Использование легковесного статического анализа для проверки настраиваемых семантических ограничений языка программирования

В.Н. Игнатьев
valery.ignatyev@ispras.ru

Аннотация. Для достижения переносимости, надёжности и безопасности программ на C и C++ может применяться введение дополнительных ограничений на язык и стиль программирования. В работе предложен новый метод формализации и классификация таких ограничений, описана система их автоматической проверки, основанная на применении быстрого и нетребовательного к ресурсам статического анализа, использующего средства компилятора, а также способ её интеграции с распространёнными системами сборки проектов.

Ключевые слова: стандарты написания исходного текста программ; статический анализ; LLVM; CLANG.

1. Введение

Языки программирования C и C++ часто используются в крупных проектах. Поиск и исправление ошибок в большом проекте является сложным и дорогим. Статические анализаторы могут применяться на ранних стадиях разработки и позволяют обнаруживать ошибки до тестирования, однако большинство из них требует продолжительного времени для анализа. Поэтому для раннего обнаружения ошибок в программе может быть использован легковесный анализатор, работающий перед компиляцией.

При разработке крупных проектов часто используется введение дополнительных ограничений на возможности языка разработки. Их можно разделить на стилистические, синтаксические и ситуационные. К стилистическим относятся правила форматирования и именования объектов в исходном тексте программ, которые повышают удобочитаемость и облегчают понимание. Синтаксические ограничения позволяют избегать потенциально небезопасных конструкций языка программирования, допускаемых стандартом. Ситуационные ограничения определяют условия, при которых возникают или могут возникать ошибки.

В программах на языках C и C++ неправильное использование некоторых возможностей, как например, неявное приведение типов, работа с памятью и указателями, конструкции, результат работы которых не определён, часто приводит к серьёзным ошибкам. Данная проблема имеет два пути решения:

1) Создание близкого по синтаксису языка, избавленного от основных недостатков. Например, Java и C# имеют строгую типизацию и поддерживают автоматическую работу с памятью.

2) Введение ограничений языка, позволяющих избегать ситуаций, приводящих к ошибкам.

Например, распространённые ошибки, возникающие при разработке на C и C++ изучены и составлены рекомендации для программистов, такие как MISRA C 2004[1], MISRA C++ 2008[2], JSF[3], HICPP[4].

Компилятор проверяет соответствие программы стандарту языка. Используя перечисленные в [1-4] рекомендации, можно составить набор ограничений для использования в конкретном проекте. В результате добавления ограничений к языку C или C++ получится новый язык, сокращающий возможности исходного. Важно, чтобы полученный язык был достаточным для написания программы и набор ограничений был непротиворечивым. Полученный язык не соответствует стандарту исходного (C или C++), поэтому проверяется компилятором не в полном объёме. Его проверка может осуществляться с помощью статического анализатора, описание которого предлагается в работе.

В связи с тем, что описание рекомендаций в сборниках[1-4] производится на естественном языке, они могут по разному интерпретироваться разработчиками и неоднозначно формализоваться. Кроме того, не существует стандартизированной или общепризнанной нотации для формализации ограничений. Различные записи одного и того же правила могут требовать принципиально отличающихся по времени работы, сложности, наличию ложных срабатываний и полученным результатам алгоритмов их проверки. В работе рассмотрено несколько возможностей задания правил и предложен формальный способ их определения. Также описана инфраструктура, на основе которой можно задавать и проверять рассматриваемые ограничения.

Данная работа имеет следующую структуру: в разделе 2 рассматриваются возможные ограничения, существующие методы их формальной записи и классификации, в разделе 3 предлагается метод формального задания ограничений, в 4 разделе рассмотрена их классификация на основе формальной записи. Далее в разделе 5 описана разработанная инфраструктура проверки правил. В заключительном разделе 6 представлены результаты работы анализатора, сформулированы выводы и предложены идеи для дальнейших исследований.

2. Обзор существующих ограничений языка, методик их формализации и классификации

Рекомендации для языков C и C++, на основе которых задаются правила, перечислены в сборниках MISRA C 2004[1], MISRA C++ 2008[2], JSF[3], HICPP[4], а также в книгах, посвящённых программированию на C и C++[6,7]. Они представлены в виде списка с описанием преимуществ использования, примеров нарушений и исправлений ошибок. Обычно этот список является одноранговым, т.е. не имеет структуры, поскольку не найдено однозначных критериев, согласно которым можно было бы систематизировать правила. Производители промышленных анализаторов группируют их по природе обнаруживаемых ошибок: форматирование текста программы; правила, связанные с обработкой исключений; конвенции именования; ошибки работы с памятью. Группировка по терминам языка, к которым правило имеет наибольшее отношение, также используется на практике. Основным недостатком описанных попыток классификации является их неоднозначность, поскольку каждое правило не имеет определённого места в предлагаемой структуре, и может быть отнесено к разным группам.

Задача классификации состоит в выборе признака, согласно которому производится разделение на группы. В статье [8] предлагается структуризация *по целям* правил, которые должны достигаться их соблюдением. Предложено 7 групп:

1. Ясность, чёткость, прозрачность: для избежания непонимания разработчиками.
2. Совместимость: для минимизации стоимости переноса.
3. Защита, безопасность.
4. Производительность.
5. Предсказуемость: для избежания неоднозначной интерпретации.
6. Простота: для сохранения простоты программы, минимизации стоимости последующего тестирования.
7. Правила, регламентирующие процесс разработки.

Систематизация правил, предложенная в [9], использует в качестве признака введённый автором термин «сложность» правила. В статье предложено 6 групп правил:

1. Тривиальные. Поиск простых шаблонов в исходном тексте.

Пример: «не использовать malloc()» (MISRA-C 20.4).

2. Синтаксические. Уточнения грамматики языка.

Пример: «нельзя использовать inline для методов класса» (HICPP 3.1.7).

3. Ограничения на типы. Требуется использование информации о типах.

Пример: «логические выражения могут использоваться в качестве операндов только для операторов &&, ||, !» (MISRA-C 12.6).

4. Структурные. Правило связано со статическими отношениями (наследование, член класса) между объектами в коде. Информация не содержится в дереве разбора, но полностью статична, как например, иерархия наследования.

Пример: «если виртуальная функция базового класса не переопределена в производных, то она не должна быть виртуальной» (HICPP 3.3.6).

5. Динамические. Правило определяет последовательность событий при выполнении. Использование графа потока управления недостаточно, требуется модель памяти.

Пример: «все автоматические переменные должны быть инициализированы» (MISRA-C 9.1).

6. Неавтоматизируемые. Правила, которые сложно или невозможно формализовать, а также использующие невычислимые свойства.

Пример: «реализация функциональности в классе не должна дублироваться» (HICPP 3.1.9).

Программные ошибки, с которыми связано правило, являются другим признаком для классификации. В основе классификации правил в этом случае лежит классификация ошибок, которая на данный момент изучена гораздо лучше. Ошибки можно группировать по их значимости, причинам, источникам. Описание методов классификации ошибок рассмотрено в статье [11].

С точки зрения [10] невозможно создать классификацию, в которой каждое правило будет принадлежать только одной категории. В связи с этим предлагается группировка небольшого количества правил по большому количеству категорий и построение дерева категорий.

Рассмотренные систематизации правил не позволяют решить задачу классификации, поскольку не задают формального критерия, согласно которому правило может быть отнесено к определённому классу. Кроме того, некоторые правила, описанные в [1-4] не могут быть отнесены ни к одной группе предлагаемых систематизаций. Для создания классификации требуется формальный язык определения правил, на основе которого могут быть заданы критерии классификации.

На данный момент не существует общепринятого языка описания правил, поэтому они задаются на естественном языке. В работах, исследующих методы их статической проверки, предложено несколько способов формального описания. Так в [9, 10] предложены декларативные языки на основе Пролога, позволяющие определять и проверять некоторое множество правил, а в [10] ещё и задавать базу знаний о программе. Например, правило «Переменные, являющиеся членами класса, не должны быть публичными (public)», записанное на языке из [10] выглядит следующим образом:

```
faulty_cldata :-
      get_classdefinition(Class),
```

```
      get_data_member_in_classdefinition(Class,
Var_decl),
      get_protection_flag_of_variable_decl(Var_decl,
Prot_flag),
      is_protection_flag_public(Prot_flag),
      writef(«Rule Violated: Public data member in
class»),
      print_name_and_location_of_classdefinition(Class).
```

Предложенный авторами язык используется не только для определения правил, но и для описания базы знаний, которая представляет собой набор фактов о программе, полученных во время её анализа. Для проверки соответствия программы заданному набору ограничений используется интерпретатор Пролога, исходными данными которого являются формализация правил, база знаний и библиотека описания интерфейсных функций, а в результате получается список предупреждений.

В большинстве крупных проектов существуют *неявные правила* [12]. К ним относятся ограничения, имеющие отношение только к конкретному проекту. Для их описания в статье [12] используется расширенный авторами язык шаблонов из статьи [13]. Правило «перед вызовом функции ChangeView необходимо установить глобальную переменную CtrlNo», записанное на предлагаемом языке выглядит следующим образом:

```
      $vg_1 = 'CtrlNo' // переменная CtrlNo – глобальная
      $f_1 = 'ChangeView' // функция ChangeView %%
      ^@$vg_1 = #; // отсутствие выражения
      @$f_1(#*);
```

Использованная нотация создавалась для рефакторинга программных систем, поэтому её применение для формализации ограничено. Она позволяет задавать лишь небольшое множество правил в связи с тем, что возможен доступ к небольшому числу объектов языка программирования.

Как видно из приведённых примеров, для записи формализации правил авторы предлагают различные синтаксические конструкции: декларативный проблемно-ориентированный язык, язык программирования общего назначения (Пролог).

3. Формализация ограничений

Ограничение языка программирования (правило) – это предикат, на вход которому подается модель программы или её части, а результатом является одно значение из двухэлементного множества {И, Л}, соответствующее истинности и ложности. Пусть **p** — корректная согласно стандарту программа на языке C или C++. Исходный текст программы является строкой символов. Пусть $\mathbf{m_s}$ – модель программы **p**, задаваемая в виде исходного текста программы и представляемая в виде строки. В процессе компиляции программа имеет несколько различных представлений. На вход лексическому

анализу компилятора подается модель $\mathbf{m_s}$ программы, а полученную в результате его работы модель назовем $\mathbf{m_{lex}}$. Она состоит из последовательности лексем и таблицы символов. Поскольку рассматриваемая программа корректна, то лексический анализ всегда успешно завершается и множество диагностических сообщений пусто. Объектом модели $\mathbf{m_{lex}}$ является лексема (токен), для которого заданы атрибуты: тип лексемы, положение в модели $\mathbf{m_s}$, ссылка на таблицу символов для некоторых типов лексем. Кроме перечисленных атрибутов, вычисляемых компилятором, анализатор создает дополнительные необходимые для анализа атрибуты. Например, компилятор не различает табуляцию, количество символов «пробел» и «перевод строки», которые используются для проверки стилистических правил.

Синтаксический анализ компилятора по модели $\mathbf{m_{lex}}$ строит модель программы $\mathbf{m_{synt}}$, состоящую из абстрактного синтаксического дерева, узлы которого делятся на два основных класса: узлы, соответствующие объявлениям функций, переменных, типов, классов и т.д. в программе, и узлы, соответствующие выражениям в программе. Объектом модели $\mathbf{m_{synt}}$ является узел дерева. Для каждого узла задан его тип, определяющий какому объекту в программе он соответствует, например, «объявление функции» или «бинарный оператор». В зависимости от типа, узел дерева имеет различные наборы атрибутов. После семантического анализа из модели $\mathbf{m_{synt}}$ строится модель $\mathbf{m_{sem}}$, отличающаяся наличием дополнительных атрибутов.

В процессе компиляции для выполнения оптимизаций обычно используется одно или несколько внутренних представлений программы. Одним из таких представлений является биткод LLVM, модель которого назовем $\mathbf{m_{llvm}}$. Объектом в данной модели является инструкция виртуальной машины LLVM с соответствующим её типу множеством атрибутов.

Программа может состоять из нескольких файлов. За один запуск компилятор обычно обрабатывает один модуль компиляции, которые затем объединяются в исполняемый файл редактором связей. Для каждого модуля компиляции, подающегося на вход редактору связей, существует представление в виде модели $\mathbf{m_{llvm}}$. Модель программы, состоящую из множества моделей $\mathbf{m_{llvm}}$ модулей компиляции, назовём $\mathbf{m_{im}}$.

Введённые модели программы задают область определения ограничений. В определении правила указано, что оно может применяться не только ко всей модели, но и к её части. Для каждой модели определён объект, являющийся минимальной неделимой частью. При разделении модели минимальная часть соответствует объекту данной модели и правило может применяться к каждому объекту или к их последовательности, определяемой моделью. Таким образом, алфавит формальной модели правил состоит из объектов модели программы. Для задания формальной модели правил необходимо введение функциональных символов и предикатов на множестве правил. Пусть r, s, q – правила, m – модель программы, тогда модель системы ограничений задается следующим образом:

Алфавит — множество объектов модели m**;**

Предикаты:

1. Двуместный предикат *равенство*:
   $\forall r, s : =(r,s) \Leftrightarrow r = s \Leftrightarrow \forall m : r(m) = s(m);$

Функциональный символы (ФС):

- Одноместный ФС *отрицание:*
  $\forall r \exists s : s = \neg(r) \Leftrightarrow s = r \Leftrightarrow \forall m : r(m) = s(m);$
- Двуместный ФС *конъюнкция*:
  $\forall r, s \exists q : q = \wedge(r,s) \Leftrightarrow q = r \wedge s \Leftrightarrow \forall m : q(m) = r(m) \wedge s(m);$

Полученная модель изоморфна модели логики первого порядка и позволяет задавать одни правила с помощью других. Множество правил R является *непротиворечивым*, если $\exists m : \wedge_{r \in R} r = И$.

В качестве примера формального определения ограничения рассмотрим правило: «нельзя использовать операции сравнения на равенство и неравенство с операндами нецелочисленного типа». Для этого будем использовать модель программы $\mathbf{m}_{\mathbf{sem}}$. Операция сравнения в указанной модели представлена в виде узла дерева *BinaryOperation* с атрибутом *BinaryOperation::kind* равным *BinaryOperation::EQ* или *BinaryOperation::NEQ*. Для данного узла заданы два операнда, также являющиеся узлами дерева разбора, и доступные с помощью атрибутов *LHS* и *RHS*. Независимо от представляемого ими объекта программы, для них определён атрибут *Type* соответствующий типу рассматриваемого узла. Введем специальное правило, принимающие на вход узел объявления типа, и проверяющее, является ли он типом с плавающей запятой.

```
isFloatingPoint (Type t) ⇔
      t.CanonicalType = BuiltinType::Float ∨
      t.CanonicalType = BuiltinType::Double ∨
      t.CanonicalType = BuiltinType::LongDouble
```

Тогда рассматриваемое правило будет иметь вид:

```
hasFpEqNeqCmp (BinaryOperator bop) ⇔
      (bop.kind == BinaryOperation::EQ ∨ bop.kind ==
BinaryOperation::NEQ) ⇒
      (isFloatingPoint(bop.LHS.Type) ∧
isFloatingPoint(bop.RHS.Type))
```

Для записи ограничений не всегда удобно использовать язык логики предикатов, потому что сформулированные подобным образом ограничения получаются громоздкими и трудны для понимания. В разработанном анализаторе для задания правил используется язык общего назначения C++. Для каждой модели программы в анализаторе присутствуют соответствующие объекты с методами доступа к их атрибутам, например, лексема представлена классом Token с атрибутами TokenKind, SourceLocation . Правило

представляет собой класс, реализующий необходимые методы базового класса. На вход методам обычно подается объект модели (лексема, узел дерева разбора требуемого типа). Разработанная библиотека вычисляет ряд дополнительных атрибутов, упрощающих задание правил.

4. Классификация правил

В разделе о формализации описаны модели программы. Ограничения задаются на определённой модели, в которой доступны все необходимые для формализации атрибуты. Для каждой модели определено подмножество правил и их пересечение является пустым множеством. Таким образом, правила однозначно разделяются на классы в соответствии с моделью программы, на которой они задаются. В компиляторе одна модель обычно строится на основе другой. Например, для построения $\mathbf{m}_{\mathbf{llvm}}$ используется $\mathbf{m}_{\mathbf{sem}}$, которая в свою очередь использует $\mathbf{m}_{\mathbf{synt}}$, строящуюся на основе $\mathbf{m}_{\mathbf{lex}}$. Предложенная классификация является однозначной и характеризует сложность правила, т.е. относительное количество ресурсов, требуемых для его проверки. Модель программы задаёт множество объектов и атрибутов. Использованное в формальной записи правила множество объектов и атрибутов определяет класс правила.

В качестве примера рассмотрим правило «не использовать malloc()»(MISRA-C 20.4), формально записанное следующим образом на основе модели $\mathbf{m}_{\mathbf{synt}}$:

isMallocUsed(CallStmt s) $\Leftrightarrow$ *s.CalledFunc.Name = «malloc»*

Узел дерева типа CallStmt имеет атрибут, указывающий на объявление вызываемой функции, который в свою очередь имеет атрибут — имя функции. Таким образом, правило задается в модели $\mathbf{m}_{\mathbf{synt}}$ и принадлежит классу синтаксических правил.

С помощью предложенного критерия все правила можно распределить на 4 основных класса, строго определяющих положение каждого правила. При этом сохраняется возможность группировки правил внутри основных пунктов классификации по тем или иным нестрогим критериям.

4.1. Лексические ограничения

В обычном режиме работы компилятор не сохраняет данные лексического анализа (информацию о лексемах), а также пропускает лексемы, соответствующие комментариям, не различает пробелы, табуляцию, перевод строки и их количество, поскольку данная информация не влияет на семантику программы. Однако для определения некоторых правил необходима данная информация. В модели $\mathbf{m}_{\mathbf{lex}}$ вся информация присутствует и доступна в анализаторе.

К классу лексических ограничений относятся правила, формализуемые на основе объектов, описывающих лексемы. В терминах неформального языка описания правил к данному классу относятся ограничения, содержащие слова:

ключевое слово языка и их полный список, строка, символ, токен, лексема, знаки операций («&&», «?:», «+» и т.д.), 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. В качестве примеров лексических правил можно привести следующие:

«каждый модуль компиляции должен содержать в начале комментарий в формате doxygen[14]» или «каждый заголовочный файл должен быть заключён в директивы #ifndef FILE_NAME_H #define FILE_NAME_H ... #endif».

4.2. Синтаксические ограничения

Синтаксический анализ или *разбор(parsing)* являются второй фазой компиляции. Анализатор использует информацию о токенах, полученных после лексического анализа для создания древовидного промежуточного представления, узлы которого являются объектами модели $\mathbf{m}_{\mathbf{synt}}$. Модель $\mathbf{m}_{\mathbf{synt}}$ определяет объекты, соответствующие каждому типу выражений, например, унарный и бинарный оператор, выражение if-then-else и каждому типу объявлений, например, функции, типа, переменной, класса. Для этих объектов определены атрибуты, такие как имя, аргументы, квалификаторы в соответствии с типом объекта.

Правила, формализованные с помощью объектов модели $\mathbf{m}_{\mathbf{synt}}$, принадлежат классу синтаксических. Связь между $\mathbf{m}_{\mathbf{synt}}$ и $\mathbf{m}_{\mathbf{lex}}$ обеспечивается в контрольных точках, соответствующих началу объекта в $\mathbf{m}_{\mathbf{synt}}$, т.е. для выражения *if* задан атрибут StartToken, указывающий на лексему - ключевое слово *if* . Таким образом, например, обеспечивается проверка правила «перед каждой функцией должен быть комментарий в формате doxygen».

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.

4.3. Контекстные ограничения

К классу контекстных правил относятся ситуационные ограничения, зависящее от состояний нескольких объектов модели программы. Например, правило «недопустимо отсутствие оператора return в не void функции» зависит от состояния объекта, соответствующего объявлению функции, и от наличия в теле функции объекта ReturnStmt на всех путях выполнения. При этом проверка второго условия производится в контексте наличия первого.

Для определения правил данного класса используются две модели программы: $\mathbf{m}_{\mathbf{sem}}$, отличающаяся от $\mathbf{m}_{\mathbf{synt}}$ наличием дополнительных атрибутов, и $\mathbf{m}_{\mathbf{llvm}}$ –

промежуточное языково-независимое внутреннее представление, объектом которой является инструкция виртуальной машины LLVM. В связи с этим, в данном классе можно выделить подклассы языково-зависимых и языково-независимых ограничений, заданных на $\mathbf{m}_{\mathbf{sem}}$ и $\mathbf{m}_{\mathbf{llvm}}$ соответственно.

Статический семантический анализ осуществляет проверку исходной программы на семантическую согласованность со стандартом языка на основе синтаксического дерева и информации из таблицы символов. Во время анализа осуществляется распространение информации о типах. На данном этапе компилятор решает похожую на поставленную задачу: осуществляет проверку правил, только их источником является стандарт языка.

Большинство правил описывается в терминах языка программирования, поэтому для них лучше подходит языково-зависимое представление программы. Проверка формализованных с помощью модели $\mathbf{m}_{\mathbf{sem}}$ правил осуществляется после семантического анализа. Все доступные для формализации атрибуты делятся на вычисляемые компилятором и анализатором. Для вычисления некоторых атрибутов на модели $\mathbf{m}_{\mathbf{sem}}$ анализатор строит дополнительные структуры данных, такие как, например, граф потока управления, граф вызовов, информация о границах значений переменных.

Для правил, формализованных на модели $\mathbf{m}_{\mathbf{llvm}}$ доступны атрибуты, вычисляемые в процессе анализа и оптимизаций компилятором. К ним относятся атрибуты для доступа к информации о потоках данных и управления, анализа псевдонимов, циклов.

В качестве примера приведено несколько групп правил, принадлежащие классу семантических:

1. Структурные правила внутри одного модуля компиляции, связанные со статическими отношениями в программе, не представленными в дереве разбора (наследование, член класса), проверка которых полностью осуществляется внутри одного модуля компиляции. Подробнее о проверке таких правил во всей программе можно прочитать в [9].

2. Ограничения на типы.
1) Нельзя использовать оператор сравнения на равенство для операндов не целочисленного (с плавающей запятой) типа .
2) Нельзя использовать переменные, квалифицированные как volatile в выражениях более одного раза.

3. Уточнения грамматики языка.
1) Отсутствие оператора return в функции, определённой не void .
2) Ограничения преобразования типов. (Неявное преобразование большего к меньшему, преобразование указателя на функцию к типу не «void *»).

В качестве примера правил, реализуемых на языково-независимом внутреннем представлении, можно привести следующие группы, использующие

1. анализ циклов: счётчик цикла должен изменяться только в одном месте, условие цикла не должно изменяться внутри цикла;
2. продвижение констант и анализ границ значений переменных: разыменование нулевого указателя, выход за границы массива.

4.4. Межмодульные статические правила.

Следующим после компиляции обычно выделяется этап связывания (linking), во время которого результирующая программа собирается из объектных файлов и устанавливаются связи между функциями и переменными, определёнными в разных модулях и библиотеках. На этом этапе современные компиляторы могут также осуществлять ряд оптимизаций, называемых оптимизациями времени связывания, а также он является подходящим для проверки межмодульных статических правил. Исходными данными для проверки этой группы правил являются файлы с промежуточным представлением (например, LLVM), а также 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.

Предлагаемая классификация является однозначной и позволяет определять оптимальную последовательность анализа правил, а также сравнительную сложность (количество ресурсов), необходимые для осуществления их проверки. Таким образом можно ограничивать количество сообщений, не проводя ресурсоёмких вычислений.

Предложенная классификация включает в себя описанные в [12] неявные правила, которые специфичны для конкретной системы, редко описаны в спецификациях и содержатся в опыте разработчиков программы. В качестве примера такого правила можно привести следующее: «перед вызовом функции A необходимо установить глобальную переменную B». В зависимости от необходимых для их проверки данных о программе они распределены по нескольким классам, например, приведённое правило принадлежит классу контекстных правил.

5. Подсистема проверки правил

5.1. Особенности основных компонентов системы

Для реализации анализатора используются открытые компиляторные инфраструктуры LLVM[15] и CLANG[16]. Разработанная система состоит из

трёх основных компонентов: подсистема определения правил, сбора информации и планировщик правил.

Для определения ограничений применяется язык C++, используемый в реализации LLVM и CLANG. Правила в анализаторе задаются в виде классов, унаследованных от базового класса Pass, определяющего набор интерфейсных методов. Список необходимых правилу объектов модели программы передается с помощью параметров шаблона базового класса. Для обработки каждого объекта правило должно определить соответствующий метод. Например, правилу проверки исключений требуются объекты CXXThrowExpr, CXXCatchStmt, CallExpr для сбора информации и необходим запуск в конце обработки модуля компиляции для анализа. Тогда его объявление будет выглядеть следующим образом:

```
class ExceptionObjectCheckRule :
   public Pass<process::ASTStmt<CXXThrowExpr>,
               process::ASTStmt<CXXCatchStmt>,
               process::ASTStmt<CallExpr>,
               process::TranslationUnitFinalize> {
  public:
    void processStmt(const CXXThrowExpr *, RCContext *)
const;
    void processStmt(const CXXCatchStmt *, RCContext *)
const;
    void processStmt(const CallExpr *, RCContext *)
const;
    void finalizeTU(const TranslationUnitDecl *,
RCContext *) const;
};
```

Подсистема определения правил предоставляет интерфейс для доступа к объектам модели программы и их атрибутам. Использование C++ позволяет создать библиотеку, в которой объекты модели программы представлены в виде классов, а доступ к их атрибутам осуществляется с помощью методов. Например, термин лексема имеет следующий интерфейс:

```
class Token {
public:
      TokenKind getKind();
      std::string getContent();
      SourceLocation getSourceLocation();
      bool hasLeadingSpace();
}
```

Подобным образом определены остальные объекты. Использование методов позволяет получать доступ к атрибутам терминов по мере их вычисления компилятором или анализатором. В качестве примера рассмотрим правило «не использовать пробелы вокруг '.', '->' ». Для его проверки недостаточно

информации об одной лексеме, поэтому на вход поступает список лексем, и реализация выглядит следующим образом:

```
class SpaceAroundMemberAccessRule : public Pass <
process::TokenList > {
public:
  void processTokenList(const TokenList lst) const {
    for (auto i = lst.begin(), e = lst.end(); i != e;
++i) {
      Token *tok = *i;
      if ((tok->getKind() == TokenKind::arrow) ||
         (tok->getKind() == TokenKind::period)) {
        Token *next = *(i + 1);
        if (tok->hasLeadingSpace() || next-
>hasLeadingSpace()))

ReportRuleCheckerMessage(rcdiag::space_around_member_acc
ess, Token->getSourceLocation());
      }
    }
  }
};
```

Класс, задающий правило, получает доступ только на чтение к общим структурам данных анализатора, что обеспечивает их неизменность. Данные, необходимые для проверки ограничения в пределах одного модуля компиляции, хранятся в полях класса. Это создает возможность параллельной обработки объектов модели разными правилами. Описанный интерфейс позволяет переписать формальную запись правил на языке C++.

Подсистема сбора информации также использует интерфейс определения правил для доступа к объектам модели программы. Результат её работы необходим при вызове методов доступа к вычисляемым анализатором атрибутам объектов. Часть из них вычисляется непосредственно в момент вызова. Это позволяет исключить избыточный анализ в случае, если атрибут объекта не потребуется ни для одного правила. Для часто используемых атрибутов анализ осуществляется перед запуском проверки ограничений. Собранная информация сохраняется в специальное хранилище – базу данных SQLite[19] и доступна несколькими независимыми правилами при анализе разных модулей компиляции, т. е. в нескольких запусках анализатора. Таким образом функционирует проверка межмодульных правил. Они состоят из классов собирающих и сохраняющих в базу необходимую информацию, обрабатываемую на этапе связывания. Хранилище используется, например, при анализе исключений, которые могут возникать и перехватываться внутри функций, определённых в разных модулях компиляции. Подсистема работы с базой данных используется менеджером правил и подсистемой сбора информации. Возможность сохранения результата работы анализатора в БД

позволяет выполнять SQL запросы и, например, собирать статистику. Планировщик сохраняет в БД информацию о просмотренных заголовочных файлах, что исключает их повторный анализ и сокращает время работы анализатора.

В процессе анализа планировщик правил обходит промежуточное представление программы (список лексем, дерево разбора, граф потока управления, граф вызовов) и вызывает необходимые интерфейсные методы правил. Поскольку правила не меняют внутреннее представление, то они могут запускаться параллельно, что сокращает время анализа. Упрощённая схема представлена на рисунке 1.

Использование достаточно низкоуровневого языка C++ совместно с планировщиком проверки ограничений сокращает время анализа и обеспечивает возможность написания новых и параметризации существующих проверок.

5.2. Использование анализатора в процессе разработки

Разработанный анализатор поддерживает различные системы сборки проектов для автоматического запуска и анализа только той части исходных текстов, которая компилируется в результирующую программу. Для каждого запущенного при сборке процесса с помощью переменной окружения LD_PRELOAD принудительно загружается разработанная динамическая библиотека, которая размещается в памяти и производится динамическое связывание. В процессе компиляции проекта функции группы exec (execve, execv, execl, execvp, execlp) заменяются описанными в библиотеке. Новые функции анализируют командную строку и запускают анализатор в случае обнаружения там пути к компилятору или редактору связей. Такая система позволяет использовать практически любой компилятор и любую систему сборки. Для удобства используется программа-оболочка, обрабатывающая входные параметры и запускающая команды сборки. Предложенный подход был также адаптирован для системы кросс-компиляции scratchbox[20].

5.3. Результаты работы анализатора

Реализованная система способна обнаруживать более 50 различных дефектов программ. В результате запуска анализатора на примере LLVM с CLANG (~1250 тыс. строк кода) были получены следующие результаты:

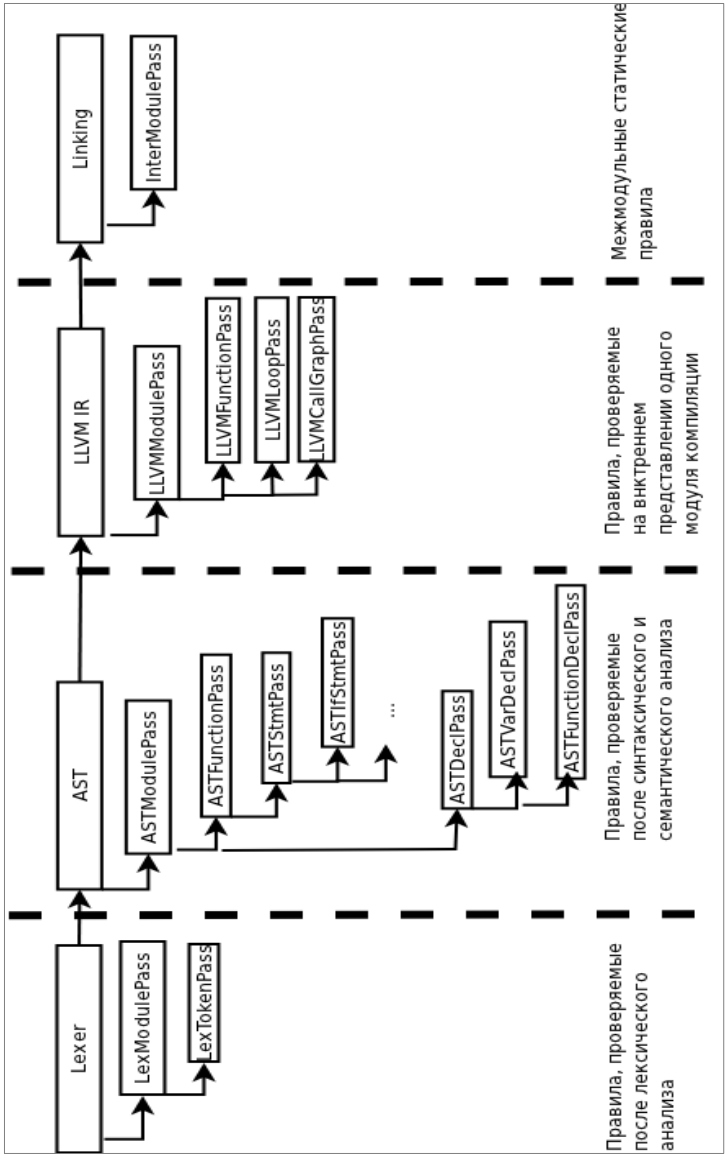

Рис. 1. Структура проверки правил

Коли-чество ошибок	Название правила	Комментарий
1	SIZE_CHECK	Тип аргумента sizeof не соответствует типу результирующего указателя
3	EXCEPT-15_DMC	Исключение, имеющие тип класс должно перехватываться по ссылке
7	MISRA2004-11_1_DMC	Недопустимое преобразование между указателем на функцию и типом не void*
11	OOP-24	Деструктор в базовом классе должен быть объявлен virtual
13	MISRA2004-12_2_d	Результат зависит от порядка вычисления аргументов
25	PB-17	Присваиваемое значение больше размера типа unsigned char
30	SECURITY-12_DMC	Использование небезопасной функции работы со строками
31	MRM-43	Переопределённый оператор присваивания должен копировать все члены класса, включая базовые
41	MISRA2004-12_8_a_DMC	Аргумент оператора сдвига должен быть больше 0 и меньше битового размера типа сдвигаемого значения
76	MISRA2004-13_3_DMC	Недопустимо сравнение на равенство неравенство не целочисленных операндов
116	PB-20	Недопустим slicing параметров функции / возвращаемого значения
118	MISRA2008-15_3_4	Исключение не имеет обработчика
124	MISRA2004-10_1_b-3	Неявное преобразование между целочисленным и не целочисленным значением
129	MISRA2004-17_2	Вычитание указателей допустимо только для указателей на один и тот же массив
150	MISRA2004-17_3	Сравнение (<, <=, >, >=) указателей допустимо только для указателей на один и тот же массив

156	MISRA2004-12_3_DMC	Аргумент sizeof имеет sideeffect
218	MISRA2004-13_1_DMC	Оператор присваивания использован в выражении логического типа
275	MISRA2004-12_2_b_DMC	Результат зависит от порядка вычисления аргументов функции
796	INIT-06	Все переменные должны быть инициализированы в конструкторе

Табл. 1. Результаты работы анализатора на LLVM и CLANG(~1250 тыс. строк кода)

В случае использования программы-оболочки, которая наряду с обычной сборкой приложения осуществляет запуск CLANG и нескольких проходов анализа представления LLVM, были получены следующие результаты:

Пакет	GCC, сек.	GCC + Анализатор LLVM, сек	Замедление, %
binutils	73.88	90.52	22.52%
glib	21.14	25.96	22.80%
sqlite	38.58	46.80	21.31%

Большая часть сообщений анализатора была проанализирована вручную и подтверждена их истинность.

Таким образом, использование анализатора замедляет сборку приблизительно на 22%. Приведённые результаты показывают, что применение встроенного в систему сборки анализатора допустимо при каждой компиляции и позволяет обнаруживать ошибки на самой ранней стадии разработки.

6. Заключение

Для решения задачи автоматической проверки настраиваемых ограничений и поиска распространённых ошибок в исходном коде программы система должна удовлетворять следующим критериям:

1. Малые дополнительные ресурсы, требуемые для проверки программы.

Промышленные статические анализаторы реализуют большое количество правил, а также способны обнаруживать сложные дефекты, требующие долговременного анализа. В разработанной системе существует планировщик правил, обеспечивающий оптимальную последовательность их проверки, а

также не используются эвристические и ресурсоёмкие алгоритмы. В результате этого время работы анализа сравнительно невелико.

2. Интеграция в процесс разработки

Описанная в разделе 5.3 подсистема интеграции обеспечивает возможность удобного использования анализатора с различными компиляторами и системами сборки проекта.

3. Использование результатов статического анализа на самых ранних этапах разработки.

Первым этапом тестирования написанного текста программы является компиляция. Разработанный анализатор может работать перед компиляцией.

4. Возможность отключения и конфигурирования существующих проверок, а также добавления новых.

В большинстве проектов конвенции написания исходных текстов уже выбраны, поэтому система имеет возможность гибко подстраиваться под существующие, а также позволяет не только параметризовать правила, но и задавать новые. Для этого применяется подсистема определения правил и планировщик, который в зависимости от принадлежности правил тому или иному пункту классификации, определяет порядок их проверки.

Дальнейшие работы будут направлены на исследование применимости различных видов анализа программ, которые позволят осуществлять проверку более сложных ограничений, реализацию новых ограничений и построение системы ограничений для каждого из языков C и C++.

Список литературы

[1] «Guidelines for the use of the C language in critical systems», MISRA-C:2004, 2004
[2] «Guidelines for the Use of the C++ Language in Critical Systems», ISBN 978-906400-03-3 (paperback), June 2008.
[3] «Joint Strike Fighter air vehicle C++ coding standards for the system development and demonstration program», Lockheed Martin Corporation, 2005
[4] «High Integrity C++ Coding Standard Manual - Version 2.4», The programming research group, 2007
[5] The annotated ANSI C Standard American National Standard for Programming Languages—C: ANSI/ISO 9899-1990
[6] «Code Complete», S. McConnel, Microsoft Press; 1 edition (January 1, 1993)
[7] «The Practice of Programming», Brian W. Kernighan, Rob Pike, Addison-Wesley Professional; 1 edition (February 14, 1999)
[8] «Coding standards for high-confidence embedded systems», Paul Anderson, MILCOM2008
[9] «A Coding Rule Conformance Checker Integrated into GCC», Marpons, Marino, Carro, Herranz, Fredlund, Moreno-Navarro, and Polo, Electronic Notes in Theoretical Computer Science, 2007
[10] «A tool for checking coding standards for C++», S. Mohammed Saleem (Master of technology thesis), 1999

[11] «Standard error classification to support software reliability assessment», John B. Bowen, afips, pp.697, 1980 Proceedings of the National Computer Conference, 1980
[12] «The Detection of Faulty Code Violating Implicit Coding Rules», Tomoko Matsumura, Akito Monden, Ken-ichi Matsumoto, isese, pp.173, 2002 International Symposium on Empirical Software Engineering (ISESE'02), 2002
[13] «A Framework for Source Code Search Using Program Patterns», S. Paul, A. Prakash, IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 463-475, June 1994
[14] Doxygen. Средство автоматической генерации документации. http://www.doxygen.org
[15] «LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation», Chris Lattner, Vikram Adve, cgo, pp.75, International Symposium on Code Generation and Optimization (CGO'04), 2004
[16] CLANG: C, Objective C, C++ frontend for LLVM, Apple, Inc, http://clang.llvm.org/
[17] «Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ.» А. Аветисян, А. Белеванцев, А. Бородин, В. Несов. Труды Института системного программирования РАН, том 21, 2011 г, стр. 23-38
[18] «Механизмы расширения системы статического анализа Svace детекторами новых видов уязвимостей и критических ошибок.» А. Аветисян, А. Бородин. Труды Института системного программирования РАН, том 21, 2011 г, Стр. 39-54
[19] «The Definitive Guide to SQLite», Allen, Grant; Owens, Mike (November 5, 2010). (2nd ed.). Apress. p. 368. ISBN 1430232250.
[20] Инфраструктура для кросс-компиляции. http://www.scratchbox.org/
[21] Standard – the C++ language. Report ISO/IEC 14882: 1998, Information Technology Council (NCTIS).
[22] «О некоторых задачах анализа и трансформации программ.» С. Гайсарян, А. Чернов, А. Белеванцев, О. Маликов, Д. Мельник, А. Меньшикова. Труды Института системного программирования РАН, том 5, 2004 г. Стр. 7-40.