М. И. Глухих, В. М. Ицыксон, В. А. Цесько, Использование зависимостей для повышения точности статического анализа программ, *Модел. и анализ информ. систем*, 2011, том 18, номер 4, 68–79

Модел. и анализ информ. систем. Т. **18**, № **4** (2011) 68–79

УДК 004.415.5+004.416.2

Использование зависимостей для повышения точности статического анализа программ

Глухих М. И.[1], Ицыксон В. М., Цесько В. А.

Санкт-Петербургский государственный политехнический университет

e-mail: glukhikh@kspt.ftk.spbstu.ru, vlad@ftk.spbstu.ru, tsesko@kspt.ftk.spbstu.ru

получена 15 сентября 2011

Ключевые слова: статический анализ, абстрактная интерпретация, анализ зависимостей, обнаружение программных дефектов

Статья посвящена разработке методов анализа зависимостей с целью повышения точности статического анализа. Рассмотрены причины, ведущие к снижению точности абстрактной интерпретации при обнаружении дефектов в программном коде. На различных примерах показана необходимость выявления и интерпретации зависимостей между программными объектами в ходе анализа. Приведена классификация зависимостей по различным признакам. Показана необходимость совокупного рассмотрения хранимых значений и зависимостей. Описан порядок выявления зависимостей при интерпретации присваиваний. Предложен метод интерпретации зависимостей на основе логического вывода, базирующегося на известных правилах логики и арифметики. Некоторые из разработанных методов реализованы в средстве обнаружения дефектов Digitek Aegis, показано значительное увеличение точности анализа.

1. Введение

В современном мире огромное внимание уделяется вопросам повышения качества программных систем. Согласно результатам исследований в широко используемых приложениях с открытым исходным кодом (Linux, Apache, MySQL и т. п.) содержится около 0.1–0.5 ошибок на 1000 строк. В ряде случаев проявления подобных ошибок наносят значительный ущерб [1].

Качество разрабатываемых программ может быть повышено за счет использования статических методов обнаружения ошибок. Подобные методы позволяют анализировать свойства программ без их запуска. Статические методы оперируют исходным текстом программ, формальными спецификациями и другими артефактами стадии проектирования для выявления свойств, которыми обладает вся программа,

[1]Работа выполнена при поддержке комитета по науке и высшей школе Правительства Санкт-Петербурга.

а не одна из возможных трасс ее выполнения. При этом могут быть обнаружены как функциональные, так и нефункциональные программные ошибки.

Нефункциональные ошибки (или программные дефекты) чаще всего являются следствием сложности программной системы и нарушений правил работы с объектами программы. К таким ошибкам относятся утечки памяти, переполнения буфера, выходы за границы массива, разыменования нулевого указателя, использование неинициализированных объектов и многие другие. Для обнаружения подобных ошибок, помимо тестирования, могут использоваться методы статического анализа [2].

Алгоритмы обнаружения дефектов с помощью статического анализа могут быть сопоставлены по трем взаимосвязанным характеристикам — ресурсоемкость, полнота и точность. *Полнота* анализа определяется как доля обнаруженных истинных программных дефектов среди всех имеющихся, а *точность* — как доля истинных программных дефектов среди всех обнаруженных.

Наиболее интересными представляются алгоритмы статического анализа, обеспечивающие полноту. Действительно, только используя подобные алгоритмы, можно обнаружить все дефекты в программе. При использовании других алгоритмов часть критических дефектов может остаться невыявленной.

Проблема использования полных алгоритмов статического анализа состоит в одновременном обеспечении приемлемой ресурсоемкости и высокой точности. Как правило, алгоритмы с низкой ресурсоемкостью имеют и низкую точность. Одним из возможных выходов из данной ситуации является использование механизма зависимостей [5, 6, 15].

Зависимость представляет собой строго определенную связь между значениями двух и более переменных, которая может быть описана предикатом вида $f(x, y)$ is $true$. Механизм зависимостей предназначен для уточнения результатов и устранения ложных обнаружений в ситуациях, когда текущее состояние программы объединяет два или более состояний, полученных на различных путях выполнения, а также при работе с множественными значениями объектов.

Целью данной работы является разработка методов анализа зависимостей, позволяющих повысить точность абстрактной интерпретации программного кода при обнаружении дефектов.

В разделе 2 приведен краткий обзор литературы по данной тематике. В разделе 3 рассмотрен ряд примеров, требующих выявления и интерпретации зависимостей для повышения точности анализа. В разделе 4 приведена классификация зависимостей. Раздел 5 посвящен практической реализации аппарата зависимостей. Наконец, раздел 6 содержит результаты экспериментальных исследований.

2. Обзор существующих работ

Методы статического анализа являются предметом многочисленных исследований. Наиболее распространенным методом статического анализа, применяемым для обнаружения дефектов, является абстрактная интерпретация [3, 4]. Абстрактная интерпретация выполняется подобно обычной интерпретации, однако оперирует не

обычными значениями переменных (числа, указатели), а значениями из абстрактного семантического домена, который выбирается в соответствии с решаемой задачей.

Для поиска нефункциональных дефектов используется домен, включающий интервальные множества значений [12, 13], множества адресов различных объектов [14, 19, 20], а также множества зависимостей между объектами.

Основополагающей работой на тему анализа зависимостей является работа Кузо [6]. Для извлечения зависимостей предлагается использовать абстрактную интерпретацию, при этом учитываются линейные операторы вида $x_j = a_1x_1 + a_2x_2 + \cdots + a_nx_n$. Для аппроксимации реальных ограничений над переменными программы используется модель замкнутых выпуклых многогранников во множестве R^n, где n — количество переменных программы.

В работе Несова и Маликова [5] рассматривается метод увеличения точности обнаружения дефектов на основе улучшенного интервального анализа с использованием систем неравенств. Для построения систем неравенств используются линейные зависимости вида $y(x) = kx + b$.

В работе Буша и др. [15] рассмотрен способ представления состояния программы на основе значений и связей между ними. Этот способ предполагает, что для некоторых переменных хранятся их точные значения. Если точное значение переменной определить невозможно, для нее могут храниться различные связующие условия, например, $a < 5$, $a > b$, $a = c + 3$. Таким образом, происходит дополнение точных значений зависимостями между переменными. Данный подход перспективен, но нуждается в расширении и дополнении — например, в анализе зависимостей более широкого класса.

В работе Вонга и др. [16] рассматривается метод доказательства корректности Java-программ с использованием доказателя теорем *HOL* [17] и платформы верификации *WHY* [18]. Для получения исходных данных используются JML-аннотации. В средстве *VCC* [21] аналогичный подход применяется для доказательства корректности многопоточных C-программ. Недостаток подобного подхода состоит в необходимости аннотирования программы, что для больших программ весьма трудоемко.

Имеется ряд реализаций средств обнаружения дефектов — например, Microsoft Code Analyzer [7], Frama-C [8], Splint [9], Fortify [10] и многие другие. Экспериментальные исследования некоторых из перечисленных средств обнаружения дефектов показали, что они не обеспечивают полноту [11].

Результаты, рассматриваемые в данной статье, получены с использованием средства *Digitek Aegis* [24]. Это средство ориентировано на обнаружение всех имеющихся в программе дефектов определенных классов, обеспечивая максимальную полноту ценой снижения точности обнаружения.

3. Причины снижения точности абстрактной интерпретации

Обнаружение дефектов на основе абстрактной интерпретации при обеспечении высокой полноты и точности характеризуется большой ресурсоёмкостью. Причина

этого кроется в необходимости анализа всех возможных путей выполнения и сопутствующем хранении значений всех доступных объектов.

Большое число путей выполнения программы связано с наличием операторов ветвления — каждый оператор ветвления потенциально удваивает количество путей. С учётом этого, для программы среднего или большого размера количество анализируемых путей становится астрономическим. Кроме того, по мере анализа отдельного пути число доступных объектов обычно возрастает. Указанные причины приводят к экспоненциальному росту затрат времени и памяти в зависимости от размера анализируемой программы, что делает практически невозможным полный и точный анализ даже сравнительно небольших программ.

Для снижения количества анализируемых путей применяется объединение состояний программы в точках слияния путей выполнения с последующим совместным анализом объединенных путей [12]. Подобное действие, называемое слиянием путей, обеспечивает линейный рост ресурсоемкости анализа при отсутствии циклов в анализируемой программе. С другой стороны, переменные программы на различных путях выполнения могут иметь различные значения. В результате их слияния образуются множественные значения переменных, например, $\texttt{x} \in [5 \dots 10]$, $\texttt{ptr} \in \{0, \&\texttt{arr}\}$ и так далее. При работе с множественными значениями информация о зависимостях между ними теряется, что снижает точность анализа. Рассмотрим примеры.

3.1. Пример на размер динамического массива

```
int *ptr = malloc(size * sizeof(int));
for (int i = 0; i < size; i++)
    ptr[i] = i;
```

Пусть $\texttt{size} \in [5 \dots 10]$. В этом случае абстрактная интерпретация цикла потребует 10 итераций. На последней итерации будет справедливо $\texttt{i} = 9$, что при размере массива меньше 10 приведет к обнаружению выхода за границу массива. Для устранения данного ложного дефекта следует учесть наличие зависимости между `size` и размером участка выделенной памяти, а также зависимости $\texttt{i} < \texttt{size}$.

3.2. Пример на число инициализированных элементов

```
int arr[MAXSIZE];
int size = 0;
while (!in.eof() && size < MAXSIZE)
    in >> arr[size++];
for (int i = 0; i < size; i++)
    cout << arr[i] << endl;
```

Поскольку размер потока `in` при статическом анализе неизвестен, по окончании интерпретации цикла `while` справедливо $\texttt{size} \in [0 \dots \texttt{MAXSIZE}]$, при этом каждый из элементов массива либо инициализирован, либо нет. В результате, при выводе

массива на консоль нет уверенности в инициализации его элементов. Учет зависимости числа инициализированных элементов массива `initcount(arr)` от значения `size` позволяет разрешить данную ситуацию.

3.3. Пример на недетерминированное выделение памяти

```
char *ptr = size > 0 ? malloc(size) : 0;
if (ptr != 0) free(ptr);
```

Пусть `size` $\in [0 \dots 10]$. В этом случае в точке слияния после тернарного оператора образуется участок динамической памяти, который может быть как выделен, так и нет, при этом `ptr` указывает на данный участок либо равен 0. При выполнении оператора `if` участок динамической памяти может быть как освобожден, так и нет. Обнаруживается потенциальная утечка памяти — ложный дефект. Учет зависимости «если `ptr` равен 0, то память не выделена» позволяет устранить этот дефект.

3.4. Пример на использование зависимых слагаемых

```
int num = 10 - shift;
fread(buffer + shift, 1, num, fin);
```

Пусть размер буфера `buffer` равен 10, а смещение `shift` $\in [0 \dots 10]$. Функция `fread()` осуществляет чтение в буфер, начиная с элемента `shift`, при этом осуществляется чтение `num` $\in [0 \dots 10]$ элементов. Таким образом, последний прочитанный элемент имеет индекс в диапазоне `shift` + `num` $- 1 \in [-1 \dots 19]$, что приводит к обнаружению ложного дефекта. Учет зависимости `num` $= 10 -$ `shift` при вычислении суммы `shift` + `num` позволяет устранить данный дефект.

Из приведенных примеров видно разнообразие зависимостей, встречающихся в программах и требующих выявления и интерпретации.

4. Классификация зависимостей

Зависимости между программными объектами обладают различными характеристиками. Рассмотрим классификацию зависимостей по различным признакам.

Зависимости могут связывать различные типы *программных объектов*, а именно: одиночные переменные, элементы составных переменных (поля структур, элементы массивов), размеры участков динамической памяти, длины строк, количества инициализированных элементов массивов.

По *характеру связи* между программными объектами зависимости делятся на следующие группы:

- зависимости эквивалентности $Equiv(x, D)$: $D \Leftrightarrow x$, где D — зависимость, а x — логическая переменная;
- арифметические зависимости $Arith(obj_3, obj_1, op, obj_2)$: $obj_3 = obj_1\ op\ obj_2$, где obj_i — программные объекты, op — арифметическая операция;

- логические зависимости $Logic(obj_3, obj_1, op, obj_2)$: $obj_3 = obj_1\ op\ obj_2$, где op — логическая операция;
- зависимости сравнения $Comp(obj_1, op, obj_2)$: $obj_1\ op\ obj_2$, где op — операция сравнения;
- другие, например, зависимость «одна из» вида $D_1\ or\ D_2\ is\ TRUE$, зависимости существования вида $D \Rightarrow$ *dynamic object* `obj` *exists*, зависимости инициализации вида $D \Rightarrow$ `obj` *initialized*.

По *механизму выявления* зависимости делятся на две группы:

- зависимости присваивания, появляющиеся в результате интерпретации конкретного оператора присваивания, например, `f = (x > 5)` или `y = x + 3`;
- зависимости слияния, появляющиеся при слиянии двух ветвей, например, `if (...) ptr = malloc(...) else ptr = 0`.

Существует три *способа использования* выявленных зависимостей:

- при интерпретации условий — в этом случае логический вывод начинается с утверждения вида `cond` $is\ TRUE$ или `cond` $is\ FALSE$, после чего с использованием выявленных зависимостей из этого утверждения делаются определенные выводы;
- при обнаружении дефектов — для этой цели во многих случаях следует проверять справедливость определенных условий, например, при обнаружении выхода за границу массива используется условие $0 \leq$ `index` $<$ `sizeof(array)`;
- при интерпретации присваиваний — в этом случае логический вывод начинается с присваивания вида `a = b - c`, после чего необходимо определить наличие связи между `b` и `c` и при ее наличии сделать выводы о значении `a`.

5. Реализация анализа зависимостей

В этом разделе описывается принцип реализации анализа зависимостей в рамках статического анализатора. Реализация включает в себя способ представления состояния программы, методы выявления и интерпретации зависимостей.

5.1. Представление состояния программы

Метод абстрактной интерпретации оперирует *состоянием программы* — множеством элементов абстрактного семантического домена [2, 3]. Для проведения анализа зависимостей наиболее удобно использовать *домен предикатов* — логических утверждений, описывающих текущее состояние программы и ее объектов. Домен предикатов в этом случае объединяет все возможные предикаты, относящиеся к объектам программы, а состояние программы является подмножеством домена, включающим только предикаты, истинные в определенный момент ее выполнения. Возможные предикаты можно разбить на два подмножества.

1. Утверждения о *значениях* объектов:
   - точные: $obj = const$, obj_1 *points to* obj_2;
   - множественные: $Comp(obj, op, const)$, *obj points to one of* $(obj_1, \ldots, obj_N)$;
2. Утверждения о зависимостях между объектами (см. раздел 4).

5.2. Выявление зависимостей

Выявление зависимостей осуществляется при интерпретации различных операторов программы. Из операторов присваивания непосредственно следуют зависимости присваивания. То же происходит при выполнении косвенного присваивания `a = *ptr` или `*ptr = a`, если *точно* известно, на какой программный объект указывает `ptr`. Псевдокод выявления зависимости для оператора `*ptr = a` может выглядеть следующим образом:

```
Set predicates; // predicates: dependencies and values
function extract(DereferenceAssignment da(ptr, a)):
    for each (p points to obj) in predicates:
        if (p == ptr):
            predicates.add(Comp(obj, =, a))
```

При изменении значения какого-либо программного объекта либо при окончании его времени жизни все связанные с ним ранее зависимости сбрасываются. В случае, когда рассматриваемый объект связан зависимостями хотя бы с двумя другими, следует построить суперпозиции этих зависимостей. Например, если $a = f(x)$ и $x = g(b)$, то по окончании времени жизни `x` строится зависимость $a = f(g(b))$.

При выделении участков динамической памяти следует построить зависимость для размера выделенного участка от значения параметра функции выделения памяти — например, для оператора `ptr = malloc(size)` строятся зависимости вида *ptr points to dynamic* и $sizeof(dynamic) = size$. Для других специальных объектов (например, строк) могут быть сформулированы аналогичные правила.

При интерпретации точек слияния путей выполнения выполняется операция объединения состояний $C = A \cup B$. На практике объединение множеств зависимостей может потребовать большого количества ресурсов из-за образования множества комбинаций зависимостей вида D_1 *or* D_2. Предлагается осуществлять преобразование $A \cup B = (A \cap B) \cup (A \backslash B) \cup (B \backslash A)$, строить пересечение множеств зависимостей $A \cap B$, а объединение $(A \backslash B) \cup (B \backslash A)$ игнорировать. В будущем планируется реализация более точных правил объединения множеств зависимостей.

5.3. Интерпретация зависимостей

В процессе интерпретации зависимостей осуществляется логический вывод из известных на данный момент предикатов. Логический вывод основывается на известных правилах доказательства теорем, которые можно разделить на несколько групп:

- общие правила логического вывода: Modus Ponens, подстановка;
- правила алгебры логики;
- правила арифметики целых чисел.

Интерпретацию зависимостей в анализаторе *Aegis* на примере условий в операторах ветвления можно описать следующим псевдокодом. Здесь x — условие, v задает истинную ветвь (вывод начинается с `x` $is\ TRUE$) или ложную ветвь (вывод начинается с `x` $is\ FALSE$).

```
function interpret(Condition x, boolean v):
    if v:
        for each Logic(f, x1, op, x2) in predicates:
            if x == f && op == and:
                useModusPonens(x1, true);
                useModusPonens(x2, true);
    else:
        for each Logic(f, x1, op, x2) in predicates:
            if x == f && op == or:
                useModusPonens(x1, false);
                useModusPonens(x2, false);
    useModusPonens(x, v);
    expandDeps();

function useModusPonens(Condition x, boolean v):
    for each Equiv(f, dep) in predicates:
        if x == f:
            predicates.add(v ? dep: dep.invert()) // Modus Ponens
```

Здесь функция *interpret* применяет правила алгебры логики для интерпретации зависимостей $Logic(x, x1, and, x2)$ и $Logic(x, x1, or, x2)$, а функция $useModusPonens$ использует правило modusPonens и зависимости $Equiv(x, D)$ для выявления справедливых в определенной ветви условий. Функция *expandDeps* расширяет список известных зависимостей с использованием правила подстановки и правил арифметики целых чисел.

5.4. Развитие подхода

В будущем планируется применить для интерпретации зависимостей существующие средства логического вывода. Их можно разделить на три основные группы:

1. *Доказатели теорем* (theorem provers). В качестве примера можно привести доказатель HOL [17]. Как правило, данные средства ориентированы на использование логики высших порядков. Без участия человека ими могут быть доказаны только достаточно простые утверждения, в сложных же случаях

пользователь должен указывать им путь решения. Кроме этого, доказатели отличаются сравнительно низкой производительностью и неудобным программным интерфейсом.

2. *SMT-решатели* (SMT solvers). В качестве примера можно привести Microsoft Z3 Solver [22]. Ориентированы на автоматическое решение задач в логике первого порядка, обычно включают мощный аппарат разрешения систем линейных уравнений и неравенств. Многие SMT-решатели используют унифицированный язык описания задач SMT-LIB [23], что позволяет достаточно просто заменить один решатель на другой.

3. *Языки логического программирования*, наиболее известным из которых является Prolog. Данный язык использует при логическом выводе подмножество логики первого порядка — дизъюнкты Хорна. Достоинством данного решения является простота его интеграции в любую программную систему, однако мощности логического вывода может быть недостаточно для решения сложных задач.

Анализ имеющихся в данный момент зависимостей и правил вывода свидетельствует о том, что основные задачи доказательства в рамках статического анализа представимы в рамках логики первого порядка. В связи с этим, наиболее подходящим средством логического вывода видятся SMT-решатели, так как доказатели теорем для этой цели слишком громоздки, а языки логического вывода не обладают достаточной мощностью.

6. Экспериментальные исследования

Описанные в разделе 5 методы реализованы в средстве *Digitek Aegis*. Проведено исследование ряда утилит с открытым исходным кодом — результаты приведены в таблице 1. Из перечисленных в таблице утилит, `base64` и `ping` входят в состав ОС Linux, `pg_resetxlog` — в состав сервера PostgreSQL, остальные проекты загружены с `http://www.sourceforge.net`.

Таблица 1. Результаты статического анализа проектов с открытым исходным кодом

Утилита	Объем, KLOC	Зависимости ВЫКЛ		Зависимости ВКЛ	
		Ложных дефектов	Время анализа, с	Ложных дефектов	Время анализа, с
base64	10	55	105	24 (-56%)	136 (+29%)
ping	2.5	24	19	15 (-37%)	22 (+15%)
bwn-ng	5	63	35	41 (-34%)	43 (+22%)
heme	2.5	18	7	8 (-56%)	12 (+71%)
rhapsody	19	8	218	2 (-75%)	228 (+4%)
resetxlog	4	9	37	1 (-88%)	45 (+21%)
Всего	**43**	**177**	**421**	**91 (-48%)**	**486 (+15%)**

Приведённые результаты наглядно демонстрируют значительный выигрыш в точности анализа в результате ввода правил анализа зависимостей — среднее количество обнаруженных ложных дефектов уменьшилось на 48% при незначительном увеличении времени анализа. Среднее количество ложных дефектов составляет около двух на тысячу строк исходного кода при анализе программ объемом порядка 10 KLOC.

7. Заключение

Проведенное исследование подтверждает, что выявление и интерпретация зависимостей является одним из важнейших способов повышения точности статического анализа кода. Могут быть выделены следующие направления дальнейшего развития:

- разработка более точных правил объединения множеств зависимостей в точках слияния;
- использование методов автоматизированного доказательства теорем при интерпретации зависимостей.

В настоящий момент выполняется формализация правил интерпретации операторов, упрощения, вывода и обнаружения дефектов в терминах языка SMT-LIB [23].

Список литературы

[1] M. Zhivich, R. Cunningham, “The Real Cost of Software Errors. IEEE Security & Privacy”, *IEEE Computer Society. 2009*, **7**:2.

[2] F. Nielson, N. Nielson, C. Hankin, *Principles of Program Analysis*, XXI, Corr. 2nd printing, Springer, 2005, 452 pp.

[3] P. Cousot, “Abstract Interpretation. ACM Computing Surveys (CSUR)”, *ACM*, **28**:2 (1996), 324–328.

[4] N. Jones, F. Nielson, “Abstract Interpretation: A Semantic-based Tool for Program Analysis”, *Handbook of logic in computer science: semantic modeling*, **4**, Oxford University Press, Oxford, 1995, 527–636.

[5] В. С. Несов, О. Р. Маликов, “Использование информации о линейных зависимостях для обнаружения уязвимостей в исходном коде программ”, *Труды ИСП РАН*, 2006, № 9, 51–57.

[6] P. Cousot, N. Halbwachs, “Automatic discovery of linear restraints among variables of a program”, Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL ’78), *ACM*, 1978, 84–96.

[7] http://msdn.microsoft.com/en-us/library/d3bbz7tz.aspx, Code Analysis for C/C++ Overview, Microsoft Corporation.

[8] http://frama-c.com, Frama-C Software Analyzers.

[9] http://www.splint.org, Splint home page.

[10] http://www.fortify.com, Fortify Software.

[11] В. М. Ицыксон, М. Ю. Моисеев, В. А. Цесько, А. В. Карпенко, "Исследование систем автоматизации обнаружения дефектов в исходном коде программного обеспечения", *Научно-технические ведомости СПбГПУ. Информатика. Телекоммуникации. Управление*, 2008, № 5(65), 119–127.

[12] M. Schwartzbach, *Lecture Notes on Static Analysis*, Aarhus, 2000, 58 pp.

[13] В. М. Ицыксон, М. Ю. Моисеев, В. А. Цесько, А. В. Захаров, М. Х. Ахин, "Алгоритм интервального анализа для обнаружения дефектов в исходном коде программ", *Информационные и управляющие системы*, 2009, № 2(39), 34–41.

[14] В. М. Ицыксон, М. Ю. Моисеев, М. Х. Ахин, А. В. Захаров, В. А. Цесько, "Алгоритмы анализа указателей для обнаружения дефектов в исходном коде", *Системное программирование*, 2009, 5–30.

[15] W. Bush, J. Pincus, D. Sielaff, "A static analyzer for finding dynamic programming errors", *Software: Practice and Experience*, **30**:7 (June 2000), 775–802.

[16] A. Wang, H. Fei, M. Gu, X. Song, "Verifying Java Programs By Theorem Prover HOL", Proceedings of the 30th Annual International Computer Software and Applications Conference, *IEEE Computer Society Washington DC*, **01** (2006), 139–142.

[17] http://hol.sourceforge.nethttp://hol.sourceforge.net, HOL 4 Kananaskis 6.

[18] http://why.lri.fr, WHY — a software verification platform.

[19] B. Steensgaard, "Points-to analysis in almost linear time", *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, USA, 1996, 32–41.

[20] D. Avots , M. Dalton, V. Livshits, M. Lam, "Improving software security with a C pointer analysis", *Proceedings of the 27th international conference on Software engineering*, New York, USA, 2005, 332–341.

[21] http://vcc.codeplex.com, VCC.

[22] http://research.microsoft.com/en-us/um/redmond/projects/z3, Z3.

[23] http://www.smtlib.org, SMT-LIB.

[24] http://www.digiteklabs.ru/aegis, Aegis—a defect detection system.

The Use of Dependencies for Improving the Precision of Program Static Analysis

Glukhikh M. I., Itsykson V. M., Tsesko V. A.

Keywords: static analysis, abstract interpretation, dependency analysis, program defect detection

The development of dependency analysis methods in order to improve static code analysis precision is considered in this paper. Reasons for precision loss is abstract interpretation methods when detecting defects in program source code are explained. The need for program object dependency extraction and interpretation is justified by numerous real-world examples. A dependency classification is presented. The necessity for aggregate analysis of values and dependencies is considered. The dependency extraction from assignment statements is described. The dependency interpretation based on logic inference using logic and arithmetic rules is proposed. The methods proposed are implemented in defect detection tool Digitek Aegis and significant increase of precision is shown.

Сведения об авторах:
Глухих Михаил Игоревич,
Санкт-Петербургский государственный политехнический университет,
доцент кафедры компьютерных систем и программных технологий;
Ицыксон Владимир Михайлович,
Санкт-Петербургский государственный политехнический университет,
доцент кафедры компьютерных систем и программных технологий;
Цесько Вадим Александрович,
Санкт-Петербургский государственный политехнический университет,
ассистент кафедры компьютерных систем и программных технологий.