Обнаружение дефектов работы с указателями в программах на языках C и C++ с использованием статического анализа и логического вывода

Татьяна Верт[1], Татьяна Крикун[1] и Михаил Глухих[2]

[1] Санкт-Петербургский государственный политехнический университет, Россия,
werttanya@mail.ru, krikuntatiana@mail.ru,
[2] Технический университет Клаусталя, Германия,
mikhail.glukhikh@gmail.com

Аннотация В статье рассматривается один из способов повышения точности обнаружения дефектов при помощи статического анализа, а именно дополнение классических алгоритмов анализом зависимостей. Предлагается в процессе абстрактной интерпретации извлекать информацию как об известных статически значениях переменных, так и о зависимостях между неизвестными значениями, представляемых предикатами логики первого порядка. Это позволяет при проверке условий наличия дефектов, а также при анализе ветвлений, использовать готовые средства логического вывода для доказательства истинности или ложности соответствующих условий. Основной упор сделан на логику анализа указателей и на правила обнаружения дефектов работы с указателями. Приведены результаты исследования программного прототипа, использующего Microsoft Z3 Solver в качестве средства вывода. Показано значительное повышение точности анализа, предложены пути снижения ресурсоёмкости данного метода.

Keywords: обнаружение дефектов работы с указателями, статический анализ, логический вывод

1 Введение

Проблема обеспечения надёжности программного обеспечения является одной из ключевых в современном мире. Известно [6], что даже в хорошо отлаженных программах содержится до 0.7 ошибок на 1000 строк исходного кода. Одним из наиболее распространённых классов ошибок являются ошибки работы с указателями.

Основным методом проверки качества программного обеспечения попрежнему является тестирование. Не умаляя достоинств данного метода, следует отметить, что тестирование не дает гарантий обнаружения всех

имеющихся в программе ошибок и поэтому в ряде случаев не может являться единственным методом проверки качества. Подобную гарантию могут предоставить только статические методы, одним из которых является статический анализ кода [23][20].

Ключевыми характеристиками методов обнаружения ошибок в программном коде являются полнота, точность и ресурсоёмкость. *Полнота* анализа вычисляется как доля обнаруженных истинных ошибок среди всех имеющихся ошибок в программе, а *точность* —как доля обнаруженных истинных ошибок среди всех обнаруженных. Как правило, три данные характеристики находятся в противоречии друг с другом, и в средствах обнаружения ошибок приходится выбирать разумный компромисс. В частности, промышленные средства статического анализа [11] обычно ориентированы на высокую точность и низкую ресурсоёмкость, но не гарантируют полноту. Исследовательские средства анализа часто стремятся к обнаружению всех имеющихся ошибок определенного класса, что приводит к понижению точности и росту ресурсоёмкости; во многих случаях подобные характеристики препятствуют широкому распространению средств статического анализа [20].

Распространённым алгоритмом статического анализа является абстрактная интерпретация [13]. Данный алгоритм подобен обычной интерпретации. Однако, абстрактная интерпретация осуществляет анализ всех путей программы, обеспечивая таким образом полноту анализа. При этом для хранения возможных значений переменных используются так называемые абстрактные семантические домены — в частности, множество целочисленных интервалов, множество указателей и т.д. Абстрактная интерпретация допускает как раздельный анализ путей программы, имеющий высокую точность и очень высокую ресурсоёмкость, так и совместный анализ путей, имеющий приемлемую ресурсоёмкость, но низкую точность за счёт объединения возможных значений переменных при слиянии путей.

Одним из способов повышения точности при совместном анализе путей является дополнение классических алгоритмов анализом зависимостей [19]. *Зависимостью* здесь называется произвольная связь между значениями двух и более переменных программы; математически, зависимость может быть представлена в виде предиката [22]. Анализ зависимостей позволяет полностью или частично компенсировать погрешность, вызванную слиянием путей в ходе статического анализа. Для реализации анализа зависимостей могут быть использованы готовые методы и средства логического вывода, такие, как аппарат дизъюнктов Хорна и средства логического программирования [18], логика первого и высших порядков [22], SMT-решатели [3]. Подобная возможность позволяет, в числе прочего, упростить другие используемые алгоритмы, в частности, использовать анализ точных значений вместо интервального анализа. В [15] Патрик Кузо рассматривает одну из возможных математических моделей объединения абстрактной интерпретации и логического вывода. Насколько известно авторам статьи, на данный

момент эта модель не реализована в рамках существующих средств статического анализа.

Ключевой идеей данной работы является использование сочетания методов статического анализа и логического вывода для обнаружения ошибок в исходном коде. В качестве основной группы ошибок была выбрана некорректная работа с указателями, однако предлагаемый подход может быть расширен и на другие группы ошибок. В разделе 2 приведён обзор существующих работ по рассматриваемой тематике. В разделе 3 рассматривается предлагаемый подход. Раздел 4 посвящен практической реализации подхода и проведённым экспериментальным исследованиям. Раздел 5 подводит итоги и намечает направления дальнейшего развития подхода.

2 Обзор существующих работ

В настоящее время, известно ряд как коммерческих, так и исследовательских средств статического анализа [9]. Ведущим коммерческим средством обнаружения дефектов методом статического анализа де-факто является платформа Coverity SAVE [8]. Концептуально данное средство ориентировано на анализ программ произвольного размера и обеспечение высокой точности — от 80% до 90% по данным авторов. Средство не гарантирует полноты анализа, но тем не менее обнаруживает довольно большое количество серьёзных ошибок — порядка 7 ошибок на 10000 строк кода — в реальных программных проектах [11,6]. Подход средства Coverity SAVE основан на извлечении из исходного кода правил использования различных его объектов [17], при этом извлекаются как точные правила вида «указатель P не NULL», так и статистические правила вида «mutex M, возможно, защищает переменную V». В дальнейшем эти правила проверяются на непротиворечивость с целью обнаружения ошибок в программе.

Исследовательское средство Astree [7] разработано под руководством Патрика Кузо. Средство ориентировано на поиск ошибок времени исполнения в программах на языке C и было использовано для анализа ряда промышленных программных проектов среднего размера (порядка 100000 строк кода). Согласно [14], средство использует как общеизвестные абстрактные семантические домены (целочисленные интервалы, множества указателей), так и ряд доменов, позволяющих отследить зависимости между переменными (в частности, домен октагонов, домен линейных выражений). После разбора исходного кода Astree вначале выполняет ряд простых видов анализа, позволяющих упростить основную задачу (в частности, выявление мёртвых ветвей и переменных), после чего выполняет основной анализ сверху вниз, начиная с точки входа в программу и кончая наиболее простыми функциями.

Исследовательское средство Saturn [4] построено на анализе программ снизу вверх [10]. Информация, полученная при анализе конкретных функций, используется в точках их вызова и анализ функций не повторяется. Тот же принцип может быть использован при анализе циклов. При ана-

лизе отдельных операторов используется вывод ограничений в различных точках исполнения программы с последующим запуском SAT-решателя для определения логической разрешимости. Для описания правил анализа используется язык логического программирования Calypso. Средство может быть использовано для анализа больших программных систем, однако, как следует из [16], имеет довольно низкую точность.

Средство PREfix также анализирует программы снизу вверх [21]. Используются традиционные алгоритмы анализа потока данных, сопоставления с образцом и некоторые другие. В частности, используется идея хранения состояния программы в виде множества точных значений, дополненных множеством предикатов [12].

Продукты PC-Lint (для MS Windows) и FlexeLint (для Unix, Linux, Mac OS X, Solaris) компании Gimpel Software [5] предназначены для обнаружения дефектов в программах на языках C/C++. Система производит синтаксический анализ исходного кода, анализ потока данных и управления, осуществляет межпроцедурный анализ. Используются специальные средства проверки, в том числе: анализ значений, дополнительная строгая проверка типов, определяемая пользователем семантическая проверка аргументов функции и возвращаемых значений.

Исследовательское средство Aegis [1] анализирует программы сверху вниз. Средство использует анализ указателей, интервальный анализ, ресурсный анализ; для повышения точности используется также простой анализ зависимостей [19].

3 Описание разработанного подхода

Предлагаемый подход ориентирован на использование средств верификации и логического вывода для доказательства различных утверждений в ходе статического анализа, в частности, утверждений о наличии либо отсутствии дефектов в отдельных операторах программы.

3.1 Архитектура

Первым этапом выполнения статического анализа является построение модели по исходному коду программы. Наиболее удобной моделью для выполнения статического анализа является *граф потока управления (ГПУ)*, в нем присутствуют только узлы, соответствующие исполнимым операторам программы.

На следующем этапе происходит применение собственно алгоритмов статического анализа с целью определить состояния программы во всех точках её исполнения. Алгоритмы, используя известную информацию о семантике языка, вычисляют возможные значения всех переменных программы.

На последнем этапе (выполняемым параллельно с предыдущим) рассчитанные состояния используются алгоритмами обнаружения дефектов для выявления и локализации имеющихся в программе ошибок.

Для того, чтобы использовать средства логического вывода в ходе абстрактной интерпретации, необходимо предоставить им входные данные в подходящей для них форме. Один из вариантов подобного представления — хранение состояния программы в виде множества *предикатов*, с последующим использованием данного множества для формирования выводов.

Общая схема анализатора, отражающая структуру предлагаемого подхода, представлена на рис.1.

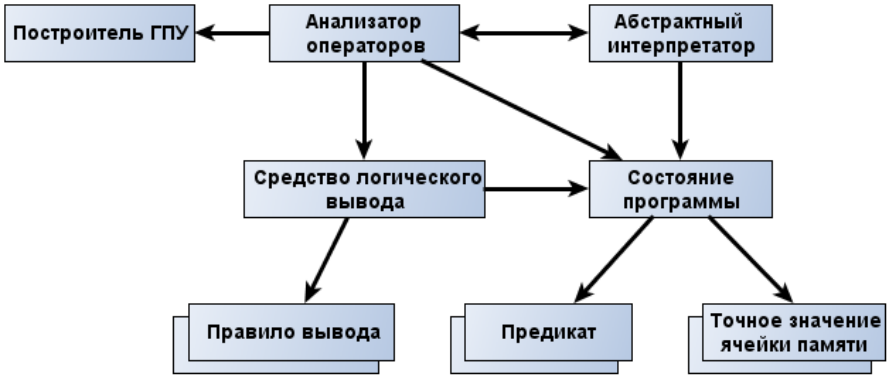

Рис. 1. Структура предлагаемого подхода

Данный способ предполагает, что для некоторых переменных хранятся их точные на данный момент значения. Если значение данной переменной вычислить статически невозможно, для неё могут храниться различные условия, связывающие её значение со значениями других переменных. Таким образом, происходит дополнение точных значений зависимостями между переменными.

Средство логического вывода отвечает за взаимодействие с внешним доказателем. Предполагается решение двух задач — доказательства истинности определенных предикатов и упрощения множества предикатов для сокращения его объёма.

Анализатор, оперируя состоянием программы и результатами вывода, осуществляет поиск дефектов в конкретных операторах.

3.2 Извлечение предикатов

В ходе анализа отдельных операторов программы происходит извлечение информации о значениях переменных и связях между ними, представляемых в виде логических утверждений — *предикатов* [22][18]. В наиболее простом случае интерпретация одного оператора приводит к формированию одного предиката, в точности описывающего семантическое преобразо-

вание, выполняемое данным оператором (например, интерпретация операторов присваивания).

Предикат в нашем случае представляется в виде формулы $p(v_1, v_2, ..., v_n)$, где p — функциональный символ, а $v_i, i = 1...n$ — *предикатные переменные*. Значение предикатной переменной в большинстве случае соответствует значению одной из переменных программы; примерами подобных предикатов являются:

- арифметические: сумма $sum(a, b, c)$ $(a = b+c)$, разность $diff(a, b, c)$ $(a = b - c)$;
- предикаты сравнения: равенство $equals(a, b)$ $(a = b)$, больше $greater(a, b)$ $(a > b)$, меньше или равно $less\_equals(a, b)$ $(a \leq b)$;

Допускаются также более сложные предикаты, переменными которых являются другие предикаты, в частности, $oneof(p_1, p_2)$ (истинен хотя бы один из двух предикатов p_1, p_2), $oppos(p)$ (предикат p ложен) и $equiv(p_1, p_2)$ (предикаты p_1 и p_2 одновременно истинны или одновременно ложны). Данные сложные предикаты также могут быть описаны в терминах логики первого порядка (без использования предикатов в качестве аргументов), поэтому их введение не увеличивает порядок используемой логики. Определение предикатов, представленное выше, выбрано для простоты реализации.

Одной переменной программы могут соответствовать несколько *предикатных переменных*, описывающих её значение в разные моменты выполнения. Это требование является важным, поскольку логика первого порядка не предполагает изменение значения задействованных переменных. Поэтому было принято решение о представлении переменных на базе статического однократного присваивания. Таким образом, при каждом прямом присваивании переменная программы, стоящая в левой части, употребляется в новом контексте, и поэтому для нее создается новая предикатная переменная. Например, после интерпретации цепочки операторов вида `a=b+c; d=a; a+=e` образуется набор предикатов $sum(a_1, b_1, c_1)$, $equals(d_1, a_1)$, $sum(a_2, a_1, e_1)$.

Описание логических операций специфично для языка C, поскольку, согласно семантике данного языка, логические выражения представляются в виде целых чисел. При этом нуль соответствует лжи, а не нуль — истине. Отсюда вытекают такие предикаты, как $zero(v)$ и $nonzero(v)$ (нуль/ложь, не нуль/истина). Предикаты $conj(a, b, c)$ (`a = b && c`), $disj(a, b, c)$ (`a = b || c`) описывают логические операции C, а предикат $equiv(p_1, p_2)$ может быть использован для связи логических переменных с арифметическими. Например, после интерпретации цепочки `g = (a>0); h = (b>=a); f = g && h` образуется набор предикатов $equiv(nonzero(g_1), greater(a_1, 0))$, $equiv(nonzero(h_1), greater\_equals(b_1, a_1))$, $conj(f_1, g_1, h_1)$.

При интерпретации операторов ветвления `if(cond)`, где `cond` — выражение, из входного состояния образуется два. Если `cond` не имеет точного значения, то используется средство логического вывода, которое на основании имеющихся предикатов позволит проверить истинность и/или ложность условия. Если какая-то ветвь оказалась мертвой (условие для нее никогда не выполняется), то дальнейший анализ для этой ветви не производится.

После проверки условия в состояние истинной ветви добавляется предикат, соответствующий истинности условия `cond`, а в состояние ложной ветви — предикат, соответствующий ложности условия `cond`. В частности, если `cond` является переменной, используются предикаты $nonzero(cond)$ и $zero(cond)$ соответственно.

Для снижения количества анализируемых путей применяется объединение состояний программы в точках слияния путей (так называемых фи-функциях) с последующим совместным анализом объединённых путей. В отличие от других операторов оператор фи-функции имеет два и более входных операторов и соответственно два или более предикатных состояния на входе. Основные правила слияния состояний:

- если одна и та же предикатная переменная v_i имеет разные значения во входных состояниях, то её возможные значения x_j объединяются по ИЛИ путём добавления предиката $oneof(equals(v_i, x_1), equals(v_i, x_2))$;
- если некоторой переменной программы `v` соответствуют разные предикатные переменные v_1 и v_2 во входных состояниях, то для неё создается новая предикатная переменная v_3, при этом в выходное состояние добавляется предикат $oneof(equals(v_3, v_1), equals(v_3, v_2))$;
- все предикаты, входящие во входные состояния, необходимо добавить в выходное состояние.

Для описания логики работы с составными переменными языка C (структурами, массивами) и с указателями на них, используется дополнительный набор предикатов:

- элемент сложного объекта $arr(a, v, s)$ — массив (структура) a содержит значение v по смещению s байт — `a[s]=v` в случае байтового массива;
- размер сложного объекта $sizeof(a, s)$ — массив (структура) a имеет размер s байт, или `sizeof(a)=s`;
- указатель на объект $ptr(a, p, s)$ — указатель p указывает на массив (структуру, переменную) a со смещением s байт, или `p=(void*)(&a)+s`;
- разадресация указателя $deref(p, v)$ — значение переменной v равно значению по адресу p, или `v=*p`;
- корректность указателя $correct\_ptr(p)$, $incorrect\_ptr(p)$ — указатель p имеет корректное (указывающее на реально существующую переменную) или некорректное (равное нулю, указывающее на несуществующую переменную) значение.

При слиянии ветвей в некоторых ситуациях важной является связь между значением указателей и выражением из условия. В общем случае, если некоторая переменная имеет разные значения в разных ветвях условного оператора, предикат, описывающий её значение, следует связать с условием оператора `if` для повышения точности. Рассмотрим пример:

```
1 if(size > 0)
2     q = malloc(size);
3 else
4     q = 0;
```

После интерпретации данной конструкции состояние в истинной ветви содержит предикаты: $greater(size_1, 0)$, $sizeof(dyn, size_1)$, $ptr(dyn, q_1, 0)$, где dyn — созданный динамически массив. В состоянии ложной ветви имеются следующие предикаты: $less\_equals(size_1, 0)$, $equals(q_2, 0)$.

При слиянии двух указанных состояний важна зависимость между значением переменной `size`, значением указателя `q` и существованием динамического массива. В дальнейшем при анализе оператора освобождения памяти необходимо знать, что динамический массив существует тогда и только тогда, когда `size > 0`, или когда `q` не является нулевым указателем. Поэтому в выходном состоянии следует учесть эту зависимость.

Таким образом, в момент слияния состояний в разных ветвях оператора условия указателю `q` соответствуют разные предикатные переменные q_1, q_2. Согласно правилам слияния состояний создается новая переменная q_3 и для неё добавляется предикат $oneof(equals(q_3, q_1), equals(q_3, q_2))$. Чтобы учесть зависимость между переменными `size` и `q`, в состояние нужно добавить также предикат $equiv(less\_equals(size_1, 0), equals(q_3, 0))$.

3.3 Правила анализа указателей

Логический вывод осуществляется при помощи некоторого набора аксиом, на основе которых делаются все логические заключения. В основе всех средств логического вывода лежит базовый набор аксиом для распространённых теорий, таких как теория целых чисел. Так как теория указателей не является стандартной для средств логического вывода, появляется необходимость в расширении базового набора аксиом. Таким образом, для осуществления анализа указателей средствами логического вывода необходимо ввести следующие правила:

- правило корректности указателя:

$$\frac{ptr(t, p, s), sizeof(t, v), greater\_equals(s, 0), less(s, v)}{correct\_ptr(p)} \tag{1}$$

  указатель p корректен, когда смещение s, с которым он указывает на цель t, не отрицательно и меньше размера v цели t;
- правила разадресации указателя:
  1)Указатель на простую переменную:

$$\frac{ptr(t, p, 0), deref(p, v)}{equals(t, v)} \tag{2}$$

  если p указывает на t со смещением 0, то значение v по адресу p равно t;
  2)Указатель на элемент составного объекта (массива или структуры):

$$\frac{ptr(a, p, s), arr(a, t, s), deref(p, v)}{equals(t, v)} \tag{3}$$

  если p указывает на a со смещением s, и t находится в a по смещению s, то значение v по адресу p равно значению t.

- правило сложения указателя с целочисленной константой:

$$\frac{ptr(a,p,s), sum(t,s,b), sum(q,p,b)}{ptr(a,q,t)} \tag{4}$$

если указатель p указывает на a со смещением s, t равно сумме s и b, а указатель q равен сумме p и b, то q указывает на a со смещением t.

Эти правила применяются для обнаружения программных дефектов, связанных с использованием указателей.

3.4 Правила обнаружения дефектов

В данной работе особое внимание уделено следующим типам дефектов:

- некорректное использование указателей;
- переполнение буферов и выходы за границы массивов.

Некорректное использование указателей — группа дефектов, связанная с разадресацией некорректного или нулевого указателя. Некорректный указатель — это указатель, в результате разадресации которого возникает ошибка. Примером некорректного указателя может служить указатель на освобождённую память. Процедура обнаружения дефектов при интерпретации операторов косвенной записи или косвенного чтения зависит от значения указателя. Если разадресуемый указатель имеет точное значение: нулевой указатель или некорректное значение, то дефект считается обнаруженным. Если указатель не имеет точного значения, то присутствие дефекта в операторе подтверждается путём доказательства *разрешимости* определённых предикатов относительно множества уже существующих предикатов. Такими предикатами являются:

- нулевой указатель: $equals(ptr, 0)$;
- некорректный указатель: $oppos(correct\_ptr(ptr))$.

Чтобы показать отсутствие дефекта, необходимо доказать *неразрешимость* противоположных предикатов:

- ненулевой указатель: $not\_equals(ptr, 0)$;
- корректный указатель: $correct\_ptr(ptr)$.

Обнаружение выхода за границу объекта производится при выполнении операции разадресации `*ptr` и операции обращения по индексу `ptr[i]`. Для каждого значения указателя `(obj, shift)` следует проверить, что смещение $shift \geq 0$ и $shift < sizeof(obj)$. Если это условие доказуемо, то дефект отсутствует, в противном случае дефект считается обнаруженным.

4 Экспериментальные исследования

Для проведения экспериментальных исследований описанного подхода было разработано средство-прототип. Для получения графа потока управления из исходного кода программы был использован плагин к компилятору gcc, в качестве ядра прототипа была использована библиотека статического анализатора Aegis [1]. В качестве средства логического вывода был выбран SMT-решатель Z3 [3]. Z3 —это свободно распространяемый SMT решатель с открытым исходным кодом, разрабатываемый в Microsoft Research и имеющий API доступа для множества языков программирования. В настоящее время Z3 считается самым мощным средством работы с SMT-предикатами по данным соревнования SMT-COMP [2].

Исследование проводилось на различных тестовых программах (всего порядка 30 программ, объем каждой программы 30-50 строк), содержащих и не содержащих дефекты. Данные программы также были протестированы с помощью средства FlexeLint компании Gimpel Software и базовым анализатором Aegis, не использующим механизм зависимостей. Сводные результаты анализа приведены в таблице 1.

Таблица 1. Результаты анализа

Средство анализа	Точность	Полнота	Среднее время анализа
Прототип	95%	95%	53 сек
FlexeLint	60%	45%	4 сек
Aegis	68%	75%	5 сек

Приведенные результаты наглядно демонстрируют значительный выигрыш в точности анализа в результате ввода правил анализа зависимостей. Недостатком разработанного прототипа являются временные затраты на анализ, которые в зависимости от сложности программы составляли от одной секунды до десяти минут.

Наиболее ресурсоемким является анализ циклических участков программ из-за большого количества создаваемых предикатов на каждой итерации цикла. Для снижения ресурсоемкости анализа можно предложить несколько решений. Одним из таких решений является применение алгоритмов сборки мусора. Этот подход подразумевает удаление из состояния программы предикатов, значения которых не используются при проведении дальнейшего анализа. Другим возможным решением является применение алгоритмов упрощения состояния программы на основе правил трансформации предикатов. Применение данного подхода позволит снизить количество вызовов внешнего решателя.

5 Заключение

В статье был рассмотрен подход к статическому анализу, основанный на комбинировании анализа точных значений и анализа предикатов с использованием средств логического вывода. Приведены правила извлечения предикатов из различных операторов анализируемой программы. Разработаны логические правила вывода для анализа указателей и обнаружения дефектов работы с указателями. Приведённые алгоритмы анализа реализованы в исследовательском прототипе на базе анализатора Aegis и SMT-решателя Microsoft Z3. Показано значительное повышение точности по сравнению с базовым анализатором.

Направления дальнейшего развития данной работы в основном относятся к проработке и реализации методов снижения ресурсоёмкости анализа, что сделает возможным применение подхода для анализа промышленных программ.

Список литературы

1. Aegis static analysis framework. http://www.digiteklabs.ru/en/aegis/platform.
2. Smt-comp 2012. http://smtcomp.sourceforge.net/2012/.
3. Z3 SMT solver. http://z3.codeplex.com/.
4. The saturn program analysis system. http://saturn.stanford.edu, 2006.
5. PC-lint and Flexelint static analysis tools. http://www.gimpel.com/html/products.htm, 2011.
6. Coverity scan: 2012 open-source report. http://scan.coverity.com, 2012.
7. The astree static analyzer. http://www.astree.ens.fr, 2013.
8. Coverity save. http://www.coverity.com/products/coverity-save.html, 2013.
9. Static analysis tools for c code. http://spinroot.com/static, 2013.
10. A. Aiken, S. Burgara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins. An overview of the saturn project. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 43–48. ACM, 2007.
11. A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A fex billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.
12. W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30:775–802, 2000.
13. P. Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2):324–328, 1996.
14. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniax, and X. Rival. Combination of abstractions in astree static analyzer. In *Proceedings of the 11th Annual Asian Computing Science Conference (ASIAN'06)*, pages 272–300, Berlin, 2008.
15. P. Cousot, R. Cousot, and L. Mauborgne. Theories, solvers and static analysis by abstract interpretation. *Journal of the ACM*, 59(6), 2012.
16. I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. *ACM SIGPLAN notices*, 43:270–280, 2008.
17. D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in system code. In *Symposium on Operating System Principles*, pages 57–72. ACM, 2001.

18. Jean H. Gallier. *Logic for Computer Science. Foundation of Automatic Theorem Proving*. Philadelphia, PA, USA, 2003.
19. M. Glukhikh, V. Itsykson, and V. Tsesko. Using dependencies to improve precision of code analysis. *Automatic Control and Computer Science*, 46(7):338–344, 2012.
20. B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, pages 672–681, 2013.
21. N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the 27th international conference of Software Engineering*, pages 580–586. ACM, 2005.
22. W. Rautenberg. *A Concise Introduction to Mathematical Logic*. New York, NY, USA, 2010.
23. В. Ицыксон и М. Глухих. *Программная инженерия. Обеспечение качества программных средств методами статического анализа*. Изд-во Политехн. ун-та, СПб., 2011.