На правах рукописи

Игнатьев Валерий Николаевич

Статический анализ программ для проверки настраиваемых ограничений языков программирования C и C++

Специальность 05.13.11 – математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

АВТОРЕФЕРАТ
диссертации на соискание учёной степени
кандидата физико-математических наук

Москва – 2015

Работа выполнена в Федеральном государственном бюджетном учреждении науки Институте системного программирования Российской академии наук.

Научный руководитель: **Белеванцев Андрей Андреевич**, кандидат физико-математических наук, ведущий научный сотрудник.

Официальные оппоненты: **Галатенко Владимир Антонович** доктор физико-математических наук, старший научный сотрудник, заведующий сектором автоматизации программирования Федерального государственного бюджетного учреждения науки Научно-исследовательского института системных исследований Российской академии наук,

Бахтин Владимир Александрович кандидат физико-математических наук, заведующий сектором отдела "Автоматизации параллельного программирования" Федерального государственного бюджетного учреждения науки Института прикладной математики им. М.В. Келдыша Российской академии наук.

Ведущая организация: Федеральное государственное бюджетное учреждение науки Вычислительный центр им. А.А. Дородницына Российской академии наук.

Защита состоится «28» мая 2015 г. в 15 часов на заседании диссертационного совета Д.002.087.01 при Институте системного программирования РАН по адресу: 109004, Москва, ул. А. Солженицына, д.25.

С диссертацией можно ознакомиться в библиотеке и на сайте Федерального государственного бюджетного учреждения науки Институт системного программирования Российской академии наук.

Автореферат разослан «27» апреля 2015 г.

Учёный секретарь
диссертационного совета Д.002.087.01,
кандидат физико-математических наук Зеленов С. В.

Общая характеристика работы

Актуальность работы

Языки программирования C и C++, несмотря на свой возраст, остаются незаменимыми и занимают первое и четвертое место в рейтинге распространённости «TIOBE». Однако бесконтрольное использование их возможностей обычными программистами часто приводит к серьезным ошибкам. Решением проблемы употребления «опасных» конструкций является введение ограничений на язык с помощью уточнения или изменения его семантики. Это достигается внедрением правил программирования, которые могут быть ориентированы на решение различных проблем: форматирования кода, безопасности, производительности, переносимости. Минимальные наборы правил состоят из конвенций форматирования кода и именования сущностей в программе. В самых ответственных проектах используются отраслевые стандарты кодирования, например, JSF для программного обеспечения самолета, MISRA для встраиваемых автомобильных систем. Так как обычно используется значительное количество правил на большом объеме исходного кода, требуется автоматизация проверки программ на соответствие им, а формулировка большинства правил в терминах языка программирования требует применения статического анализа исходного текста.

Основной проблемой при этом является отсутствие универсального статического анализатора, сочетающего качественную проверку ограничений с малым временем работы, сравнимым со временем компиляции. Под проверкой ограничений (правил) здесь и далее будет пониматься проверка кода программ на соответствие этим ограничениям (правилам). Так, практически мгновенно работающие анализаторы ограничиваются только лишь проверкой стилистических правил и не способны выявить остальные ошибки. Инструменты, осуществляющие глубокий анализ, учитывающий контекст и поток управления, требуют огромное количество ресурсов и времени, вплоть до нескольких суток, имеют высокую рыночную стоимость, поэтому используются очень редко и только в больших проектах. Таким образом, оба типа анализаторов непригодны для регулярного поиска распространенных ошибок в типичных проектах. Существующие быстрые анализаторы осуществляют проверку либо очень ограниченного и специфического набора правил, например, связанных только с переполнением буфера или переносимостью, либо находят большое количество ситуаций, формально нарушающих стандарт, но не провоцирующих ошибку, фактически являющихся ложными предупреждениями. Например, реализация полезного правила «не использовать вызовы функций, зависящие от

порядка вычисления параметров» фактически запрещает параметры вызовов с побочными эффектами, поскольку лучшее качество анализа недостижимо на уровне синтаксического анализа. Поэтому инструменты такого уровня также неприменимы для прикладных программ. Таким образом, инструмент проверки правил кодирования должен работать именно на этапе написания текста программы, иметь малое время работы и минимальное количество ложных срабатываний, что требует более глубокого анализа, чем синтаксический. В связи с этим возникает необходимость создания анализатора для решения задачи проверки большинства описанных ограничений языков C и C++.

Исследования, посвященные статическому анализу, ведутся в отечественных и зарубежных научных организациях (ИСП РАН, СПбГУ, МГУ, MIT, Stanford University) и исследовательских центрах ведущих промышленных компаний (Samsung, Intel, Microsoft, HP). Несмотря на это на данный момент не существует языка формального определения правил, не выработано единой системы их классификации и отсутствует быстрый универсальный анализатор.

Таким образом, разработка формальной модели ограничений и реализация на её основе быстрого статического анализатора для проверки настраиваемых ограничений языков C и C++ является важной научной задачей, имеющей большое практическое значение.

Цель диссертационной работы состоит в создании формальных моделей, алгоритмов и структур данных программного средства для автоматической проверки исходного кода программ на соответствие ограничениям языков программирования C и C++ с малым временем работы, сравнимым со временем компиляции и минимальным количеством ложных предупреждений, а также реализации разработанных средств в виде легковесного статического анализатора.

Для достижения поставленной цели было необходимо решить следующие задачи:

1. Предложить формальную модель, позволяющую задавать большую часть существующих правил, нотацию для их записи, и их классификацию.

2. Разработать модель программы, включающую модель памяти для проверки программ на соответствие различным классам ограничений, основанную на представлениях программы, используемых при компиляции.

3. Разработать алгоритмы статического анализа для быстрой проверки программ на соответствие рассматриваемому множеству ограничений с минимальным количеством ложных срабатываний и межпроцедурный ме-

тод обнаружения побочных эффектов, взаимных побочных эффектов, межмодульный метод анализа исключений.

4. Реализовать разработанные модели и алгоритмы на базе компиляторной инфраструктуры Clang.

5. Провести испытания анализатора на реальных программных проектах и проанализировать полученные результаты.

Научная новизна

В работе получены следующие основные результаты, обладающие научной новизной:

1. Разработана модель C и C++ программы и её памяти для применения в статическом анализе;

2. Построена и обоснована формальная модель ограничений языка программирования, позволяющая с помощью предложенной нотации на основе оригинальной модели программы задавать стилистические, синтаксические и ситуационные правила. Предложена классификация ограничений.

3. Разработаны, обоснованы и экспериментально протестированы модель памяти C и C++ программы и алгоритмы межпроцедурного анализа программ для обнаружения побочных эффектов, взаимных побочных эффектов, необработанных или неправильно обработанных исключений языка C++.

Теоретическая и практическая значимость

Предложена модель программы на основе аннотированного абстрактного семантического графа, позволяющая эффективно решать задачи, возникающие при статическом анализе. На её основе задана модель памяти, а также семантика модели памяти в виде системы переходов на модели программы. Предложен метод формального определения ограничений языков C и C++ на основе разработанных модели программы и памяти.

Результаты, изложенные в диссертации, использованы для создания подсистемы статического анализа в компиляторе Clang, на основе которой создано средство статического анализа. Оно используется для автоматической проверки подмножества правил из сборников MISRA, HICPP и JSF, а также стандартов написания исходного кода в программах на языках C и C++. Предложенные и протестированные алгоритмы анализа потока данных могут

быть использованы для быстрого и эффективного решения этой задачи в других статических анализаторах для поиска дефектов программ. Разработанное средство внедрено в коммерческой компании Samsung, а также используется для проверки студенческих программ на факультете ВМК МГУ.

На защиту выносятся следующие основные положения:

1. модель программы, включающая модель памяти, используемая как для задания правил, так и для их проверки программ на соответствие им;
2. формальная модель ограничений языка программирования, позволяющая с помощью предложенной нотации задавать правила на основе разработанной модели программы;
3. классификация ограничений по уровням сложности анализа, необходимого для проверки программ на соответствие им, основанная на предложенной формализации;
4. алгоритм построения модели памяти, алгоритм межмодульного анализа программ для обнаружения побочных эффектов, взаимных побочных эффектов, необработанных или неправильно обработанных исключений языка C++;
5. инструмент статического анализа, реализующий разработанные модели и алгоритмы.

Апробация работы

Основные результаты диссертации докладывались на следующих конференциях и семинарах:

1. IEEE Seventh International Conference on Software Testing, Verification and Validation (ICST) (Cleveland, Ohio, USA, 2014);
2. «Майоровские чтения» (Санкт-Петербург, Россия, 2013);
3. XXI Международная научная конференция студентов, аспирантов и молодых учёных «Ломоносов-2014» (Москва, Россия, 2014);
4. научно-исследовательский семинар Института системного программирования РАН

Публикации

Материалы диссертации опубликованы в 5 печатных работах, из них 2 статьи в рецензируемых журналах [1, 2], 3 статьи в сборниках трудов конференций [3, 4, 5]. В совместной работе [2] личный вклад автора состоит в разработке описанного в диссертации статического анализатора, являющегося частью системы Svace.

Личный вклад автора

Все представленные в диссертации результаты получены лично автором.

Структура и объем диссертации

Диссертация состоит из введения, 4 глав, заключения и библиографии. Общий объем диссертации 121 страница, включая 8 рисунков и 8 таблиц. Библиография включает 85 наименований.

Содержание работы

Во введении обоснована актуальность диссертационной работы, сформулирована цель и аргументирована научная новизна исследований, показана практическая значимость полученных результатов.

В первой главе приводится обзор существующих методов решения задач, имеющих отношение к теме диссертации, вводятся основные термины, используемые в тексте диссертации. Рассматриваются существующие способы формальной записи ограничений, принципы их классификации, а также представления программ, используемые при их проверке.

Из проведенного обзора можно сделать вывод, что качественный анализатор не может ограничиваться синтаксическим анализом. Это подтверждается простым примером правила «не использовать вызовы функций, зависящие от порядка вычисления параметров». Стандарт языка не определяет порядок обработки аргументов функции, что приводит к зависимости от компилятора и неопределенности результата. Синтаксического анализа вполне достаточно, чтобы понять, что в строке 5 ограничение нарушено, а в строке 6 нет. Однако консервативная версия алгоритма, работающая на абстрактном синтаксическом дереве (АСД), будет считать, что в строках 7 и 8 правило также не соблюдается, хотя с помощью межпроцедурного анализа потоков данных можно показать отсутствие ошибки в строке 8.

Листинг 1. Пример, показывающий недостаточность синтаксического анализа для качественной проверки правил

```
1 void bar(int, int);
2 int inc(int& x) { return ++x; }
```

```
3 void foo() {
4         int a = 1, b = 2;
5         bar(a++, a); // ошибка
6         bar(a++, b); // правильно
7         bar(inc(a), inc(a)); // ошибка
8         bar(inc(a), inc(b)); // правильно
9 }
```

Многие ограничения языков C и C++ содержатся в уже опубликованных стандартах (JSF, MISRA) и сборниках (HICPP). Одна из проблем создания анализатора состоит в том, что данные ограничения интерпретируются неоднозначно и неточно, т.к. сформулированы на естественном языке. Это приводит либо к их неполной проверке, либо к огромному количеству ложных срабатываний на прикладных программах. Поэтому, в первую очередь, для создания эффективно работающего анализатора необходимо провести их формализацию.

Стандартизированной или общепринятой модели и нотации для ограничений на данный момент не существует, поэтому приводится обзор существующих методов их формальной записи и группировки. Рассмотрены различные модели программ, используемые для определения и проверки ограничений: в виде абстрактного синтаксического дерева, что позволяет добиться высокой производительности, однако, без использования дополнительных видов анализа, ограничивает количество доступных для проверки правил, либо имеет невысокое качество; в виде фактов на языке Пролог, что позволяет задавать правила на этом же языке, однако имеет невысокую скорость работы. Другие декларативные нотации правил требуют разработки интерпретатора и имеют невысокую производительность, т.к. не определяют алгоритм проверки. Простого императивного языка без условных операторов и циклов недостаточно для задания правил. Таким образом, не предложено эффективного решения задачи формализации правил.

Для вычисления данных, необходимых статическому анализатору, используются многие методы из области компиляторных технологий: алгоритмы синтаксического и семантического анализа, построения абстрактного синтаксического дерева, графов потока управления и вызовов и т.д. Поэтому в разделе 1.4 приводится описание существующих алгоритмов для извлечения необходимой для проверки правил информации. К ним относятся алгоритмы поиска побочных эффектов, алгоритмы построения модели памяти, анализ псевдонимов, а также анализ исключений в программе на C++.

Анализ существующих статических анализаторов, проведенный в разделе 1.5 показал, что инструменты, разработанные в научных целях, являются специализированными и не подходят для промышленного применения по причине невысокой скорости или частичной поддержки языка программирования, а также плохо интегрируются в процесс разработки. Основным недостатком коммерческих инструментов является их закрытость. Часть из них, осуществляя глубокий анализ исходного кода, требует огромного количества ресурсов, другие ограничиваются лишь синтаксическим анализом, что либо сокращает набор реализованных правил, либо значительно снижает качество результатов. Таким образом показано, что существующие инструменты не решают поставленных задач.

Во второй главе приводится описание разработанной *модели программы*, которая используется как для формального её описания, так и для задания правил, а также предложенной *модели памяти программы*.

Для проверки некоторых правил требуется незначимая для компилятора информация об исходном тексте программы, например, данные о комментариях. Для реализации других правил необходимы сведения о потоке управления программы или состоянии памяти программы. Абстрактного синтаксического дерева (АСД) недостаточно для эффективного хранения и манипуляций с такими разнородными данными. Поэтому в качестве универсальной модели программы для задания правил и их проверки будем использовать *аннотированный абстрактный семантический граф* (ААСГ) – структуру данных, представляющую собой абстрактный семантический граф, каждая вершина которого дополнена *аннотацией*, а каждое ребро помечено. Под *аннотацией ААСГ* будем понимать множество пар $\langle$*имя, значение*$\rangle$ атрибута.

В каждой вершине ААСГ задан тип, определяющий её семантику. Для него введем специальный атрибут $\tau\colon \mathcal{V} \mapsto \mathcal{T}$, где $\mathcal{V}$ – множество вершин ААСГ, $\mathcal{T}$ – конечное множество возможных типов. Тип узла обозначает, какой сущности в программе соответствует рассматриваемая вершина графа. Поскольку ААСГ создан для хранения данных с разных этапов анализа программы, в нем содержатся вершины для лексем, узлов АСД и графа потока управления, причем каждая из них имеет заданное множество типов:

$$\begin{aligned}
\mathcal{T} &= \mathbf{Lex} \cup \mathbf{AST} \cup \mathbf{CFG} \\
\mathbf{Lex} &= \{\mathbf{Lex}\mathit{::Keyword}, \mathbf{Lex}\mathit{::Identifier}, \dots\} \\
\mathbf{AST} &= \{\mathbf{AST}\mathit{::IfStmt}, \mathbf{AST}\mathit{::BinaryOperator}, \dots\} \\
\mathbf{CFG} &= \{\mathbf{CFG}\mathit{::BasicBlock}, \mathbf{CFG}\mathit{::Stmt}, \mathbf{CFG}\mathit{::Terminator}\}
\end{aligned}$$

Например, для вершины, соответствующей условному оператору `if-then-else` абстрактного синтаксического дерева

$$v_{\texttt{if-then-else}}.\tau = AST::IfStmt.$$

Для каждого типа определены множества обязательных и допустимых атрибутов. Предложенная модель решает одновременно несколько задач:

- на основе формализаций из теорий лексического, синтаксического анализа, атрибутных грамматик задаются соответствия между используемыми в задании правил терминами и объектом в модели, что частично устраняет неоднозначность понимания естественного языка определения правил и позволяет автоматизировать проверку.

- абстрактная сущность модели позволяет *одновременно* обрабатывать и хранить разнородные данные: языково-специфичную информацию и результаты сразу нескольких видов анализа.

Определение и проверка ряда правил требует задание модели памяти, которая позволит формализовать работу с типами памяти, указателями, преобразованиями типов. Для этого введем понятие *область памяти* $A = StorageClass \times \mathbb{N}_0$ – декартово произведение класса памяти и размера в битах, где $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$ – множество натуральных чисел и 0.

Таблица 1. Классы памяти C/C++ программы

Класс	**Имя**	**Подкласс**	**Имя**
Стековый	Stack	Автоматические переменные	*AutoStack*
		Параметры функций	*FuncParamStack*
		Выделенная `alloca`	*AllocaStack*
Код	Text	Указатель на функцию	*FuncPtrText*
		Метки	*LabelText*
Данные	Data	Глобальные системная	*GlobalSystem*
		Глобальная `static`	*FileStaticData*
		Глобальная	*GlobalData*
Динамическая	Heap	Выделенная `malloc`	*MallocHeap*
		Выделенная `new`	*NewHeap*

Использование битового размера областей памяти позволяет учитывать битовые операции и структуры с битовыми полями. В общем случае, информация о смещениях и выравниваниях полей в структурах платформо-специфична, поэтому в реализации используются предоставленные компилятором

данные, а в модели – выравниванием пренебрегаем и считаем все структуры упакованными. Иерархия классов памяти представлена в таблице 1. В программах на C и C++ статически определены не все размеры выделяемой памяти, что поддерживается моделью.

В отличие от работ, посвященных символьной интерпретации, в предлагаемой схеме никак не используются виртуальные адреса. Вместо этого область памяти является «абстрактным адресом», а для поддержки элементов массива и полей структур, классов используется *подобласть памяти* – декартово произведение базовой области памяти и размера в битах $A_{\sqsubset} = A_{base} \times \mathbb{N}_0$ назовем *подобластью памяти*.

Это позволяет хранить значения отдельных элементов массива или полей структур без всего их содержимого. Для выражений `a[2]` будем записывать $A^2_{\sqsubset}(A^a)$, для `s.f` – $A^f_{\sqsubset}(A^s)$. Смещение относительно начала базовой области памяти не является свойством подобласти, а определяется в отображении, задающем состояние областей памяти.

Теорема 1. *Подобласть памяти является областью памяти.*

Будем использовать запись $\mathcal{V}^{type}$ для обозначения всех вершин ААСГ G некоторого типа *type*. Тогда состояние памяти S_G в модели программы задается как декартово произведение

$$S_G = \mathcal{V}^{CFGStmt} \times Env \times Mem,$$

$\mathcal{V}^{CFGStmt}$ – множество всех вершин типа *CFGStmt* в ААСГ,

соответствующих операторам графа потока управления всех функций

$$Env = \begin{cases} \alpha\colon \mathcal{V}^{\mathrm{VarDecl}} \mapsto A, \\ \alpha'\colon A_{\mathrm{base}} \times \mathcal{V}^{\mathrm{FieldDecl}} \mapsto A_{\sqsubset} \\ \alpha''\colon A_{\mathrm{base}} \times \mathbb{N}_0 \mapsto A_{\sqsubset} \end{cases} \text{– состояние переменных,}$$

$$Mem = \begin{cases} \lambda\colon A \mapsto \{0,1\}* \\ \hat{\lambda}\colon A \mapsto A \end{cases} \text{– состояние области памяти.}$$

Обозначение $\{0,1\}*$ использовано для последовательности бит в памяти.

Начальное состояние S_G обозначим $S_0 = \{(\mathcal{V}^{entry}(f), e_0, m_0)\}$, где $\mathcal{V}^{entry}(f) = \underbrace{f.\mathrm{entry}}_{\text{начальный ББ } f}.\mathrm{child}, f \in t(\mathcal{V}, \{FunctionDecl\})$ – первый оператор начального базового блока функции f, e_0 содержит отображение всех глобальных переменных и фактических параметров вызова функции f в соответствующие области памяти, m_0 – отображение указанных областей памяти в их значения.

Семантика модели памяти программы задается с помощью *системы переходов*, введенной Г. Плоткиным (G.D. Plotkin): $\mathrm{TS}_G = (S_G, S_0, \xrightarrow{G})$, где S_G, S_0 – множество состояний памяти программы и начальное состояние, а $\xrightarrow{G} \subseteq S_G \times S_G$ – правила перехода от одного состояния к другому, которые определяются на графе потока управления в модели программы в виде:

$$\xrightarrow{G} = \begin{cases} \xrightarrow{next}: \dfrac{n_1,n_2 \in \mathcal{V}^{CFGStmt},\ n_1 \xrightarrow{next} n_2}{(n_1,e,m) \xrightarrow{next} (n_2,e',m')} \\ \xrightarrow{succ}: \dfrac{b_1 = n_1.parent,\ b_1 \xrightarrow{succ} b_2,\ c \in \mathcal{V}_{b_1}^{CFGTerminator},\ c = b_1.Terminator,\ P((c,b_2)) \neq 0,\ n_2 = b_2.child}{(n_1,e,m) \xrightarrow{succ} (n_2,e',m')} \end{cases} \tag{1}$$

Переход $\xrightarrow{next}$ осуществляется между двумя «соседними» узлами типа *CFGStmt* при условии наличия между ними ребра с меткой *next*. При этом состояния переменных и областей памяти e, m преобразуются в новые e', m', вычисляемые по приведенным далее правилам. Переход между базовыми блоками $\xrightarrow{succ}$ чуть сложнее:

- во-первых, $\nexists n_2 \colon n_2 = n_1.next$, иначе выполнилось бы первое правило;
- во-вторых, должно существовать ребро, помеченное *succ*, из $b_1 = n_1.parent$ в некоторый b_2;
- в-третьих, должно выполняться условие перехода в b_2, хранящееся в узле типа *CFGTerminator* базового блока предка (b_1), записанное как $P(c, b_2) \neq 0$;
- в-четвертых, базовый блок b_2 не пустой, т.е. $\exists n_2 \colon n_2 = b_2.child$.

В соответствии с терминологией, введенной Г. Плоткиным, система переходов $(\Gamma, \Gamma_0, \longrightarrow)$ *детерминированная*, если

$$\forall \gamma, \gamma_1, \gamma_2 \in \Gamma \colon \gamma \to \gamma_1 \wedge \gamma \to \gamma_2 \Rightarrow \gamma_1 = \gamma_2$$

На основе приведенного определения в работе сформулирована и доказана теорема:

Теорема 2. *Семантика модели памяти является детерминированной системой переходов.*

Вычисление условия перехода $\xrightarrow{succ}$ при статическом анализе в явном виде невозможно, поэтому оно заменяется на систему уравнений или неравенств на основе состояний памяти программы, которая соответствует данному переходу.

В случае разветвления потока управления программы при этом возникает неоднозначность выбора пути анализа, которая разрешается путем копирования состояния, превращая линейный анализ в дерево с множеством допустимых конечных состояний. Тем не менее, получившаяся система переходов может быть смоделирована недетерминированной машиной Тьюринга.

Обозначим как C – систему уравнений и неравенств, содержащую множество условий перехода в некоторой точке анализа, тогда

$$\xrightarrow{succ^*} = \frac{S \xrightarrow{succ} S', S \xrightarrow{succ} S'', S' \neq S''}{C' = C \wedge P', C'' = C \wedge P''} \tag{2}$$

Перейдем к рассмотрению изменения компонент e и m в семантике. Для этого требуется ввести вспомогательные функции. Функционал $\theta\colon \mathcal{V}^{VarDecl*} \mapsto \mathcal{V}^{TypeDecl}$ возвращает языковой тип объявленной переменной, при этом данные берутся из модели программы, $\theta^*\colon \mathcal{V}^{Expr} \mapsto \mathcal{V}^{TypeDecl}$ – для типов выражений. Функционал $\sigma\colon \mathcal{V}^{TypeDecl} \mapsto \mathbb{N}_0$ возвращает размер соответствующего типа в битах. Даже в языке C, и тем более в C++, существует немало типов, для которых размер не известен на этапе компиляции (массивы переменной длины, зависящие от шаблонных параметров типы, предварительные объявления типов). В таких случаях значение функционала не определено ($\sigma = \top$), однако в некоторых случаях может быть уточнено во время анализа. Поскольку значение области памяти представлено в виде последовательности бит, то для его интерпретации в значение некоторого типа требуется введение функционала $\iota\colon \{0,1\}^* \times \mathcal{V}^{TypeDecl} \mapsto \mathbb{Z} \cup \mathbb{F} \cup \{\mathbf{true}, \mathbf{false}\}$ и $\iota^-\colon \mathbb{Z} \cup \mathbb{F} \cup \{\mathbf{true}, \mathbf{false}\} \times \mathcal{V}^{TypeDecl} \mapsto \{0,1\}*$ для представления значений в виде последовательности бит заданного типа. Для данных функционалов значения платформо-специфичных параметров, таких, как последовательность байт, размеры базовых типов определяются с помощью компиляторной части анализатора.

Выражения языков C и C++ могут иметь семантику селекторов (l-value) и значений (r-value). В работе записаны правила их вычисления для большинства выражений, определенных в модели программы. Они задают операционную часть семантики модели памяти, т.е. как выполнение разных типов выражений и операторов программы отражается на состояниях переменных и памяти. В качестве примера приведены правила для целочисленных констант, подмножества бинарных операторов и обращения к массиву.

Константы. Пусть $\tau(v) = IntegerLiteral$, тогда

$$l((e,m),v) = \bot,\ r((e,m),v) = \iota^-((v.value), \theta^*(v)), \tag{3}$$

где $\perp$ обозначает «не определено», а $v.value$ – взятие атрибута, содержащего целочисленное значение константы. Введенные обозначения для «не определено» $\perp$ и $\top$ следует трактовать следующим образом: $\perp$ подразумевает, что значение не имеет смысла и не должно вычисляться в принципе, а $\top$, что оно может иметь произвольное значение в рамках некоторого домена.

Бинарные операции $\{+, -, *, /, \%, \&\&, \|, >, <, \leq, \geq, =, \neq\}$. Пусть $\tau(v) = BinaryOperator,\ v = v_1 \star v_2,\ v.OpKind = \square \in \{+, -, *, /, \%, \&\&, \|, >, <, \leq, \geq, =, \neq\}$

$$\begin{aligned} &l((e,m),v) = \perp, \\ &r((e,m),v) = \iota^{-}(\iota(r((e,m),v_1),\theta^*(v_1)) \square \iota(r((e,m),v_2),\theta^*(v_2)),\theta^*(v)) \end{aligned} \tag{4}$$

Доступ к элементу массива. Пусть $\tau(v) = ArraySubscriptExpr, v = v_1[v_2]$.

$$\begin{aligned} &l((e,m),v) = \alpha''(l((e,m),v_1),\iota(r((e,m),v_2),\mathbb{N}_0), \\ &r((e,m),v) = \lambda(l((e,m),v)) \end{aligned} \tag{5}$$

Для пояснения данного случая рассмотрим пример выражения `a[j]`, для которого селектор является подобластью области памяти переменной `a`.

$$l((e,m),v) = \alpha''(A^a, \lambda(A^j)) = A^{\iota(\lambda(A^j),\mathbb{N}_0)}_{\sqsubset}(A^a)$$

Описанная семантика модели памяти похожа на абстрактную интерпретацию программы. В результате анализа программы состояния представляют собой множество трасс $\{(n,e,m)\ldots(n_{end},e_{end},m_{end})\}$ выполнения всех ветвлений потока управления программы. Это означает, что для анализа и построения множества таких трасс потребуется огромное количество ресурсов. Кроме того, полученная информация избыточна для проверки интересующего подмножества правил. Поэтому введен ряд ограничений для снижения времени анализа за счет ухудшения точности, в частности на повторный анализ уже просмотренных вершин *CFGStmt*. Это позволит существенно сократить время анализа циклов. Рассмотрим стандартную ситуацию, когда в базовый блок приходит несколько входящих ребер из предков. Без введения дополнительных ограничений, при этом создавалось ветвление в трассе-результате анализа. Это позволяло производить потоко-чувствительный анализ. Так как для рассматриваемых языков характерны небольшие функции, более существенное влияние оказывает контекстная чувствительность по сравнению с потоковой. Применение стандартных техник экстраполяции *расширения (widening)* и *сужения (narrowing)* ведет к утрате относительно небольшой доли информации. Расширение в случае отображений *Env* и *Mem* реализуется простым объединением

множеств их значений, а сужение – пересечением. Обозначив расширение/сужение как $\triangle / \triangledown \in \underbrace{(v', e', m')}_{S'_G} \times \underbrace{(v'', e'', m'')}_{S''_G} \mapsto \underbrace{(v, e, m)}_{S_G}$ его реализация может быть записана в виде:

$$\begin{aligned} \triangle(S', S'') &= \frac{S' \xrightarrow{succ} S,\ S'' \xrightarrow{succ} S}{e = e' \cup e'', m = m' \cup m'' C = C' \vee C''} \\ \triangledown(S', S'') &= \frac{S' \xrightarrow{succ} S,\ S'' \xrightarrow{succ} S}{e = e' \cap e'', m = m' \cap m'', C = C' \vee C''} \end{aligned} \tag{6}$$

На основе изложенных понятий в работе сформулированы и доказаны теоремы:

Теорема 3. *Объединение систем* (6), (2) *и* (1) *определяют детерминированную систему переходов, реализующую семантику модели памяти.*

Теорема 4. *Заданная система переходов, моделирующая память программы, завершается.*

Для пояснения построения состояний памяти рассмотрим пример:

Листинг 2. Пример программы для демонстрации возможностей модели памяти

```
1 struct S {  int x; };
2 struct C { S *ps; };
3 int f(void) {
4     S s; C *pc;
5     pc = new C;
6     pc->ps = &s;
7     pc->ps->x = 10;
8     return s.x;
9 }
```

Тело функции f состоит из одного базового блока. Его анализ осуществляется путем последовательного обхода вершин модели программы типа *CFG::Stmt* и применения к ним описанных правил. Согласно прототипу функции резервируется область памяти для возвращаемого значения

$$Mem = \langle A^{retval} = (FuncParamStack, \sigma(\texttt{int})), \top \rangle.$$

Первым рассматриваются определения переменных `s, pc` с учетом вызова неявного конструктора. В результате получаем:

Env		*Mem*	
		A^{retval}	$\top$
$s_{VarDecl}$	$A^s = (AutoStack, \sigma(\texttt{S})$	A^s	$\top$
$(s_{VarDecl}, x_{FieldDecl})$	$A^x_{\sqsubset}(A^s) = (A^s, \sigma(\texttt{int}))$	$A^x_{\sqsubset}$	$\top$
$pc_{VarDecl}$	$A^{pc} = (AutoStack, \sigma(\texttt{C*}))$	A^{pc}	$\top$
$(pc_{VarDecl}, ps_{FieldDecl})$	$A^{ps}_{\sqsubset}(A^{pc})$	$A^{ps}_{\sqsubset}(A^{pc})$	$\bot$

Оператор на 5 строке листинга реализует присваивание указателю `pc` адреса выделенной в `CXXNewExpr` области памяти для класса `C`. Согласно приведенным в тексте работы правилам, требуется вычислить l-value $l(\texttt{DeclRefExpr}(pc)) = A^{pc}$, r-value $r(\texttt{CXXNewExpr}(C)) = (NewHeap, \sigma(\texttt{C}))$, проанализировав при этом неявный конструктор $\texttt{CXXConstructExpr}(C)$, и записать результаты в отображение *Mem*. Таким образом,

Env		*Mem*	
	$A^{new} = (NewHeap, \sigma(\texttt{C}))$	A^{new}	$\top$
$pc_{VarDecl}$	A^{pc}	A^{pc}	A^{new}
$(pc_{VarDecl}, ps_{FieldDecl})$	$A^{ps}_{\sqsubset}(A^{pc})$	$A^{ps}_{\sqsubset}(A^{pc})$	$\top$

Аналогично присваивания на 6 строке, применяя правила для доступа к полю и унарного оператора &, получаем

Env		*Mem*	
$(pc_{VarDecl}, ps_{FieldDecl})$	$A^{ps}_{\sqsubset}(A^{pc})$	$A^{ps}_{\sqsubset}(A^{pc})$	A^s

Анализ оператора присваивания на 7 строке нестрого можно записать в виде:

$$l(\texttt{pc->ps->x}) = A^{\texttt{x}}_{\sqsubset}(\hat{\lambda}(l(\texttt{pc->ps})) = A^{\texttt{x}}_{\sqsubset}(\hat{\lambda}(A^{\texttt{ps}}_{\sqsubset}(A^{\texttt{pc}})) = A^{\texttt{x}}_{\sqsubset}(A^{\texttt{s}}), \quad r(\texttt{10}) = 10,$$

Env	*Mem*	
	$A^{\texttt{x}}_{\sqsubset}(A^{\texttt{s}})$	10

Наконец, оператор `return`: $r(\texttt{s.x}) = \lambda(l(\texttt{s.x})) = \lambda(A^{\texttt{x}}_{\sqsubset}(A^{\texttt{s}})) = 10$

Приведенный пример показывает, как с помощью описанной модели памяти проводить достаточно точный анализ состояния памяти программы. Рассмотрено использование подобласти памяти для манипуляций с элементами

класса, что является частичным, однако достаточным для поставленной задачи анализа, решением проблемы псевдонимов. Также, пример позволяет детально понять, как, применяя набор заданных формальных правил, производить анализ реальных программ, записанных в виде модели.

Предложенные модели программы и памяти позволяют анализировать все множество конструкций языков C и C++, а не только искусственно выделенное подмножество, обычно рассматриваемое в работах, посвященных статическому анализу программ. При этом осуществляется нечувствительный к потоку управления, контекстно-чувствительный анализ заданных свойств программы, необходимых для проверки ограничений. Модель специально адаптирована для работы со стандартными представлениями сущностей языка в компиляторах, что позволяет её использовать не только для рассматриваемых языков, но и включить поддержку, например, C# или Java. Доказана детерминированность и завершаемость анализа на основе предложенной модели, в процессе чего показано, что для большинства операторов требуется всего один проход, а для вложенных циклов – не более, чем количество изменяемых переменных в теле цикла.

Результаты второй главы опубликованы в [4, 3]

В третьей главе описана разработанная формализация ограничений языков C и C++, основанная на логике предикатов. Поскольку для эффективной реализации в анализаторе требуется императивное задание правил, предлагается использовать одновременно обе формализации: в форме предиката для достижения однозначности трактовки, поскольку формулировка на естественном языке допускает неточность интерпретации; в виде реализации на C++ для эффективной проверки в анализаторе. Оба представления правила объединяет единая модель программы, что позволяет упростить переход от декларативного представления к императивному.

Определим *ограничение (правило)* как предикат, заданный на модели программы.

Под *t-селектором* будем понимать функционал выбора из ААСГ всех вершин с заданным множеством типов: $t\colon \mathcal{P}(\mathcal{V}) \times \mathcal{P}(\mathcal{T}) \mapsto \mathcal{P}(\mathcal{V})$, где $\mathcal{P}$ – степень множества (множество всех подмножеств).

Под *e-селектором* будем понимать функционал, выбирающий множество всех вершин ААСГ, в которые существует путь из заданных в первом аргументе вершин, причем все ребра пути помечены строкой S: $e\colon \mathcal{P}(\mathcal{V}) \times \mathcal{S} \mapsto \mathcal{P}(\mathcal{V})$.

Для простоты записи будем использовать привычные символы $=, \neq$ вместо соответствующих предикатных символов $= (t_1, t_2), \neq (t_1, t_2)$, а также, где

это допустимо, отношений для множеств ($\in, \notin, \subset, \subseteq, ...$). Тогда правило `MISRA2004-20_4-3`, запрещающее использование перечисленных функций, будет записываться в виде:

$$\begin{aligned} &\forall n \in t(\mathcal{V}, \{AST::FunctionDecl\}): \\ &\quad n.name \notin \{\text{''}\texttt{malloc}\text{''}, \text{''}\texttt{free}\text{''}, \text{''}\texttt{calloc}\text{''}, \text{''}\texttt{realloc}\text{''}\} \end{aligned} \tag{7}$$

С помощью предложенных обозначений легко записывать правила, связанные с работой с памятью. Например,

- проверка неинициализированной переменной [MISRA2008-8_5_1-3]:

$$\forall n \in t(\mathcal{V}, \{AST::DeclRefExpr\}): r(n) \neq \top$$

- выход за границы массива [BD-PB-ARRAY-1]:

$$\begin{aligned} &\forall n \in t(\mathcal{V}, \{AST::ArraySubscriptExpr\}): \\ &\quad 0 \leq \underbrace{\iota(r(n.Offset))}_{\substack{\text{значение выражения-индекса} \\ \text{массива}}} * \underbrace{\sigma(\tau(n.Base))}_{\substack{\text{размер типа} \\ \text{элемента массива}}} \leq \underbrace{l(n.Base).Size}_{\substack{\text{размер области} \\ \text{памяти массива}}} \end{aligned}$$

Таким образом, используя аппарат логики предикатов и набор предопределенных функционалов и предикатов, можно задавать и записывать ограничения на приведенной модели программы. Однако запись правил с помощью предложенной нотации все же требует достаточной квалификации, т.к. необходимо учитывать неочевидные на первый взгляд случаи.

Для реализации в анализаторе применяется другая формализация, которая фактически является реализацией на языке C++ на основе разработанной инфраструктуры алгоритма проверки правила, позволяя эффективней тратить ресурсы. Переход от математической записи к программной осуществляется вручную, однако для многих случаев может быть автоматизирован.

Для решения задачи автоматического распараллеливания проверки правил, а также автоматического формирования оптимальной последовательности их верификации предложена классификация, которая позволяет однозначно распределить множество правил по группам. Как видно из примеров записи ограничений, каждое из них использует определенное множество вершин ААСГ (определенных типов) и атрибутов, подразумевая их существование в модели программы. Однако во время работы анализатора узлы и атрибуты вычисляются и связываются друг с другом поэтапно. Поэтому одни правила

могут быть проверены раньше других по мере построения ААСГ. Анализ программы компилятором принято разделять на 5 этапов: лексический, синтаксический, семантический, требуемый при оптимизациях, и межмодульный во время компоновки, причем каждый следующий требует результат предыдущего. Аналогично распределим правила по классам. Предложенная классификация является однозначной и характеризует сложность правила, т.е. относительное количество ресурсов, требуемых для его проверки. В случае, когда для проверки необходима информация с нескольких уровней, будем считать, что правило принадлежит к классу наиболее позднего этапа. С помощью предложенного критерия все правила можно распределить на 5 классов.

Лексические ограничения. К классу лексических ограничений относятся правила, формализуемые на основе вершин ААСГ, описывающих лексемы (с типом $\tau \in Lex::*$). Неформально, к данному классу относятся ограничения, содержащие слова: ключевое слово языка и их полный список, строка, символ, лексема, знаки операций, комментарий, директивы препроцессора и их список, идентификатор. Большую часть ограничений данного класса составляют правила форматирования исходного текста программы, правила использования комментариев, директив препроцессора и макроопределений.

Синтаксические ограничения. К этому классу относятся правила, задаваемые на уровне, соответствующем синтаксическому анализу компилятора. С помощью дуг и атрибутов вершин ААСГ обеспечена связь с лексической частью графа, например, для выражения типа *AST::IfStmt* построена дуга к соответствующему узлу типа *Lex::Keyword* лексемы ключевого слова `if`, помеченная как *StartToken*. Большую часть синтаксических правил составляют конвенции именования переменных, функций, типов; конвенции использования функций стандартных библиотек; проверка форматной строки; проверка наличия комментариев для функций, переменных, блоков и т.д.; уточнение грамматики языка, не требующее информации о типах.

Контекстные ограничения, не требующие дополнительного анализа. К классу относятся ситуационные ограничения, зависящие от состояний нескольких объектов модели программы. Например, правило «недопустимо отсутствие оператора `return` в не `void` функции» зависит от вершины графа *AST::FunctionDecl* для соответствующей функции, и от наличия в теле функции узла *AST::ReturnStmt* на всех путях выполнения. Большинство правил описывается в терминах языка программирования и их проверка осуществляется после семантического анализа. К классу семантических относятся структурные правила внутри одного модуля компиляции, связанные со статическими

отношениями в программе, не представленными в дереве разбора (наследование, член класса); ограничения на типы; уточнения грамматики языка.

Контекстные ограничения, требующие дополнительного анализа программы. Большая часть атрибутов ААСГ строится на основе данных, вычисленных компилятором в процессе работы. Однако проверка ряда ограничений требует дополнительный анализ. Анализатор, используя свои алгоритмы, структуры данных и внутреннее представление дополняет ААСГ дугами и атрибутами. Подобным образом строится, например, граф потока управления, граф вызовов, получается информация о границах значений переменных и побочных эффектах выражений. В качестве примера правил рассматриваемого класса можно привести ограничения, использующие анализ побочных эффектов или модель памяти.

Межмодульные статические правила. К данному классу относятся правила, для проверки которых недостаточно данных, определённых внутри одного модуля компиляции. Их тестирование осуществляется на этапе, при компиляции соответствующем компоновке. Исходными данными анализа является специально экспортированная и сохранённая информация, собранная на предыдущих этапах. Для стадии связывания можно выделить следующие группы правил: анализ исключений, поиск взаимных блокировок, и race condition; проверка ряда структурных правил, требующих информацию обо всей программе; проверка tainted данных с использованием всей программы.

Результаты третьей главы опубликованы в работах [1, 4].

В четвертой главе приводится описание алгоритмов и эвристик статического анализа, специально разработанных для проверки требуемого подмножества ограничений, а также особенности реализации разработанного инструмента RuleChecker.

Основными принципами анализатора являются малое время работы (сравнимое со временем компиляции) и минимальное количество ложных срабатываний. Эти критерии определяют, допустимо ли включение некоторого правила во множество поддерживаемых. Кроме того, в условиях промышленного использования разработанный анализатор RuleChecker является первым этапом статического анализа программы, за которым следует применение более «мощного» анализатора Svace, использующего языково-независимое внутреннее представление, которое не содержит исходной информации о конструкциях языка. Поэтому требующие языково-специфичной информации правила должны поддерживаться в RuleChecker. Лексические, синтаксические и семантические правила, не зависящие от дополнительного анализа, хорошо подходят для

включения в анализатор, поскольку имеют точные алгоритмы реализации, не допускающие (при условии отсутствия ошибок кодирования) ложных срабатываний. При этом они составляют преобладающую часть языково-специфичных правил. Оставшиеся языково-специфичные правила определяют набор необходимых для их проверки анализов и алгоритмов: побочные эффекты, межмодульный анализ исключений, указателей и т.д.

Для реализации анализатора используется инфраструктура статического анализа компилятора Clang. Разработанная система состоит из трёх основных компонентов: подсистема определения правил, сбора информации и планировщик правил. Основные этапы анализа схожи с компиляцией программы. Лексический анализ как отдельный этап не применяется, а осуществляется по запросу из парсера. На этапе синтаксического и семантического анализа подсистема сбора информации осуществляет построение AST и Lex частей модели программы, устанавливая между ними связи. На основе неявно заданных при реализации правил зависимостей от типов вершин модели программы строится граф зависимостей между правилами, который определяет группы для параллельного запуска и порядок в группах. После завершения построения синтаксической части модели запускается проверка ограничений путем нескольких различных обходов графа программы и вызова переопределенных правилами обработчиков разных типов узлов ААСГ. Затем подсистема сбора информации выполняет базовые алгоритмы дополнительного анализа: достраивает граф потока управления и статический граф вызовов, тем самым разблокируя проверку оставшихся контекстных правил. Для функций, в которых алгоритм проверки правила столкнулся с недостаточным количеством данных, запускаются дополнительные виды анализа: строится модель памяти, вычисляются побочные эффекты. Если информации в данном модуле компиляции недостаточно, то в межмодульную базу данных сохраняется условное предупреждение, которое может быть удовлетворено при анализе других модулей компиляции. После завершения всех проверок планировщик правил запускает обработку условных предупреждений из базы данных, а также сериализацию и сохранение заданной части модели программы для межмодульного анализа.

На этапе анализа, соответствующем компоновке при компиляции, производится проверка межмодульных правил с помощью экспортированной информации. Запуск этой фазы, как и остальных, осуществляется путем перехвата старта компилятора и компоновщика из системы сборки. Механизм перехвата основан на переопределении переменной окружения `LD_PRELOAD`. Это позволяет принудительно загружать у всех дочерних процессов разработанную дина-

мическую библиотеку, которая содержит реализацию функций группы `exec` (`execve`, `execv`, `execl`, `execvp`, `execlp`) и переопределяет их. Реализованные функции анализируют командную строку и в случае обнаружения выполнения компилятора или компоновщика, запускают скрипты исправления параметров, путей, окружения, а затем анализатор. Такая система позволяет с небольшими доработками использовать практически любой компилятор и любую систему сборки и обеспечивает анализ только той части исходных текстов, которая компилируется в результирующую программу.

Для сохранения информации между запусками анализатора для различных модулей компиляции использована реляционная база данных SQLite. В подсистеме сбора информации разработан механизм сериализации и десериализации необходимых данных о типах, переменных и их областях памяти, аннотаций функций, позволяющий (1) не анализировать несколько раз функции в заголовочных файлах, (2) сохранять информацию о побочных эффектах и исключениях на уровне функций, (3) сохранять условные предупреждения, требующих данные о функции, в еще не анализированном модуле компиляции.

Для эффективного анализа были разработаны специальные алгоритмы. Основным является *построение модели памяти*. Для этого производится последовательное применение сформулированных во второй главе правил, фактически реализующее обход графа потока управления функции. Наибольший интерес представляет анализ ветвлений и циклов, который был предложен в теоретической части. На практике условия ветвлений сериализуются в систему уравнений и неравенств с целью проверки инвариантности с помощью библиотеки внешнего решателя. Такие данные вычисляются только в вершинах ветвлений и задействуют минимальное количество переменных, участвующих в формировании условий перехода. Поэтому решение получающихся небольших систем не является слишком ресурсоемким.

Анализ циклов при построении модели памяти осуществляется итеративно до достижения стационарности окружения и памяти. Каждая новая итерация выбирает измененную в цикле область памяти, значение которой устанавливается в $\top$. Поскольку количество измененных в цикле переменных с учетом иерархии областей памяти (для структур и массивов) невелико, процесс быстро завершается, однако с непредсказуемым качеством результата. Решение системы неравенств и анализ инвариантов цикла позволяют получать более точные результаты, однако не укладываются в ограничения по времени.

Анализ побочных эффектов. Под *побочным эффектом* выражения или функции в программе обычно понимается изменение нелокальных данных. В

контексте языков C и C++ побочный эффект возникает, если выражение или функция: (1) осуществляет вызов функции с побочными эффектами; (2) содержит изменение значения переменной с большей областью видимости (глобальной переменной, локальной статической переменной функции, локальной переменной с большей областью видимости); (3) выполняет изменение состояния памяти с помощью указателя или ссылки в C++; (4) генерирует исключение C++; (5) содержит ассемблерную вставку. Зададим множество генерируемых исключений в виде пар $\langle \mathcal{V}^{TypeDecl}, A \rangle$, где $\mathcal{V}^{TypeDecl}$ определяет тип исключения, а A – соответствующую область памяти. Оператор `throw` без параметров описывается как $\langle \top, \top \rangle$, а `throw <имя типа>` как $\langle$*TypeDecl(имя типа)*$, \top \rangle$.

Согласно перечисленным вариантам, если для вершины ААСГ, соответствующей выражению или функции, необходимо знать, имеет ли она побочные эффекты, достаточно вычислить 3 значения: множество вызываемых функций, чтобы проверить наличие в них побочных эффектов; множество модифицируемых областей памяти: тех, значение которых изменяется в отображении *Mem* при построении модели памяти; множество исключений, возможных при выполнении функции. При построении модели памяти для функции побочные эффекты вычисляются практически без накладных расходов путем объединения в родительской вершине результатов вычислений дочерних вершин АСД. По умолчанию консервативно предполагается, что функция имеет побочные эффекты, если это не опровергнуто анализом или явно указанным атрибутом, например, `__attribute__ ((pure))`.

Два выражения имеют *взаимные побочные эффекты*, если результат их вычисления может зависеть от порядка выполнения. Это понятие используется для разрешения ситуаций, подобных примеру в первой главе (листинг 1). В строках 5 и 7 происходит вызов функции `bar`, порядок вычисления аргументов которой не определён стандартом, и каждый аргумент содержит побочный эффект. Вычислив множество не только измененных, но и используемых областей памяти (тех, к которым применяются функционалы $\lambda, \hat{\lambda}$), можно легко находить подобные ситуации, проверяя на пустоту их пересечение.

Разработанный анализатор RuleChecker включает более 50 различных семантических и синтаксических правил из сборников MISRA, JSF, HICPP, требующих минимальное количество ресурсов и работающих без ложных срабатываний. Классы межмодульных и ситуационных правил, требующих дополнительного анализа программы, содержат 14 правил, часть из которых представлены в виде нескольких отдельных в MISRA, однако имеют общую реализацию. Установленные критерии качества ограничивают количество ложных

срабатываний, которое не должно превышать 15% для каждого отдельного правила.

Тестирование анализатора проводится несколькими различными способами. Использование тестового набора для статических анализаторов Juliet не является репрезентативным, поскольку включает большое количество неподдерживаемых ошибок и не содержит тестов для большинства реализованных ограничений, т.к. имеет иное назначение. Тем не менее, было выделено 20 типов поддерживаемых ошибок, 16 из которых покрыты полностью и без ложных срабатываний. На 2 группах было обнаружено 100% ошибок, однако доля ложных срабатываний составила 38% и 50%, что связано эвристиками в потоковой чувствительности, плохо подходящими для искусственных тестов набора. На оставшихся двух группах обнаружено 99% и 60% ошибок в связи с не полностью поддерживаемыми анализатором шаблонами ошибок.

Тестирование производительности RuleChecker производилось на различных типах проектов – от небольших (десятки тысяч строк кода) до огромных (десятки миллионов строк кода). Результаты соответствуют установленным требованиям и представлены в таблице 2:

Таблица 2. Результаты времени работы RuleChecker на открытых программных проектах

Название проекта	**Время сборки, с**	**Время анализа, с**
android 4.3 (24,5 млн. LOC)	1983	771 (39%)
LLVM+Clang 3.5 (2,5 млн. LOC)	434	301 (69 %)
openssl 0.9.8 (450 тыс. LOC)	94	71 (76 %)

В тексте работы также приведены выборочные результаты анализа репрезентативного с точки зрения автора проекта LLVM + Clang. Поскольку компилятор, как правило, разрабатывается высококвалифицированными программистами, имеющими глубокое понимание языка, количество обнаруженных недочетов или, с другой стороны, возможных улучшений наглядно демонстрирует потенциал регулярного использования даже легковесного анализатора при разработке. С другой стороны, при разработке компилятора нередко используется специфичный код, корректный в контексте разработки компилятора, однако требующий сообщения в другом проекте. Это требует более глубокого анализа с целью снижения процента ложных срабатываний. Небольшая выборка результатов анализа представлена в таблице 3. Большая часть этих сообщений была проанализирована вручную и подтверждена их истинность. Таким образом, разработанная система статического анализа обеспечивает высокое качество результатов, требует малое количество ресурсов, а также предоставляет

Таблица 3. Результаты работы анализатора на исходном тексте Clang и LLVM

Правило	**Кол-во**
Тип аргумента sizeof не соответствует типу результирующего указателя (SIZE_CHECK)	1
Исключение, имеющее тип класса, должно перехватываться по ссылке (EXCEPT-15)	3
Недопустимое преобразование между указателем на функцию и типом не void* (MISRA2004-11_1)	7
Деструктор в базовом классе должен быть виртуальным (OOP-24)	11

удобные механизмы для легкого встраивания в обычный процесс разработки любого проекта.

Разработанный анализатор RuleChecker в составе набора средств статического анализа Svace внедрен и используется в подразделениях компании Samsung.

Результаты четвертой главы опубликованы в работах [3, 5, 2].

В Заключении формулируются результаты диссертационной работы и приводятся направления дальнейших исследований. Одним из направлений дальнейших исследований является адаптация моделей программы и памяти для других C-подобных языков программирования (C# и Java) и разработка для них аналогичного анализатора с соответствующим языку набором правил.

Основные результаты диссертационной работы

В диссертационной работе были разработаны математические модели, алгоритмы и программное обеспечение, предназначенные для быстрой и качественной проверки исходного кода на соответствие набору правил. Получены результаты:

1. проведена классификация ограничений по уровням анализа, требуемого для проверки кода программ на соответствие им;

2. разработана модель программы, включающая модель памяти, содержащая всю необходимую информацию для задания правил и проверки кода на соответствие им;

3. построена формальная модель ограничений языка программирования, позволяющая с помощью предложенной нотации задавать стилистические, синтаксические и ситуационные правила на основе разработанной

модели программы;

4. на основе модели программы и модели ограничений языков разработаны и реализованы алгоритмы межпроцедурного и межмодульного анализа программ для обнаружения побочных эффектов, взаимных побочных эффектов, необработанных или неправильно обработанных исключений языка C++, необходимые для проверки программы на соответствие правилам;

5. разработанные модели и алгоритмы реализованы в подсистеме статического анализа компилятора Clang открытой инфраструктуры LLVM.

Список публикаций

1. Игнатьев В. Н. Использование легковесного статического анализа для проверки настраиваемых семантических ограничений языка программирования // Труды Института системного программирования РАН. 2012. Т. 22. С. 169–188.
2. Иванников В. П., Белеванцев А. А., Игнатьев В. Н., Бородин А. Е., Журихин Д. М., Леонов М. И. Статический анализатор Svace для поиска дефектов в исходном коде программ // Труды Института системного программирования РАН. 2014. Т. 26. С. 231–250.
3. Ignatyev V. Static Analysis Usage for Customizable Semantic Checks of C and C++ Programming Languages Constraints // Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on / IEEE. 2014. P. 241–242.
4. Игнатьев В. Н. Формализация ограничений языков C и C++ для их проверки методами статического анализа // Пятый сборник трудов молодых ученых и сотрудников кафедры Вычислительной Техники ИТМО / Под ред. Т. И. Алиева. 2014. С. 17–20.
5. Игнатьев В. Н. Проверка семантических ограничений языков C и C++ с помощью статического анализа // Сборник трудов XXI Международной научной конференции студентов, аспирантов и молодых учёных «Ломоносов-2014». 2014. С. 79897(2).