уменьшение пространства состояний до приемлемого размера.

Модели часто проверяются решателями задач проверки выполнимости булевых формул (SAT), а точнее их подвидом на базе решателей для задачи выполнимости формул в теориях (SMT) [122]. Они позволяют решать задачи

в доменах битовых векторов, чисел с плавающей точкой, списков, массивов, указателей, строк. Чаще всего они используются методы DPLL(T) [123], CDCL [124].

Другим методом анализа в условиях экспоненциального взрыва стала абстрактная интерпретация [125], предложенная Патриком Кузо [58]. Основная идея метода заключается в трансляции конкретных значений в абстрактные домены путем над-аппроксимации. К примеру, набор значений $(1, 2, 5, 8)$ может быть представлен как интервал $[1, 8]$. Безусловно, трансляция в абстрактные домены влияет на способность анализатора делать точные выводы о значениях: если переменная имеет значение в интервале $[1, 8]$, то результат проверки значения i на равенство 5 не определено. Любое абстрактное значение может быть представлено либо как $\bot$ (отсутствие значения), либо как $\top$ (любое значение), либо значением абстрактного домена. Выбор абстрактного домена может варьироваться в зависимости от требуемой точности: это может быть знаковый домен (значения $-$, $+$), численный домен, интервальный домен, многогранные домены. Для проведения анализа циклов используются операции *расширения* абстрактных доменов и *сужения*. Операция расширения позволяет увеличить диапазон значений, включая в домен новые значения без учета ограничений (например, если в цикле происходит инкрементирование переменной i, имеющей начальное значение $[4, 4]$, то операция расширения дает абстрактное значение $[4, +\inf]$. Операция сужения уменьшает диапазон значений до определенного порога, обычно заданного условием цикла. Например, при условии $i < 8$ (представимом как $[-\inf, 7]$) применение операции сужения даст переменной i абстрактное значение $[4, 7]$, что корректно отображает возможные значения переменной внутри цикла.

Приведенные методы решают не все проблемы, например, не покрывается случай рекурсивных проблем и нетривиальных циклов. Применение конверсии в конечную модель из-за ограничений этого метода не дает точное решение, а надмножество значений, получаемое в результате абстрактной интерпретации, может не иметь достаточной репрезентативности для проверки всех интересующих пользователя свойств. Одним из решений может являться применение ограниченных дизъюнктов Хорна (Constrained Horn Clauses) [61].

При обращении к решателю (Spacer [126]), поддерживающего этот метод, можно добиться разрешения рекурсии. Также необходимо провести корректную трансляцию из промежуточного представления в SMT- (CHC-) представление. В исследовании [127] и в проекте [128] применяется вывод регулярных инвариантов для улучшения обработки различных свойств алгебраических типов данных.

Во всех приведенных методах, основывающихся на SMT (CHC) представлениях, есть важный недостаток — производительность. Трансляция всей программы в такое представление может осуществляться в короткие сроки, однако *верификация* крупных проектов практически невозможна при наличии большого числа переменных. В практических ситуациях верификация функции размером в 100 строк может осуществляться в течение нескольких секунд для каждой верификационной цели.

Решением проблемы может оказаться применение «легких» методов анализа. Одну из таких процедур представляет символьное исполнение [129]. В этом методе для каждой переменной создается символьная трасса, содержащая нетривиальное или тривиальное выражение в противовес явному значению переменной, выраженному числом, интервалом или другим численным доменом. Вычисление таких выражений собственным решателем может осуществляться нативно, что обеспечивает значительный прирост производительности в самых используемых сценариях. Метод реализуется в проекте Klee [130] для генерации юнит-тестов, а также расширяется компанией Huawei в проекте [131] путем добавления «двунаправленного символьного исполнения».

Значительный вклад в развитие методов верификации внесла логика разделения (Separation Logic) [132; 133], разработанная Джоном Рейнольдсом и Питером О’Хирном. Она применяется для верификации программ, содержащих изменяемые структуры данных и указателей. Основными понятиями в этой логике являются *утверждения*, константа *emp* для неопределенных указателей, бинарный оператор отображения кучи на значение, оператор $*$ — разделяющая конъюнкция, оператор $-\!\!*$ — *разделяющая импликация* и *правило фрейма*, которое дает возможность масштабировать верификацию на большие пространства состояний. Логика используется в анализаторе Infer [134].

1.5 Методы интеграции

Методология интеграции статических анализаторов во многом зависит от их сервисных моделей [26].

В абсолютном большинстве анализаторов кода, включая Infer [134], cppcheck [45], PVS Studio [40], Svace [35] и т.п., применяется консольный интерфейс. Такие программы запускаются как отдельные процессы (через fork/exec для UNIX-подобных систем [135] и CreateProcess для Windows [136]). Для интеграции такой сервисной модели в среды разработки чаще всего применяется перенаправление ввода-вывода с помощью конвейеров (pipes). Это самый простой путь, хотя он и не обладает большой эффективностью вследствие обязательной сериализации ввода.

Альтернативой служат сетевые серверы, опирающиеся на протоколы TCP, UDP [137; 138]. Для подсветки кода и выполнения других рутинных действий часто используются языковые серверы [139]. Ярким решением является протокол JetBrains Reactive Distributed Communication Framework [140], используемый в IDE Rider и обеспечивающий высокую производительность этой среды. Несмотря на то, что передача данных по сети — относительно медленный процесс, скорость *петлевых* соединений (loopback) крайне велика, а TCP/UDP значительно ускоряются [141] с помощью аппаратного обеспечения, что нивелирует нагрузку на центральный процессор.

Подтипом сетевой интеграции является *облачная модель*, на базе которой работают такие проекты, как SonarQube [47], Coverity [44] и т.п.

Сетевая интеграция может быть без состояния [142], с сессиями через какой-либо другой протокол (например, JSON-RPC [143]). Взаимодействие через протоколы *масштабируемости* наподобие ZeroMQ [144] и nanomsg [145] становится все более используемым.

Для статических анализаторов разработан формат SARIF [71], унифицирующий их вывод. Одновременно разрабатывается стандарт SASP [72], который позволяет интегрировать анализаторы в среды.

Приведенные программы и протоколы основываются преимущественно на *вертикальной* модели, то есть во взаимодействии явно присутствует главный и зависимый модули. На момент написания данного обзора автор

не нашел примеры интеграции между утилитами, основанными на *горизонтальной* модели.

Глава 2. Архитектура программного решения

Разработанный статический анализатор является приложением для Windows, Linux и MacOS.

Полная схема работы анализатора приведена в Приложении А. Следует отметить, что для простоты данная схема не включает над-уровень реализации серверного режима.

В основе статического анализатора лежит модульный подход. Основные модули (Приложение Б) статически связаны между собой, в то время, как реализации конкретных языков программирования, решателей и т.п. загружаются динамически (кроме случаев, когда такая загрузка технически невозможна).

Анализ начинается с разбора рабочей среды (п. 3), то есть чтения исходного кода с построением *обобщенных синтаксических деревьев*, которые подвергаются редукции (п. 3.4). Далее обобщенные синтаксические деревья транслируются в промежуточное представление (п. 4.3) на языке Midair (п. 4.1), которое выполняется при помощи виртуальных машин. Применение гибридного решателя с моделированием различных аспектов программы (п. 5) дает в качестве результата некоторое *аналитическое представление* программы, или *гибридную модель*, которая подвергается анализу. Для обеспечения работы внутри комплексной технологии программирования применяются хранилища семантики и различные методы межпроцессного взаимодействия (п. 6).

Анализатор выявляет различные классы ошибок (Приложение В) во внутрипроцедурных и межпроцедурных режимах.

Глава 3. Особенности чтения исходных текстов

Методика чтения состоит из нескольких основных этапов. Для начала необходимо ввести формальную модель последовательной программы (п. 3.1). П. 3.3 рассматривает алгоритмические основы языково-независимого анализа. В п. 3.4 вводится алгоритм чтения нескольких файлов и объединения их результатов, в том числе и для распределенного анализа. В п. 3.5 разбирается иерархическая система типов. П. 3.6 описывает механизм сериализации данных.

3.1 Формальная модель программы

Рассмотрим абстрактное синтаксическое дерево (АСД) программы. Оно сформировано из синтаксических ветвей, имеющих определенные свойства (тип, операцию) и подвыражения. Тогда в форме, аналогичной форме Бэкуса-Наура [146], можно ввести рекурсивное определение синтаксической ветви AbstractSyntaxNode через Null (аналог нулевого указателя) и Property (свойство произвольного вида):

$$AbstractSyntaxNode ::= Null \mid Property * \ AbstractSyntaxNode*$$

Рассмотрим обобщенное синтаксическое дерево программы и его элементы.

Определение 3.1. Обобщенное синтаксическое дерево — древовидная структура программы, состоящая из синтаксических ветвей унифицированного языкового базиса.

Пусть $r \in O^R$ — один ресурс из множества ресурсов программы, $f \in O^F$ — один фрагмент из множества фрагментов программы, $n \in O^N$— пространство имен из множества пространств имен, а $t \in O^T$ — один тип из множества типов. Ресурсы, фрагменты и пространства имен имеют собственный уникальный идентификатор, в то время как типы имеют уникальный

хеш, который тоже является идентификатором. Добавим множество идентификаторов к множеству объектов, образуя пары:

$$\begin{aligned}
\forall(id_i, r_i), (id_j, r_j) \in (ID^R, O^R) \quad i \neq j \quad id_i \neq id_j, r_i \neq r_j \\
\forall(id_i, f_i), (id_j, f_j) \in (ID^F, O^F) \quad i \neq j \quad id_i \neq id_j, f_i \neq f_j \\
\forall(id_i, n_i), (id_j, n_j) \in (ID^N, O^N) \quad i \neq j \quad id_i \neq id_j, n_i \neq n_j \\
\forall(id_i, t_i), (id_j, t_j) \in (ID^T, O^T) \quad i \neq j \quad id_i \neq id_j, t_i \neq t_j
\end{aligned}$$

Аналогично абстрактному синтаксическому дереву, оно имеет определенные свойства и подвыражения:

$$Expression ::= Null \mid Property * \ Expression*$$

Введем рекурсивную структуру карты пространства имен, начинающуюся от корневого пространства имен. Пусть NamespaceNode — одно пространство имен с определенным ID, NamespaceNodeList — список дочерних пространств имен, NamespaceMap — вся карта пространств имен. Тогда в форме, аналогичной форме Бэкуса-Наура, описание структуры будет выглядеть следующим образом:

$$\begin{aligned}
NamespaceNode ::= Null \mid Id\ NamespaceNodeList \\
NamespaceNodeList ::= Null \mid NamespaceNode+ \\
NamespaceMap ::= NamespaceNode
\end{aligned}$$

Таким образом, целью алгоритма чтения исходных текстов является конверсия из AbstractSyntaxNode в множество указанных структур данных:

$$\begin{aligned}
Parser : AbstractSyntaxNode \rightarrow ((ID^R, O^R), (ID^F, O^F), \\
(ID^N, O^N), (ID^T, O^T), \\
NamespaceMap)
\end{aligned}$$

Чтение (разбор) исходных файлов осуществляется по схеме из Приложения Г. Введем операции предварительной обработки, трансляции и

постобработки:

$$PreResult ::= Continue \mid Stop \tag{3.1}$$

$$Pre(AbstractSyntaxNode) \rightarrow (O^R, O^F, O^N, O^T)', PreResult, NewAbstractSyntaxNode \tag{3.2}$$

$$(PreResult = Continue) \implies Convert(NewAbstractSyntaxNode) \rightarrow Expr \tag{3.3}$$

$$(Expression \neq Null) \implies Post(NewAbstractSyntaxNode, Expr) \rightarrow put(f \in O^F, Expr) \tag{3.4}$$

Метод предварительной обработки (3.2) заключается в трансляции абстрактной синтаксической ветви в набор ресурсов, фрагментов и пространств имен. В результате метод предлагает вердикт: продолжить трансляцию (Continue) или завершить (Stop). Он также решает задачу *обхода* прямой трансляции [147] в случае, когда в ней нет необходимости. Например, блок выражений представлен в виде фрагмента, и трансляция в выражение (Expression) будет бессмысленной. Такой обход лишнего этапа позволяет ускорить трансляцию.

Трансляция (3.3) заключается в прямом конвертировании абстрактной синтаксической ветви в выражение обобщенного синтаксического дерева, если в результате предварительной обработки получен результат Continue. Результатом трансляции может быть нулевое или ненулевое выражение.

Постобработка (3.4) производит добавление выражения во фрагмент $f \in O^f$, если в результате трансляции получилось ненулевое выражение.

3.2 Семантики языков программирования

Каждый язык программирования имеет свои особенности. Например, в языках C, C++ есть такие механизмы, как *неявная конверсия массивов в указатели (array to pointer decaying)* [148], *целочисленное повышение (integer promotion)* [149], *перегрузка операторов (в C++)*. Это означает, что в реализации анализатора должны быть точки расширения. Такой точкой расширения может быть *языковая семантика*.

Определим семантику в соответствии с работой [150].

Определение 3.2. Языковая семантика — сущность, определяющая набор свойств, отличающих ассоциированный язык программирования от других языков программирования, и предоставляющая способ моделирования указанных свойств.

Пусть $p \in P$ — произвольное свойство из множества свойств. Тогда каждый язык программирования $l \in L$ подразумевает существование ассоциированного множеств свойств (n — количество свойств):

$$Props : L \to P^n$$

Языки могут организовывать иерархии. Зададим оператор $\oplus$, добавляющий свойства к языку:

$$l_2 = l_1 \oplus P^+ \tag{3.5}$$

$$Props(l_2) = Props(l_1) \cup P^+ \tag{3.6}$$

Зададим оператор $\ominus$, удаляющий свойства из языка:

$$l_2 = l_1 \ominus P^- \tag{3.7}$$

$$Props(l_2) \cup P^- = Props(l_1) \tag{3.8}$$

На основе данных операторов 3.5 и 3.7 можно задать иерархические последовательности языков:

$$l_1 \quad l_2 = l_1 \oplus P_1^+ \ominus P_1^-$$
$$l_3 = l_2 \oplus P_2^+ \ominus P_2^- \quad \ldots \quad l_n = l_{n-1} \oplus P_n^+ \ominus P_n^-$$

В дальнейшем подобные семантики можно использовать при выполнении статических проверок, производя запросы к конкретным семантическим свойствам.

3.3 Организация языково-независимого чтения

Независимость анализа по отношению к языку читаемой программы строится на основе *обобщенных синтаксических деревьев* и *языковых*

семантик. Рассмотрим языковой базис для императивных языков программирования. Согласно проведенному автором анализу языковых конструкций языков C, C++ и практической проверке, базис должен содержать некоторые обязательные конструкции (Приложение Д).

Обобщенное синтаксическое дерево не является полным отображением абстрактного синтаксического дерева исходной программы. Часть выражений формулируется в терминах *фрагментов* и их взаимосвязей. Приведем некоторые примеры таких конструкций. **Ветвления** записываются в виде *условий входа* во фрагменты. Также во фрагмент записывается конкретный тип ветвления. **Циклы** оформляются как условия входа во фрагменты. **Break**, **Continue** — остановка действия текущей конструкции или переход к следующей итерации цикла. Они транслируются посредством установки идентификатора *следующего* фрагмента во фрагментах. **Label** — метки для goto. Добавляются во фрагменты в виде строк и используются при определении *следующего* фрагмента. **GoTo** — явно задают идентификаторы меток следующего фрагмента.

Принцип трансляции синтаксических конструкций для языков C/C++ приведен в Прил. И, для языка РуСи — в Прил. К. Аналогично транслируются синтаксические конструкции языков аннотаций ACSL (Прил. Л) и контрактов РуСи.

Языковой базис не фиксирован. Если в процессе добавления возможности чтения программ на новом языке возникнет необходимость внедрить новые конструкции, это допустимо при условии добавления поддержки во все сопутствующие модули. Однако во множестве случаев это не требуется, так новые функции языков часто реализуются как «синтаксический сахар» поверх языкового базиса.

Не каждый язык программирования может быть транслирован в указанный базис напрямую, но для доказательства возможности трансляции можно воспользоваться **сведением задачи к трансляции ассемблера в данный языковой базис**.

Примем без доказательства следующие утверждения:

Утверждение 3.1. Ассемблеры основных современных кремниевых компьютеров являются императивными языками программирования.

Утверждение 3.2. Исполняемые на основных современных кремниевых компьютерах программы представимы в виде программ на соответствующем ассемблере.

Замечание 3.1. Альтернативой ассемблеру может являться низкоуровневое представление, такое как LLVM IR, MSIL и т.п.

Корректность указанных утверждений строится на основе состояния науки и техники на момент написания данной работы.

Теорема 3.1. Исполняемые на процессоре X86 программы представимы в виде языкового базиса (Приложение Д).

Доказательство. Предположим, что возможность транслировать основные конструкции ассемблера в языковой базис определяет представимость указанных программ в языковом базисе. Рассмотрим основные команды ассемблера X86 [8].

– **Стековые инструкции**. Команды `push`, `pop` отвечают за сохранение регистров в стеке и выгрузку данных из стека в регистры. Как правило, это необходимо для восстановления регистров после выхода из области, условно считающейся функцией. Данный процесс может быть эмулирован при помощи массивов, доступа к ним через *ArraySubscript* и присваивания через выражение типа *Binary*. Команды `sub`, `add` и другие в приложении к стеку отвечают за выделение пространства на стеке непосредственно для хранения переменных. Выделение пространства в стеке возможно через выражение типа *Declaration* и формирование ресурсов при помощи высокоуровневого транслятора. Команда `call` транслируется в команду *Invocation*, а `ret` — в *Return*. Эти команды скорее относятся к управлению потоком, но они также и управляют стеком.
– **Арифметические целочисленные инструкции и инструкции для арифметики с плавающей точкой**. Данный тип команд может быть представлен комбинациями выражений типа *Binary*, *PrefixUnary*, *PostfixUnary* и т.п.
– **Инструкции SIMD** [151]. Инструкции по параллелизации вычислений могут быть оформлены как вызовы отдельных псевдо-функций

через `Invocation`, а также приведены к последовательному виду через последовательности фрагментов с ветвлениями.

– **Инструкции по управлению памятью**. Загрузка данных из памяти в регистры и сохранение данных регистров в памяти возможны через прямое использование арифметических выражений (*Binary*, *PrefixUnary*, *PostfixUnary* и т.п.), обращение к памяти через *ArraySubscript* и *MemberAccess*.
– **Инструкции по управлению потоком управления**. Команды сравнения регистров можно эмулировать при помощи условий входа во фрагменты с правильным заданием типа условия входа. Команды условных и безусловных переходов должны быть транслированы в метки *goto* и указание следующего фрагмента в каждом фрагменте. Команды `call`, `ret` могут быть транслированы в `invoke` и `return`. Команда `int` также может быть транслирована в `invoke`.

Таким образом, по построению, команды ассемблера x86 могут быть транслированы в указанный языковой базис. □

Аналогичным образом можно доказать эту теорему для процессоров ARM [152] и MIPS [153; 154]. Исходя из данных утверждений, можно вывести алгоритм (3.1) трансляции программ в данный языковой базис.

Алгоритм 3.1. Схема безусловной трансляции программ на разных языках программирования в языковой базис

Последовательность действий:

1. Если язык программирования допускает трансляцию в языковой базис «как есть», то необходимо её осуществить.
2. В противном случае необходимо транслировать программу в низкоуровневое представление (ассемблер, LLVM IR, MSIL и т.п.) и после этого транслировать в языковой базис.

Отметим, что в статическом анализе в работе [33] поверх указанного базиса была реализована поддержка C# (императивного языка с функциональными элементами), что подтверждает применимость указанной схемы для императивных языков с функциональными элементами [155]. Так как C# основан на CIL [156], то сведением F# [157] к CIL можно доказать и применимость базиса к данному функциональному языку.

Указанного базиса достаточно для работы с синтаксическими конструкциями различных языков программирования и подготовки данных к следующим этапам разбора.

3.4 Метод чтения нескольких файлов и объединения их семантики

Множества O^R, O^F, O^N, O^T представляют собой семантический контекст программы. Однако при работе с несколькими исходными файлами требуется объединение семантических контекстов для более полного анализа.

Рассмотрим алгоритм объединения семантических данных (3.2) на примере объединения двух контекстов.

Алгоритм 3.2. Объединение семантических контекстов

Последовательность действий:

1. Объединение фрагментов разных семантических контекстов осуществляется тривиально через объединение фрагментов. Пусть $O^{F'}$ — множество фрагментов в объединенном семантическом контексте, а O_1^F, O_2^F — множества фрагментов в семантических контекстах отдельных файлов. Аналогично введем множества идентификаторов ID.

   Введем функцию трансляции идентификаторов: $TranslateID$. Эта функция принимает на вход множества ID_1, ID_2 произвольных идентификаторов и видоизменяет множество ID_2 так, чтобы не было пересечений идентификаторов между данными семантическими контекстами:

   $$TranslateID : ID_1 \times ID_2 \to ID_2' \tag{3.9}$$

   $$ID_1 \cap ID_2 = \varnothing \tag{3.10}$$

   Тогда их объединение будет выглядеть следующим образом:

   $$O^F = O_1^F \cup O_2^F \tag{3.11}$$

   $$ID = ID_1^F \cup TranslateID(ID_1^F, ID_2^F) \tag{3.12}$$

2. Объединение ресурсов происходит по схеме, включающей *хеширование* ресурса. Введем понятие функции хеширования ресурса. Пусть каждый ресурс $r \in O^R$ имеет хеш $\bar{r} \in \bar{O^R}$. Тогда функцию хеширования можно ввести следующим способом:

   $$Hash : O^R \to \bar{O^R} \tag{3.13}$$

   В то время, как идентификатор является уникальным идентификатором объекта в рамках семантического контекста, хеш является уникальным идентификатором

объекта в рамках всей сессии анализа. Таким образом, задача объединения ресурсов сводится к выявлению хешей всех объектов и корректировке идентификаторов объектов в остальных объектах.

Введем функцию очистки дублирующихся ресурсов. Она будет принимать на вход множества объектов O_1^R, O_2^R и формировать множество объектов $O_2^{R'}$, являющееся подмножеством O_2^R без дублирующихся объектов:

$$ClearClones : O_1^R \times O_2^R \to O_2^{R'} \tag{3.14}$$

$$O_2^{R'} \subset O_2^R \tag{3.15}$$

$$\forall r_1 \in O_1^R, r_2 \in O_2^{R'}, Hash(r_1) \neq Hash(r_2) \tag{3.16}$$

Аналогично можно ввести функцию $ClearIDClones$ для идентификаторов объектов (за тем исключением, что она будет дополнительно принимать на вход множество изменяемых идентификаторов).

Объединение будет выглядеть следующим образом:

$$O^R = O_1^R \cup ClearClones(O_1^R, O_2^R) \tag{3.17}$$

$$ID = ID_1^R \cup TranslateID(ID_1^R, ClearIDClones(O_1^R, O_2^R, ID_2^R)) \tag{3.18}$$

3. Рассмотрим объединение пространств имен. Оно происходит аналогично ресурсам, за тем лишь исключением, что функция хеширования будет полностью зависеть от строкового представления пространств имен (со специальной обработкой анонимных пространств имен). Также, в отличие от ресурсов, пространства имен не одинаковы и должны быть явным образом объединены.

   Введем функцию объединения двух пространств имен n_1 и n_2 в пространство n':

$$MergeNamespace : n_1 \times n_2 \to n' \tag{3.19}$$

   Детали её реализации могут различаться для технически отличающихся анализаторов кода, но основная суть заключается в необходимости добавить дочерние пространства имен и объединить объекты в них.

   Для удобства работы введем оператор $Join(X_1, X_2, hash)$, который объединит элементы множеств X_1, X_2 в пары при равенстве их хеша, задаваемого оператором $hash$.

   В итоге функция объединения двух множеств пространств имен выглядит следующим образом:

$$MergeNamespaces(O_1^N, O_2^N) = O^N \tag{3.20}$$

$$\tilde{O} = Join(O_1^N, O_2^N, NamespaceHash) \tag{3.21}$$

$$O_N = [\ldots MergeNamespace(\tilde{O}_{i_1}, \tilde{O}_{i_2}), \ldots] \tag{3.22}$$

4. Объединение типов осуществляется через прямое объединение посредством метода хеширования аналогично ресурсам.

$$O^T = O_1^T \cup ClearClones(O_1^T, O_2^T) \tag{3.23}$$

$$ID = ID_1^T \cup ID_2^T \tag{3.24}$$

Технически объединение контекстов осуществляется по алгоритму 3.3.

Алгоритм 3.3. Объединение семантических контекстов

Последовательность действий:

1. Осуществляется проход карт пространств имен: целевой (куда происходит объединение) и вторичной (с которой происходит обновление).
    а) Если в целевом пространстве имен нет объекта с таким названием, то объект переносится в новое пространство имен с корректировкой ID.
    б) Если в целевом пространстве имен есть объект с таким названием, то объекты объединяются. Если объекты семантически разные, то они добавляются под одним названием.
    в) Строится карта корректировок ID, которая применяется при объединении фрагментов.
2. Объединение фрагментов.
    а) Фрагменты с непересекающимися ID переносятся напрямую в новое пространство имен.
    б) При пересечении ID фрагментов, происходит корректировка ID фрагмента. Измененные ID заносятся в карту корректировок. После корректировки фрагмент переносится в дерево.
    Важное обстоятельство: так как файлы являются самостоятельными единицами трансляции, то в дерево фактически переносится только верхний уровень.
    в) В выражениях всех перенесенных фрагментах корректируются ID объектов.

Важно отметить, что для минимизации пересечений ID между единицами трансляции анализатор делает *аппроксимацию* диапазона идентификаторов. В частности, разбор каждого читаемого файла начинается с другой группы динамически вычисляемых ID. Если анализ происходит последовательно, то ID вычисляются точно, что полностью ликвидирует проблему пересечений. Если анализ происходит параллельно, то анализатор осуществляет предварительную подготовку набора ID при помощи эвристических вычислений на основе размеров входных файлов.

Описанный метод работы с семантикой может использоваться в комбинации с алгоритмом Map-Reduce [158] для проведения *распределенного анализа*. Для этого выполняются все основные условия: контексты файлов могут читаться отдельно друг от друга и потом *коммутативно* объединяться.

3.5 Система типов данных

Универсальный базис состоит в том числе из типов данных, организация хранения которых существенно отличается от других элементов. Выделим три основные сущности, характерные для системы типов данных:

1. **Описание типа (дескриптор)** — сырое описание целевого типа. Например, `int*` описывает тип «указатель на переменную типа int», при этом независимо от фактического размера `int` данный дескриптор различим при сравнении с `int32_t*`, `int64_t*` и т.п. Это позволяет компиляторам приводить точную с точки зрения пользователя диагностику.
2. **Структура типа** — описание внутренней иерархической структуры типа.
3. **Реализация типа** — описание методов хранения и конверсии объектов.

Система типов в проекте представляет собой иерархическую систему, самым «низким» уровнем в которой является непосредственно тип данных. Например, тип данных «знаковое число разрядности 32 бита» указывает на фактическую длину в 4 байта и наличие знака в одном из разрядов числа, что уменьшает возможный диапазон значений в абсолютных числах.

В реальных программах большое значение имеют *квалификаторы*. Например, константность параметров или аргументов функции явным образом влияет на разрешение имен при перегрузке функций в языке C++.

Процесс ассоциирования типа данных с квалификатором или несколькими квалификаторами порождает *квалифицированный* тип данных. Типы данных могут включать себя другие квалифицированные типы данных. Например, структура может содержать именованный объект константного интегрального типа.

3.5.1 Принципы хранения данных

В C и РуСи система типов фактически является плоской: разрешение типа фактически ограничивается сравнением имени. В C++ и других языках существуют *пространства имен* [159], что значительно усложняет поиск типов и заставляет производить поиск не только внутри иерархической структуры непосредственно типов, но и внутри над-иерархии пространств имен. Название, квалификаторы и последовательность пространств имен, таким образом, составляют *идентификатор* типа. В связи с тем, что один и тот же тип может быть квалифицирован по-разному, порождаются отдельные множества квалифицированных и неквалифицированных типов.

Не умаляя предыдущих выводов, рассмотрим O^T как множество квалифицированных типов, а $t \in O^T$ — как один из квалифицированных типов. Тогда O^{T_u} — множество неквалифицированных типов, а t^u — один из неквалифицированных типов. Введем также $s \in S$ — произвольного вида структуру данных из множества всевозможных структур данных. Справедливо соотношение 3.25:

$$|O^{T_u}| \leqslant |O^T| \tag{3.25}$$

Определим структуру типа:

$$t = (n_t, q_t, t_t^u) \tag{3.26}$$

где n_t — ассоциированное с квалифицированным типом пространство имен, q_t — ассоциированные с типом квалификаторы, а t_t^u — ассоциированный неквалифицированный тип данных.

Тогда под неквалифицированным типом данных будем понимать:

$$t^u = (n_{t^u}, s_{t^u}) \tag{3.27}$$

где n_{t^u} — ассоциированное с неквалифицированным типом пространство имен, s_{t^u} — структура данных типа.

В вершине иерархической структуры стоит глобальное пространство имен, то есть пространство с объектами и типами, использование которых допустимо из всех модулей без ограничений. В одном пространстве имен

типы могут иметь одинаковую структуру, но разные имена, при этом автоматическая дедупликация не является возможной за единственный просмотр исходного текста в свете существования *неполных типов данных*.

При использовании анализатора в реальных проектах важно обеспечить *кэширование типов*, предоставляемых парсером. Для этого в реализацию кэша встроены ассоциативные таблицы для *адресов типов из парсера* (прямая ассоциация до целевого типа), *пространств имен* (ассоциации контекста абстрактного синтаксического дерева с пространства имен).

Введем функции кэширования типа по различным признакам. Пусть $p \in P$ — произвольный признак. Тогда введем функции Cache для прямого ассоциирования признака с типом и BackCache для обратной ассоциации типа с признаком.

$$Cache : P \to O^T \tag{3.28}$$

$$BackCache : O^T \to P \tag{3.29}$$

Не все свойства есть смысл кэшировать. Указатель на тип в терминах исходного парсера является хорошим свойством для добавления в множество P, так как при чтении исходного текста этот указатель будет биективно связан с квалифицированным типом в терминах анализатора. Свойства возможно кэшировать только на время парсинга непосредственно данного исходного текста до высвобождения памяти контекста чтения. Однако предполагается, что типы в этих терминах в дальнейшем и не требуются.

Важно обеспечить *хеширование* типа, то есть механизм формирования уникального идентификатора объекта. Оно осуществляется простым алгоритмом djb2 [160]. Процедура хеширования проходит по всем значимым для целевого языка полям типа и ведет расчет контрольной суммы.

Пусть $h \in H$ — произвольный хеш во множестве хешей. Тогда процедура хеширования представлена следующим отображением:

$$Hash : O^T \to H \tag{3.30}$$

Таким образом, система типов состоит из следующих объектов:

$$TypeSystem = (O^T, O^{T_u}, Cache, BackCache, Hash) \tag{3.31}$$

В практической реализации *создание типов* происходит по следующей схеме (Алгоритм 3.4).

Алгоритм 3.4. Алгоритм создания типов

Последовательность действий:

1. Пользователь отправляет запрос на создание типа, явно указывая параметры типа (в том числе типы вложенных выражений), предоставляя указатель на тип в терминах модуля чтения.
2. В случае, если указатель в терминах парсера присутствует в ассоциативном кэше, то из него извлекается указатель на целевой тип, и процедура завершается.
3. Производится хеширование типа с учетом информации, явно представленной в определении типа в целевом языке программирования. Например, в случае C/C++ декларатор типа «структура» не содержит сведений о полях структуры, поэтому они не учитываются при хешировании.
4. Производится поиск хеша в кэширующей подсистеме:
    а) Если хеш найден, то сравниваются все объекты с данным хешем во избежание коллизий.
        1) Если найден объект с идентичными характеристиками, то именно он и возвращается как результат запроса на создание типа.
    б) Если хеш или тип не найдены, то новый объект создается в целевом пространстве имен.

Приведенная процедура создания типов имеет несколько преимуществ.

1. Важнейшее влияние имеет отсутствие дубликатов неквалифицированных типов. Это позволяет сэкономить значительный объем оперативной памяти, так как количество типов относительно ограничено, а выражений может быть много.
2. Отсутствие дубликатов неквалифицированных типов приводит к уменьшению накладных расходов на выделение памяти.
3. Наличие уникального идентификатора типа благодаря хешированию, которое позволяет быстро проверять существование типа в хранилище типов. Это ускоряет процесс добавления, так как ликвидирует необходимость прямого сравнения типов по их свойствам.
4. Высокая производительность благодаря ассоциативным кэшам.
5. Поддержка неполных типов через корректировку алгоритма хеширования с целью пропустить хеширование отдельных частей объекта (например, полей класса).

У процедуры есть следующие недостатки:

1. Хеширование может быть длительным процессом. Отчасти эта проблема нивелируется через использование ассоциативных кэшей для

разных случаев (например, в случае отображения «указатель на тип в терминах парсера» — «указатель на тип в терминах анализатора»).

2. Неэффективное хранение квалификаторов типов. Квалифицированный тип является отдельным объектом с несколькими полями, хотя, например, Clang решает эту проблему через введение служебных битов в указателе на неквалифицированный тип, что несомненно быстрее.

Создание типов данных по данной схеме позволяет эффективно анализировать большие файлы.

3.5.2 Структура типов

Типы формируются через простую иерархическую структуру. Например, тип `int*` является указателем на простой тип `int`, соответственно в иерархии находятся два типа: указатель, простой тип. Важно отметить, что каждый тип является также и самостоятельным объектом.

Введем формальное описание неквалифицированного и квалифицированного типов через расширение формы Бэкуса-Наура для более удобного описания структурных типов. Для этого перед членами структур добавлено название поля через двоеточие, если это улучшает читаемость.

$$\begin{aligned} Type ::= {} & IntegralType \mid ClassType \mid AlignType \mid FunctionType \mid \\ & PointerType \mid ReferenceType \mid ArrayType \mid TemplateType \mid \\ & EllipsisType \mid OpaqueType \end{aligned}$$

$$QualType ::= Qualifiers\ (Type)+$$

Типы делятся на следующие группы:

1. **Скалярные типы**: простые арифметические типы фиксированной разрядности.

$$IntegralType ::= IntegralTypeKind$$

2. **Структурные типы**: типы, наследованные от каких-либо типов и содержащие некоторый набор полей.

$$ClassType ::= Members : (Name\ MemberType : QualType)*$$

3. **Выравнивание**: определенное количество незначимых байтов.

$$AlignType ::= NumBytes : Number$$

4. **Функции**: прототипы функций, имеющие определенный тип возвращаемого значения и параметры.

$$FunctionType ::= ReturnType : QualType$$
$$Parameters : (NameParamType : QualType)*$$

5. **Указатели**: указатель на объект другого типа данных.

$$PointerType ::= Pointee : QualType\ PointerType : Type$$

6. **Ссылки**: ссылки на другой тип данных.

$$ReferenceType ::= NumReferences : Number$$
$$ReferencedType : QualType$$

7. **Массивы**: некоторый набор объектов заданного размера.

$$ArrayType ::= ArraySize\ ElementType : QualType$$

8. **Шаблоны**: тип с шаблонными параметрами.

$$TemplateType ::= (TemplateArgument) *\ TargetType : Type$$

9. **Эллипсис** ($EllipsisType$): знак многоточия, который может означать переменное число аргументов. Не имеет структуры.
10. **Непрозрачный тип** ($OpaqueType$): тип без структуры, который может быть полезен при чтении произвольного описателя типа.

Простые типы также имеют расширение в виде информации о низкоуровневом типе данных. Эта информация позволяет при необходимости конвертировать типы на поздних этапах.

3.6 Организация чтения и записи сериализированных данных

Одним из требований комплексной технологии программирования является организация инкрементального анализа. Для этого в идеальном случае

требуется работа с дифференциальными образами, которые, в свою очередь, должны формироваться на основе прочитанных и обработанных синтаксических структур.

В связи с этим, одной из важных задач анализатора является организация чтения и записи сериализированных данных. Для этого в анализаторе реализована поддержка MongoDB [161] — СУБД из нереляционного семейства [162]. Ориентированность на *документы* — то есть отдельные объекты произвольной структуры — позволяет прозрачно выгружать и загружать объекты без построения схем баз данных и частых миграций.

Для каждой рабочей среды формируется *отдельная база данных*. Для хранения данных программ используются основные концепции ресурсов, фрагментов, выражений и команд виртуальной машины (Гл. 4). Ресурсы и фрагменты имеют собственные идентификаторы, которые могут использоваться для идентификации объектов в БД. Каждый ресурс со всеми свойствами сохраняется в отдельную таблицу ресурсов. Каждому ресурсу-функции сопоставляется фрагмент, содержащий набор выражений в виде рекурсивной структуры обобщенных синтаксических деревьев. Также им сопоставляется контейнер команд виртуальной машины (Гл. 4), если анализ дошел до данного этапа. Каждая команда виртуальной машины содержит идентификатор целевого ресурса, а также данные применяемых выражений.

По окончанию анализа в БД сохраняется резюме все прочитанных функций и контрольные суммы файлов, в которых они заданы. Это позволяет существенно уменьшить время инкрементального анализа благодаря отсутствию повторной обработки.

Глава 4. Виртуальная машина и промежуточное представление Midair

Практически все компиляторы и анализаторы имеют свое промежуточное представление (Intermediate Representation, или IR). Не стал исключением и разрабатываемый проект. В нем реализован собственный язык Midair (п. 4.1), программы на котором читаются виртуальной машиной и гипервизором (п. 4.2) для осуществления дальнейшего межпроцедурного анализа.

4.1 Язык Midair

Для хранения подпрограмм используется язык промежуточного представления Midair [163]. Необходимость собственного языка обуславливается тем, что анализатор требует представления, *представляющего свойства исходной программы достаточно подробно для проведения глубокого анализа*, но в то же время без избыточных синтаксических конструкций исходного языка. Важна также *возможность абстрагироваться от низкоуровневого типа и легко осуществить перебазирование на другой центральный процессор, операционную систему, компилятор*. Для обеспечения трансляции в SMT требуется *хранение метаданных для каждой команды*. Для межпроцедурного анализа имеет значение *отсутствие прямого декларирования переменных и выделения памяти под них*, что позволяет подменять переменные на их копии. Однако самая главная особенность представления заключается в **раздельном хранении семантических данных, позволяющем осуществить межъязыковой анализ**.

Инструкции языка Midair представляют собой простой трехадресный код следующего вида:

$$x := y \ op \ z$$

Это налагает дополнительную ответственность на транслятор: необходимо максимально упростить код анализируемой программы, избавиться от синтаксического сахара как исходного языка, так и языка обобщенных синтаксических деревьев. Однако факт использования такого типа команд

упрощает разработку, уменьшая размер обработчиков команд. За счет этого увеличивается и производительность решателя.

Операнды большинства команд делятся на две группы: ресурсы и выражения. Ресурсы могут использоваться лишь при присваивании, так как это наиболее частый случай использования (например, $x = 1 + 5$, где x — ресурс). Однако присваивание поддерживает альтернативный режим, предполагающий использование обобщенные выражения в качестве левой части (например, $x[0] = 1 + 5$). Операнды y и z всегда представлены выражениями. Строго говоря, правая часть ($y\ op\ z$) представлена одним выражением, на которое накладываются ограничения трехадресного кода.

Эффективное хранение кода также возможно. Основной предполагаемый механизм: отдельное хранение ресурсов с доступом к ним по уникальным идентификаторам, «документное» хранение выражений со всей структурой в NoSQL СУБД. Таким образом, команды редуцируются до наборов идентификаторов и кодов команд.

4.2 Гипервизор и виртуальные машины

Гипервизор [164] выполняет множество задач — от создания и вызова виртуальных машин до оркестрации виртуальных машин для выполнения различных типов анализа.

Виртуальная машина выполняет разбор промежуточного представления и подготовку данных для гибридного решателя.

Упрощенная схема работы гипервизора представлена в Приложении E. Ей соответствует алгоритм 4.1.

Алгоритм 4.1. Механизм работы гипервизора

Последовательность действий:

1. Производится обход точек входа в программу.
    а) Отбираются функции-кандидаты на трансляцию. Игнорируются функции стандартной библиотеки.
    б) Запускаются виртуальные машины для каждой функции-кандидата.
    в) Осуществляется трансляция подпрограмм в промежуточное представление. Транслированные команды виртуальной машины помещаются в

контейнеры команд, адресуемые по идентификаторам ассоциированных ресурсов.

г) Осуществляется первичный анализ процедуры с построением аналитического представления (П. 5.1).

д) На основе итогового аналитического представления, формируется *резюме* процедуры.

е) Проводится подбор пар кандидатов подпрограмм и комбинирование контейнеров программ.

ж) Комбинированные контейнеры подпрограмм выполняются с помощью решателя.

з) Формируется результат анализа.

2. Результат анализа функций разными виртуальными машинами комбинируется.

Более подробно гипервизор и виртуальная машина описаны в работах [32; 164; 165].

Одна из задач, которую выполняет гипервизор — *девиртуализация* [166; 167] вызовов виртуальных функций классов. В C++ не всегда известно, какой именно метод будет вызван вследствие их многозначности. Для решения этой проблемы гипервизор составляет перечень возможных целей вызова. Если целей несколько, то их аннотации будут объединены.

4.3 Трансляция подпрограмм из обобщенных деревьев в промежуточное представление

Перед работой виртуальных машин возникает необходимость конверсии представления из обобщенных синтаксических деревьев в промежуточное представление. К этому моменту у анализатора накапливается достаточно информации: функциям и параметрам присвоены теги, которые говорят о характере работы функции (например, контекст исполнения, производимые аллокации), накоплены данные об аннотациях.

Все команды опираются на две основных концепции: *ресурсы* и *выражения* (Гл. 8.5). Это дает преимущество в дальнейшем анализе, поскольку не ограничивает характер производимых ими операций. Следовательно, анализ может осуществляться для любого языка, который представим указанными командами.

Низкоуровневые языки программирования для основных архитектур центрального процессора опираются на концепции регистров, стека, памяти, а также операций по манипуляции этими понятиями.

Теорема 4.1. Регистры могут быть транслированы в переменные.

Доказательство. Регистры представляют собой области памяти фиксированного размера. Основные команды ассемблера по управлению регистрами включают применение арифметических операций к операндам-регистрам, загрузку данных из регистров и сохранение данных регистров в памяти. Манипуляция значениями переменных-регистров возможна путем применения `assign`, а работа с загруженными значениями — через выражения типа `Reference`. □

При трансляции программ в Midair возникает проблема реализации условных и безусловных переходов. Язык не поддерживает явные переходы в стиле `goto`, потому прямая трансляция с языков с ними является затруднительной. Для решения этой проблемы предлагается предварительная обработка условных и безусловных переходов (Алгоритм 4.2).

Алгоритм 4.2. Трансляция условных и безусловных переходов
Последовательность действий:

1. При обнаружении безусловного перехода транслятор следует к целевому фрагменту, как будто команды безусловного перехода нет.
2. При обнаружении условного перехода транслятор генерирует инструкции **branch** / **end branch** с выполнением кода внутри ветки аналогично безусловному переходу.

Приведем две теоремы. Сначала объясним алгоритм приведения программ на ассемблере x86 к программам на языке Midair. Затем уточним понятие Тьюринг-полноты в приложении к промежуточному представлению.

Теорема 4.2. Программы на ассемблере для x86 представимы в виде программ на языке Midair при условии предварительной трансляции условных и безусловных переходов.

Доказательство. Рассмотрим основные команды ассемблера x86 [8]. Предположим, что в случае возможности транслировать основные конструкции низкоуровневых языков программирования в представление Midair, существует и возможность представления программ на них в целом.

– **Стековые инструкции**. Команды `push`, `pop` отвечают за сохранение регистров в стеке и выгрузку данных из стека в регистры. Как правило, это необходимо для восстановления регистров после выхода из области, условно считающейся функцией. Данный процесс может быть эмулирован при помощи `assign`.

  Команды `sub`, `add` и другие в приложении к стеку отвечают за выделение пространства на стеке непосредственно для хранения переменных. Выделение пространства на стеке для хранения переменных в рамках Midair возможно путем создания структур данных и генерации соответствующей переменной с помощью команды `declare`, которая, в свою очередь, в ходе анализа создаст регион памяти необходимого размера.

  Команда `call` транслируется в команду `invoke`, а `ret` — в `return`. Эти команды скорее относятся к управлению потоком, но они управляют и стеком.

– **Арифметические целочисленные инструкции и инструкции для арифметики с плавающей точкой**. Данный тип команд не требует трансляции в Midair, так как фактически представим множеством *выражений*, описанным в Гл. 3.

– **Инструкции векторных расширений**. Инструкции по параллелизации вычислений могут быть оформлены как вызовы отдельных псевдо-функций через `invoke`, а также приведены к последовательному виду через последовательности `branch`, `assign`, `end branch`.

– **Инструкции по управлению памятью**. Загрузка данных из памяти в регистры и сохранение данных регистров в памяти возможны путем использования команд `assign`.

– **Инструкции по управлению потоком управления**. Команды по сравнению регистров возможно эмулировать при помощи `branch`.

  Команды условных и безусловных переходов должны быть транслированы на более раннем этапе, так как Midair не поддерживает переходы в стиле goto.

  Команды `call`, `ret` могут быть транслированы в `invoke` и `return`. Команда `int` также может быть транслирована в `invoke`.

Таким образом, по построению, команды ассемблера x86 могут быть транслированы в команды Midair. □

Теорема 4.3. Язык Midair является Тьюринг-полным.

Доказательство. Проверку Тьюринг-полноты проведем путем конструирования способа эмуляции машины Тьюринга.

В машине Тьюринга существует три типа операций:

1. Сдвиг по ленте на соседнюю ячейку.
2. Запись в текущую ячейку.
3. Изменение текущего состояния машины

Формально язык Midair не задает ограничения на объекты, поэтому любой массив может по существу являться бесконечной лентой, так как восприятие этой ленты будет зависеть от исполнителя кода Midair. Следовательно, в качестве бесконечной ленты можно взять любой объект типа «массив»: $declare\ x:\ Integer\ [\,]$.

Добавим индекс текущего элемента при помощи той же команды: $declare\ i:\ Integer$.

Далее, изменяя индекс при помощи команды `assign`, можно добиться выполнения сдвига по бесконечной ленте (п. 1): $assign\ i = i + 1$.

Запись в текущую ячейку может осуществляться при помощи той же команды: $assign\ x[i] = y$.

Изменение и проверка состояния автомата может осуществляться при помощи тех же операторов `assign`, `branch`/`end branch`. Следовательно, трансляция Машины Тьюринга в программу на языке Midair возможна, следовательно, язык является Тьюринг-полным. □

Сам факт Тьюринг-полноты не является достаточным условием возможности *проверять* программы на любом языке. Более подробно о трансляции программ в верифицируемое аналитическое представление написано в Гл. 5.

Введем формализмы для описания команд.

Пусть $v \in V$ — одна команда из множества команд. Тогда последовательность команд записывается следующим образом:

$$v_1, v_2, \ldots, v_n \tag{4.1}$$

Для уточнения характера команды введем следующий формализм:

$$v1 = (v1| < kind >) \tag{4.2}$$

В частности, команда **branch** может быть представлена как $(v_1|branch)$.

Для получения грамматически правильных последовательностей команд необходимо ввести понятие *группы* команд и ввести требования к их корректности. Пусть $g \in G$ — группа команд из множества групп команд, отвечающих конструктивным требованиям корректности. Введем их:

1. Группа может состоять из пустого множества команд:

$$g = \varnothing \tag{4.3}$$

2. Группа может состоять из множества отдельно стоящих команд и групп:

$$g = v*, g*, v* \tag{4.4}$$

3. Каждая команда **branch** должна закрываться командой **end branch**:

$$g = (v_1|branch), g_2, (v_3|end\ branch) \tag{4.5}$$

4. Каждая команда **enter** должна быть продолжена декларацией возвращаемого значения, переменных, и закрываться командой **exit**:

$$\begin{aligned} g = {} & (v_1|enter), \\ & (v_2|declare\ returnVar), \\ & (v_3|declare\ p_1), \\ & \ldots, \\ & g_{n-1}, \\ & (v_n|exit) \end{aligned} \tag{4.6}$$

Грамматически корректная программа (Well-formed formula) состоит из комбинации команд и групп, соответствующих этим требованиям.

4.3.1 Метод обобщения

Алгоритм конверсии в промежуточное представление начинается с процедуры *обобщения*. Идея обобщения (Алгоритм 4.3) заключается в формировании подпрограммы, устойчивой к изменению аннотаций внутренних

вызовов процедур и изменению способа раскрытия вызовов. Значения параметров в формируемых программах остаются неопределенным.

Алгоритм 4.3. Обобщающая трансляция

Последовательность действий:

1. Осуществляется поиск точек входа в программу среди ресурсов-функций. Их выбор может быть обусловлен настройками среды, целевой операционной системы, компилятора и т.п.
2. Для обнаруженных точек входа добавляются пред- и пост-условия как команды $constraint$. В случае аннотаций с описанием поведений в разных ситуациях (именованных поведений) может формироваться подпрограмма с последовательностью $branch$, $constraint$ и $end\ branch$. Пусть c_i — одно из условий именованного поведения, а c_i' — ограничение, следующее из справедливости условия c_i. Тогда формируемая последовательность выглядит следующим образом:

$$\begin{Bmatrix} (v_1^i|branch\ c_i), \\ (v_2^i|constraint\ c_i'), \\ \dots, \\ (v_n^i|end\ branch) \end{Bmatrix} \tag{4.7}$$

3. Начинается обход фрагментов, связанных с найденными точками входа:
   - Формируется команда включения семантики, соответствующей языку l анализируемой процедуры:

$$(v_1|semantics\ l) \tag{4.8}$$

   - Входные условия c преобразуются в соответствующие команды $branch$ и $end\ branch$:

$$(v_1|branch\ c), \dots, (v_n|end\ branch) \tag{4.9}$$

   - Начинается обход выражений фрагмента.
     а) Производится упрощение всех выражений фрагмента до простых. Например, выражение $f()+(5+7)$ упрощается до $var1 = f()$, $var2 = 5 + 7$, $var1 + var2$.
     б) Производится анализ каждого выражения. Как правило, предполагается, что почти все значимые выражения на данном этапе являются бинарными.
     Присваивания вида $a = b$ преобразуются в команду $assign$:

$$(v_1|assign\ a = b)$$

     Задание псевдонима преобразуется в команду $alias$. Условием для этого в большинстве языковых семантик является факт использования указательного или ссылочного типа в левой части

присваивания.

$$(v_1|alias\ a = b)$$

Вызов функции f с параметрами p_1, p_2, …, p_n преобразуются в команду $invoke$:

$$(v_1|invoke\ result = f(p_1, p_2, \ldots, p_n))$$

Все аннотации связанных с вызовом функций обрабатываются согласно правилу преобразования аннотаций.

Таким образом, в результате работы алгоритма создается модель программы в форме промежуточного представления, не раскрывающего вызовы процедур, которые могут быть проанализированы и дополнены тегами позднее.

4.3.2 Метод инстанцирования

Под инстанцированием подразумевается формирование программы, подходящей для межпроцедурного анализа (Алгоритм 4.4). Основным её отличием является подстановка полных тел функций, известных значений переменных или более подробных резюме на место вызовов функций.

Алгоритм 4.4. Инстанцирующая трансляция

Последовательность действий:

1. Для небольших целевых функций (размером меньше 20 строк) осуществляется прямая подстановка функции на место применения ($invoke$).
   а) Все используемые функцией переменные клонируются с обновлением идентификатора, с сохранением аннотаций и ссылок на них.
   б) Команды $return$ подменяются на прямые присваивания переменных на $assign$.
2. Для больших функций происходит подстановка действительных значений в резюме функций. В случае, если целевой метод — виртуальный, виртуальная машина осуществляет запрос в гипервизор с целью предоставления перечня возможных исходов данного вызова. В таком случае, происходит объединение резюме функций посредством логической связки $\wedge$.

После проведения инстанцирования промежуточное представление приобретает свойства, необходимые для проведения детального межпроцедурного анализа.

4.4 Модифицирование и упрощение команд в результате анализа

В ходе анализа возникает необходимость модифицировать команды, чтобы получить более легкую для последующего анализа последовательность. Для этого существуют два процесса: упрощение и переписывание.

Упрощение призвано избавиться от конструкций, представимых более простым способом, путем их трансляции в группу команд. Представим процесс более формально:

$$Simplify : V \to G \tag{4.10}$$

Процесс упрощения работает на уровне выражений, встречающихся в командах, поэтому модифицируется не столько сама команда, сколько входящее в нее выражение.

Приведем примеры модификации:

1. Выделение памяти на этапе конверсии осуществляется через введение фантомных переменных нужного типа. На данном этапе для них нет команды $declare$ и её необходимо добавить.
   Пусть v_1 — команда, содержащая в себе доступ к фантомной переменной var_1. Тогда команда будет переписана следующим образом:
   $$Simplify(v_1) = (v_1''|declare\ phantom), v_1'$$
2. Анализ псевдонимов требует переписывания доступа к переменным в конструкцию $constraint$. Пусть v_1 — команда, содержащая в себе доступ к переменной-псевдониму. Тогда команда будет переписана на использование временной переменной с явным выводом возможных псевдонимов. Пусть переписанная команда будет обозначена как v_1':
   $$Simplify(v_1) = (v_1''|constraint\ al = \ldots), v_1'$$
   Альтернативно, в ограниченном числе случаев корректно переписать такой доступ на последовательность условных присваиваний. Пусть функция $Aliases$ возвращает список псевдонимов для x в формате (A_c, A_e), где A_c — вектор условий, A_e — вектор выражений, причем условие A_{c_i} относится к выражению A_{e_i}:
   $$Aliases(x) = (A_c, A_e)$$

Тогда упрощение такой конструкции можно представить следующим образом:

$$BranchAlias(x, A) = \Big\{(v_{b_i}|branch\ A_{c_1}),$$
$$(v_{b_i}|assign\ x = A_{e_i}),$$
$$(v_{b_i}|end\ branch)\Big\}$$

$$Simplify(v_1) = BranchAlias(x, Aliases(x)), v_1'$$

Альтернативно, на части присваиваний команда $assign$ может быть заменена на $alias$.

3. Корректировка строковых литералов. В зависимости от семантики целевого языка может быть необходимо заменить строковые литералы на ссылки на массивы. Пусть v_1 — команда, содержащая в себе доступ к строковому литералу. Пусть переписанная команда на использование фантомной переменной var команда будет обозначена как v_1'. Тогда последовательность выглядит следующим образом:

$$Simplify(v_1) = (v_1'|declare\ var|), (v_1''|assign\ var = \ldots), v_1' \quad (4.11)$$

4. Конверсии переменных. В случае реинтерпретации переменных необходимо переписать её конверсию на использование $constraint$:

$$Simplify(v_1) = (v_1'|declare\ var|), \Big\{(v_1'''^i|constraint\ldots)\Big\}, v_1' \quad (4.12)$$

4.5 Поддержка различных языков программирования в одном контексте

Предполагается, что к моменту трансляции программы в промежуточное представление обобщенное синтаксическое дерево по большей части лишено конструкций, явным образом зависящих от языковой семантики. Однако языковая семантика может влиять на согласование типов, на семантическое сравнение типов, а также влияет на включение или выключение диагностических правил.

Использование инструкции **semantics** позволяет изменить языковую семантику в конкретной части программы.

Обозначим за X некоторый вектор из команд промежуточного представления, а $l \in L$ — за произвольную языковую семантику из множества языковых семантик L. Тогда $sema(X, l)$ — функция применения семантики l к вектору команд X. Пусть итоговые векторы с семантической информацией обозначаются как X^s, в то время как векторы с семантической информацией для языка l обозначаются как X^{s^l} (назовем их *семантическими векторами*). Отметим, что справедливо следующее соотношение:

$$X^{s^l}, l \in L, X^{s^l} \in X^s \tag{4.13}$$

Следовательно, любая программа $p \in P$ представляет собой набор семантических векторов. Предположим, что в программе n векторов. Тогда $l_i, 1 \leqslant i \leqslant n$ представляет собой произвольный язык из L. Все объединение можно представить следующим образом:

$$p = (X_1^{s^{l_1}}, X_2^{s^{l_2}}, \ldots, X_n^{s^{l_n}}) \tag{4.14}$$

Разберем основные «связки», необходимые при совмещении семантик нескольких программ:

– **Конверсии переменных**. Так как за базовыми типами закреплены низкоуровневые типы конкретной архитектуры ЦП, то в случае одинаковой архитектуры ЦП конверсии между типами не требуются. В противном случае, необходимо провести конверсию типов заранее заданными правилами конверсии базовых типов.
– **Конверсии структур**. В случае, если объекты передаются по указателю, никакой конверсии не требуются[1]. Случай передачи объекта в случае структурно одинаковых типов данных требует пошагового преобразования каждого значимого поля в аналогичное поле целевой структуры данных через формирование команды $constraint$.

Отметим, что анализатор формирует код конверсий автоматически при выполнении программ и выявлении несовпадений типов данных.

[1]Более того, случай сдвига физического местоположения данных будет отслежен анализатором автоматически при пересечении барьера языков за счет приведения типов блоков памяти

Глава 5. Структура и функции гибридного решателя

Программы на языке промежуточного представления содержат достаточно информации для анализа, но они неудобны для поиска ошибок. Для решения проблемы вводится *гибридный решатель*, формирующий специализированный подвид Midair — *аналитическое представление*, в которое записывается информация о потоках данных и управления, информация о псевдонимах, абстрактные и символьные домены значений (предоставляемые *решателем абстрактной интерпретации*). Эта информация впоследствии поступает в *модуль подготовки данных для решателя* (SolverFeeder), транслирующий программу в формат SMT. SMT-решатель может вызываться *вычислителем* при необходимости проверить выполнимость определенных задач. Более подробно взаимосвязи решателей описаны в работах [32; 34].

5.1 Аналитическое представление программы

При анализе программы важно сохранять промежуточные результаты вычислений. Это необходимо для упрощения работы с формой единственного статического присваивания (при таком случае значительная часть SMT-представления может быть получена прямым отображением аналитического представления), для применения функций $prev(x)$, $next(x)$ и $old(x)$ (позволяющих получать предыдущие и следующие версии переменных), а также для уменьшения сложности отдельных выражений.

Введем аналитическое представление программы. Пусть a — одно из аналитических представлений. Тогда A_p — множество аналитических представлений для программы p.

Выразим структуру a. Пусть r — произвольный ресурс из множества ресурсов O^r, а E — произвольное множество ресурсов. Таким образом, a состоит из множества пар «ресурс — множество соответствующих выражений»:

$$a = \{(r, E)\}$$

Следует отметить, что в практическую реализацию также добавляются различные кэши для более быстрого доступа к данным и прочие контекстуальные данные, которые также могут быть представлены парами (r, E), если для их хранения добавить фантомные ресурсы.

Аналитическое представление существует в каждой точке программы, то есть оно представлено в виде древовидной иерархической структуры, в которой на каждом уровне любое последующее представление выполняется после предыдущего. Также каждое вложенное представление (то есть являющееся иерархически ниже в иерархии по сравнению с текущим представлением) является более ранним. Такая форма организация удобна для процесса упрощения программ, так как эффект от команды явно ассоциирован с самой командой.

Представим данные факты об аналитическом представлении более формально. Пусть $t : A \to N$ — функция получения условного монотонного времени, когда эффект применяется. Пусть $sub(a)$ — функция получения множества аналитических представлений, являющихся дочерними для представления a. Тогда:

$$\forall a' \in sub(a), t(a') < t(a)$$

Для полноценной работы следующих этапов требуется ввести дополнительно выражение, переключающее фокус вычислителя на другое аналитическое представление. Пусть $e \in E$ — произвольное выражение. Тогда $Located(e, a) \in E$ (Приложение Д) — значение данного выражения относительно представления a. В дальнейшем все выражения, зависящие от выбора представления, будут неявно записываться через функцию $Located$. Это существенно расширяет возможности по анализу выражений и вычислению их значений.

5.2 Моделирование памяти

Для аналитического представления особый интерес представляет информация о потоках данных и управления. Её можно получить с помощью

методики моделирования памяти. Память в анализаторе *разбита на регионы*, *версионирована*, *реинтерпретируема* и *поддерживает анализ псевдонимов*.

5.2.1 Версионирование переменных

Изначально объекты в анализаторе не имеют версии. Учет изменений переменных производится анализатором потока данных по описанному далее принципу. Введем понятия *переменной*, *связанной переменной*, *ревизии* и *контрольной точки*.

Определение 5.1. Переменная — именованное хранилище значения.

Определение 5.2. Связанная переменная — переменная, связанная с основной переменной и выявленная анализом псевдонимов.

Определение 5.3. Ревизия — количество изменений переменной с последней контрольной точки.

Определение 5.4. Контрольная точка — количество объединений ревизий переменной при ветвлениях.

Изменение переменной может генерировать изменения нескольких связанных переменных, при этом только одна из них может быть реально изменена вследствие принципа детерминированности последовательного исполнения.

Опишем процесс более формально.

Пусть $v \in V$ — произвольная переменная из множества переменных. Тогда v_{0_0} — её начальная версия, v_{c_r} — версия переменной с контрольной точкой c и ревизией r.

Тогда процесс работы с переменными можно представить следующим образом:

1. Присваивание нового значения для переменной v_{c_r} порождает новую переменную $v_{c_{r+1}}$:

$$v_{c_{r+1}} = value \tag{5.1}$$

2. Рассмотрим обработку цепочки `branch - end branch`. Предположим, что изначально переменная имеет вид v_{c_r}. Пусть в каждой ветке этой цепочки рекурсивно порождается переменная $v_{c_{ir_i}}$, где $c_i \geqslant c, c_i = c \implies r_i \geqslant r$. Тогда выход из цепочки `branch - end branch` с присваиванием нового значения переменной v_{c_r} (один или несколько раз) внутри цепочки порождает новую переменную $v_{c'_0}$, где $c' = max(c_i) + 1$.

Теперь рассмотрим работу со связанными переменными (псевдонимами). Пусть V' – множество связанных переменных, тогда $B(V) = V'$ — отображение множества переменных во множество связанных переменных. Тогда для любой переменной v существует некоторое множество V'_v, которое состоит из связанных с ней переменных. Пусть для связывания каждой переменной v_i (где i — номер связанной переменной) с v существует условие $e_i \in E$. Тогда в процессе работы любое присваивание переменной формирует присваивания всех связанных переменных, удовлетворяющих условиям:

$$True \implies v_{c_{r+1}} = value \tag{5.2}$$

$$\begin{aligned} e_1 &\implies v^1_{c_{1r_1+1}} = value \qquad \neg e_1 \implies v^1_{c_{1r_1+1}} = v^1_{c_{1r_1}} \\ e_2 &\implies v^2_{c_{2r_2+1}} = value \qquad \neg e_2 \implies v^2_{c_{2r_2+1}} = v^2_{c_{2r_2}} \\ &\dots \\ e_n &\implies v^n_{c_{nr_n+1}} = value \qquad \neg e_n \implies v^n_{c_{nr_n+1}} = v^n_{c_{nr_n}} \end{aligned}$$

Таким образом, невыполнение условия связывания не меняет переменную, но порождает новую версию, эквивалентную предыдущей.

Формализуем алгебру ревизий и контрольных точек. Пусть r — ревизия переменной, а c — её контрольная точка. Тогда сложение пар (r, c) в случае последовательного обхода осуществляется по следующему правилу:

$$(c_1, r_1) + (c_2, r_2) = (c_1 + c_2, r_1 + r_2) \tag{5.3}$$

В случае сложения ревизий различных ветвей, сложение осуществляется по следующему правилу (применим скобку Айверсона для удобства написания):

$$(c_1, r_1) + (c_2, r_2) = (c_1 + c_2 + [(r_1 + r_2) \geqslant 0], [(c_1 + c_2) = 0] \cdot (r_1 + r_2)) \tag{5.4}$$

Таким образом, всякое изменение переменной увеличивает версии самой переменной и всех связанных с ней переменных в инкрементальной (свойство *инкрементальности*) и аккумуляционной (свойство *аккумуляции*) карте использования переменных не более, чем на одну версию. По окончанию ветвления аккумуляционная версия переменной увеличивается на суммарную инкрементальную версию внутренних блоков, устанавливается новая контрольная точка.

5.2.2 Разбиение на регионы

Для любой реалистичной архитектуры компьютера характерна модель цельной памяти, когда все объекты в программе хранятся в единственном регионе (архитектура Фон Неймана [168]). Однако хранение информации обо всем объеме памяти, свойствах её виртуального строения (в случае служебных областей, векторов прерываний и т.п.) практически невозможно вследствие ограничений современных компьютеров и применяемых технологий защиты данных, таких как ASLR [169].

Следовательно, так или иначе разбиение на непересекающиеся регионы памяти необходимо [170; 171].

Пусть $\mathbb{M}$ — множество регионов памяти. Тогда $m \in \mathbb{M}$ – произвольный регион памяти.

Пусть $\mathbb{I}$ — множество диапазонов адресов. Тогда $i \in \mathbb{I}$ — произвольный диапазон адресов.

Пусть $Addr(m) : \mathbb{M} \to \mathbb{I}$ — отображение из множества регионов памяти в множество диапазонов адресов. Тогда $Mem(i) : \mathbb{I} \to \mathbb{M}$ — отображение из множества диапазонов адресов во множество регионов памяти.

Тогда для предлагаемой модели справедливо следующее:

$$\forall m \in \mathbb{M}, m' \in \mathbb{M}, Addr(m) \cap Addr(m') = \varnothing \tag{5.5}$$

Такая модель минимизирует расходы на проверку памяти в связи с отсутствием необходимости моделировать несуществующие в конкретной точке программы регионы.

Модель не закрывает возможность использовать отдельные цельные регионы, как, например, стек программы в конкретной точке. Такое моделирование стека позволяет эффективно обнаруживать повреждения памяти [172].

Каждый регион памяти обладает следующими свойствами:

– **Размер региона**, который может задаваться как интервальным численным значением, так и символьной переменной. Пусть размер региона задается отображением $size(m) : m \to Z$. Будем обозначать размер отдельного региона как $size$. Размер переменной хранится символьно как вектор возможных значений-выражений. Корректный размер выбирается с использованием оператора «исключающее ИЛИ». При этом если телом выражения является выражение со встроенным указателем на место в коде, такое выражение переписывается как импликация с применением оператора «И» с побочным эффектом с применением оператора «исключающее ИЛИ» к условиям. Одновременно могут быть либо выражения без указателя на место в коде, либо выражения с его указанием.
– **Набор значений в конкретных субдиапазонах адресов**. Каждое значение обладает семантическим смыслом: значение, псевдоним или ссылка на другой блок памяти. Значения хранятся в карте, индексированной диапазонами (символьными или прямыми).

  Пусть субдиапазоны обозначаются как $a \in A$, где A — множество субдиапазонов, а a — отдельный субдиапазон. Тогда для конкретного региона m будет справедливо: $0 \leqslant min(a) \leqslant max(a) \leqslant size(m)$.

  Отдельно оговоримся, что доступ к выражениям региона памяти осуществляется через $m[a]$, а добавление и присваивание возможно через следующие конструкции:

$$m[a] = x \quad m[a] += x$$

  Пусть наборы значений обозначается как $values$.
– **Тип памяти**: статическая память, стек, куча и т.п. Введем отображение $type(m) : M \to P_{type}$. Для конкретного региона памяти будем обозначать тип памяти как $type$.

– **Теги памяти**: выделена, очищена и т.п. Введем отображение $tag(m) : M \rightarrow P_{tag}$. Для отдельного региона памяти будем обозначать поле с тегами как tag.

Отдельный регион памяти может быть представлен набором следующего вида:

$$m = (size, values, type, tag) \tag{5.6}$$

Важно подчеркнуть, что регион памяти не является типизированным. При приведении указательных типов блок данных остается неизменным, что позволяет сохранить ассоциированную с ними информацию (provenance [173]).

Доступ к структурным объектам возможен как к отдельным полям, так и как к массивам байтов, но данные хранятся с разбиением по полям. Вычисление пересечения агрегированного состояния для значений субдиапазонов позволяет достичь *реинтерпретируемости* регионов памяти.

5.2.3 Анализ псевдонимов

Предполагается, что разные объекты могут указывать на одну память, тем самым порождая необходимость анализа псевдонимов. Анализ псевдонимов реализуется аналогично классическому алгоритму **Points-To** [174], который предписывает формировать перечни псевдонимов для каждого из объектов, однако в реализованном варианте есть важные отличия, касающиеся способа хранения информации о псевдонимах в регионах памяти.

Рассмотрим произвольную переменную x типа $uint64\_t$. Её регион памяти состоит из 8 байтов. При этом, если объявить переменную псевдонимом переменной y, то в субдиапазон $[0, 8]$ будет добавлен псевдоним на данную область памяти. Таким образом, случай наложения псевдонима на весь объект не представляется сложным для разбора.

Рассмотрим структурную переменную s, состоящую из двух полей $first : uint32\_t$, $second : uint64\_t$. В таком случае псевдоним может быть наложен на конкретную часть объекта ($first$ или $second$) путем добавления

соответствующего этой части субдиапазона со значением $alias$, указывающим на целевой объект.

Обработка псевдонимов осуществляется по алгоритму 5.1, в то время как доступ к переменным, имеющим в качестве значения псевдонимы, осуществляется при помощи алгоритма 5.2.

Алгоритм 5.1. Обработка присвоения псевдонима в терминах Midair и аналитического представления

Последовательность действий:

1. Проводится разбор команды `alias [object] = expression` и сопоставление object с выражением или целевым ресурсом.
2. Для целевого объекта, заданного выражением или целевым ресурсом, выбирается существующий или создается новый регион памяти подходящего размера. Пусть это будет $m_1 \in M$.
3. Определяется целевой субдиапазон $a_1 \in A$, который может задаваться либо прямым доступом к объекту, либо структурным доступом, либо доступ к элементу массива (существуют и другие варианты, но они, в основном, являются производными этих вариантов).
4. К целевому выражению добавляется информация о месте в коде, где данное выражение задается. Для этого целевое выражение x преобразуется к выражению $x' = Located(l_x, x)$, где $location$ — внутренний по отношению к реализации анализа объект, указывающий на точку в программе.
5. Осуществляется добавление псевдонима x' к найденному субдиапазону:

$$m_1[a_1] = x'$$

Алгоритм 5.2. Доступ к переменным, имеющим в качестве значения псевдоним, в терминах Midair и аналитического представления

Последовательность действий:

1. Предположим, что осуществляется доступ к объекту o, имеющему псевдонимы $x'_1, \ldots, x'_n$, где n — количество псевдонимов, а x' — формализм описания псевдонимов из алгоритма 5.1.
2. Выполняется оценка выполнимости условий псевдонимов: каждому из них присваивается 0, 1 или $\top$ (неопределенное значение). Пусть такое значение задается как c. Тогда получаем подобное множество:

$$(x'_1, c_1), \ldots, (x'_n, c_n)$$

3. Из множества псевдонимов $x'_1, \ldots, x'_n$ фильтруются те, у которых оценка c — 1 или $\top$. Пусть новое количество задается как m, тогда все множество может выглядеть следующим образом:

$$x''_1, \ldots, x''_m$$

4. Пусть у каждого псевдонима x_i'' есть условие $c_{x_i''}$. Тогда доступ к объекту, содержащему ссылку, переписывается на использование переменной r следующим образом:

$$c_{x_1''} \implies r = x_1'' \quad \dots \quad c_{x_m''} \implies r = x_m''$$
$$c_{x_1''} \oplus c_{x_2''} \dots \oplus c_{x_m''}$$

Сам доступ описывается через обращение к переменной r.

На практике при вычислении значений выражений с псевдонимами выполняются дополнительные оптимизации согласно алгоритмам 5.3 и 5.4.

Алгоритм 5.3. Оптимизация вычисления псевдонимов с равным значением в случае полного покрытия пространства состояния
Последовательность действий:

Если выражение содержит несколько условных псевдонимов с одинаковым значением, то проверка корректности псевдонимов не осуществляется при условии полноты заполнения пространства состояния.

Замечание 5.1. Это означает, что для выполнения данной оптимизации условия псевдонимов должны покрывать все возможные состояния объекта.

Алгоритм 5.4. Оптимизация единственного псевдонима
Последовательность действий:

Если выражение содержит единственный псевдоним, и семантика языка разрешает единственную инициализацию псевдонима, то можно считать этот псевдоним целью.

Замечание 5.2. Это позволяет оптимизировать случай применения ссылочных типов, которые, например, присутствуют в языке C++.

Алгоритм 5.5. Обработка вызовов функций
Последовательность действий:

Вызовы функций оборачивается в конструкции $invoke$, которые раскрываются непосредственно в момент анализа, что позволяет подменить результат применения резюме вызываемой функции в ходе дальнейшего анализа, обеспечивая бесшовный переход к более точной межпроцедурной модели. Это работает и для псевдонимов, делая анализ контекстно-чувствительным.

Реализованный метод обладает следующими свойствами:

– **Потоко-чувствительность**. Псевдонимы анализируются по ходу анализа функций, и наборы возможных целей различаются в разных точках.
– **Контекстно-чувствительность**. Набор целей может зависеть от вызываемых функций и их резюме.
– **Чувствительность к полям**. Псевдонимы устанавливаются на часть объекта, а не на весь объект, что позволяет реализовать свойства вне рамок существующих решений, применяемых в императивных языках программирования.

5.2.4 Инкрементальные и кумулятивные модели

В каждой точке программы строятся *инкрементальные* и *кумулятивные* модели.

Определение 5.5. Инкрементальная модель — аналитическое представление программы, содержащее образ изменений, выполняемых в рассматриваемой точке программы.

Определение 5.6. Кумулятивная модель — аналитическое представление программы, содержащее образ значений, псевдонимов переменных и т.п., накопленных с начала программы.

Замечание 5.3. Данные кумулятивной модели позволяют рассчитать значение каждой переменной.

Замечание 5.4. Кумулятивная модель строится через последовательное наложение инкрементальных моделей.

Для выполнения указанного наложения инкрементальных моделей требуется два алгоритма: один для случая последовательного наложения (5.6), другой для случая «параллельного» наложения (5.7), то есть когда происходит ветвление.

Алгоритм 5.6. Последовательное наложение инкрементальной модели региона памяти
Последовательность действий:

1. Рассмотрим случай с *модифицируемым* и *целевым* регионами памяти. На модифицируемый регион происходит наложение, а данные целевого используются как аргументы при наложении.
2. В случае, если целевой регион памяти содержит слой со значениями переменных, значения переменных полностью заменяет оные в модифицируемом регионе.
3. В случае, если целевой слой содержит карту применения переменных, она объединяется с картой модифицируемого слоя.
4. Если целевой слой содержит теги, они последовательно переносятся на теги модифицируемого региона.

Алгоритм 5.7. Параллельное наложение инкрементальной модели региона памяти

Последовательность действий:

1. Рассмотрим случай с *модифицируемым* и *целевым* регионами памяти. На модифицируемый регион происходит наложение, а данные целевого используются как аргументы при наложении.
2. В случае, если целевой регион памяти содержит слой со значениями переменных, значения переменных добавляются в модифицируемый регион.
3. В случае, если целевой слой содержит карту применения переменных, она объединяется с картой модифицируемого слоя.
4. Если целевой слой содержит теги, они параллельно переносятся на теги модифицируемого региона.

Инкрементальная модель функции является прообразом её **резюме**, которое отличается от неё лишь дополнительной систематизацией и упрощением свойств.

5.2.5 Свойства и их перенос

Одним из элементов аналитического представления является набор *тегов* — ассоциативный массив с данными о наличии определенного свойства у объекта. Каждый тег представляет собой следующую структуру в форме Бэкуса-Науэра:

$$Tag ::= Unknown \mid Set \mid NotSet \mid MayBe$$

Примерами тегов являются свойства «память выделена», «мьютекс заблокирован» и т.п.

Основным преимуществом тега является его простота. В большинстве случаев работа с ним происходит через переаппроксимацию пространства состояний и потому крайне эффективна, что позволяет находить значительное число ошибок, не выполняя никаких проверок через SMT-решатель.

Для функционирования тегов необходимо осуществлять перенос свойств между объектами и псевдонимами. Это позволяет сохранить свойства при передаче данных различного характера.

Рассмотрим случай с *модифицируемым* и *целевым* регионом памяти. Модифицируемый регион памяти — регион, на который переносятся теги. Целевой регион памяти — регион, с которого переносятся теги.

Существует три основных режима объединения тегов:

Последовательный режим: свойства целевого региона памяти перекрывают свойства модифицируемого региона по алгоритму 5.8.

Алгоритм 5.8. Последовательный перенос тегов

Последовательность действий:

1. Рассмотрим модифицируемый (относящийся к модифицируемому региону памяти) и целевой (относящийся к целевому региону памяти) теги.
2. За исключением случая $Unknown$, целевой тег переносится из целевого в модифицируемый регион без изменений. В случае $Unknown$, модифицируемый тег остается без изменений.

Параллельный режим: происходит объединение свойств регионов памяти, относящихся к разным ветвям, по алгоритму 5.9.

Алгоритм 5.9. Параллельный перенос тегов

Последовательность действий:

1. Рассмотрим модифицируемый (относящийся к модифицируемому региону памяти) и целевой (относящийся к целевому региону памяти) теги.
2. Модифицируемый тег $Unknown$ остается таковым, если целевой тег тоже $Unknown$. В противном случае он преобразуется в $MayBe$.
3. Модифицируемый тег Set остается таковым, если целевой тег $Unknown$ или Set. В противном случае он преобразуется в $MayBe$.
4. Модифицируемый тег $NotSet$ остается таковым, если целевой тег $Unknown$ или $NotSet$. В противном случае он преобразуется в $MayBe$.
5. Модифицируемый тег $MayBe$ остается таковым во всех вариантах целевого тега, кроме $Unknown$. В этом случае он будет преобразован в $Unknown$.

Дополняющий режим: добавляются свойства, относящиеся к изначальному кумулятивному состоянию модифицируемого региона до параллельного переноса, по алгоритму 5.10.

Алгоритм 5.10. Дополняющий перенос тегов

Последовательность действий:

1. Рассмотрим модифицируемый (относящийся к модифицируемому региону памяти) и целевой (относящийся к целевому региону памяти) теги.
2. Модифицируемый тег $Unknown$ остается таковым, если целевой тег тоже $Unknown$. В противном случае он преобразуется в $MayBe$.
3. Модифицируемый тег Set остается таковым, если целевой тег $Unknown$ или Set. В противном случае он преобразуется в $MayBe$.
4. Модифицируемый тег $NotSet$ остается таковым, если целевой тег $Unknown$ или $NotSet$. В противном случае он преобразуется в $MayBe$.
5. Модифицируемый тег $MayBe$ остается таковым во всех вариантах целевого тега, кроме $Unknown$. В этом случае он будет преобразован в $Unknown$.

Все три алгоритма используются при обработке ветвлений (Приложение М). В этом случае на кумулятивную модель региона памяти последовательно накладывается инкрементальная модель ветви, результат параллельно объединяется с другими ветвями (в примере их всего две), и результат параллельного переноса дополняется первоначальной кумулятивной моделью модифицируемого региона.

5.3 Метод трансформации программ в SMT-представление

Переменные на языке промежуточного представления модифицируются множество раз, что является препятствием для конверсии в SMT-представление, требующее формы единственного статического присваивания. Однако его применение усложняет применение диагностических правил. Поэтому необходимо сохранять обе формы — *обычного* и *объектного* присваивания.

Определение 5.7. Форма обычного присваивания — форма программы, в которой присваивания могут осуществляться в произвольные выражения.

Рассмотрим в качестве левой части обычного присваивания следующие типы выражений:

1. **Переменные**. Фактически, такой вариант равноценен далее рассматриваемой форме объектного присваивания.

2. Выражения с **доступом к членам структурных объектов**.
3. Выражения с **доступом к элементам массива по произвольному индексу**.
4. **Префиксно-унарные выражения**. В частности, $assign\ *x = 5$ — присваивание переменной по адресу x. Особое внимание требуется уделить выражениям взятия адреса переменной и разыменования.
5. **Бинарные выражения**, в частности, те, которые по итогу имеют свойства $lvalue$. Как пример, пусть и в составе префиксно-унарного выражения: $assign\ *(x + 1) = 5$.
6. **Комбинированные выражения** (п. 1–5).

Будем считать данные варианты *ограничениями присваивания*.

Определение 5.8. Форма объектного присваивания — форма программы, при которой все присваивания осуществляются непосредственно в объекты.

В отличие от обычного присваивания, где могут быть команды вроде $assign\ x[5] = 6$, в форме объектного присваивания существуют только команды $assign\ x = b$, где b — произвольное выражение.

Теорема 5.1. Любое присваивание с левой частью в заданных ограничениях представимо в форме объектного присваивания.

Доказательство. Рассмотрим способы трансляции указанных выражений в форму объектного присваивания.

1. Трансляция левых частей — переменных не требуется, так как такие левые части уже в форме объектного присваивания.
2. Выражения с доступом к членам структурных объектов вида $x.a = b$ требуют ввода нового оператора $with$: $x = x\ with\ .a = b$.
3. Выражения с доступом к элементам массива по произвольному индексу вида $x[a] = b$ требуют ввода нового оператора $with$: $x = x\ with\ [a] = b$.
4. Префиксно-унарные выражения имеют различия в реализации для разных операторов. Рассмотрим операторы взятия адреса переменной и разыменования.
    а) Оператор взятия адреса переменной формирует новую переменную, являющуюся контейнером адреса:

$$assign\ \&x = b$$

Данное выражение эквивалентно следующей последовательности:

$$assign\ newvar = \&x$$
$$assign\ newvar = b$$

Эта операция выглядит малоосмысленной, но при этом она может использоваться в комплексных выражениях, где смысл, безусловно, присутствует.

б) Оператор разыменования переменной наиболее сложен, так как его обработка требует помощи анализа псевдонимов.
Рассмотрим следующую команду:

$$assign\ *x = b$$

Введем соотношение $alias(a) = V$, где a — произвольная переменная, а V — множество фактических псевдонимов этой переменной.
Пусть $v_1, ..., v_n$ — переменные из этого перечня. Тогда операция разыменования транслируется в последовательность:

$$assign\ v_1 = b \quad assign\ v_2 = b \quad assign\ v_3 = b$$
$$...$$
$$assign\ v_n = b$$

5. Трансляция бинарных выражений осуществляется тривиально.
Рассмотрим выражение:

$$assign\ *(x + 1) = 5$$

Избавим его от префиксно-унарной части, которая хоть и формирует смысл выражения, но не имеет решающего значения для данного доказательства, так как рассматривается отдельно:

$$assign\ x + 1 = b$$

Тогда преобразование будет иметь вид:

$$assign\ newvar = x + 1$$
$$assign\ newvar = b$$

Если вернуть префиксно-унарную часть, то можно отметить, что выражение остается тем же:

$$assign\ newvar = x + 1$$
$$assign\ (*newvar) = b$$

Это выражение по п. 4 может быть переписано через последовательность присваиваний псевдонимов.

Таким образом, для основных групп выражение присваивание представимо в объектной форме. □

Объектная форма присваивания значительно упрощает конверсии программ в SMT-представление.

Рассмотрим трансляцию остальных типов команд.

Алгоритм 5.11. Трансляция промежуточного представления в SMT-представление.

Последовательность действий:

1. Осуществляется обход программы с построением дерева промежуточного представления. Данный этап отличается от Control Flow Graph тем, что в графе потока управления применяются простые выражения, а в дереве программмы — разобранные выражения.
    а) Конструкции `branch` генерируют ветви, в том во время как `end branch` их завершают.
    б) Остальные конструкции перекладываются в дерево, как есть.
2. Осуществляется обход дерева программы с целью предварительной обработки семантики.
    а) Конструкции `alias` формируют в блоке памяти целевого ресурса (вычисленного при помощи алгоритма для объектного присваивания) ссылки на пространственно-чувствительные выражения из правой части.
    б) Конструкции `assign` формируют в блоке памяти целевого ресурса (вычисленного при помощи алгоритма для объектного присваивания) ссылки на значения в терминах в пространственно-чувствительных выражений.
    в) Выражения, содержащие ссылки на псевдонимы, транслируются в более сложные конструкции с элементом выбора (Листинг 5.1).

Листинг 5.1: Пример преобразования псевдонимов

```
...
branch a = 5 start or continue
    alias z = x
end branch
```

```
branch a != 5
    alias z = y
end branch
...
constraint z = 5
```

Выражение `constraint` преобразуется в серию конструкций (Листинг 5.2).

Листинг 5.2: Пример преобразования constraint

```
declare z1
constraint (a = 5 => z1 = 6) ^ (a != 5 => z1 = y)
constraint z1 = 5
```

г) Для всех ветвей строится инкрементальная и кумулятивная карты использования объектов.

3. Производится обход дерева программы с прикрепленными данными предварительной обработки с целью получения формул в формате SMT.
   а) Формулы строятся с учетом их местоположения в программе и условий их применения. Например, фрагмент из Листинга 5.3 может быть транслирован во фрагмент из Листинга 5.4.

Листинг 5.3: Пример преобразования constraint

```
branch a = 5 start or continue
assign b = 7
end branch
```

может быть транслирован в:

Листинг 5.4: Пример преобразования constraint

```
a_1 == 5 => b_2 = 7
```

   б) По окончанию серии ветвлений происходит операция *самоприсваивания* (вариация операции объединения из SSA), при которой контрольной версии переменной присваивается значения каждой из веток.
4. В список формул добавляются транслированные в SMT версии проверок свойств.

5.4 Обнаружение ошибок при помощи SMT-решателя

Некоторые ошибки невозможно обнаружить без помощи SMT-решателя. Один из примеров — проверка контрактов, то есть инвариантов, выполнение которых критически для программы.

В таком случае, программа преобразуется в SMT-представление в соответствии с методом из п. 5.3. Так как значительное число выражений в кодах программ тривиальны, то непосредственный вызов SMT-решателя производится вычислителем только в случае необходимости, которая определяется способностью вычислителя определить значение условия в коде. Это является значительным расхождением с алгоритмом из [32; 34].

Проверка справедливости условий производится через проверку формулы, обратной условию. В случае её выполнимости считается, что условие нарушается.

5.5 Обнаружение ошибок с помощью вычислителя

Для обнаружения значительного числа ошибок достаточно обхода команд виртуальной машины с вычислением значений соответствующих им выражений и последующим сравнением с эталоном. Для этого может использоваться алгоритм 5.12.

Алгоритм 5.12. Поиск тривиальных ошибок

Последовательность действий:

1. Осуществляется обход модели программы, составленной из команд виртуальной машины:

$$v_1, v_2, \ldots, v_n$$

2. Для каждой точки программы выполняется обход соответствующих команд виртуальной машины и выражений, с проверкой значений у интересующих выражений. Пусть $Expr(v)$ (где $v \in V$ — одиночная команда) — функция получения выражения из команды, а $Check(e)$ (где $e \in E$ — одиночное выражение) — функция рекурсивной тривиальной статической проверки выражения. Тогда применение функции проверки к каждой команде будет выглядеть следующим образом:

$$\forall v_i \in V, Check(Expr(v_i))$$

3. Выражение направляется в простой вычислитель.
   а) Вычислитель осуществляет обход выражений с вычислением известных частей через функцию $Evaluate(e)$. $Evaluate$ возвращает либо точное значение, либо неопределенное значение. Обозначим за $\odot$ произвольный бинарный оператор, за α — произвольный унарный оператор.

Пусть $Left$, $Right$, $Subexpr$, $Literal$ возвращают соответствующие части выражения. Следовательно, общий принцип работы можно выразить следующим образом.

$$binop(e_1, e_2) = e_1 \odot e_2 \tag{5.7}$$

$$unaryop(e_1) = \alpha e_1 \tag{5.8}$$

$$\begin{aligned} Evaluate(e) ::= Conditional(e) \mid binop(Left(e), Right(e)) \mid \\ unaryop(Subexpr(e)) \mid Literal(e) \mid \\ Unknown \end{aligned} \tag{5.9}$$

Если задать предикат $IsDeterminate$, определяющий детерминированность формулы, и предположить, что выражение e имеет подвыражения $e_1, \ldots, e_n$, то можно составить следующее общее правило вывода детерминированности:

$$\frac{IsDeterminate(e_1), \ldots, IsDeterminate(e_n)}{IsDeterminate(e)} \tag{5.10}$$

Выделим классы эквивалентности среди условных выражений. Пусть (e', c) — пара из выражения и условия его применения. Тогда классом эквивалентности назовем такую пару (e', C), которая состоит из выражения и множества связанных с ним условий. Пара является определенной, если для неё справедливо следующее условие:

$$\frac{IsDeterminate(c_1), \ldots, IsDeterminate(c_n)}{IsDeterminate((e', C))} \tag{5.11}$$

Отметим, что каждое выражение e, которое вычисляется данным алгоритмом, может иметь несколько классов эквивалентности:

$$EquivalencyClasses(e) = \{(e'_1, C_1), \ldots, (e'_n, C_n)\} \tag{5.12}$$

Положим для простоты, что $EquivalencyClasses(e) = K$, $IsDet = IsDeterminate$. Тогда результат вычисления значения всего выражения по этой паре может быть определенным только при условии, что хотя бы одно условие является истинным, при этом другие классы эквивалентности не являются определенными.

$$\frac{IsDet((e', C)), (\forall k \in K, k \neq (e', C),\ \neg IsDet(k))}{IsDet(e, (e', C), K)} \tag{5.13}$$

Следовательно, в случае детерминированности указанного класса эквивалентности несложно вычислить все выражение, положив, что его результат равен e' указанного класса. Тогда значение функции $Conditional$ с учетом указанного вычисления будет таким:

$$Conditional(e) = e' \tag{5.14}$$

б) Если выражение имеет точную оценку, то результат выражения проверяется функцией $Check(Evaluate(e)))$, то есть $Check(Evaluate(Expr(v_i)))$.

Следует учесть, что такой алгоритм подходит для вычисления явно заданных выражений вроде $5 + i$, где $i = 7$. Если выражение задается в цикле или требует доказательства корректности, то требуется вариант с автоматическим применением SMT-решателя.

Случай абстрактной интерпретации детально описан в работе [34]. В реализации описываемого проекта он является неотъемлемой частью вычислителя, то есть все вычисляемые значения принадлежат какому-либо домену, подвергающемуся операции объединения после ветвлений и операциям *расширения* (widening) и *сужения* (narrowing) [125] после циклов. Выбор подходящего алгоритма осуществляется при помощи метода, похожего по своей сути на CEGAR [175].

5.6 Ошибки типов

Ошибки типов наиболее просты в поиске, но между тем требуют сравнительно объемного математического аппарата для их проверки.

У каждого выражения может быть встроенный тип, содержащийся в самом выражении, и тогда должен применяться алгоритм 5.13.

Алгоритм 5.13. Определение квалифицированного типа выражения, содержащего в себе указание типа
Последовательность действий:

Пусть функция $Type$ возвращает квалифицированный тип, содержащийся в самом выражении:

$$Type(e) = t, e \in E, t \in O^t$$

Тогда результатом определения квалифицированного типа является результат применения функции $Type$ к выражению e: $Type(e)$.

Другая ситуация складывается, если выражение не содержит в себе тип и тот должен быть выведен через типы подвыражений с помощью алгоритма 5.14.

Алгоритм 5.14. Определение типа выражения через выведение типа

Последовательность действий:

1. Пусть $InferType$ — функция выведения типа. Тогда итоговое выражение выглядит следующим образом:

$$InferType(e) = t, e \in E, t \in O^t$$

2. Определим тип выражения.
   а) Если выражение содержит в себе тип, то применяется функция $Type$:

$$ContainsType(e) \implies InferType(e) = Type(e)$$

   б) Если выражение не содержит в себе тип, то тип определяется с учетом типов все подвыражений. Введем для этого функцию $NegotiateType(e, e_1, e_2, \dots e_n)$.

$$\neg ContainsType(e) \implies$$
$$InferType(e) = NegotiateType(e, e_1, e_2, \dots, e_n)$$

   Вид функции $NegotiateType$ сложно формализовать, так как он зависит от семантики целевого языка.

Тогда с помощью алгоритмов 5.13 и 5.14 можно обнаруживать ошибки соответствия типов (алгоритм 5.15):

Алгоритм 5.15. Поиск ошибок типов

Последовательность действий:

1. Осуществляется обход модели программы.
2. Выполняется обход соответствующей команды $v_i \in V$ и подвыражений её выражения, задаваемого через $Expr(v_i)$. Пусть $Subexprs(e) = \{e_1, e_2, \dots, e_n\}$.
3. Для каждого подвыражения определяется тип с помощью функции $InferType$.
4. Выполняется зависимое от семантики целевого языка сравнение типов подвыражений между собой или с заранее определенным эталоном.
5. В случае несоответствия типов подвыражений требованиям конкретной языковой семантики, необходимо выдать ошибку.

Пример 5.1. Классический пример на проверку типов выражений — работа с масками `scanf`, `printf`. В таком случае тип подвыражения будет зависеть от маски выражения.

Глава 6. Интеграции с языковыми инструментами

6.1 Контракты в языке РуСи

Контракты в языке РуСи представляют собой записанные в тексте программы инварианты, исполнение которых проверяется при компиляции.

Перед постановкой требований к контрактам, автор провел открытый опрос разработчиков в сети «Интернет» с целью выяснения основных затруднений при работе с контрактами в других языках программирования. Основной аудиторией являлись профессиональные программисты. В опросе участвовали члены команды разработки языка C#, разрабатывавшие решение по контрактам, начиная с 2010 года. В итоге по результатам опроса, были определены следующие требования к контрактам: *синхронизируемость* (аннотации не должны «устаревать»), *привязка к документации*, присутствие *валидаторов предусловий и постусловий*.

Компилятор РуСи занимается чтением аннотационной секции каждой функции самостоятельно и заполняет соответствующую секцию таблицы аннотаций. Анализатор читает ассоциированную с функцией секцию таблицы и применяет к ней автоматически сгенерированный парсер грамматики контрактов[1]. Альтернативным языком является ACSL[2].

Грамматика контрактов основывается на конструкциях **require**, **ensure**, **assume**, **assign**, которые составляют тело контракта. Формально тело контракта можно описать следующим образом:

$$\begin{aligned} any\_clause ::= & \; require\_clause \mid assign\_clause \mid \\ & ensure\_clause \mid assume\_clause \\ contract\_body ::= & \; any\_clause+ \end{aligned} \tag{6.1}$$

Также в языке поддерживаются именованные поведения, необходимые для разделения различных случаев работы функции. Их грамматика триви-

[1] https://github.com/maximmenshikov/contract-grammar

[2] https://github.com/maximmenshikov/acsl-grammar

альна:

$$named\_behavior ::= \text{`}behavior\text{`}\ id\ \text{`}:\text{`}\ contract\_body \tag{6.2}$$

$$named\_behaviors ::= named\_behavior+ \tag{6.3}$$

Контракт функции состоит либо из тела контракта, либо из набора именованных поведений:

$$function\_contract ::= contract\_body \mid named\_behaviors \tag{6.4}$$

Компилятор РуСи производит начальную синтаксическую проверку исходного кода, и в случае отсутствия явных ошибок направляет данные рабочей среды в статический анализатор.

Этот процесс осуществляется посредством упрощенного API библиотеки **libasp** [165], созданного специально для интеграций с внешними (локальными) инструментами. Статический анализатор получает данные рабочей среды и производит повторное чтение с построением внутренних аналитических моделей. Вывод о выполнимости контракта делается с учетом всей информации о программной среде.

Результат проведения анализа присылается в компилятор РуСи и направляется во внутренний диагностический движок, где и происходит сопоставление ошибки с её местом в коде.

6.2 Проверка комментариев Doxygen

Для документирования кода часто используется проект Doxygen [113]. Он позволяет оформить специальные комментарии к коду, которые впоследствии могут быть преобразованы в электронную или печатную документацию в форматах PDF [176], TeX [177] и т.п. Пример комментария приведен в листинге 6.1.

Листинг 6.1: Пример Doxygen-комментария

```
/**
 * Calculate function result based on provided exponent @c e
 *
 * @param[out] result   Pointer to a variable receiving invocation
   result
```

```
 * @param[in]  e        The exponent that is used during calculation
 *
 * @return @c 0 on success, any other value on failure.
 */
int f(int *result, double e);
```

Анализатор вызывает Doxygen для чтения комментариев. Если комментарий содержит ссылку на неизвестную переменную (например, если в секции `@param` написать название несуществующей переменной), то анализатор сигнализирует об ошибке. Также верифицируются секции `@annotation` — `@endannotation`, которые содержат аннотации — это одно из расширений, уникальных для анализатора.

6.3 Анализ групп связности

Интегрированная среда разработки имеет необходимость в анализе связности исходных текстов. Это может быть полезно для выделения библиотек среди разрозненных исходных текстов, определения явных и неявных взаимосвязей различных кодов, наличие которых может отрицательно влиять на самодостаточность модулей, безопасность, скорость компоновки. Для выполнения этой задачи предлагается алгоритм 6.1.

Алгоритм 6.1. Анализ групп связности.

Последовательность действий:

1. Парсер конфигурируется на **игнорирование ошибок** при сборке. Это важно, так как в момент импорта ошибки допустимы вследствие не гарантированно точного выбора определений препроцессора и путей до заголовочных файлов.
2. Производится полноценное **чтение всех требуемых файлов с запоминанием карты пространства имен каждого файла**, но анализ не производится.
3. Осуществляется **конверсия карт пространств имен в линейные группы символов**. Каждый символ имеет несколько основных свойств: имя объекта, пространство имен (в формате `((ns::)*(ns))?`), тип ресурса (локальный, глобальный, статический, принадлежность к методам или параметрам, принадлежность к стандартной библиотеке или библиотеке builtin-определений, признак существования или отсутствия), источник символа (из исходного текста, внешний объект, стандартная библиотека). К названиям объектов и их пространствам имен применяются правила декорирования.
4. **Хеширование символов с формированием хеш-таблицы**.

5. **Объединение символов разных файлов в группы**. Предполагается, что если в разных группах есть пересекающиеся символы, группы могут быть объединены в одну единую группу связности.

Данный подход позволяет эффективно определить возможные «закладки» [178] в коде, даже скрывающиеся за макро-определениями.

6.4 Анализ зависимостей

В процессе анализа групп связности можно произвести дополнительный анализ на прямые зависимости. Он позволяет проверить возможность импорта внешнего проекта, составить графы зависимостей, определить «неявные закладки в коде», помочь в поиске клонов [179; 180] подготовить документацию для сертификации. Для выполнения данного вида анализа требуется алгоритм 6.2.

Алгоритм 6.2. Анализ зависимостей.
Последовательность действий:

1. Записать пары **«вызывающий» – «вызываемый»** при обработке вызовов функций.
2. **Сформировать области связности** на основе данных пар. Если при анализе двух проектов вызывающий символ находится в одной области связности, а вызываемый — в другой области, тогда такие проекты можно назвать *связными по вызовам*.

Следует отметить, что анализ групп связности и анализ зависимостей дают похожую информацию: в обоих случаях это информация о связях подпрограмм. Это связано с тем, что сам по себе факт вызова функции из внешнего проекта отражается фактом наличия символа с признаком несуществования (non-existent) вследствие его присутствия в заголовочных файлах. Однако анализ зависимостей дает возможность построить граф вызовов, а не только области связности.

6.5 Анализ языка

Интегрированная среда разработки в ходе работы может иметь необходимость определить целевой язык. Определение языка по расширению может оказаться неэффективным, так как не все языки имеют выделенные расширения. Неправильное определение целевого языка может выражаться в невозможности провести компиляцию из-за отличий в поведении [181], заголовочных файлах и поддерживаемых расширениях [182].

Алгоритм определения языка основывается на формировании внутреннего *рейтинга* языков. По ходу работы алгоритм присваивает языкам программирования балл в зависимости от соответствия программу этому языку. Язык программирования с наибольшим баллом выбирается как наиболее подходящий язык.

Алгоритм 6.3. Определение языка программирования исходного текста.
Последовательность действий:

1. **Проверить расширение файла для определения наиболее вероятного языка**. Эта информация не всегда корректна с учетом сложившейся практики, поэтому она добавляет лишь 100 баллов к рейтингу языка.
2. **Произвести практическую проверку корректности выбора языка чтением исходного текста**. Если исходный код читается определенным парсером без ошибок, то такой язык попадает в число приоритетных кандидатов. В случае, если приоритетный кандидат лишь один, то такой язык получает статус результирующего, и алгоритм останавливается.
3. **Проверить специфичные для языка синтаксические конструкции** (Приложение Н). Их присутствие существенно увеличивает балл языка в рейтинге, а в случае однозначно идентифицирующих конструкций — влияет на перенос языка в список приоритетных кандидатов.

   Наличие однозначно идентифицирующих конструкций в отношении определенного языка останавливает процесс анализа языка и позволяет задать соответствующий язык как результирующий.

   Если таких конструкций нет, то рейтинг языков никак не меняется.
4. **Проверяется ширина и глубина читаемого синтаксического дерева программы**. К баллу языка добавляется количество прочитанных парсером синтаксических ветвей. Эти сведения могут быть неточными в случае наличия синтаксических ошибок и отсутствия возможности восстановления контекста парсером.

5. **Выбирается результирующий язык программирования**. В случае, если результирующий язык не был выбран на ранних этапах, выбор производится путем вывода языка с наибольшим количеством баллов.

Использование данного подхода позволяет отойти от неточности методов, основанных на применении нейронных сетей [183] и байесовских классификаторов [184].

6.6 Композиционный анализ

В процессе анализа составляются резюме для всех функций. Резюме включает краткие предположения о том, что *делает функция*, *является ли она «чистой» в смысле побочных эффектов*, *вызывает ли она другие функции*.

Процедуры приобретают различные *контексты* в процессе анализа. Например, функция, отправленная как аргумент в `pthread_create`, приобретает контекст «поток». Функции, которые создают объекты, приобретают контекст «конструктор». Функции, в которых инициализируется значительная доля полей структуры, приобретает контекст «инициализатор». Аналогично, если происходит уничтожение выделенной на структуру памяти, то объект приобретает контексты «деинициализатор» или «деструктор». Функции, которые выполняют получение или присваивание одного поля структуры, становятся «свойствами».

Обнаруженные контексты используются для межпроцедурного анализа, а также присылаются в заинтересованные инструменты для выполнения *композиционного анализа* [185]. Польза от такого анализа достаточно очевидна в безопасных средах: он позволяет разработчикам следить за зависимостями и «чистотой» методов, чтобы они выполняли строго заявленный объем работ [186].

6.7 Интерфейс клиент-серверного взаимодействия

Интерфейс клиент-серверного взаимодействия позволяет интегрировать анализ во внешние приложения. Следует отметить, что он реализован не *поверх* анализа, порождая поверхностный вариант клиент-серверного взаимодействия. Проект изначально проектировался с поддержкой такого взаимодействия, и потому поддерживает различные способы управления.

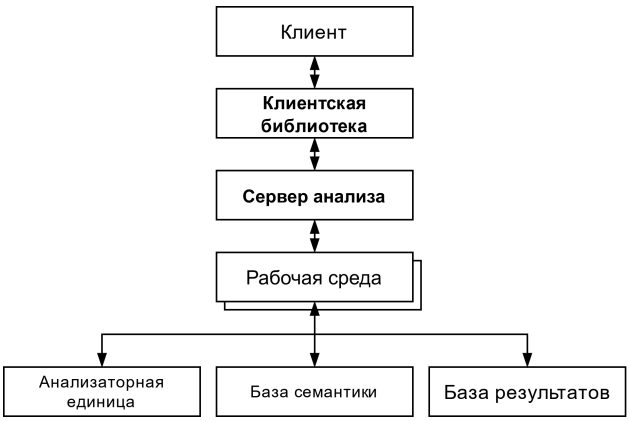

Рисунок 6.1 — Схема клиент-серверного взаимодействия

Работа в клиент-серверном режиме организована посредством библиотеки **asp** [165] (Рис. 6.1), не привязанной к конкретному транспорту. Пользователь библиотеки создает на сервере *рабочую среду* с различными параметрами, анализ которой начинается по отдельному запросу.

Сервер может перенаправлять запросы на любую из анализаторных единиц. Данные могут храниться как локально, так и удаленно [187] (в этом случае данные могут быть переданы произвольным способом, указанным в универсальном идентификаторе файла в описании рабочей среды). Результаты закодированы при помощи индустриальных стандартов, таких как JSON RPC [143] и SARIF [71].

6.8 Интерактивное управление

Внешний пользователь может осуществлять интерактивное управление анализом [188] с целью проведения поиска по коду [189], интерактивной

проверки доказательств, поиска уязвимостей в коде [190]. Для этого используется сервер статического анализа, запускающий отдельный поток для конкретной задачи. По мере выполнения анализа, анализ приостанавливается и ожидает от пользователя ввода.

Интерактивный ввод может осуществляться посредством библиотеки **libasp**. В таком случае анализатор получает запрос от пользователя и продолжает анализ на основе скорректированной пользовательской задачи. Более детально способ выполнения запросов и их язык описаны в статье [32].

6.9 Интеграция различных архитектур и компиляторов

Анализ программ является затруднительным без информации об архитектуре процессора, о компиляторе. На результат анализ влияет порядок байтов целевого процессора, размерность типов в компиляторе и их знакомость в ABI[3]. Также при несоответствии компилятора и анализатора можно столкнуться с различиями **builtin**-функций.

В реальных программах встречаются проверки на компилятор, его версию, целевую платформу. Иногда встречаются и проверки на конкретные опции компилятора. Например, в проекте QT встречается проверка состояния опций PIC/PIE [191], необходимая для раннего указания на ошибку при сборке (Листинг 6.2).

Листинг 6.2: Пример программы, зависящей от определений препроцессора

```
#if !defined(QT_BOOTSTRAPPED) && defined(QT_REDUCE_RELOCATIONS) &&
    defined(__ELF__) && \
     (!defined(__PIC__) || (defined(__PIE__) && defined(Q_CC_GNU) &&
    Q_CC_GNU >= 500))
#error "You must build your code with position independent code if Qt
    was built with -reduce-relocations. "\
          "Compile your code with -fPIC (-fPIE is not enough)."
#endif
```

[3]Например, в GCC для платформы x86_64 `char` является знаковым, а в ARM он беззнаковый: https://developer.arm.com/documentation/den0013/d/Porting/Miscellaneous-C-porting-issues/unsigned-char-and-signed-char

При отсутствии каких-либо определений препроцессора данная проверка не приведет к ошибке, но их отсутствие само по себе изменяет семантику кода, что неприемлемо для точного анализа кода.

Любой компилятор предварительно задает размеры основных типов в составе определений препроцессора. Например, на основе `WORDSIZE` заголовочный файл `stdint.h` из состава стандартной библиотеки glibc[4] и включаемые им заголовочные файлы формируют типы `int*_t`, `uint*_t` и т.п.

Таким образом, статическому анализатору кода безусловно нужно знать целевую архитектуру ЦП, свойства компилятора и реализуемых им типов.

Приведем формальное описание методики учета низкоуровневой архитектуры.

Пусть C — множество центральных процессоров. Тогда C' — множество поддерживаемых статическим анализатором центральных процессоров, а $c \in C'$ — один из поддерживаемых процессоров.

Пусть I — множество встроенных скалярных типов. К ним можно отнести типы без привязки к низкоуровней архитектуре.

Пусть T — множество низкоуровневых типов. Низкоуровневый тип определяет принцип хранения типа в выделенном массиве байтов. Пусть $e(v, b)$ — символьный шаблон конверсии из высокоуровневого типа в низкоуровневый (один из потенциально бесконечного множества E), представляющий собой некоторое отношение между типами, $v \in V$ — произвольное значение в высокоуровневых терминах, а b — произвольный массив байтов (из различных массивов байтов B). Тогда если обозначить произвольную последовательность операторов как $\cdot$, то операции над низкоуровневыми типами можно определить следующим образом:

$$\forall t \in T, \forall v \in V, \exists b \in B, e(v, b) = t \cdot b$$

Таким образом, для любой трансформации произвольного значения в массив байтов существует шаблон трансформации $e(v, b)$, подстановка точного значения в которую дает готовый механизм трансляции в массив байтов.

[4] https://www.gnu.org/software/libc/

Отметим, что всякому v соответствует b, причем единственный. При этом массив байтов не является уникальным и может быть реинтерпретирован другим значением v, поэтому обратная конверсия, строго говоря, имеет смысл только при указании целевого типа.

Определим основные элементы каждого **центрального процессора**. Пусть $i \in I$ — произвольный из внутренних скалярных типов. Тогда $type : [c, i] \to t, i \in I, t \in T$ — функция выбора низкоуровневого типа. Необходима функция выбора указательного типа:

$$pointer\_type : c \to t, t \in T$$

Также важна функция выбора типа, соответствующего машинному слову:

$$word\_type : [c, prop] \to t, t \in T$$

где $prop$ — пропорция размера слова: полуслово, слово, двойное слово и т.п.

Тогда процессор может быть представлен следующим образом:

$$CPU = (pointer\_type_{CPU}, word\_type_{CPU}) \tag{6.5}$$

Определим основные элементы каждого **компилятора**. Пусть $s \in S$ — произвольное словесное написание названия типа (из множества названий). Тогда необходимо следующее преобразование:

$$integral\_type : s \to I, s \in S, i \in I$$

которое реализует получение интегрального типа, соответствующего произвольному словесному написанию. Добавим также множество определений препроцессора $preprocessor\_defs \in S$, соответствующее текстовому написанию стандартных определений препроцессора.

Тогда компилятор может быть представлен следующим образом:

$$Compiler = (integral\_type_{Compiler}, preprocessor\_defs_{Compiler}) \tag{6.6}$$

С помощью инструментов вроде `crosstool-ng`[5] практически каждый инженер может собрать свой инструментарий для сборки под практически любую платформу и с любыми настройками. В таких условиях практически невозможно учесть все варианты компиляторов, ABI, процессоров.

[5] https://crosstool-ng.github.io

В качестве решения предлагается концепция *гибких таргетов* (*flexible target*), позволяющая сформировать новый таргет на основе информации, вычисленной по пробному запуску (алгоритм 6.4) целевого компилятора (*пробированию*).

Алгоритм 6.4. Настройка гибкого таргета

Последовательность действий:

1. Назовем «целевыми» компилятор и архитектуру ЦП для того компилятора, который предлагается эмулировать.
   «Новыми» назовем компилятор и архитектуру ЦП, которые порождаются данным алгоритмом.
2. За основу нового инструментария берется инструментарий, состоящий из центрального процессора необходимой архитектуры и компилятора одного семейства.
3. Совершается *пробный запуск* целевого компилятора. Для семейства GCC команда выглядит следующим образом (Листинг 6.3).

Листинг 6.3: Командная строка пробного запуска компилятора

```
LC_ALL=C.UTF-8 gcc -std=gnu++17 -xc++ -E -dM -v probe.txt
```

Для clang-cl команда выглядит следующим образом (Листинг 6.4).

Листинг 6.4: Командная строка пробного запуска компилятора

```
LC_ALL=C.UTF-8 clang-cl /clang:-E /clang:-dM /clang:-v probe.
   txt
```

Добавление префикса необходимо для обхода ограничений clang-cl, связанных с поддержкой эмуляции параметров из cl.exe от Microsoft.

4. Из полученного вывода можно выделить автоматически обнаруженные пути до директорий с заголовочными файлами (Листинг 6.5).

Листинг 6.5: Обнаружение путей в выводе компиляторов

```
#include <...> search starts here:
...
#include "..." search starts here:
...
```

Определения препроцессора, в том числе те, которые определяют типы переменных и их размеры, начинаются со строки `#define`.

Использование данного метода позволяет эффективно настроить анализатор на использование пользовательского компилятора и добиться высокой степени схожести среды.

6.10 Моделирование поддержки различных операционных систем

Одной из наиболее популярных операционных систем является Microsoft Windows. Моделирование сборки C/C++ программ под эту ОС возможно при наличии Windows SDK. В таком случае первичными целями являются:

1. Подстановка путей до заголовочных файлов Windows SDK для препроцессора.
2. Модификация определений препроцессора в соответствии с требованиями CL.
3. Установка целевого триплета windows msvc.

При попытке применить сборку на операционных системах с регистрозависимыми файловыми системами [192] (или хотя бы при размещении основных компонентов на такой файловой системе) можно столкнуться с тем, что Windows SDK хранит файлы с названием в фиксированном регистре, но при этом обращается к файлам в произвольном регистре. В MacOS такая проблема не возникает при стандартных настройках чувствительности к регистру в файловой системе. В Linux, где стандартная файловая система ext4 является регистрочувствительной, эта проблема может быть решена несколькими способами:

1. Через монтирование образа регистронезависимой файловой системы, содержащего Windows SDK, с помощью `mount` или `udisks2`.
2. Через предзагрузку библиотеки, подменяющей названия файлов при обращении к ним, с помощью механизма `LD_PRELOAD` [193].
3. В некоторых модулях чтения, например, в Clang, есть поддержка виртуальных файловых систем (Virtual File System, VFS), которые могут быть применены для решения проблемы с регистрами названий файлов. Аналогичный механизм может быть реализован в самом анализаторе и/или в других парсерах.

Для моделирования сборки программ под Linux, как правило, ничего не требуется: достаточно адаптироваться под целевой компилятор и процессор. В практических случаях, известных автору, нет необходимости контроля регистрозависимости (нет файлов с одинаковым названием в разных регистрах).

Для сборки программ под MacOS требуется поддержка путей до SDK, задание переменной `__ENVIRONMENT_MAC_OS_X_VERSION_MIN_REQUIRED__` и включение опции `__BLOCKS__`.

6.11 Сканирующий запуск компиляции

Применение статических анализаторов может быть достаточно непростым при наличии крупных промышленных проектов с развитой системой сборки. Это связано с тем, что необходимо снабдить анализ теми же сведениями, что подаются компилятору. Предварительный анализ сборочных файлов без их исполнения может оказаться неточным, так как некоторые системы сборки позволяют добавить нестандартные вставки, например, код на произвольном языке программирования, зависящий от переменных среды. В некоторых проектах используется кодогенерация, и тогда получить все необходимые сведения о проекте представляется невозможным. Таким образом, статический анализатор вынужден записывать аргументы командной строки всех компиляторов, используемых в процессе компиляции.

Для выявления параметров запуска компилятора используются несколько основных методов:

1. Формирование перечня аргументов компилятора для каждого исходного файла системой сборки. Этот вариант доступен для многих систем сборки. Одним из стандартных форматов является `compile_commands.json` [194].
2. Сканирование аргументов сборки при помощи перехвата аргументов процессов. Данный вариант доступен на всех основных операционных системах. В Linux доступна подмена функций `exec` через механизм `LD_PRELOAD` [195]. Аналогичный механизм для Mac OS основывается на переменной среды `DYLD_INSERT_LIBRARIES` [196]. В Windows можно использовать подмену `CreateProcessA`/`CreateProcessW` с помощью пакета **Detours** [197].

Таким образом, сканирующий запуск компиляции предоставляет перечень аргументов для компилятора, который может использоваться ана-

лизатором для построения рабочей среды и правильного воспроизведения окружения.

6.12 Эмуляция окружения на основе аргументов компиляторов

Не повторяя алгоритм анализа вывода компиляторов из п. 6.9, распространим эмуляцию окружения на аргументы компиляторов. Алгоритм 6.5 является *компиляторо-специфичным*, но рассмотрим его для C/C++.

Алгоритм 6.5. Анализ аргументов компиляторов C/C++

Последовательность действий:

1. На вход поступает набор аргументов, где нулевой элемент — название компилятора (или путь до него), а остальные аргументы — прямые аргументы указанного компилятора.
2. По названию компилятора определяется его семейство.
    а) Если название содержит `ccache` и т.п., то этот аргумент игнорируется, и название берется из следующего аргумента.
    б) Если название содержит `gcc` или `g++`, то компилятор определяется как `gcc`. Если название содержит `clang`, `clang++`, то компилятор определяется как `clang`. Если содержится `cl`, то компилятор определяется как CL-совместимый (это может быть как Clang-CL, так и компилятор CL из набора Microsoft Visual C++). Сам путь до компилятора добавляется в перечень точных путей, относительно которых будет произведен поиск наборов инструментов.
3. Из оставшихся аргументов командной строки интерпретируются значимые аргументы компиляторов:
    а) `-I`, `-isystem`, `/I` — пути до заголовочных файлов, обычно они добавляются в описатель окружения и применяются перед чтением исходного текста;
    б) `-include` — путь до явно включаемого заголовочного файла, обычно добавляется в описатель окружения и применяется перед чтением исходного текста;
    в) `-D` — определение препроцессора, добавляется в перечень определений препроцессора и может применяться как в преамбуле исходного текста программы, так и непосредственно на объекте парсера исходного текста.
    г) `-f<название>` — дополнительные опции компилятора.
4. Выполняется конвертация опций, требующих сложной обработки. Например, существуют сложные правила взаимодействия опций `-fpic`, `-fpie`, `-fPIC`, `-fPIE` и выработкой значений определений препроцессора `__pic__`, `__PIC__`, `__pie__`,

`__PIE__` со значениями 1 или 2, которые и добавляются в перечень определений препроцессора.

5. Все определения препроцессора, пути до заголовочных файлов и прочие настройки применяются на объекте парсера (в случае Clang — CompilerInstance).

Используя данный алгоритм, можно добиться высокого уровня соответствия между анализируемой и компилируемой рабочими средами.

Глава 7. Обеспечение качества решения

Обеспечение качества проекта, обеспечивающего анализ программ на разных языках, для множества компиляторов, процессоров и операционных систем, требует комплексного подхода. Приведем основные техники.

Юнит-тестирование позволяет проверить исходный код помодульно. Для реализации тестирования используется фреймворк Google Test [198]. Основные группы тестов приведены в Приложении О.

Интеграционное тестирование заключается в тестировании комбинации модулей. Для этого также применяется формат юнит-тестов в приложении к *мультирешателю*, *модулям чтения*.

Под **системным тестированием** понимается тестирование анализатора в комплексе. Основным фреймворком является **OpenVEF** [199], главными особенностями которого являются *группировка тестов по различным категориям* (ветвления, аллокации памяти, работа с типами и т.п.), *наличие позитивных и негативных тестов*; *внимание к возможности читать распространенные синтаксические конструкции и оперировать ими*; *поддержка конструкций EXPECT и GLOBAL-EXPECT* для проверки появления предупреждений на конкретных строчках или в файле в целом (без учета местоположения); *существование тестов на нескольких языках программирования*.

Схема работы фреймворка относительно проста (Приложение П):

1. Пользователем формируется перечень свойств среды, в которой необходимо запускать статические анализаторы.
2. В зависимости от выбранной среды, фреймворк запускает статический анализатор под корректным интерпретатором и с правильными путями до системных SDK.
3. Производится анализ тестов необходимых групп и сравнение результатов анализа с эталоном. В зависимости от степени соответствия, результат анализа может быть *точным*, *неполным* (то есть не все ошибки были найдены), *избыточным* (если были найдены неожиданные ошибки), *комбинационным* (то есть часть ошибок не найдена, часть избыточна) или *фатальным* (когда анализ закончился с ошибкой).

4. Генерируются результаты анализа в необходимых форматах (JUnit XML, HTML).

Самоанализ позволяет эффективно контролировать качество статического анализа через проверку кода самого анализатора. Это возможно, так как анализатор построен на языке C++ с применением компиляторов GCC/Clang (в зависимости от целевой системы) и собирается при помощи CMake [200].

Не менее важен и **статический анализ внешними средствами**, так как ошибки в проекте могут препятствовать поиску дефектов в нем самом. Для этого используется Clang-Tidy [201]. Также не игнорируется существование Klee [130], который запускается на основных алгоритмах гибридного решателя.

Увеличить качество кода позволяет **кодогенерация** [199], которая уменьшает количество ошибок вследствие некорректного копирования кода. Она используется для *выражений*, *перечислений*, *команд виртуальной машины* и других сущностей, представленных в коде.

Значительное влияние имеет техника фаззинга [202]. Реализованный в проекте генератор случайных исходных текстов позволяет найти ошибки в чтении и обработке основных синтаксических конструкций [199].

7.1 Тестирование в различных режимах и средах

Для тестирования работы статического анализа в различных режимах и средах используется подход с привлечением средств контейнеризации и эмуляции сред (Приложение Р).

Общая схема тестирования следующая:

1. В качестве средства контейнеризации выступает Docker [203]. Он используется для сборки базового образа операционной системы, используемого для запуска проекта. Такой операционной системой является Ubuntu 20.04.
2. В базовый образ системы устанавливаются системы Wine [204] (для эмуляции приложений Windows) и Darling [205] (для запуска приложений для MacOS).

3. В образ системы также устанавливаются целевые SDK и компиляторы. Для Windows — Visual C++ Tools, MinGW-w64, Clang, MIPS Toolchain, ARM Toolchain. Для Linux — MinGW, Clang, а также различные кросс-компиляторы для MIPS, ARM. Также добавляется Windows SDK и Visual C++ Tools, которые могут использоваться для анализа Windows-программ в дистрибутивах Linux. Для MacOS — XCode SDK.
4. Запускаются тесты OpenVEF в разных режимах работы (консольном и серверном) и под разными интерпретаторами (Wine для Windows, Darling для MacOS). В качестве параметра запуска выбираются различные компиляторы, операционные системы и процессоры, применяемые в тестировании.

Данный подход позволяет составить матрицу результатов тестирования. По ней можно определить, насколько корректно работает статический анализатор под всеми основными операционными системами и компиляторами.

Следует отметить главный недостаток данного подход: в качестве основы он требует операционной системы на базе Linux, что фактически сводит вопрос качества работы на операционной системе к вопросу качества работы на её эмуляторе. Однако проблемы с данным подходом не встречались.

Глава 8. Апробация решения

В данной главе приведены результаты применения статического анализатора в различных условиях и с акцентом на основные случаи использования.

Почти во всех тестах использовался процессор Apple M1 Max (ARM), 64ГБ ОЗУ. Intel Core i7-7700HQ, 16ГБ использовался в тестах проиводительности и в случаях Windows, Linux.

8.1 Апробация в синтетических тестах

Качество анализа было проверено на фреймворке Toyota ITC benchmark [206] (таблица 1) на более ранней версии из исследования [32]. С момента проверки были добавлены новые детекторы, улучшено качество анализа отдельных аспектов, но это не сказалось на указанных тестах.

Таблица 1 — Качество анализа в сравнении

Тест	Проект	Frama-C	Clang	cppcheck	Всего
`bit_shift`	17	17	14	11	17
`buffer_overrun_dynamic`	30	32	1	2	32
`buffer_underrun_dynamic`	35	39	2	3	39
`data_lost`	19	3	—	—	19
`data_overflow`	25	16	—	9	25
`data_underflow`	12	8	—	5	12
`littlemem_st`	11	11	—	—	11
`null_pointer`	15	16	13	12	17
`overrun_st`	47	54	2	21	54
`ptr_subtraction`	2	1	—	—	2
`underrun_st`	13	13	2	5	13
`uninit_pointer`	10	16	11	5	16
`zero_division`	16	16	13	8	16

Качество обнаружения ошибок в этих тестах лучше **Clang** и **cppcheck** по большинству категорий, кроме `uninit_pointer`, что объясняется необходимостью дополнительной обработки `alias`, и сравнимо с **Frama-C**.

Также проводилось тестирование на этом же фреймворке без сравнения с другими программами (Приложение С). Результаты для анализаторов CodeSonar (GrammaTech), Code Prover/Bug Finder (MathWorks) приведены в основной статье авторов бенчмарка [206]. Авторские результаты также приведены в исследованиях [34; 41].

8.2 Апробация на промышленных проектах

Проект статического анализатора проверялся на закрытых промышленных проектах, содержащих более чем 100000 строк кода на C++. Использовалась экспериментальная версия статического анализатора с более глубоким (за счет более активного использования SMT) анализом.

Были выявлены следующие особенности применения подхода:

– Перед комплексной технологией программирования была поставлена задача обеспечить компиляцию при помощи Clang, для чего потребовалось скорректировать исходный текст. Такое же требование транзитивно распространилось и на анализ.
– Реализации методов в заголовочных файлах приводят и потере производительности примерно на 10-15%.
– 7-15% производительности теряется на работе с глобальными ресурсами, которых много в C++.
– Выполнение анализа с помощью SMT-решателей происходит медленно (потери до 90% производительности), поэтому был разработан комбинированный подход для обеспечения производительного анализа (Гл. 5).
– Резюме функций отражает не все их важные свойства, что выражается в необходимости формирования ручных аннотаций для критического кода.

Организацией ООО «СофтКом» был предоставлен акт о внедрении продукта в комплексную технологию программирования.

Проект был апробирован на ядре Linux, отдельных драйверах и реализации USB-контроллера [33]. Новая версия анализатора в 3.3-4.5 раза быстрее обнаруживает рассмотренные в указанной работе ошибки благодаря переходу на более эффективную технологическую основу (с Roslyn на C++/Clang).

Была проведена верификация масштабируемости [164] анализа (таблица 2). Были взяты варианты с запуском виртуальных машин на одном узле, 4 узлах, 8 узлах, а также в варианте с одним узлом, но более простым IR (в ассемблерном стиле без структурных ветвлений), а также с одним узлом и без виртуальных машин (аналогично работе [33]). Для параллелизации запуска применялись MPI и протокол asp.

Таблица 2 — Масштабируемость анализа

Среда	Производительность (%)	Точность (%)
ВМ (1 узел)	100	100
ВМ (4 узла)	369	100
ВМ (8 узлов)	720	100
ВМ с простым IR	114	63
Без ВМ	176	80

Из результатов можно сделать вывод, что производительность анализа возрастает почти линейно. Однако это справедливо лишь в контексте тестирования, и затраты на редукцию семантических контекстов и передачу данных, скорее всего, существенно вырастут по мере увеличения количества узлов, что довольно сложно проверить в условиях ограниченных ресурсов.

8.3 Тестирование межъязыкового анализа

Для большей репрезентативности межъязыкового анализа, был применен транслятор C# из работы [33], несмотря на то, что он не является частью текущего исследования. Были отмечены следующие особенности применения схемы:

1. Конвертация структур между нативными языками происходит быстро вследствие отсутствия маршалинга. В проведенных экспериментах на это тратится не более 0.5% времени.
2. Не все типы имеют прямое отображение в другой язык, что не является проблемой для анализатора, но в целом препятствует корректному переносу семантики.
3. Для переноса семантики были разработаны «языковые пространства имен», позволяющие наладить биективные отображения между вызываемыми ресурсами.

Также была проверена корректность межъязыкового анализа (таблица 3). Особый интерес представляла способность переносить структуры между разными языками — для этого брались только поддерживаемые обоими языками возможности.

Таблица 3 — Качество межъязыкового анализа

Среда	Перенос структур (%)	Точность (%)
РуСи/C	100	100
РуСи/C++	97	96
C/C++	100	100
C#/C	88	78
C#/C++	74	81

Между смежными языками существуют прямые отображения типов, что дает 100% результат. При конверсиях РуСи и C++ нашлись незначительные ошибки. Между C# и нативными языками нашлось значительно большее число технических ошибок. Точность же анализа, практически идеальная при смежных языках, падает на $\approx 20\%$ при проверках межъязыкового взаимодействия различающихся языков.

8.4 Тестирование технологического решения по интеграции

Интеграционное решение тестировалось следующими способами:

1. Было испытано ускорение (таблица 4), которое достигается при использовании решения по интеграции. Время анализа вычислялось по формуле:

$$T = T_w + \sum_{i=1}^{N} \cdot (T_{c_i} + T_{f_i} + T_{p_i}) + \sum_{j=1}^{M} T_{q_j}$$

   где T_w, T_c, T_f, T_p, T_q — время подготовки рабочей среды, контролируемой компиляции, загрузки ресурсов, первичного анализа и анализа по запросу; N — количество файлов в среде, M — количество файлов, просмотр которых необходимо для выполнения запроса. Считается, что для самостоятельной версии требуется полный запуск ядра анализа («разогрев»), инкрементальная версия производит работу с уже сохраненными данными (поэтому времени на «разогрев» не требуется), а резидентная версия требует «разогрева» лишь единожды. Выполнялись следующие запросы:

   а) Запрос #1: проверка инициализированности возвращаемого значения функций.

   б) Запрос #2: проверка, что у функции константные возвращаемые значения.

   в) Запрос #3: проверка, что функции, вызывающие блокировку мьютекса, также вызывают разблокировку этого же мьютекса.

   В результате можно отметить, что резидентная версия анализа значительно быстрее обрабатывает запросы, чем самостоятельная версия (1.53%, 18.6% и 21.7% от полного времени). Резидентный анализ при этом медленнее простого инкрементального запуска на 6.9-41.6%, так как он требует ресинхронизации (сохранения) семантических контекстов.

2. В работе [165] также рассматриваются потери при остановке анализирующих узлов, но этот основанный на MapReduce [207] алгоритм [158] не рассматривается в данной работе.

Таблица 4 — Скорость выполнения запросов

Тест	Время (мсек)			Разница (%)
	Самост. (1)	Инкрем. (2)	Резид.(3)	(2) - (3)
«Разогрев»	12536	0	12682	—
Запрос #1	14854	2318	2772	−19.5
Запрос #2	12673	137	194	−41.6
Запрос #3	15742	3206	3430	−6.9

3. В исследовании [187] приводятся результаты тестирования разных способов хранения данных, в том числе и на разных узлах кластера.
4. Производился обмен данными разного размера (от 20 байтов до 80.000 байтов) между статическим анализатором и компилятором. В результате проверки было отмечено, что протокол NNG более стабильно реализует фрагментацию пакетов (0% потерь) и имеет незначительную ретрансмиссию ($\approx 1\%$) в случае применения 5 ГГц WiFi адаптера в качестве передатчика. TCP быстрее (на 30-50%) в сетях с большим MTU/MRU, было отмечено 3% потерь из-за алгоритмических ошибок при обработке пакетов.
5. При переносе компьютеров в разные по удаленности сети разницы между NNG и TCP отмечено не было.
6. NNG показал себя значительно лучше в испытаниях с обрывами связи и повышением латенции сети (через `tc` [208]), что связано со специализацией протокола на данных ситуациях. В TCP возникли проблемы с обработкой вызова `asp_connection_wait_analysis`, который может выполняться все время до окончания анализа.

Была оценена производительность механизма пробирования компиляторов (таблица 5). В Windows были установлены MSYS2, LLVM/Clang 13. В Ubuntu и MacOS были установлены GCC, LLVM/Clang через соответствующие пакетные менеджеры. В таблице указано количество активных/всех проверяемых конфигураций, а также общее и среднее время пробирования.

По результатам тестирования можно определить, что механизм пробирования для Windows требует доработок, так как он значительнее медленнее,

Таблица 5 — Производительность пробирования

Среда	Конфиг. (ед)	Общ. t (сек)	Ср. t (сек/комп)
Windows 10	47/120	73.685	1.6740
Ubuntu 20.04	33/33	0.604	0.0180
MacOS (ARM)	33/33	0.293	0.0088

чем в других операционных системах. Вероятно, сказывается проблема с производительностью перенаправления. В MacOS и Ubuntu пробирование осуществляется со сравнимой производительностью, а превосходство M1 объясняется более высокой производительностью на ядро.

Была проведена проверка производительности перехвата вызовов компиляторов (таблица 6).

Таблица 6 — Производительность перехвата вызовов

Среда	$\triangle t$ сборки (%)
Windows 10	1.11
Ubuntu 20.04	0.93
MacOS (ARM)	0.98

По результатам проверки отмечено, что в Windows наблюдается чуть более значительное увеличение времени анализа, а разница между Ubuntu и MacOS практически отсутствует.

8.5 Апробация в научном сообществе

Основные результаты по теме выпускной квалификационной работы изложены в 15 печатных изданиях, 2 из которых изданы в журналах, рекомендованных ВАК, 5 — в периодических научных журналах, индексируемых Web of Science и Scopus, 13 — в трудах по итогам конференций. Зарегистрированы 4 программы для ЭВМ. Различные фрагменты работы были также

представлены на международных конференциях и симпозиумах, таких как International Conference on Computational Science and its Applications 2019 (ICCSA-2019) [32; 165], International Conference on Computational Science and its Applications 2020 (ICCSA-2020) [163], International Conference on Tools and Methods of Program Analysis (TMPA-2017) [41], Spring/Summer Young Researchers' Colloquium on Software Engineering 2020 (SYRCoSE 2020) [199], Spring/Summer Young Researchers' Colloquium on Software Engineering 2021 (SYRCoSE 2021) [26], International Conference "Distributed Computing and Grid Technologies in Science and Education" 2018 (GRID-2018) [164], Всероссийской научной конференции по проблемам информатики СПИСОК-2019 [209], а также на Международной научной конференции аспирантов и студентов «Процессы управления и устойчивость» (2016 [210], 2017 [158], 2019 [187], 2020 [147], 2021 [150]).

Результаты работы были представлены на семинаре кафедры системного программирования Санкт-Петербургского государственного университета 7 декабря 2021 г.

За научную работу по данной теме за 2019–2020 и 2020–2021 учебные годы автором были получены стипендии Президента РФ для аспирантов, обучающихся по приоритетным направлениям экономического развития. Также в 2016 году автору была присуждена стипендия биотехнологической компании BIOCAD и Ассоциации выпускников Санкт-Петербургского государственного университета, выдаваемая, согласно Положению о стипендии, по результатам анализа качества и значимости научно-исследовательской работы независимым экспертным советом.

Заключение

Основные выносимые на защиту результаты работы заключаются в следующем:

1. В рамках исследования была разработана методология проведения статического анализа для комплексной технологии программирования на базе C, C++, РуСи с учетом требования по анализу программ на этих языках в одном контексте. Это требовало создания *языково-независимых методов чтения исходных текстов*, *промежуточного представления Midair*, *фреймворка виртуальных машин* и *гибридного решателя*, поддерживающих анализ многоязычных программ на разных уровнях.
2. Сформирована методическая основа для обеспечения взаимодействия различных частей программного комплекса.
3. Автором была спроектирована архитектура анализатора, воплощающая разработанную методологию.
4. Спроектированная система была практически реализована на языках C/C++ с частичным использованием ANTLR, LLVM/Clang, CVC4/Z3 и других открытых индустриальных модулей.
5. Была произведена апробация системы, которая выявила высокое качество алгоритмов обнаружения ошибок, показала возможность применения межъязыкового анализа, а также подтвердила способности проекта по интеграции с внешними средствами комплексной среды программирования.

Рекомендуется использовать наработки данного исследования в системах статического анализа, предназначенных для комплексных сред программирования, где необходимо эффективно и единообразно анализировать программы на разных языках программирования.

В дальнейшем планируется испытать анализатор на большем числе открытых проектов, поучаствовать в различных соревнованиях по качеству анализа. В систему будут добавлены детекторы других языков программирования, увеличена точность анализа за счет применения решателей дизъюнктов Хорна. С большой долей вероятности проект будет открыт для использования внешними участниками.

В заключение автор выражает благодарность Терехову А. Н. за научное руководство, поддержку и помощь при работе над выпускной квалификационной работой. Также автор благодарит коммерческие организации, в которых он работал, за помощь в подготовке тестового материала и за поддержку на ранних этапах развития проекта. Значительный вклад в проект внесли анонимные рецензенты поданных автором на различные конференции статей, отзывы участников конференций. Отдельное спасибо супруге Светлане и дочери Софье за помощь в подготовке исследования.

Словарь терминов

Этап чтения исходной программы

Синтаксический анализатор (парсер, *от англ. parser*) : программа или часть программы для чтения исходного текста на целевом языке программирования и формирования синтаксических структур.

Синтаксическая ветвь : ветвь в терминах синтаксического анализатора целевого языка программирования, связанная с единственной синтаксической конструкцией.

Абстрактная синтаксическая ветвь : синтаксическая ветвь в терминах статического анализатора, связанная с единственной синтаксической конструкцией.

Языковая семантика : набор данных о характеристиках целевого языка программирования, включающий информацию о различных свойствах, связанных с обработкой синтаксических и семантических конструкций.

Языковой базис : набор синтаксических конструкций, пригодный для чтения программ на разных языках программирования.

Обобщенная синтаксическая ветвь : синтаксическая ветвь в терминах языкового базиса.

Обобщенное синтаксическое дерево : абстрактное синтаксическое дерево программы после его чтения анализатором и преобразования в модель из *ресурсов, фрагментов, выражений* (обобщенных синтаксических ветвей) и *типов*.

Пространство имен : именованное или безымянное пространство объектов, имеющих идентификаторы.

Карта пространств имен : древовидная структура, состоящая из вложенных пространств имен, и обеспечивающая доступ к объектам, хранящимся в них.

Семантический контекст : набор данных о семантике анализируемого участка, содержащий карту пространств имен и другие извлеченные признаки.

Ресурс : объект (переменная или функция), имеющий некоторую закрепленную за ним область памяти.

Выражение : объект, ассоциированный с некоторым оператором, выполняющимся над множеством операндов-выражений и операндов типов.

Фрагмент : множество выражений, обладающее некоторой общностью свойств.

Система типов

Тип : описание структуры данных, используемой в анализируемой программе, которое представляет собой последовательность свойств, членов классов, аргументов, ссылок на квалифицированные типы и другие извлеченные признаки, характерные для структуры данных определенного семейства.

Неполный тип : тип данных, не имеющий полностью определенной структуры в определенный момент времени.

Квалификатор : описание некоторого качества, ассоциирование которого с типом данных изменяет качественный состав типа данных.

Квалифицированный тип : описание типа с добавленными к нему модификаторами и именем типа.

Виртуальные машины и гипервизор

Виртуальная машина : объект в анализаторе, симулирующий работу исполнителя (центрального процессора) и отвечающий за однопоточные проверки.

Транслятор виртуальной машины : транслятор из языка обобщенных синтаксических деревьев в выходной язык виртуальной машины.

Язык виртуальной машины : упрощенное представление программы, сформированное транслятором виртуальной машины и исполняемое при дальнейшем анализе.

Гипервизор : объект в анализаторе, комбинирующий работу нескольких виртуальных машин для выполнения межпроцедурного анализа.

Команда виртуальной машины : единственная инструкция в терминах промежуточного представления, предназначенная для выполнения в виртуальной машине.

Контейнер команд виртуальной машины : набор команд виртуальной машины, подготовленный для выполнения в виртуальной машине.

Процесс поиска ошибок

Решатель Satisfiability Modulo Theories (SMT) : программа или часть программы для решения математических неравенств в терминах SMT.

Транслятор в SMT : модуль для трансляции программ на языке виртуальной машины в язык SMT.

Абстрактный интерпретатор : модуль для вычисления аппроксимированного представления программы в рамках абстрактной интерпретации.

Вычислитель : программа или часть программы для вычисления значений выражений с помощью абстрактного интерпретатора и/или решателя SMT.

Резюме : краткое описание свойств функции или её фрагмента, применимое в качестве простой замены метода во время поиска решений задач верификации.

Эмуляция сред

Среда : часть данных о среде компиляции и исполнения, позволяющая произвести анализ в соответствии с характеристиками среды.

Рабочая среда : часть данных о среде, сохраненная пользователем на сервере.

Набор инструментов (тулчейн, от англ. toolchain) : множество образов программ для компиляции, компоновки и других действий, необходимых для трансляции исходных текстов в исполняемые двоичные образы.

Компилятор : набор данных о характеристиках компилятора целевого языка, включающий размерности типов, знаковость типов и прочие свойства.

Процессор : набор данных о характеристиках целевого процессора, включающий информацию о внутреннем строении базовых типов, желаемом выравнивании и о прочих свойствах.

Пробирование компилятора : процесс искусственного запуска компилятора для получения данных среды.

Таргет (*от англ. target*) : краткое описание платформы, используемое в компиляторах и анализаторах при выборе целевой среды.

Гибкий набор инструментов : набор из компилятора и процессора, подстраивающийся под данные, полученные в результате пробирования компилятора.

Список литературы

1. *Schaller, R. R.* Moore's law: past, present and future [Text] / R. R. Schaller // IEEE spectrum. — 1997. — Vol. 34, no. 6. — P. 52—59.
2. *Shalf, J. M.* Computing beyond Moore's Law [Text] / J. M. Shalf, R. Leland // Computer. — 2015. — Vol. 48, no. 12. — P. 14—23.
3. *Kuon, I.* FPGA architecture: Survey and challenges [Text] / I. Kuon, R. Tessier, J. Rose. — Now Publishers Inc, 2008.
4. *Koch, D.* FPGAs for Software Programmers [Text] / D. Koch, F. Hannig, D. Ziener. — Springer International Publishing, 2016.
5. Survey of machine learning accelerators [Text] / A. Reuther [et al.] // 2020 IEEE High Performance Extreme Computing Conference (HPEC). — IEEE. 2020. — P. 1—12.
6. *Singh, M. P.* Evolution of processor architecture in mobile phones [Text] / M. P. Singh, M. K. Jain // International Journal of Computer Applications. — 2014. — Vol. 90, no. 4. — P. 34—39.
7. *Haase, C.* Androids: The Team That Built the Android Operating System [Text] / C. Haase. — Chet Haase, 2021.
8. *Ledin, J.* Modern Computer Architecture and Organization: Learn X86, ARM, and RISC-V Architectures and the Design of Smartphones, PCs, and Cloud Servers [Text] / J. Ledin. — Packt Publishing, 2020.
9. Apple announces Mac transition to Apple silicon [Текст]. — URL: https://www.apple.com/newsroom/2020/06/apple-announces-mac-transition-to-apple-silicon (дата обр. 30.05.2022).
10. *Le Lann, G.* An analysis of the Ariane 5 flight 501 failure-a system engineering perspective [Text] / G. Le Lann // Proceedings International Conference and Workshop on Engineering of Computer-Based Systems. — 1997. — P. 339—346.
11. *Leveson, N.* An investigation of the Therac-25 accidents [Text] / N. Leveson, C. Turner // Computer. — 1993. — Vol. 26, no. 7. — P. 18—41.

12. Guidelines for the Use of the C Language in Critical Systems [Text] / M. T. M. I. S. R. Association [et al.]. — 2004.

13. *Klabnik, S.* The Rust Programming Language (Covers Rust 2018) [Text] / S. Klabnik, C. Nichols. — No Starch Press, 2019.

14. CompCert-a formally verified optimizing compiler [Text] / X. Leroy [et al.] // ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress. — 2016.

15. *Hobbs, C.* Embedded software development for safety-critical systems [Text] / C. Hobbs. — CRC Press, 2019.

16. *Rierson, L.* Developing safety-critical software: a practical guide for aviation software and DO-178C compliance [Text] / L. Rierson. — CRC Press, 2017.

17. *Blandy, J.* Programming Rust: Fast, Safe Systems Development [Text] / J. Blandy, J. Orendorff. — O'Reilly Media, 2017.

18. *Martin, E.* Go in Action [Text] / E. Martin, W. Kennedy, B. Ketelsen. — Manning, 2015.

19. *Srikant, Y.* The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition [Text] / Y. Srikant, P. Shankar. — CRC Press, 2018.

20. *Skeet, J.* C# in Depth: Fourth Edition [Text] / J. Skeet. — Manning Publications, 2019.

21. *Jemerov, D.* Kotlin in Action [Text] / D. Jemerov, S. Isakova. — Manning, 2017.

22. *Gaynor, A.* Linux kernel modules in rust [Text] / A. Gaynor, G. Thomas // Proceedings of the Linux Security Summit North America. — 2019. — Vol. 2019.

23. *Leun, V. van der*. Introduction to JVM Languages [Text] / V. van der Leun. — Packt Publishing, 2017.

24. *Hewardt, M.* .NET Internals and Advanced Debugging Techniques [Text] / M. Hewardt. — Addison Wesley Professional, 2014. — (Addison-wesley Microsoft Technology).

25. *Lattner, C.* LLVM and Clang: Next generation compiler technology [Text] / C. Lattner // The BSD conference. Vol. 5. — 2008.

26. *Menshikov, M. A.* Review of static analyzer service models [Текст] / M. A. Menshikov // Труды института системного программирования РАН. — 2021. — Т. 33, № 3. — С. 27—40.

27. *Терехов, А. Н.* Инструментальное средство обучения программированию и технике трансляции [Текст] / А. Н. Терехов // Компьютерные инструменты в образовании. — 2016. — № 1.

28. *Терехов, А. Н.* Проект РуСи для обучения и создания высоконадежных программных систем [Текст] / А. Н. Терехов, М. А. Терехов // Известия высших учебных заведений. Северо-Кавказский регион. Технические науки. — 2017. — 3 (195). — С. 70—75.

29. *Архипов, И. С.* Генерация оптимального объектного кода [Текст] / И. С. Архипов. — 2021.

30. *Терехов, А. Н.* Виртуальная машина для проекта РуСи [Текст] / А. Н. Терехов, А. В. Митенев, М. А. Терехов // Компьютерные инструменты в образовании. — 2016. — № 6. — С. 33—41.

31. *Medvedev, O.* Using hardware-software codesign language to implement CANSCID [Text] / O. Medvedev, I. Posov // Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010). — 2010. — P. 85—88.

32. *Menshikov, M.* Equid - a static analysis framework for industrial applications [Text] / M. Menshikov // International Conference on Computational Science and Its Applications. — Springer. 2019. — P. 677—692.

33. *Меньщиков, М. А.* Нахождение условий гонки в коде на C методом статического анализа [Текст] / М. А. Меньщиков. — 2016.

34. *Меньщиков, М. А.* Гибридная система статического анализа с доказательной проверкой инвариантов [Текст] / М. А. Меньщиков. — 2018.

35. Static analyzer Svace for finding defects in a source program code [Text] / V. Ivannikov [et al.] // Programming and Computer Software. — 2014. — Vol. 40, no. 5. — P. 265—275.

36. *Бородин, А. Е.* Статический анализатор Svace как коллекция анализаторов разных уровней сложности [Текст] / А. Е. Бородин, А. А. Белеванцев // Труды Института системного программирования РАН. — 2015. — Т. 27, № 6.

37. Aegis [Текст]. — URL: http : / / www . digiteklabs . ru / aegis (дата обр. 30.05.2022).

38. *Akhin, M. K.* Software defect detection by combining bounded model checking and approximations of functions [Text] / M. K. Akhin, M. A. Belyaev, V. M. Itsykson // Automatic Control and Computer Sciences. — 2014. — Vol. 48, no. 7. — P. 389—397.

39. *Akhin, M.* Borealis bounded model checker: the coming of age story [Text] / M. Akhin, M. Belyaev, V. Itsykson // Present and Ulterior Software Engineering. — Springer, 2017. — P. 119—137.

40. PVS-Studio [Текст]. — URL: https : / / pvs - studio . com / ru (дата обр. 30.05.2022).

41. *Menshchikov, M.* 5W+1H static analysis report quality measure [Text] / M. Menshchikov, T. Lepikhin // International Conference on Tools and Methods for Program Analysis. — Springer. 2017. — P. 114—126.

42. *Фадин, А. А.* Appchecker - инструмент статического анализа [Текст] / А. А. Фадин, С. Борзых, П. Гусев // Control Engineering Россия. — 2017. — № 2. — С. 26—29.

43. *Швед, П.* Опыт развития инструмента статической верификации BLAST [Текст] / П. Швед, В. Мутилин, М. Мандрыкин // Программирование. — 2012. — Т. 38, № 3. — С. 24—35.

44. A few billion lines of code later: using static analysis to find bugs in the real world [Text] / A. Bessey [et al.] // Communications of the ACM. — 2010. — Vol. 53, no. 2. — P. 66—75.

45. cppcheck - A tool for static C/C++ code analysis [Текст]. — URL: https://cppcheck.sourceforge.io (дата обр. 30.05.2022).

46. Cppcheck Premium [Текст]. — URL: https://www.cppchecksolutions.com (дата обр. 30.05.2022).

47. *Campbell, G. A.* SonarQube in action [Text] / G. A. Campbell, P. P. Papapetrou. — Manning Publications Co., 2013.

48. Semgrep [Текст]. — URL: https://semgrep.dev (дата обр. 30.05.2022).

49. QL for source code analysis [Text] / O. De Moor [et al.] // Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007). — IEEE. 2007. — P. 3—16.

50. CLion - A cross-platform IDE for C and C++ [Текст]. — URL: https://www.jetbrains.com/clion (дата обр. 30.05.2022).

51. *Gąsior, Ł.* ReSharper Essentials [Text] / Ł. Gąsior. — Packt Publishing, 2014. — (Community experience distilled).

52. What is clangd? [Текст]. — URL: https://clangd.llvm.org (дата обр. 30.05.2022).

53. The IntelliJ Platform: A Framework for Building Plugins and Mining Software Data [Текст] / Z. Kurbatova [и др.] // 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW). — 2021. — С. 14—17.

54. *Schubert, P. D.* Phasar: An inter-procedural static analysis framework for C/C++ [Text] / P. D. Schubert, B. Hermann, E. Bodden // International Conference on Tools and Algorithms for the Construction and Analysis of Systems. — Springer. 2019. — P. 393—410.

55. *Vermeir, N.* .NET Compiler Platform [Text] / N. Vermeir // Introducing .NET 6: Getting Started with Blazor, MAUI, Windows App SDK, Desktop Development, and Containers. — Berkeley, CA : Apress, 2022. — P. 275—295.

56. *Bock, J.* .NET Development Using the Compiler API [Text] / J. Bock. — Apress, 2016.

57. The ASTRÉE analyzer [Text] / P. Cousot [et al.] // European Symposium on Programming. — Springer. 2005. — P. 21—30.

58. *Cousot, P.* Abstract interpretation [Text] / P. Cousot // ACM Computing Surveys (CSUR). — 1996. — Vol. 28, no. 2. — P. 324—328.

59. *Delmas, D.* Astrée: From research to industry [Text] / D. Delmas, J. Souyris // International Static Analysis Symposium. — Springer. 2007. — P. 437—451.

60. *Hojjat, H.* The ELDARICA horn solver [Text] / H. Hojjat, P. Rümmer // 2018 Formal Methods in Computer Aided Design (FMCAD). — IEEE. 2018. — P. 1—7.

61. Horn clause solvers for program verification [Text] / N. Bjørner [et al.] // Fields of Logic and Computation II. — Springer, 2015. — P. 24—51.

62. *Fedyukovich, G.* Competition Report: CHC-COMP-21 [Text] / G. Fedyukovich, P. Rümmer // arXiv preprint arXiv:2109.04635. — 2021.

63. Frama-C [Text] / P. Cuoq [et al.] // International conference on software engineering and formal methods. — Springer. 2012. — P. 233—247.

64. Frama-C: A software analysis perspective [Text] / F. Kirchner [et al.] // Formal Aspects of Computing. — 2015. — Vol. 27, no. 3. — P. 573—609.

65. The SeaHorn verification framework [Text] / A. Gurfinkel [et al.] // International Conference on Computer Aided Verification. — Springer. 2015. — P. 343—361.

66. Moving fast with software verification [Text] / C. Calcagno [et al.] // NASA Formal Methods Symposium. — Springer. 2015. — P. 3—11.

67. *Kettl, M.* The static analyzer Infer in SV-COMP (competition contribution) [Текст] / M. Kettl, T. Lemberger // International Conference on Tools and Algorithms for the Construction and Analysis of Systems. — Springer. 2022. — С. 451—456.

68. Build your own Resource Leak analysis [Текст]. — URL: https://github.com/facebook/infer/blob/main/infer/src/labs/README.md (дата обр. 30.05.2022).

69. Why don't software developers use static analysis tools to find bugs? [Text] / B. Johnson [et al.] // 2013 35th International Conference on Software Engineering (ICSE). — IEEE. 2013. — P. 672—681.

70. *Anderson, P.* Modernizing Static Analysis Tools to Facilitate Integrations [Text] / P. Anderson // ACM SIGAda Ada Letters. — 2020. — Vol. 39, no. 1. — P. 101—108.

71. *Fanning, M.* Static Analysis Results Interchange Format (SARIF) Version 2.0 [Текст] / M. Fanning, L. J. Golding. — 2019. — URL: https://docs.oasis-open.org/sarif/sarif/v2.0/sarif-v2.0.html (дата обр. 30.05.2022).

72. *Anderson, P.* Modernizing Static Analysis Tools to Facilitate Integrations [Text] / P. Anderson // Ada Lett. — New York, NY, USA, 2020. — Vol. 39, no. 1. — P. 101—108.

73. Compilers: Principles, Techniques, and Tools [Текст] / A. Aho [и др.]. — Pearson Addison-Wesley, 2014.

74. *Cooper, K.* Engineering a Compiler [Text] / K. Cooper, L. Torczon. — Elsevier Science, 2011.

75. *Yelland, P.* A New Approach to Optimal Code Formatting [Текст] / P. Yelland. — 2016. — Technical note for open source project rfmt; https://github.com/google/rfmt.

76. *Hatton, L.* Safer language subsets: an overview and a case history, MISRA C [Текст] / L. Hatton // Information and Software Technology. — 2004. — Т. 46, № 7. — С. 465—472.

77. Designing the McCAT compiler based on a family of structured intermediate representations [Текст] / L. Hendren [и др.] // International Workshop on Languages and Compilers for Parallel Computing. — Springer. 1992. — С. 406—420.

78. *Merrill, J.* GENERIC and GIMPLE: A new tree representation for entire functions [Text] / J. Merrill // Proceedings of the 2003 GCC Developers' Summit. — Citeseer. 2003. — P. 171—179.

79. GNU Compiler Collection (GCC) internals: RTL [Текст]. — URL: https://gcc.gnu.org/onlinedocs/gccint/RTL.html (дата обр. 30.05.2022).

80. SUIF: An infrastructure for research on parallelizing and optimizing compilers [Текст] / R. P. Wilson [и др.] // ACM Sigplan Notices. — 1994. — Т. 29, № 12. — С. 31—37.

81. Soot - a Java Bytecode Optimization Framework [Текст] / R. Vallée-Rai [и др.] // Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research. — Mississauga, Ontario, Canada : IBM Press, 1999. — С. 13. — (CASCON '99).

82. *Arzt, S.* Towards Cross-Platform Cross-Language Analysis with Soot [Text] / S. Arzt, T. Kussmaul, E. Bodden // Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis. — Santa Barbara, CA, USA : Association for Computing Machinery, 2016. — P. 1—6. — (SOAP 2016).

83. Apache Commons BCEL [Текст]. — URL: https://commons.apache.org/proper/commons-bcel (дата обр. 10.06.2020).

84. *Hovemeyer, D.* Finding bugs is easy [Text] / D. Hovemeyer, W. Pugh // Acm sigplan notices. — 2004. — Vol. 39, no. 12. — P. 92—106.

85. An overview of AspectJ [Text] / G. Kiczales [et al.] // European Conference on Object-Oriented Programming. — Springer. 2001. — P. 327—354.

86. LLVM Language Reference [Текст]. — URL: https://llvm.org/docs/LangRef.html (дата обр. 01.02.2020).

87. *Lattner, C.* MLIR Primer: A Compiler Infrastructure for the End of Moore's Law [Text] / C. Lattner, J. Pienaar. — 2019.

88. *Abadi, M.* A computational model for TensorFlow: an introduction [Text] / M. Abadi, M. Isard, D. G. Murray // Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages. — 2017. — P. 1—7.

89. CIL: Intermediate language and tools for analysis and transformation of C programs [Text] / G. C. Necula [et al.] // International Conference on Compiler Construction. — Springer. 2002. — P. 213—228.

90. *Dillig, I.* SAIL: Static analysis intermediate language with a two-level representation [Text] / I. Dillig, T. Dillig, A. Aiken // Technical report. — 2009.

91. *Leino, K. R. M.* This is Boogie 2 [Text] / K. R. M. Leino // manuscript KRML. — 2008. — Vol. 178, no. 131.

92. *Torlak, E.* Growing solver-aided languages with Rosette [Text] / E. Torlak, R. Bodik // Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software. — 2013. — P. 135—152.

93. *Bonetta, D.* GraalVM: metaprogramming inside a polyglot system (invited talk) [Text] / D. Bonetta // Proceedings of the 3rd ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection. — 2018. — P. 3—4.

94. Bringing low-level languages to the JVM: efficient execution of LLVM IR on Truffle [Text] / M. Rigger [et al.] // Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages. — 2016. — P. 6—15.

95. *Wimmer, C.* Truffle: a self-optimizing runtime system [Text] / C. Wimmer, T. Würthinger // Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity. — 2012. — P. 13—14.

96. *Dullien, T.* REIL: A platform-independent intermediate representation of disassembled code for static code analysis [Text] / T. Dullien, S. Porst. — 2009.

97. *Yellin, F.* The Java virtual machine specification [Text] / F. Yellin, T. Lindholm. — 1996.

98. Standard ECMA-335 - Common Language Infrastructure (CLI) [Текст]. — URL: http://www.ecma-international.org/publications/standards/Ecma-335.htm (дата обр. 30.05.2022).

99. *Eagle, C.* The IDA Pro Book, 2nd Edition [Text] / C. Eagle. — No Starch Press, 2011.

100. *Eagle, C.* The Ghidra Book: The Definitive Guide [Text] / C. Eagle, K. Nance. — No Starch Press, 2020.

101. *Mushtaq, Z.* Multilingual source code analysis: A systematic literature review [Text] / Z. Mushtaq, G. Rasool, B. Shehzad // IEEE Access. — 2017. — Vol. 5. — P. 11307—11336.

102. *Pfeiffer, R.-H.* Taming the Confusion of Languages [Text] / R.-H. Pfeiffer, A. Wąsowski // Modelling Foundations and Applications / ed. by R. B. France [et al.]. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2011. — P. 312—328.

103. *Pfeiffer, R.-H.* Language-Independent Traceability with Lässig [Text] / R.-H. Pfeiffer, J. Reimann, A. Wąsowski // Modelling Foundations and Applications / ed. by J. Cabot, J. Rubin. — Cham : Springer International Publishing, 2014. — P. 148—163.

104. Pangea: A Workbench for Statically Analyzing Multi-language Software Corpora [Text] / A. Caracciolo [et al.] // 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation. — 2014. — P. 71—76.

105. *Storm, T. van der*. The Rascal language workbench [Text] / T. van der Storm // CWI. Software Engineering [SEN]. — 2011. — Vol. 13. — P. 14.

106. *Зубов, М. В.* Применение универсальных промежуточных представлений для статического анализа исходного программного кода [Текст] / М. В. Зубов, А. Н. Пустыгин, Е. В. Старцев // Доклады Томского государственного университета систем управления и радиоэлектроники. — 2013. — 1 (27). — С. 64—68.

107. *Emerson, E. A.* Characterizing correctness properties of parallel programs using fixpoints [Text] / E. A. Emerson, E. M. Clarke // International Colloquium on Automata, Languages, and Programming. — Springer. 1980. — P. 169—181.

108. *Browne, M. C.* Characterizing finite Kripke structures in propositional temporal logic [Text] / M. C. Browne, E. M. Clarke, O. Grümberg // Theoretical computer science. — 1988. — Vol. 59, no. 1/2. — P. 115—131.

109. *Kripke, S. A.* Semantical Considerations on Modal Logic [Text] / S. A. Kripke // Acta Philosophica Fennica. — 1963. — Vol. 16.

110. *Holzmann, G.* The Spin Model Checker: Primer and Reference Manual [Text] / G. Holzmann. — Addison-Wesley, 2004.

111. ACSL: ANSI/C specification language [Text] / P. Baudin [et al.] // CEA-LIST, Saclay, France, Tech. Rep. v1. — 2008. — Vol. 2.

112. *Filliâtre, J.-C.* The Why/Krakatoa/Caduceus platform for deductive program verification [Text] / J.-C. Filliâtre, C. Marché // International Conference on Computer Aided Verification. — Springer. 2007. — P. 173—177.

113. *Beningo, J.* Documenting Firmware with Doxygen [Text] / J. Beningo // Reusable Firmware Development. — Springer, 2017. — P. 121—148.

114. *Itsykson, V.* Automated program transformation for migration to new libraries [Text] / V. Itsykson, A. Zozulya // 2011 7th Central and Eastern European Software Engineering Conference (CEE-SECR). — IEEE. 2011. — P. 1—7.

115. *Meyer, B.* Design by contract [Text] / B. Meyer. — Prentice Hall Upper Saddle River, 2002.

116. The Java Modelling Language (JML) [Текст]. — URL: http://www.eecs.ucf.edu/~leavens/JML/index.shtml (дата обр. 30.05.2022).

117. *Pnueli, A.* The temporal logic of programs [Text] / A. Pnueli // 18th Annual Symposium on Foundations of Computer Science (SFCS 1977). — 1977. — P. 46—57.

118. *Pnueli, A.* The temporal semantics of concurrent programs [Text] / A. Pnueli // Theoretical computer science. — 1981. — Vol. 13, no. 1. — P. 45—60.

119. *Büchi, J. R.* On a decision method in restricted second order arithmetic [Text] / J. R. Büchi // The Collected Works of J. Richard Büchi. — Springer, 1990. — P. 425—435.

120. *Emerson, E.* The design and synthesis of synchronization skeletons using temporal logic [Text] / E. Emerson // Workshop on Logics of Programs. Vol. 131. — 1981. — P. 52—72.

121. Symbolic model checking without BDDs [Text] / A. Biere [et al.] // International conference on tools and algorithms for the construction and analysis of systems. — Springer. 1999. — P. 193—207.

122. *De Moura, L.* Satisfiability modulo theories: introduction and applications [Text] / L. De Moura, N. Bjørner // Communications of the ACM. — 2011. — Vol. 54, no. 9. — P. 69—77.

123. DPLL (T): Fast decision procedures [Text] / H. Ganzinger [et al.] // International Conference on Computer Aided Verification. — Springer. 2004. — P. 175—188.

124. *Marques Silva, J. P.* GRASP—a new search algorithm for satisfiability [Text] / J. P. Marques Silva, K. A. Sakallah // The Best of ICCAD. — Springer, 2003. — P. 73—89.

125. *Rival, X.* Introduction to Static Analysis: An Abstract Interpretation Perspective [Text] / X. Rival, K. Yi. — MIT Press, 2020.

126. *Komuravelli, A.* SMT-based model checking for recursive programs [Text] / A. Komuravelli, A. Gurfinkel, S. Chaki // Formal Methods in System Design. — 2016. — Vol. 48, no. 3. — P. 175—205.

127. *Kostyukov, Y.* Beyond the elementary representations of program invariants over algebraic data types [Text] / Y. Kostyukov, D. Mordvinov, G. Fedyukovich // Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. — 2021. — P. 451—465.

128. RInGen: Regular Invariant Generator [Текст]. — URL: https://github.com/Columpio/RInGen (дата обр. 30.05.2022).

129. A survey of symbolic execution techniques [Text] / R. Baldoni [et al.] // ACM Computing Surveys (CSUR). — 2018. — Vol. 51, no. 3. — P. 1—39.

130. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. [Text] / C. Cadar, D. Dunbar, D. R. Engler, [et al.] // OSDI. Vol. 8. — 2008. — P. 209—224.

131. UnitTestBot [Текст]. — URL: https://unittestbot.github.io (дата обр. 30.05.2022).

132. *Reynolds, J. C.* Separation logic: A logic for shared mutable data structures [Text] / J. C. Reynolds // Proceedings 17th Annual IEEE Symposium on Logic in Computer Science. — IEEE. 2002. — P. 55—74.

133. *O'Hearn, P.* Separation logic [Text] / P. O'Hearn // Communications of the ACM. — 2019. — Vol. 62, no. 2. — P. 86—95.

134. Building your own modular static analyzer with Facebook Infer [Текст]. — URL: https://pldi17.sigplan.org/details/pldi-2017-workshops-and-tutorials/9/Building-your-own-modular-static-analyzer-with-Facebook-Infer (дата обр. 30.05.2022).

135. *Love, R.* Linux System Programming: Talking Directly to the Kernel and C Library [Text] / R. Love. — O'Reilly Media, Incorporated, 2013.

136. *Russinovich, D.* Windows Internals, Sixth Edition, Part 1 [Text] / D. Russinovich, A. Ionescu. — Microsoft Press, 2012.

137. *Tanenbaum, A.* Computer Networks, eBook, Global Edition [Text] / A. Tanenbaum, D. Wetherall. — Pearson Education, 2021.

138. *Benvenuti, C.* Understanding Linux Network Internals [Text] / C. Benvenuti. — O'Reilly Media, Incorporated, 2006. — (Guided tour to networking on Linux).

139. Langserver.org - A community-driven source of knowledge for Language Server Protocol implementations [Текст]. — URL: https://langserver.org (дата обр. 30.03.2021).

140. Reactive Distributed communication framework [Текст]. — URL: https://github.com/JetBrains/rd.git (дата обр. 30.05.2022).

141. *Currid, A.* TCP offload to the rescue: Getting a toehold on TCP offload engines—and why we need them [Text] / A. Currid // Queue. — 2004. — Vol. 2, no. 3. — P. 58—65.

142. *Masse, M.* REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces [Text] / M. Masse. — O'Reilly Media, 2011.

143. Mastering API Architecture [Text] / J. Gough [et al.]. — O'Reilly Media, Incorporated, 2022.

144. *Hintjens, P.* ZeroMQ: messaging for many applications [Text] / P. Hintjens. — " O'Reilly Media, Inc.", 2013.

145. *Kleppmann, M.* Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems [Text] / M. Kleppmann. — O'Reilly Media, 2017.

146. *Knuth, D. E.* Backus normal form vs. backus naur form [Text] / D. E. Knuth // Communications of the ACM. — 1964. — Vol. 7, no. 12. — P. 735—736.

147. *Меньшиков, М. А.* Эффективная трансляция направленных ациклических графов программ в промежуточное представление [Текст] / М. А. Меньшиков // Процессы управления и устойчивость. — 2020. — Т. 7, № 1. — С. 271—275.

148. *Meyers, S.* Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14 [Text] / S. Meyers. — O'Reilly Media, 2014.

149. *Seacord, R.* Effective C: An Introduction to Professional C Programming [Text] / R. Seacord. — No Starch Press, 2020.

150. *Меньшиков, М. А.* Систематизация абстракций для поддержки различных языков программирования и сред в статических анализаторах кода [Текст] / М. А. Меньшиков // Процессы управления и устойчивость. — 2021. — Т. 8, № 1. — С. 294—302.

151. *Cockshott, P.* SIMD Programming Manual for Linux and Windows [Text] / P. Cockshott, K. Renfrew. — Springer London, 2013. — (Springer Professional Computing).

152. *Kusswurm, D.* Modern Arm Assembly Language Programming: Covers Armv8-A 32-bit, 64-bit, and SIMD [Text] / D. Kusswurm. — Apress, 2020.

153. *Britton, R.* MIPS: assembly language programming [Text] / R. Britton. — Prentice Hall, 2004.

154. *Harris, D.* Digital Design and Computer Architecture [Text] / D. Harris, S. Harris. — Elsevier Science, 2010. — (Computer organization bundle, VHDL Bundle).

155. *Buonanno, E.* Functional Programming in C#, Second Edition [Text] / E. Buonanno. — Manning, 2022.

156. *Bock, J.* CIL Programming: Under the Hood of .NET [Text] / J. Bock. — Apress, 2008. — (.NET developer series).

157. *Syme, D.* Expert F# 4.0 [Text] / D. Syme, A. Granicz, A. Cisternino. — Apress, 2015.

158. *Меньщиков, М. А.* Применение модели MapReduce в архитектуре статического анализатора кода [Текст] / М. А. Меньщиков, Т. А. Лепихин // Процессы управления и устойчивость. — 2017. — Т. 4, № 1. — С. 433—444.

159. *Horton, I.* Beginning C++20: From Novice to Professional [Text] / I. Horton, P. Van Weert. — Apress, 2020.

160. *Gayoso Martı□nez, V.* State of the art in similarity preserving hashing functions [Text] / V. Gayoso Martı□nez, F. Hernández Álvarez, L. Hernández Encinas. — 2014.

161. MongoDB Fundamentals: A hands-on guide to using MongoDB and Atlas in the real world [Text] / A. Phaltankar [et al.]. — Packt Publishing, 2020.

162. *Meier, A.* SQL & NoSQL Databases: Models, Languages, Consistency Options and Architectures for Big Data Management [Text] / A. Meier, M. Kaufmann. — Springer Fachmedien Wiesbaden, 2019.

163. *Menshikov, M.* Midair: An Intermediate Representation for Multi-purpose Program Analysis [Text] / M. Menshikov // International Conference on Computational Science and Its Applications. — Springer. 2020. — P. 544—559.

164. *Menshchikov, M.* Scalable semantic virtual machine framework for language-agnostic static analysis [Text] / M. Menshchikov // Distributed Computing and Grid-technologies in Science and Education. — 2018. — P. 213—217.

165. *Menshikov, M.* Towards a Resident Static Analysis [Text] / M. Menshikov // International Conference on Computational Science and Its Applications. — Springer. 2019. — P. 62—71.

166. *Padlewski, P.* Devirtualization in LLVM [Text] / P. Padlewski // Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity. — 2017. — P. 42—44.

167. *Godbolt, M.* Optimizations in C++ Compilers: A practical journey [Text] / M. Godbolt // Queue. — 2019. — Vol. 17, no. 5. — P. 69—100.

168. *Von Neumann, J.* First Draft of a Report on the EDVAC [Text] / J. Von Neumann // IEEE Annals of the History of Computing. — 1993. — Vol. 15, no. 4. — P. 27—75.

169. ASLR on the Line: Practical Cache Attacks on the MMU. [Text] / B. Gras [et al.] // NDSS. Vol. 17. — 2017. — P. 26.

170. *Wang, W.* Partitioned memory models for program analysis [Text] / W. Wang, C. Barrett, T. Wies // International Conference on Verification, Model Checking, and Abstract Interpretation. — Springer. 2017. — P. 539—558.

171. *Xu, Z.* A memory model for static analysis of C programs [Text] / Z. Xu, T. Kremenek, J. Zhang // International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. — Springer. 2010. — P. 535—548.

172. A survey of prevention/mitigation against memory corruption attacks [Text] / T. Saito [et al.] // 2016 19th International Conference on Network-Based Information Systems (NBiS). — IEEE. 2016. — P. 500—505.

173. Exploring C semantics and pointer provenance [Text] / K. Memarian [et al.] // Proceedings of the ACM on Programming Languages. — 2019. — Vol. 3, POPL. — P. 1—32.

174. *Steensgaard, B.* Points-to analysis in almost linear time [Text] / B. Steensgaard // Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. — 1996. — P. 32—41.

175. Counterexample-guided abstraction refinement for symbolic model checking [Text] / E. Clarke [et al.] // Journal of the ACM (JACM). — 2003. — Vol. 50, no. 5. — P. 752—794.

176. Document management — Portable document format — Part 2: PDF 2.0 [Text] : Standard / International Organization for Standardization. — Geneva, CH, 12/2020.

177. *Knuth, D. E.* Breaking paragraphs into lines [Text] / D. E. Knuth, M. F. Plass // Software: Practice and Experience. — 1981. — Vol. 11, no. 11. — P. 1119—1184.

178. *Dowd, M.* The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities [Text] / M. Dowd, J. McDonald, J. Schuh. — Pearson Education, 2006.

179. *Higo, Y.* Enhancing quality of code clone detection with program dependency graph [Text] / Y. Higo, S. Kusumoto // 2009 16th Working Conference on Reverse Engineering. — IEEE. 2009. — P. 315—316.

180. *Gautam, P.* Non-trivial software clone detection using program dependency graph [Text] / P. Gautam, H. Saini // International Journal of Open Source Software and Processes (IJOSSP). — 2017. — Vol. 8, no. 2. — P. 1—24.

181. *Calder, B.* Quantifying behavioral differences between C and C++ programs [Text] / B. Calder, D. Grunwald, B. Zorn // Journal of Programming languages. — 1994. — Vol. 2, no. 4. — P. 313—351.

182. *Von Hagen, W.* The definitive guide to GCC [Text] / W. Von Hagen. — Apress, 2011.

183. Guesslang documentation [Текст]. — URL: https://guesslang.readthedocs.io/en/latest (дата обр. 30.05.2022).

184. Detecting programming language from source code using bayesian learning techniques [Text] / J. N. Khasnabish [et al.] // International Workshop on Machine Learning and Data Mining in Pattern Recognition. — Springer. 2014. — P. 513—522.

185. Automatically assessing vulnerabilities discovered by compositional analysis [Text] / S. Ognawala [et al.] // Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis. — 2018. — P. 16—25.

186. The Race to the Vulnerable: Measuring the Log4j Shell Incident [Text] / R. Hiesgen [et al.]. — 2022. — URL: https://arxiv.org/abs/2205.02544.

187. *Меньшиков, М. А.* Подход к организации хранения семантики в статическом анализаторе кода [Text] / М. А. Меньшиков // Процессы управления и устойчивость. — 2019. — Vol. 6, no. 1. — P. 313—320.

188. *Holzmann, G. J.* Cobra: a light-weight tool for static and dynamic program analysis [Text] / G. J. Holzmann // Innovations in Systems and Software Engineering. — 2017. — Mar. 1. — Vol. 13, no. 1. — P. 35—49. — URL: https://doi.org/10.1007/s11334-016-0282-x.

189. Improving GitHub code search [Текст]. — URL: https://github.blog/2021-12-08-improving-github-code-search (дата обр. 30.05.2022).

190. What questions remain? an examination of how developers understand an interactive static analysis tool [Text] / T. W. Thomas [et al.] // Twelfth Symposium on Usable Privacy and Security (SOUPS 2016). — 2016.

191. Advanced compiler design implementation [Text] / S. Muchnick [et al.]. — Morgan kaufmann, 1997.

192. All File Systems Are Not Created Equal: On the Complexity of Crafting {Crash-Consistent} Applications [Text] / T. S. Pillai [et al.] // 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). — 2014. — P. 433—448.

193. *Lee, H.-c.* Experimenting with system and Libc call interception attacks on ARM-based Linux kernel [Text] / H.-c. Lee, C. H. Kim, J. H. Yi // Proceedings of the 2011 ACM Symposium on Applied Computing. — 2011. — P. 631—632.

194. *Lopes, B.* Getting Started with LLVM Core Libraries [Text] / B. Lopes, R. Auler. — Packt Publishing Limited, 2014. — (Community experience distilled).

195. *Andriesse, D.* Practical Binary Analysis: Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly [Text] / D. Andriesse. — No Starch Press, 2018.

196. *Singh, A.* Mac OS X Internals: A Systems Approach [Text] / A. Singh. — Pearson Education, 2006.

197. *Brubacher, D.* Detours: Binary interception of Win32 functions [Text] / D. Brubacher // Windows NT 3rd Symposium (Windows NT 3rd Symposium). — 1999.

198. *Langr, J.* Modern C++ Programming with Test-Driven Development: Code Better, Sleep Better [Text] / J. Langr. — Createspace Independent Pub, 2014.

199. *Menshikov, M. A.* Static analyzer debugging and quality assurance approaches [Текст] / M. A. Menshikov // Труды института системного программирования РАН. — 2020. — Т. 32, № 3. — С. 33—47.

200. *Berner, D.* CMake Best Practices: Discover proven techniques for creating and maintaining programming projects with CMake [Text] / D. Berner, M. Gilor. — Packt Publishing, 2022.

201. Building Secure and Reliable Systems: Best Practices for Designing, Implementing, and Maintaining Systems [Text] / H. Adkins [et al.]. — O'Reilly Media, 2020.

202. *Li, J.* Fuzzing: a survey [Text] / J. Li, B. Zhao, C. Zhang // Cybersecurity. — 2018. — Vol. 1, no. 1. — P. 1—13.

203. *Miell, I.* Docker in practice [Text] / I. Miell, A. Sayers. — Simon, Schuster, 2019.

204. *Smith, R.* Linux in a Windows World [Text] / R. Smith. — O'Reilly, 2005. — (O'Reilly Series).

205. DarlingHQ [Текст]. — URL: https : / / www . darlinghq . org (дата обр. 30.05.2022).

206. *Shiraishi, S.* Test suites for benchmarks of static analysis tools [Text] / S. Shiraishi, V. Mohan, H. Marimuthu // 2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). — 2015. — P. 12—15.

207. *Dean, J.* MapReduce: simplified data processing on large clusters [Text] / J. Dean, S. Ghemawat // Communications of the ACM. — 2008. — Vol. 51, no. 1. — P. 107—113.

208. *Calavera, D.* Linux Observability with BPF: Advanced Programming for Performance Analysis and Networking [Text] / D. Calavera, L. Fontana. — O'Reilly Media, 2019.

209. *Меньшиков, М. А.* Статический анализ РуСи [Текст] / М. А. Меньшиков //. — ВВМ, 2019. — С. 12—18.

210. *Меньщиков, М. А.* Определение контекстов выполнения пользовательских функций в программном коде [Текст] / М. А. Меньщиков, Т. А. Лепихин // Процессы управления и устойчивость. — 2016. — Т. 3, № 1. — С. 435—439.

Приложение А

Схема работы анализатора

Полная схема работы анализатора приведена на Рис. А.1.

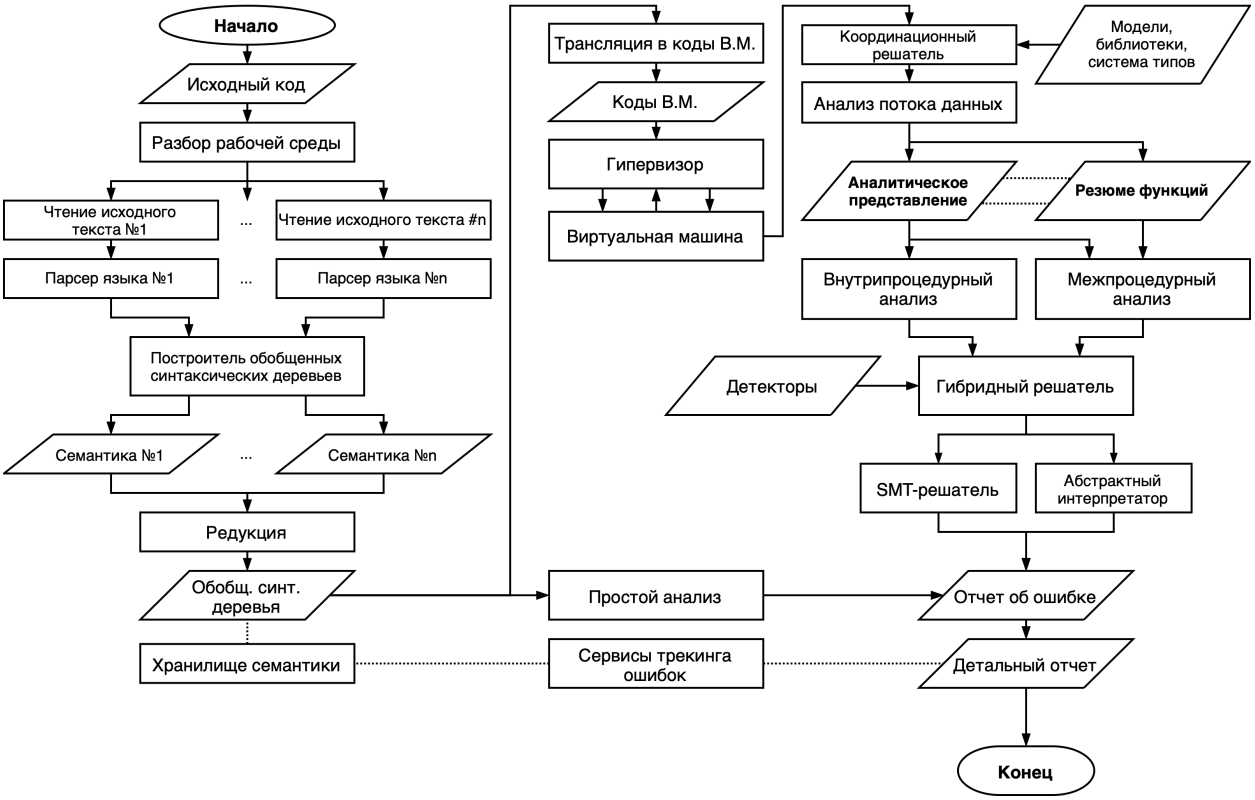

Рисунок А.1 — Полная схема работы анализатора

Приложение Б

Поддерживаемые дополнения

В анализаторе реализовываются следующие плагины:

1. **Компиляторы**: Clang, Clang-CL, GCC, GCC-Like (поддержка компиляторов, похожих на GCC), RuC.
2. **Процессоры**: ARM, ARM64, x86_64, x86, RuC-VM, MIPS (Little Endian), MIPS, MIPSEL64, MIPS64.
3. **Операционные системы**: Linux, Windows, MacOS, RuC-VM, Baremetal.
4. **Языковые семантики**: C, C++, RuC.
5. **Модули чтения исходных текстов**: C (Clang), C++ (Clang), RuC-Native (через собственные механизмы РуСи), RuC (Clang), GCC-Like, Clang, Clang++.
6. **Решатели**: CVC4, Z3 (экспериментальный), абстрактная интерпретация, *вычислитель*, *гибридный решатель*.
7. **Стандартные библиотеки**: C/C++, RuC, общая библиотека.

Приложение В

Обнаруживаемые классы ошибок

1. **Нарушения контрактов**, заданных аннотациями ACSL (в том числе встроенными в Doxygen), РуСи. К этому же классу относятся **ошибки использования стандартной библиотеки**: ошибки printf/scanf, некорректные блокировки, нарушения в работе с файлами и хэндлами. Используется комбинация вычислителя и SMT-решателя.
2. **Логические ошибки**: бессмысленные сравнения, применение неизвестного порядка вычисления, некорректное использование операций, неправильные приведения типов, некорректные аргументы и т.п. Как правило, для таких проверок достаточно вычислителя.
3. **Ошибки памяти**: утечки памяти, использование неинициализированных данных, аллокации пустых блоков, двойная деаллокация блоков, выход за границы массивов. Используются вычислитель, теги памяти из аналитического представления и SMT-решатель для проверки условий.
4. **Достижимый и недостижимый код**: всегда выполняемые и не выполняемые условия, неиспользуемые переменные и т.п. Необходимо использовать SMT-решатель.
5. **Ошибки уровня абстрактного синтаксического дерева**: повторяющиеся операнды и т.п.
6. **Арифметические ошибки**: деление на ноль, неправильные сдвиги, некорректные битовые маски, переполнения, сравнения чисел с плавающей точкой. Для выявления ошибок достаточно вычислителя.

Приложение Г

Схема работы модуля чтения

На Рис. Г.1 приведена упрощенная схема работы модуля чтения.

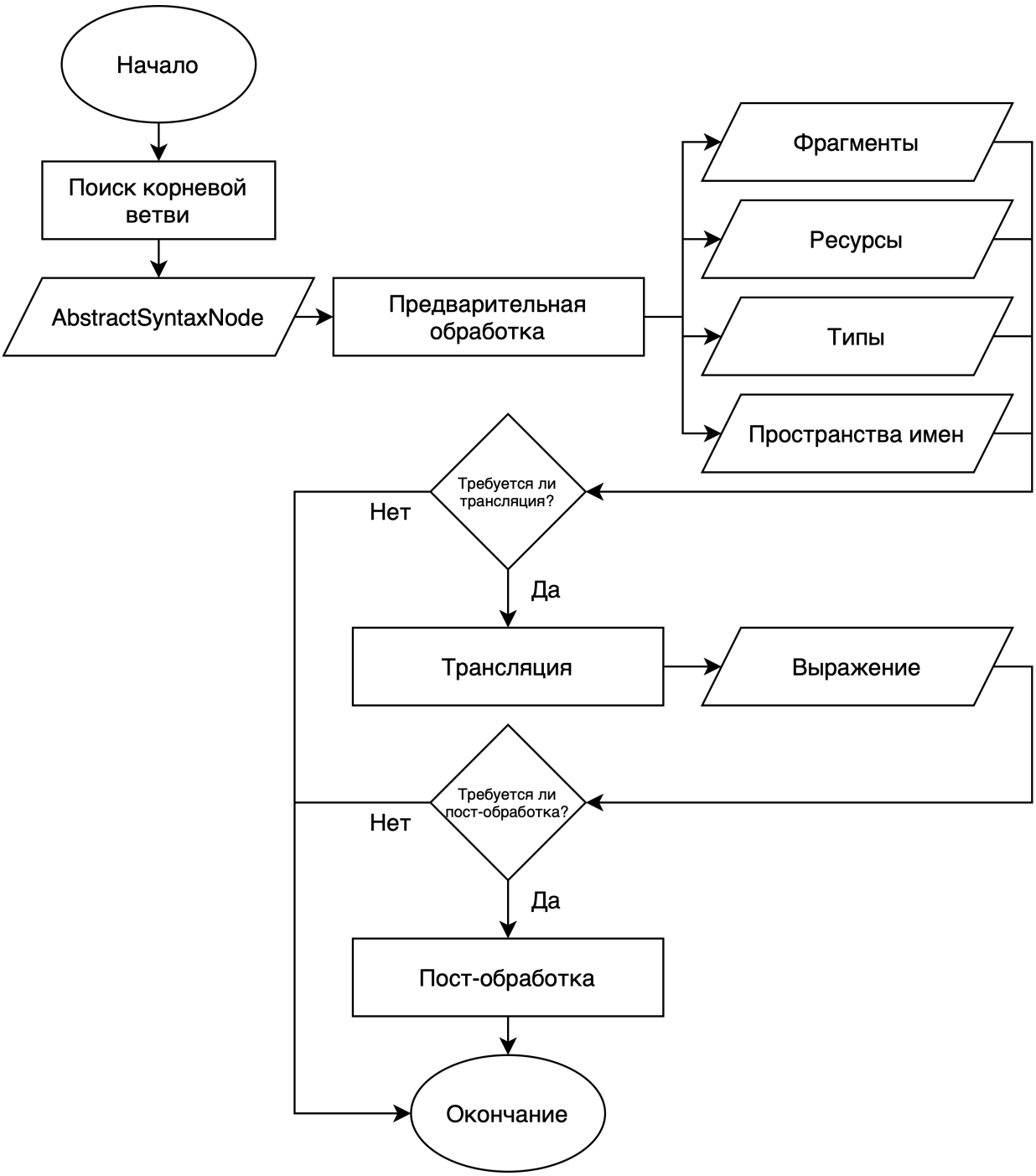

Рисунок Г.1 — Схема работы модуля чтения

Приложение Д

Универсальный языковой базис

Языковой базис состоит из следующих *выражений*:

1. **ArraySubscript** — доступ к элементам массивов.
2. **Binary** — бинарные выражения.
3. **Cast** — конверсии типов.
4. **Compound** — блоки, блоки с результатами.
5. **CompoundLiteral** — структурный литерал.
6. **ConstValue**, **ConstReference** — константные значения и ссылки на константные переменные.
7. **Declaration** — декларация переменной.
8. **ImplicitInit** — неявная инициализация.
9. **InitList** — список инициализации.
10. **Invocation** — вызов функций.
11. **Literal** — литерал численного и строкового типа.
12. **MemberAccess** — доступ к элементам структур.
13. **Paren** — скобочные выражения.
14. **PostfixUnary** — постфиксно-унарные выражения.
15. **Predefined**— заранее определенное выражение (например, $True$ в ACSL).
16. **Predicate** — предикатное выражение.
17. **PrefixUnary** — префиксно-унарные выражения.
18. **Quantified** — выражения с кванторами.
19. **Reference** — ссылки на объекты.
20. **Return** — возврат значения из функции.
21. **Set** — множества.
22. **Ternary** — тернарные выражения.
23. **UnassignedDeclaration** — не сопоставленные с реальными объектами декларации.
24. **UnassignedReference** — не сопоставленные с реальными объектами ссылки.
25. **With** — корректировка структурного объекта или массива через присваивание поля структурного объекта или элемента массива с возвратом самого объекта.

В дополнение к ним, вводится несколько внутренних выражений:

1. **Annotation** — аннотации к объектам.
2. **Internal** — внутренние выражения.
3. **Located** — выражение, прикрепленное к определенной точке в аналитической модели.
4. **LocationConstraint** — выражение, обозначающее конъюнкцию условий в определенной точке программы.

5. **Variable** — разовые переменные.

Также в языковом базисе требуются концепции фрагментов и ресурсов:

1. **Ресурсы** — переменные и функции.
2. **Фрагменты** — описатели линейных блоков, имеющие:
    а) тип условия входа (простое ветвление; циклы for, while, do);
    б) условие входа в блок;
    в) установленная метка для goto;
    г) требования к следующему фрагменту (следующая метка goto; метки прерывания и продолжения конструкций break, continue; явное указание на следующий фрагмент).

Приложение Е

Принцип работы подсистемы виртуальных машин

На рисунке Е.1 представлена схема работы подсистемы гипервизора и виртуальных машин.

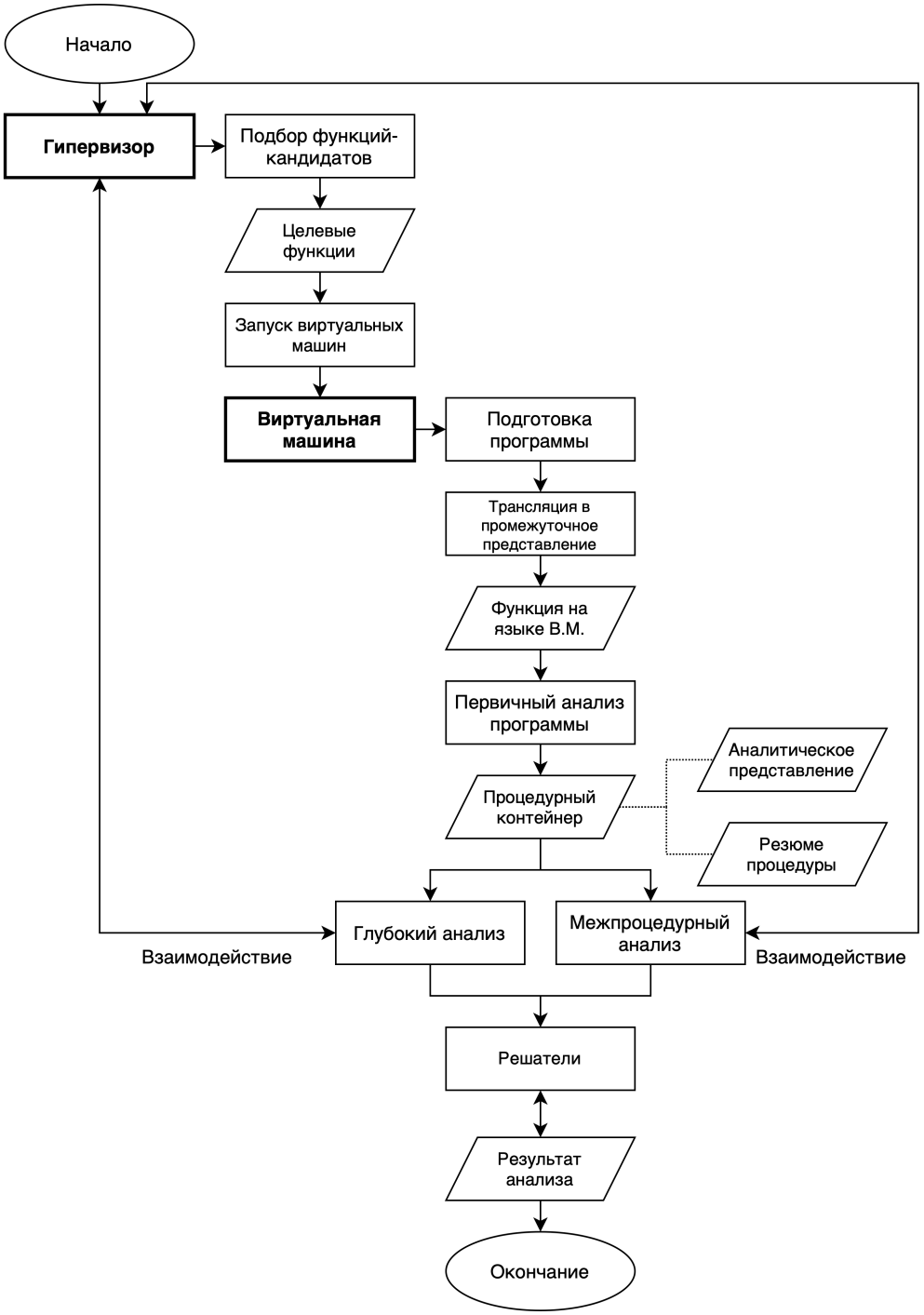

Рисунок Е.1 — Схема работы гипервизора и виртуальных машин

Приложение Ж

Команды языка Midair

Существует несколько основных категорий команд:

1. **Команды, относящиеся к анализу**. Данный набор команд является специфичным для анализа. Основное его назначение: скорректировать результаты проверки выполнимости систем неравенств в процессе анализа. Команды приведены в таблице 7.
2. **Команды для изменения потока управления**. Команды из этого перечня влияют на поток управления. Анализатор использует их для оптимизации процесса проверки, для формирования корректных с точки зрения потока управления систем неравенств. Команды приведены в таблице 8.
3. **Команды по манипулированию переменными**. Команды из этого перечня (таблица 9) создают, загружают и изменяют переменные.
4. **Внутренние команды**. Различные внутренние команды, которые генерируются анализатором в служебных целях. Приведены в таблице 10.

Таблица 7 — Команды, относящиеся к анализу

Команда	Характеристика
constraint *выражение*	Установить требование об обязательном выполнении *выражения*.
check *выражение*	Установить требование об обязательном невыполнении *выражения*.
annotate (*аннотация*)	Подключить ACSL-спецификацию к следующей инструкции.
assume *выражение*	Принять факт, что *выражение* выполняется.

Таблица 8 — Команды для изменения потока управления

Команда	Характеристика
enter → *var*	Начать блок, определяющий видимость переменных. *var* определяет переменную, куда будет помещен результат работы функции.
exit	Закончить блок, определяющий видимость переменных.
branch *cond* **[loop] [post]** ***followup***	Начать ветвление, где *followup* может быть *or continue* (указывает на ветви с возможным else-блоком), *fallthrough* (указывает на ветвь вида switch), or *follows inc* (для инкременторов).
end branch *[obj1, …, objN]*	Закончить ветвление с указанием на объекты, которые изменились в ветке. Данная информация может использовать для удаления лишних переменных.
invoke [*var* **=] *fn*(*arg1*, …, *argN*)**	Вызывает функцию *fn* и сохраняет результат в *var*. Эффект отличается для внутрипроцедурного и межпроцедурного видов анализа.
return *expression*	Присвоить значение переменной, которая указана в блоке **enter**.
context *contextval*	Назначить текущему исполнению контекст *contextval*

Таблица 9 — Команды по манипулированию переменными

Команда	Характеристика
declare *res* **[**= *expr***]**	Добавить ресурс с данным ID в текущую область видимости.
load *res*	Загрузить ресурс с данным ID, что полезно в том случае, если данные о переменной физически располагаются на другом вычислительном узле.
init *var* **:=** *expr*	Установить значение временной переменной. Это присваивание может быть пропущено, если команда выполняется на вычислительном узле, где уже проведено упрощение переменных.
assign *res* = *expr*	Установить значение переменной или объекта.
alias *res* = *expr*	Назначить объект *res* псевдонимом *expr*.
fuse *res*	Выполнить смешивание значений переменных в последнем ветвлении.
widen *res* = *expr*	Выполнить операцию расширения диапазона значений *res* с учетом диапазона *expr*.
narrow *res* = *expr*	Выполнить операцию сужения диапазона значений *res* с учетом диапазона *expr*.
tag *res* = *tagval*	Назначить тег *tagval* на объект *res*.

Таблица 10 — Внутренние команды

Команда	Характеристика
system *internal-expression*	Вызвать внутреннюю команду для манипулирования потоком данных или потоком управления. Например, одна из команд — assert.
augment *name***: (***data***)**	Добавить внешний объект к потоку данных.
semantics *language*	Изменить текущую языковую семантику.

Приложение 3

Типы и их размеры

В таблице 11 представлены основные поддерживаемые анализатором базовые типы данных.

Таблица 11 — Типы и их размеры

Тип	Характеристика	Размер (битов)
Void	«Пустой» тип	0
Pointer16	Указатель	16
Pointer32	Указатель	32
Pointer64	Указатель	64
Pointer128	Указатель	128
Boolean	Логический тип	1 (8)
U8	Беззнаковое целое	8
U16	Беззнаковое целое	16
U32	Беззнаковое целое	32
U64	Беззнаковое целое	64
U128	Беззнаковое целое	128
S8	Знаковое целое	8
S16	Знаковое целое	16
S32	Знаковое целое	32
S64	Знаковое целое	64
S128	Знаковое целое	128
Float	Число с плавающей точкой	32
Double	Число с плавающей точкой	64
Float80	Число с плавающей точкой	80
Float128	Число с плавающей точкой	128

Приложение И

Отображение конструкций C++ в языковой базис

В таблице 12 приводятся синтаксические конструкции C++ и их эквиваленты в терминах обобщенных синтаксических деревьев.

Таблица 12 — Конструкции C++ и их эквиваленты в терминах обобщенных синтаксических деревьев

Языковая конструкция	Итог трансляции
BinaryOperator	Бинарный оператор
CompoundAssignOperator	–//–
UnaryExprOrTypeTraitExpr	Унарный оператор
UnaryOperator	Префиксный или постфиксный унарные операторы
ConditionalOperator	Тернарный оператор
BinaryConditionalOperator	–//–
DeclStmt	Разбивается на множество отдельных деклараций
DeclRefExpr	ReferenceExpression на целевой ресурс
NullStmt	Выражение типа Nop
ImaginaryLiteral	Временно игнорируется по причине отсутствия интереса к трансляции конструкции
CharacterLiteral	Литерал типа S8/U8
CXXNullPtrLiteralExpr	Нулевой литерал типа U32
GNUNullExpr	–//–
noexcept	Игнорируется
CXXBoolLiteralExpr	Литерал типа Boolean
IntegerLiteral	Литерал из семейства SignedInteger или UnsignedInteger
FloatingLiteral	Литерал из семейства Float
StringLiteral	Строковый литерал
CXXMemberCallExpr	Заменяется на Invocation с this-параметром
CXXOperatorCallExpr	–//–
CallExpr	Заменяется на Invocation
CXXTemporaryObjectExpr	Формируются ссылки на временные ресурсы целевого типа
CXXConstructExpr	–//–
ImplicitCastExpr	Cast с нужным подвидом приведения типа
CStyleCastExpr	–//–

Продолжение таблицы 12

Языковая конструкция	Итог трансляции
CXXStaticCastExpr	–//–
CXXReinterpretCastExpr	–//–
CXXConstCastExpr	–//–
CXXAddrspaceCastExpr	–//–
CXXFunctionalCastExpr	–//–
BuiltinBitCastExpr	–//–
CXXDynamicCastExpr	–//–
CXXScalarValueInitExpr	Результат — ресурс целевого типа
CXXUuidofExpr	–//–
ConceptSpecializationExpr	–//–
ShuffleVectorExpr	–//–
ConvertVectorExpr	–//–
PseudoObjectExpr	Пробрасывается подвыражение
ConstantExpr	–//–
CXXBindTemporaryExpr	–//–
CXXDefaultArgExpr	–//–
AttributedStmt	–//–
CapturedStmt	–//–
CXXDefaultInit	–//–
ExprWithCleanups	–//–
MaterializeTemporaryExpr	–//–
MemberExprClass	Выражение MemberAccess
ImplicitValueInitExpr	Выражение ImplicitInit
ParenExpr	Выражение Paren
ArraySubscriptExpr	Выражение ArraySubscript
InitListExpr	Выражение InitList
CXXStdInitializerListExpr	Приведение подвыражения к целевому типу через Cast
CompoundLiteralExpr	Выражение CompoundLiteral
SizeOfPackExpr	Формируется ресурс целевого типа
VAArgExpr	Формируется ресурс целевого типа
PredefinedExpr	Выражение Predefined
StmtExpr	Выражение Compound
IfStmt	Обычно обрабатывается высокоуровневым транслятором, но в редких случаях трансформируется в тернарный оператор
CXXThisExpr	Формируется ссылка на ресурс this, содержащийся в пространстве имен текущего метода

Продолжение таблицы 12

Языковая конструкция	Итог трансляции
CXXTypeId	Префиксно-унарное выражение с соответствующей операцией.
SubstNonTypeTemplateParmExpr	Пробрасывается подстановка
CXXNewExpr	Создается пустой ресурс целевого типа с названием “!newVariable’
CXXDeleteExpr	Формируется вызов псевдо-функции “!delete_call”
AtomicExpr	В настоящее время пробрасывается подвыражение
TypeTraitExpr	Значение type trait транслируется в численный литерал
LambdaExpr	Транслируются в отдельные классы
OffsetOfExpr	Модулем чтения вычисляется реальное значение и транслируется в численный литерал
CXXTryStmt (Try)	Формируется блок со свойством try
SEHTryStmt	Формируется блок со свойством try
SEHFinallyStmt	Формируется блок со свойством finally
CXXRewrittenBinaryOperator	Пробрасывается семантическая форма оператора
UnresolvedLookup	Формируется неразрешенная ссылка
CXXPseudoDestructor	–//–
CXXInheritedCtorInitExpr	–//–
DependentScopeDeclRefExpr	–//–
GCCAsmtStmt	Временно игнорируется до появления поддержки ассемблера, но в целом поддерживается через обобщенное дерево
CXXFoldExpr	Преобразуются в Nop
CXXThrowExpr	–//–
MSAsmtStmt	–//–
CoreturnStmt	–//–
SEHExceptStmt	–//–
SEHLeaveStmt	–//–
AddrLabelExpr	–//–
ArrayInitLoopExpr	–//–
ArrayInitIndexExpr	–//–
ArrayTypeTraitExpr	–//–
NoInitExpr	–//–
NoStmt	Пробрасывается в высокоуровневый транслятор для преобразования в фрагменты и свойства фрагментов.
BlockExpr	–//–

Продолжение таблицы 12

Языковая конструкция	Итог трансляции
WhileStmt	–//–
BreakStmt	–//–
CXXCatchStmt	–//–
CXXForRange	–//–
CompoundStmt	–//–
ContinueStmt	–//–
DoStmt	–//–
ForStmt	–//–
GotoStmt	–//–
IndirectGotoStmt	–//–
CaseStmt	–//–
SwitchStmt	–//–
LabelStmt	–//–
ReturnStmt	–//–

Приложение К

Отображение конструкций РуСи в языковой базис

В таблице 13 приводятся синтаксические конструкции РуСи и их эквиваленты в терминах обобщенных синтаксических деревьев.

Таблица 13 — Конструкции РуСи и их эквиваленты в терминах обобщенных синтаксических деревьев

Языковая конструкция	Итог трансляции
OP_IDENTIFIER	Ссылка на ресурс
OP_LITERAL	Литералы типов «нулевой указатель», «символ», «целое число», «число с плавающей точкой», «массив», «перечисление»
OP_CALL	Выражение Invocation
OP_SELECT	Выражение MemberAccess
OP_SLICE	Выражение ArraySubscript
OP_CAST	Выражение Cast для приведения подвыражения к целевому типу
OP_UNARY	Префиксное или постфиксное унарное выражение (PrefixUnary, PostfixUnary)
OP_BINARY	Выражение Binary
OP_TERNARY	Выражение Ternary
OP_ASSIGNMENT	Выражение Binary с присваиванием
OP_INITIALIZER	Выражение InitList
OP_CASE	Обрабатывается высокоуровневым транслятором
OP_DEFAULT	–//–
OP_BLOCK	–//–
OP_IF	–//–
OP_SWITCH	–//–
OP_WHILE	–//–
OP_DO	–//–
OP_FOR	–//–
OP_CONTINUE	–//–
OP_BREAK	–//–
OP_RETURN	–//–
OP_DECL_STMT	Транслируется в множество деклараций переменных
OP_DECL_VAR	Обрабатывается высокоуровневым транслятором

Продолжение таблицы 13

Языковая конструкция	Итог трансляции
OP_DECL_TYPE	Обрабатывается высокоуровневым транслятором
OP_DECL_DEF	Обрабатывается высокоуровневым транслятором
OP_NOP	Игнорируется

Приложение Л

Отображение конструкций ACSL в языковой базис

В таблице 14 приводятся синтаксические конструкции ANSI C Specification Language и их эквиваленты в терминах обобщенных синтаксических деревьев. Основанием для выбора названия языковой конструкции служит грамматика в формате ANTLR[1], используемая в проекте.

Таблица 14 — Термы и предикаты ACSL и их эквиваленты в терминах обобщенных синтаксических деревьев

Языковая конструкция	Итог трансляции
id	Ссылка на ресурс
string	Строковый литерал
bin_op	Бинарная операция — свойство выражения Binary
unary_op	Унарная операция — свойство выражений PrefixUnary и PostfixUnary
literal	Численный литерал
poly_id	Ссылка на ресурс
unary_op_term	Выражения PrefixUnary/PostfixUnary
binary_op_term	Выражения Binary
array_access_term	Выражение ArraySubscript
array_func_modifier	Выражение With
struct_field_access_term	Выражение MemberAccess
field_func_modifier_term	Выражение With
pointer_struct_field_access_term	Выражение MemberAccess
cast_term	Выражение Cast
func_application_term	Выражение Invocation
parentheses_term	Выражение Paren
ternary_cond_term	Выражение Ternary
local_binding_term	Генерация временной переменной посредством высокоуровневого транслятора
sizeof_term	Выражение PrefixUnary с операцией SizeOf
sizeof_type_term	–//–
syntactic_naming_term	Не реализовано
old_term	Выражение PrefixUnary с операцией Old
result_term	Выражение Predefined с операцией Result

[1] https://github.com/maximmenshikov/acsl-grammar

Продолжение таблицы 14

Языковая конструкция	Итог трансляции
null_term	Выражение Predefined с операцией Null
base_addr_term	Выражение PrefixUnary с соответствующей операцией
block_length_term	–//–
length_term	–//–
offset_term	–//–
allocation_term	–//–
exit_status_term	Выражение Predefined
at_term	Выражение Located
rel_op	Бинарная операция — свойство выражения Binary
logical_true_pred	Литерал типа Boolean со значением 1
logical_false_pred	Литерал типа Boolean со значением 0
comparison_pred	Цепочка выражений типа Binary
predicate_application_pred	Выражение Predicate
parentheses_pred	Выражение Paren
conjunction_pred	Выражение Binary
disjunction_pred	–//–
implication_pred	–//–
equivalence_pred	–//–
negation_pred	Выражение PrefixUnary
exclusive_pred	Выражение Binary
ternary_condition_term_pred	Выражение Ternary
ternary_condition_pred	–//–
local_binding_pred	Генерация временной переменной посредством высокоуровневого транслятора
universal_quantification_pred	Выражение Quantified
existential_quantification_pred	Выражение Quantified
syntactic_naming_pred	Не реализовано
old_pred	Выражение PrefixUnary с операцией Old
set_includion_pred	Выражение Set
set_membership_pred	–//–
allocable_pred	Выражение Predicate
freeable_pred	–//–
fresh_pred	–//–
valid_pred	–//–
initialized_pred	–//–
valid_read_pred	–//–
separated_pred	–//–

Приложение М

Перенос тегов между регионами памяти

Перенос тегов в регионах памяти осуществляется согласно Рис. М.1.

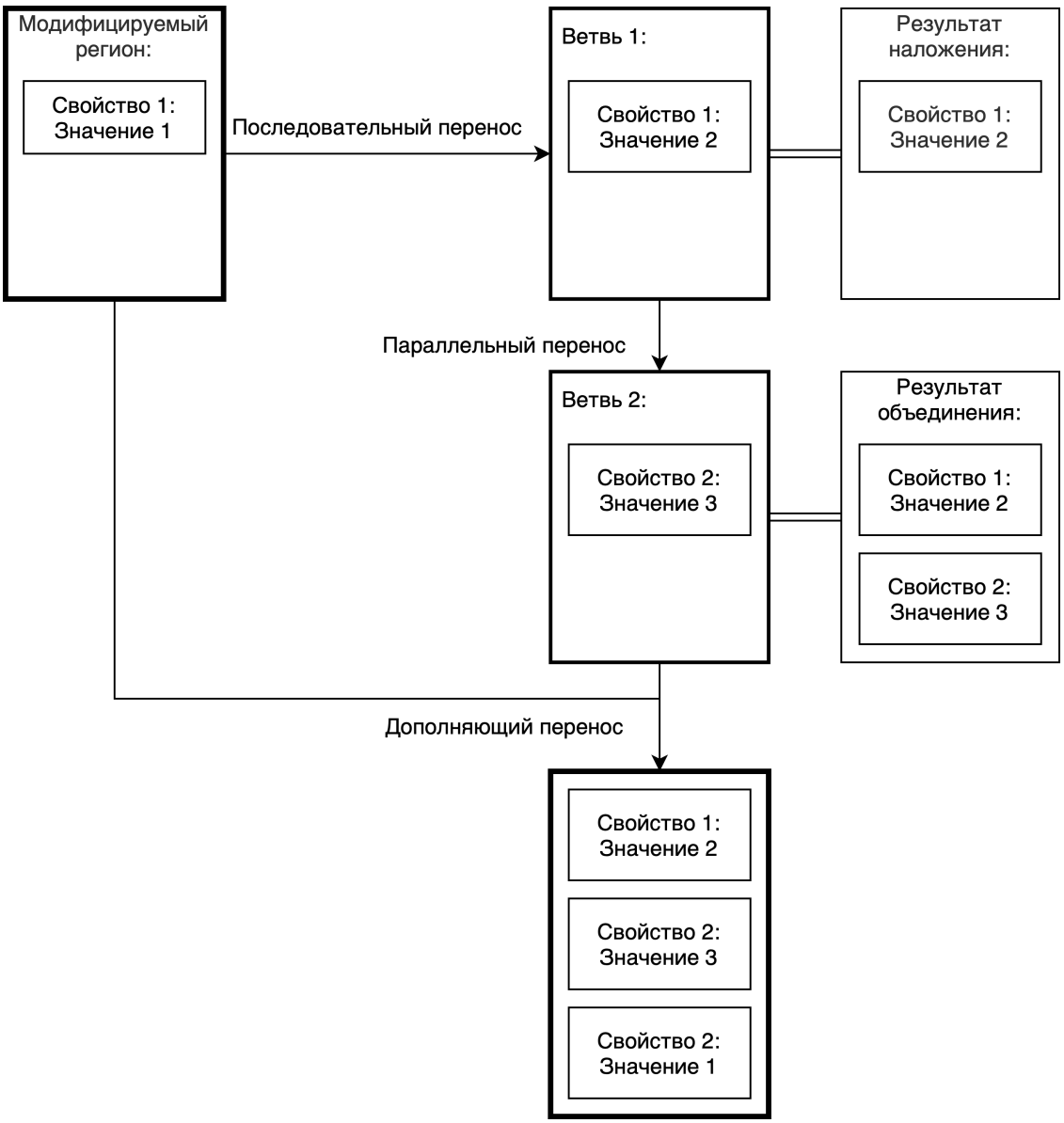

Рисунок М.1 — Схема переноса свойств при обработке ветвей

Приложение Н

Уникальные конструкции языков

В таблице 15 приведены конструкции языков, которые однозначно их идентифицируют.

Таблица 15 — Однозначно идентифицирующие конструкции языков

Язык программирования	Особенности языка
C++	Классы, шаблоны, лямбда-функции, `for` с диапазонами, `try`, операторы, вызов операторов, `typeid`, `uuidof`, `this`, аргументы по умолчанию и т.п.
C	Расширения GNU
РуСи	Ненулевые указатели, отсутствие необходимости подключать заголовочные файлы для встроенных функций.

Приложение О

Основные группы юнит-тестов

Далее приведены основные группы тестов, проверяющие качество работы статического анализатора:

- AI — тесты абстрактной интерпретации.
- Annotation — тесты правильности чтения аннотаций всех типов.
- Concept — тесты основных концепций, таких как ID объектов, ресурсные ключи, ресурсы, фрагменты.
- Core — тесты ядра анализатора.
- Diagnostics — тестирование способов выдачи отчетов об ошибках.
- Environment — проверка методов чтения информации о существующих компиляторах.
- Index — тесты объектного пула.
- Interpretation — тесты интерпретации регионов памяти.
- Lattice — тесты калькулятора решеток.
- LowLevel — тесты концептов «Центральный процессор», «Компилятор», «Диапазон значений» и т.п.
- Math — тестирование канонических типов, локаций кода.
- Memory Region — тестирование многоуровневых регионов памяти.
- Midair — тестирование языка внутреннего представления.
- Namespace — тестирование пространств имен и их объединения.
- Parsing — тестирование чтения `compile_commands.json`.
- Sarif — тесты чтения SARIF [71].
- Solving.AccessPath — проверка путей доступа к переменным.
- Solving.CrossConversion — проверка конверсий между разными типами данных.
- Solving.MultiSolverNg — тестирование гибридного решателя.
- Type — тестирование корректности чтения и записи различных низкоуровневых типов.
- TypeArithmetic — тестирование корректности реализации чтения и записи через паттерны низкоуровневых типов.
- UniType — тесты унифицированной системы типов.
- Utils — тесты прочего незначительных утилитарных классов.

Приложение П

Организация системного тестирования

На Рис. П.1 приведен принцип работы фреймворка, предназначенного для выполнения системного тестирования.

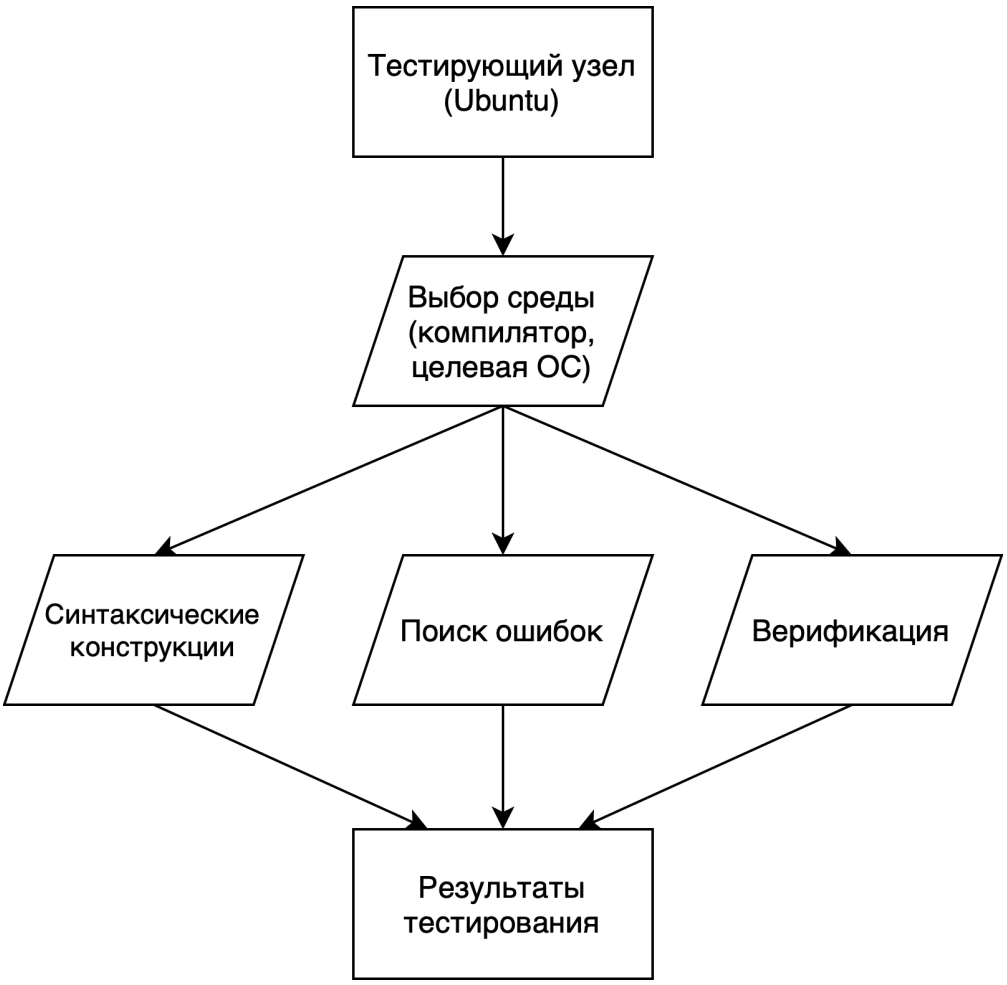

Рисунок П.1 — Схема организации системного тестирования

Приложение Р

Организация тестирования в различных средах

Принцип организации тестирования в различных средах приведен на Рис. Р.1.

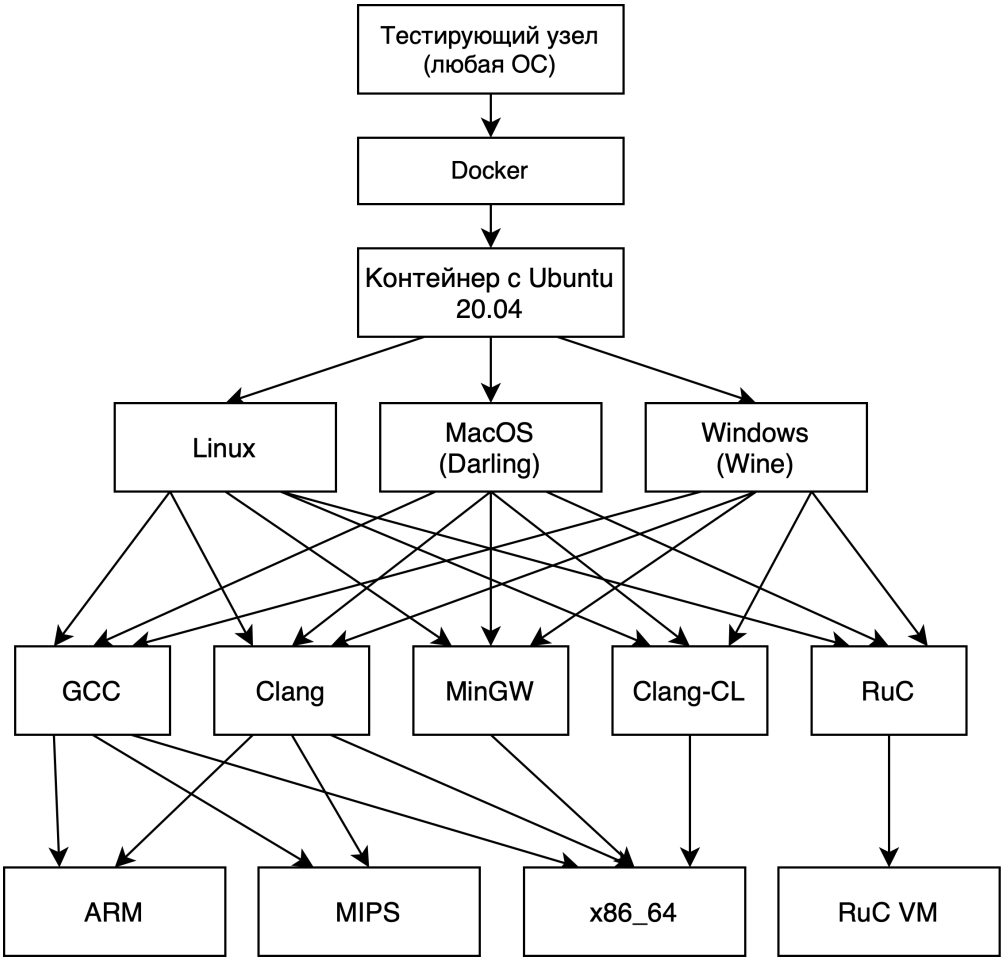

Рисунок Р.1 — Схема организации тестирования в различных средах

Приложение С

Результат тестирования на бенчмарке Toyota ITC

В таблице 16 приведены результаты тестирования статического анализатора на Toyota ITC Benchmark [206].

Таблица 16 — Тестирование на бенчмарке Toyota

Тест	Выявлено	Всего	Комментарий
`bit_shift`	17	17	–
`buffer_overrun_dynamic`	30	32	–
`buffer_underrun_dynamic`	35	39	–
`cmp_funcadr`	2	2	–
`conflicting_cond`	10	10	–
`data_lost`	19	19	–
`data_overflow`	25	25	–
`data_underflow`	12	12	–
`dead_code`	11	13	–
`dead_lock`	–	5	Проверка не осуществляется
`delet._of_data_struct._sent.`	–	3	–
`double_free`	10	12	Отмечена фатальная ошибка на одном из тестов (исправлена)
`double_lock`	4	4	–
`double_release`	5	6	–
`endless_loop`	–	9	Проверка не осуществляется
`free_nondyn_alloc._mem.`	–	16	Проверка не осуществляется
`free_null_pointer`	–	14	Не является ошибкой согласно авторскому пониманию
`func_pointer`	–	15	Проверка не осуществляется
`func._ret._value_uncheck.`	–	16	Проверка не осуществляется
`impr._term._of_block`	–	4	Проверка не осуществляется

Продолжение таблицы 16

Тест	Выявлено	Всего	Комментарий
insign_code	1	1	–
invalid_extern	6	6	–
invalid_memory_access	11	17	–
littlemem_st	11	11	–
livelock	–	1	Проверка не осуществляется
lock_never_unlock	–	9	Проверка не осуществляется
memory_allocation_failure	–	16	Проверка не осуществляется
memory_leak	15	18	–
not_return	–	4	Проверка не осуществляется
null_pointer	15	17	–
overrun_st	47	54	–
overrun_st	47	54	–
ow_memcpy	–	2	Проверка не осуществляется
pow_related_errors	–	29	Проверка не осуществляется
ptr_subtraction	2	2	–
race_condition	8	8	–
redundant_cond	7	14	–
return_local	2	2	–
sign_conv	19	19	–
sleep_lock	3	3	–
st_cross_thread_access	–	6	Проверка не осуществляется
st_overflow	7	7	–
st_underrun	–	7	Проверка не осуществляется
underrun_st	13	13	–
uninit_memory_access	7	15	–
uninit_pointer	10	16	–
uninit_var	10	16	–
unlock_without_lock	8	8	–
zero_division	16	16	–