Использование анализа недостижимого кода в статическом анализаторе для поиска ошибок в исходном коде программ

Р.Р. Мулюков <eygz@ispras.ru>
А.Е. Бородин <alexey.borodin@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. В статье описывается поиск недостижимого кода в исходном коде программ, написанных на языках Си и Си++. Приведена классификация видов недостижимого кода. Описаны цели, достигаемые поиском недостижимого кода: выдача предупреждений о возможных ошибках в анализируемой программе и улучшение точности других анализов. Формально поставлены три задачи анализа потока данных: анализ интервалов значений, анализ выколотой точки, предикатный анализ. Решения этих задач применены для поиска инвариантных условий ветвления программы. Показаны особенности поиска недостижимого кода в статических анализаторах, предназначенных для поиска ошибок. Отмечены общие ситуации, в которых нет необходимости сообщать пользователю о найденном недостижимом коде. Описанные алгоритмы реализованы в статическом инструменте Svace, разрабатываемом в ИСП РАН. Оценка результатов детекторов произведена для исходного кода операционных систем Android-5.02 и Tizen-2.3 в виде количественного сравнения предупреждений, выданных каждым из анализов, и их пересечения между собой.

Ключевые слова: статический анализ; недостижимый код; анализ потока данных; Svace; поиск ошибок.

DOI: 10.15514/ISPRAS-2016-28(5)-9

Для цитирования: Р.Р. Мулюков, А.Е. Бородин. Использование анализа недостижимого кода в статическом анализаторе для поиска ошибок в исходном коде программ. Труды ИСП РАН, том 28, вып. 5, 2016, стр. 145-158. DOI: 10.15514/ISPRAS-2016-28(5)-9

1. Введение

В статье описывается реализация анализов недостижимого кода в инструменте статического анализа Svace, предназначенного для поиска ошибок в исходном коде программ, написанных на языках Си и Си++ [1-3].

Общая схема реализации анализа недостижимого кода в Svace и находимые ситуации приведены в [4], в данной статье формально описываются используемые алгоритмы и произведена оценка результатов анализа.

1.1 Классификация недостижимого кода

Недостижимым кодом называются инструкции программы, которые ни при каком её выполнении не могут быть достигнуты. Можно выделить несколько разновидностей недостижимого кода:

(1) “По управлению” – это ситуации, когда инструкции в графе потока управления недостижимы от входной вершины.

```
void foo() {
    ...
    return;
    a = 1; // недостижимый код
}
```

```
void bar() {
    ...
    goto label;
    a = 1; // недостижимый код
label:
    a = 2;
}
```

(2) Инструкции недостижимы из-за вызова функций, завершающих выполнение программы.

```
void foo() {
    ...
    fatal_error(1);
    a = 1; // недостижимый код
}
```

(3) “Из-за сравнения” – инвариантность сравнения при ветвлении программы.

```
void foo() {
    int a = 0;
    int b = 1;
    if (a > b) {
        ... // недостижимый код
    }
}
```

```
void bar(char *p) {
    if (!p)
        return;

    ... // p не меняется

    if (p) {
        ...
    } else {
        ... // недостижимый код
    }
}
```

1.2 Цели поиска недостижимого кода

В статическом анализаторе для поиска ошибок в исходном коде программ анализ недостижимого кода используется как для выдачи предупреждений об ошибках в программе, так и для улучшения точности других анализов.

Поскольку программисты, за исключением некоторых ситуаций, нарочно не пишут код, который не может быть исполнен, то наличие такого кода вполне наверняка свидетельствует

- об ошибке в реализации задуманного алгоритма,
- об устаревшем коде, который забыли удалить,
- о непонимании программистом программы, которую он изменяет.

Поэтому существует необходимость в поиске недостижимого кода и выдачи пользователю предупреждения.

```
void foo() {
    ...
    for (i = 0; i < n; ++i) { // "++i" недостижима
        a[i] = i;
        if (m--);  // ошибочная точка с запятой
            break;
    }
}
```

Рис. 1. Недостижимый код как следствие допущенной ошибки

Fig. 1. Unreachable code as a result of a defect

Также обнаружение недостижимого кода полезно и самому статическому анализатору в качестве предварительного этапа анализа, для того чтобы на последующих этапах исключать из рассмотрения найденные недостижимые инструкции программы, что должно благоприятно сказываться на точности его работы.

1.3 Используемые анализы для поиска недостижимого кода

Из-за временных ограничений компиляторы находят недостижимый код для относительно простых случаев. В компиляторах обычно реализован поиск недостижимого кода “по управлению” при построении графа потока управления. Также находятся некоторые инвариантные сравнения при помощи анализа распространения констант.

К статическим анализам применяются менее жёсткие ограничения по времени работы, благодаря чему можно реализовывать более сложные виды анализов. Были реализованы следующие виды анализов: анализ интервалов значений, анализ выколотой точки, предикатный анализ. Для повышения точности анализов применялась нумерация значений.

Отметим, что при анализе не используется анализ алиасов, поэтому описываемые далее анализы потока данных оперируют только локальными переменными, чей адрес не передавался в функции и другим переменным.

Для поиска недостижимого кода “из-за вызова функций”, завершающих выполнение программы, был реализован обратный межпроцедурный анализ, который для каждой функции выясняет, что её вызов приведёт к завершению программы.

2 Анализ недостижимого кода

В этом разделе формально описаны реализованные анализы потока данных, которые применялись для поиска инвариантных сравнений в программе.

2.1 Анализ интервалов значений

Анализ интервалов значений вычисляет в каждой точке программы интервалы, покрывающие все возможные значения целочисленных переменных.

Интервалы значений позволяют обнаруживать инвариантные сравнения в программе. Так как интервалы покрывают все возможные значения переменных, то если на основе возможных значений переменных будет сделан вывод о недостижимости некоторой инструкции, то и при выполнении программы, эти инструкции не будут достигнуты.

```
if (…)
    x = 1;
else
    x = 3;

// x ∈ [1, 3]
if (x > 5) {
    ... // недостижимый код
}
```

Полурешётка анализа интервалов значений описывается следующим образом:

- Элементами полурешётки для одной целочисленной переменной являются интервалы [a, b]. Полурешётка для точки программы – это декартово произведение полурешёток для переменных.
- Частичный порядок определяется отношением включения $\subseteq$.
- Наименьшая верхняя граница для нескольких интервалов вычисляется при помощи объединения $\cup$.

Передаточные функции для арифметических операций основываются на интервальной арифметике [5], например (табл. 1):

Табл. 1. Передаточные функции для переменных `x` и `y` с интервалами значений `[a, b]` и `[c, d]` и константы `k`.

Table 1. Transfer functions for variables x and y with value intervals [a, b] and [c, d] and for a constant k

Выражение	Вычисление интервала значений для результата выражения
x ← k	x: [k, k]
x + y	[a, b] + [c, d] = [a + c, b + d]
x − y	[a, b] − [c, d] = [a − c, b − d]
x * y	[a, b] * [c, d] = [min(ac, ad, bc, bd), max(ac, ad, bc, bd)]

Сравнение	Вычисление интервалов значений для x и y в результате сравнения
assume x = y	x: [a, b] ∩ [c, d] y: [a, b] ∩ [c, d]
assume x ≤ y	x: [a, b] ∩ [-inf, d] y: [a, +inf] ∩ [c, d]
assume x > y	x: [a, b] ∩ [c + 1, +inf] y: [-inf, b - 1] ∩ [c, d]
assume x ≠ k	x: [a + 1, b], если k = a x: [a, b + 1], если k = b x: [a, b], если k ∈ [a + 1, b - 1]

Отметим, что полурешётка интервалов значений имеет бесконечную высоту. Это означает, что в обычном виде анализ потока данных не будет сходиться. Чтобы анализ сошёлся, мы применяем технику *widening* [6]: внутри компонент сильной связности графа потока управления на очередной итерации анализа при перевычислении интервала в точке программы:

- при расширении границы интервала, она сразу заменяется на бесконечность,
- левая граница интервала не смещается вправо, а правая - не смещается влево.

Подсчитанные значения уточняются в режиме *narrowing*: передаточные функции работают, как было описано выше.

На инструкциях сравнения может быть вычислен пустой интервал значений. Пустой интервал свидетельствует о том, что сравнение инвариантно: результат сравнения всегда ложный.

```
// x ∈ [0, 1]
// y ∈ [3, 7]
if (x == y) { // assume x = y
    ... // [0, 1] ∩ [3, 7] = ∅
    ... // недостижимый код
}
```

Недостижимому коду соответствует значение ⊥ из полурешётки для всех переменных.

2.2 Анализ выколотой точки

Анализ интервалов значений не учитывает выколотые точки интервалов, что приводит к недостаточной точности: анализ не обнаруживает многие инвариантные сравнения. Например:

```
if (x != 0) { // assume x ≠ 0
    // ???
    if (x == 0) {
        // x ∈ [0, 1]
    }
}
```

Для таких ситуаций можно описать анализ, который бы вычислял в каждой точке программы для каждой переменной множество констант, которым она в этой точке не равна.

Мы реализовали простой анализ, вычисляющий в каждой точке программы, какие переменные не равны нулю. Константа ноль является наиболее популярной, и работа только с ней тем не менее позволяет обнаруживать много ситуаций недостижимого кода (см. гл. 3).

Элементом полурешётки для одной переменной является утверждение о том, что переменная не равна нулю.

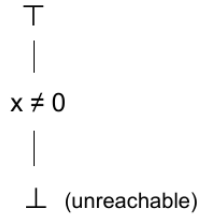

Рис. 2. Частичный порядок полурешётки анализа выколотой точки

Fig. 2. Partial order for semilattice of missing value analysis

Передаточные функции описаны следующим образом (табл. 2):

Табл 2. Передаточные функции для переменных `x` и `y` и константы k

Table 2. Transfer functions for variables x and y and for a constant k

Инструкция	Значение потока данных
`x = 0`	⊤
`x = k`	`x ≠ 0 , если k ≠ 0`
`y = *x`	`x ≠ 0`
`assume x ≠ 0`	`x ≠ 0`
`assume x = 0`	`⊥ , если x ≠ 0`

Недостижимый код по приведённой схеме будет обнаружен в примере, который приводился выше:

```
if (x != 0) { // assume x ≠ 0
    // x != 0
    if (x == 0) {
        // недостижимый код
    }
}
```

2.3 Предикатный анализ

В описанных выше анализах не учитываются сравнения тех переменных, которым сопоставлены значения ⊤, например:

```
void foo(int a, int b) {
    if (a > b) {
        if (a <= b) {
            // недостижимый код
            ...
        }
    }
}
```

Учесть подобные ситуации можно при помощи вычисления необходимых условий достижения точек программы.

Для этого был реализован предикатный анализ. Этот анализ производится над программой, переведённой в SSA-форму. Вычисляемые необходимые условия достижения точек программы – это конъюнкции, состоящие из предикатов инструкций сравнения.

```
void foo(int a1, int b1) {
    if (a1 > b1) {
        // a1 > b1
        if (b1 != 3) {
            // a1 > b1 && b1 != 3
        }
        // a1 > b1
    } else {
        // a1 <= b1
    }
}
```

Рис. 3. Пример вычисляемых необходимых условий достижения

Fig. 3. Example of computed necessary conditions

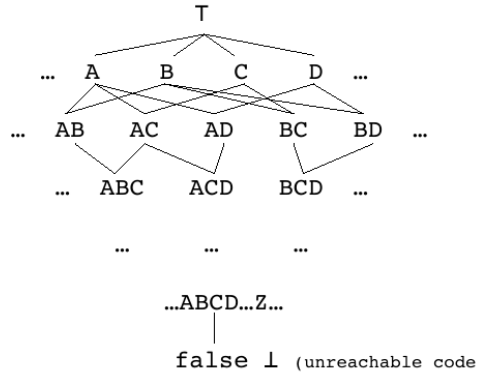

Рис 4. Частичный порядок полурешётки конъюнкций

Fig. 4. Partial order for semilattice of conjunctions

Полурешётка предикатного анализа описывается следующим образом:

- Элементами полурешётки для точки программы являются конъюнкции предикатов, содержащихся в программе (предикаты обозн. `A`, `B`, `C`, …).
- Предикат – атомарное сравнение двух операндов (операнды – SSA-определения и константы).
- Каждая конъюнкция рассматривается как множество предикатов.
- Частичный порядок конъюнкций определяется отношением вложенности этих множеств ⊆.
- Наименьшая верхняя граница для набора конъюнкций вычисляется при помощи их пересечения ∩.

Передаточные функции анализа предикатов описываются только инструкций сравнения:

Для конъюнкции `IN[instr]` на входе в инструкцию `instr` вычисляется конъюнкция `OUT[instr]` на выходе:

Инструкция	Значение потока данных
`instr:` `assume A`	`OUT[instr] ← A && IN[instr]`

При этом `OUT[instr]` сокращается в `false`, если `IN[instr]` включает в себя противоположный предикат `!A` (рис. 5).

```
void foo(int a1, int b1) {
    if (a1 > b1) {       // assume A
        ...
        if (a1 <= b1) { // assume !A
            // недостижимый код
        }
    }
}
```

Рис. 5. Пример обнаружения недостижимого кода

Fig. 5. Example of unreachable code detection

2.4 Нумерация значений

Нумерация значений – это анализ, при помощи которого переменные разбиваются на классы эквивалентности в каждой точке программы. Номерами значений обозначаются классы эквивалентности.

Рис. 6. Разбиение переменных на классы эквивалентности

Fig. 6. Variables partitions into equivalence classes

Это разбиение на классы эквивалентности применяется для того, чтобы анализ потока данных проводился не над переменными, а над их номерами значений. Таким образом, анализ будет устанавливать утверждения сразу для всего класса эквивалентности.

```
x = y;
if (x > 10) {
    ... // x > 10, y > 10
}
```

Рис. 7. Утверждение для класса эквивалентности

Fig. 7. Data-flow value for equivalence class

Вычисление номеров значений производится следующим образом.

- Программа переводится в SSA-форму. Задача сводится к тому, чтобы определить, какие SSA-определения равны между собой.
- Вначале всем SSA-определениям сопоставляются уникальные номера значений:

```
for each x = op(a, b) in Definitions
    vn[x] ← uniqueValue();
eval ← ∅
```

- Итеративно вычисляя хэш от операции и от номеров значений её операндов, мы можем объединять переменные в классы эквивалентности:

```
While (changes)
    For each x = op(a, b) in Definitions
        h ← hash(op, vn[a], vn[b]);
        If (eval[h] == null)
            eval[h] ← vn[x];
        Else
            vn[x] ← eval[h]
```

В приведённом алгоритме предполагается, что хэш-функция обрабатывает коллизии таким образом, что гарантируется, что она никогда не вернёт одно и то же значение для двух разных её аргументов.

3 Сравнение результатов анализов

Описанные анализы были запущены на проектах с открытым исходным кодом: Android-5.02 и Tizen-2.3. Каждый анализ выдал предупреждения о найденных инвариантных сравнениях в коде программ. На рис. 8 приведены графики, демонстрирующие количество предупреждений, полученных от каждого из анализов и их пересечение между собой.

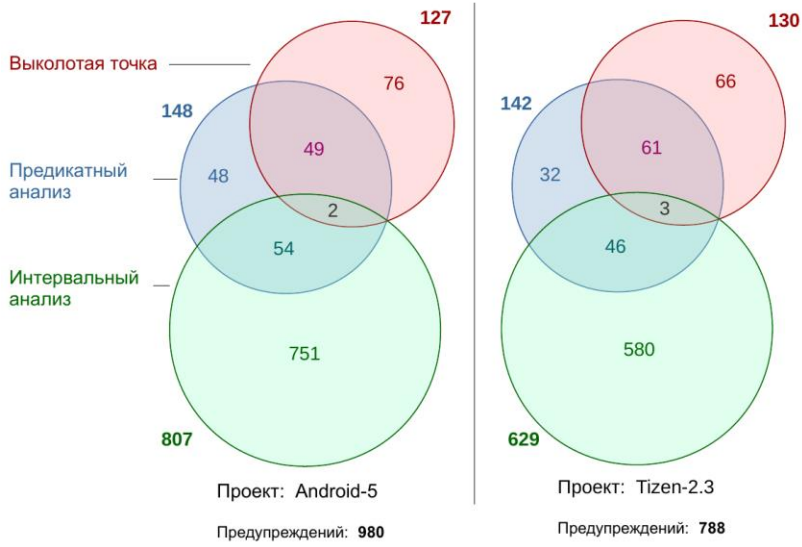

Рис 8. Пересечение предупреждений об инвариантных сравнениях

Fig. 8. Intersection of warnings about invariant comparisons

В графики не включены предупреждения, которые были автоматически отсеяны анализатором. Отсеивание производилось для многих случаев, когда предупреждение не означает ошибку в программе, и исходный код не будет исправлен программистом. Тем не менее, на текущий момент выдаваемых бесполезных предупреждений остаётся 45% от всех предупреждений, и их автоматическое отсеивание является частью нашей дальнейшей работы. Можно выделить типичные виды недостижимого кода, не являющиеся ошибками:

- Недостижимый код внутри раскрытого макроса
- Недостижимый код, возникший из-за подставленного параметра в шаблоне Си++.
- Недостижимый код, возникший из-за сравнения с константой, которую разработчик может при желании менять.
- Недостижимая метка default в инструкции switch.
- Недостижимый код из-за проверки “на всякий случай”, например:

```
free(p);
p = NULL;
goto failure;
...
return;
failure:
    if (p != NULL) {
        // недостижимый код "Won't fix"
        free(p);
        p = NULL;
    }
```

Приведённая в примере ситуация зачастую возникает в методах с большим количеством строк. Такие методы могут модифицироваться в будущем, и потому эта проверка все равно является полезной.

Как видно из графиков, большая часть предупреждений находится с помощью интервального анализа. Анализ предикатов позволяет найти больше ошибок, чем анализ выколотой точки. Но результаты при этом в большей степени пересекаются с результатами интервального анализа, что не удивительно, т. к. интервальный анализ не учитывает выколотые точки. Пересечение результатов интервального анализа и анализа выколотой точки происходит в случае, когда выколотая точка находится на границе интервала.

Заключение

В статье описано использование анализов недостижимого кода для задачи поиска ошибок в исходном коде программ. Описанные анализы позволили найти сотни ошибок для операционных систем Android 5.02 и Tizen 2.3.

Список литературы

[1]. Аветисян А. И., Бородин А. Е. Механизмы расширения системы статического анализа Svace детекторами новых видов уязвимостей и критических ошибок. Труды ИСП РАН, том 21, 2011, стр. 39-54.

[2]. Аветисян А. И., Белеванцев А. А., Бородин А. Е., Несов В. С. Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ. Труды ИСП РАН, том 21, 2011, стр. 23-38.

[3]. Ivannikov V. P., Belevantsev A. A., Borodin A. E. et al. Static analyzer Svace for finding defects in a source program code. Programming and Computer Software, vol. 40, no. 5, 2014, pp. 265–275. DOI: 10.1134/S0361768814050041

[4]. Бородин А. Е., Белеванцев А. А. Статический анализатор Svace как коллекция анализаторов разных уровней сложности. Труды ИСП РАН, том 27, вып. 6, 2015, стр. 111-134. DOI: 10.15514/ISPRAS-2015-27(6)-8.

[5]. Шокин Ю.И. Интервальный анализ. Новосибирск: Наука, 1981.

[6]. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In Proc. Int. Workshop on Programming Language Implementation and Logic Programming, volume 631 of LNCS, pages 269–295. Springer-Verlag, 1992.

Using unreachable code analysis in static analysis tool for finding defects in source code

R.R. Mulyukov <eygz@ispras.ru>
A.E. Borodin <alexey.borodin@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. The goal of finding unreachable code is to report warnings about possible bugs in the source code and an increase of other analyses accuracy. The paper describes unreachable code classification and approaches for finding unreachable code in C/C++ programs. We described three data-flow analysis problems: value interval analysis, excluded value analysis, predicate analysis. Solutions for these problems are used to detect redundant expressions in conditional statements. We described common occurrences of useless warnings. The algorithms are implemented in the Svace tool that is developed by ISP RAS. The results are evaluated for open source projects Android-5.02 and Tizen-2.3. They represent the number of found warnings and its intersection.

Keywords: static analysis; unreachable code; data-flow analysis; Svace; defects in source code.

DOI: 10.15514/ISPRAS-2016-28(5)-9

For citation: R.R. Mulyukov, A.E. Borodin. Using unreachable code analysis in static analysis tool for finding defects in source code. Trudy ISP RAN/Proc. ISP RAS, 2016, vol. 28, issue 5, 2016, pp. 145-158 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-9

References

[1]. A.I. Avetisjan, A.E. Borodin. [Mechanisms for extending the system of static analysis Svace by new types of detectors of vulnerabilities and critical errors]. Trudy ISP RAN/Proc. ISP RAS volume 21, 2011, pp. 39–54 (in Russian).

[2]. A.I. Avetisjan, A.A. Belevantsev, A.E. Borodin, V.S. Nesov. [Using static analysis for searching vulnerabilities and critical errors in the source code of programs]. Trudy ISP RAN/Proc. ISP RAS, volume 21, 2011, pp. 23–38 (in Russian).

[3]. Ivannikov V. P., Belevantsev A. A., Borodin A. E. et al. Static analyzer Svace for finding defects in a source program code. Programming and Computer Software. 2014. Vol. 40, no. 5. P. 265–275. 5. DOI: 10.1134/S0361768814050041

[4]. Borodin A., Belevancev A. [A Static Analysis Tool Svace as a Collection of Analyzers with Various Complexity Levels]. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 111-134 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-8

[5]. Y.I. Shokin. Interval analysis. Novisibirsk – Science, 1981.

[6]. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In Proc. Int. Workshop on Programming Language

Implementation and Logic Programming, volume 631 of LNCS, pages 269–295. Springer-Verlag, 1992.