УДК 004.052.42

ИТЕРАТИВНЫЙ АЛГОРИТМ СТАТИЧЕСКОГО АНАЛИЗА ДЛЯ ОБНАРУЖЕНИЯ ДЕФЕКТОВ В ИСХОДНОМ КОДЕ ПРОГРАММ

М. Ю. Моисеев,
старший преподаватель
Санкт-Петербургский государственный политехнический университет

Предлагается способ организации совместной работы алгоритмов статического анализа для обнаружения широкого класса дефектов в программах на языке С на основе итеративного алгоритма, многократно выполняющего отдельные алгоритмы анализа с уточнением результатов, получаемых на каждой итерации. Рассматриваются свойства предложенного подхода в сравнении с другими вариантами организации комплексного анализа.

Ключевые слова *— надежность программного обеспечения, обнаружение программных дефектов, статический анализ, интервальный анализ, анализ указателей.*

Введение

Одной из важных составляющих качества программного обеспечения является надежность программной системы. Основной причиной недостаточной надежности программных систем являются ошибки, сделанные разработчиками на разных стадиях проектирования. Большинство нефункциональных программных ошибок вносится на стадии кодирования, будем называть такие ошибки программными дефектами. Каждый из дефектов может приводить к серьезным последствиям — аварийным завершениям программы, выдаче некорректных результатов, нарушению конфиденциальности хранимой информации.

Среди существующих подходов к обнаружению программных дефектов наибольшее распространение получили:

— аудит исходного кода программ;
— различные виды тестирования;
— анализ трасс выполнения программ;
— верификация программ на основе моделей;
— методы статического анализа.

Обнаружение дефектов является трудоемкой задачей и может занимать значительную часть от всего времени разработки, поэтому автоматизация ее решения является актуальной. Среди перечисленных подходов наиболее перспективными с точки зрения автоматизации представляются методы статического анализа (СА).

Статический анализ — группа методов, которые используют исходный код для определения требуемых свойств программы; применяются для обнаружения программных дефектов с 70-х гг. прошлого века.

Рассматриваемые в литературе подходы зачастую предлагают решение узкой задачи — поиска одного или нескольких типов дефектов для определенного класса программ [1, 2]. Анализ потока управления выполняется приближенно: точный анализ выражений условий не проводится, межпроцедурный анализ либо не выполняется, либо является контекстно-нечувствительным. Автор статьи [3] считает возможным выполнение универсального высокоточного СА для программ, размер которых не превышает несколько тысяч строк. Обзор предлагаемых в настоящее время подходов свидетельствует о высокой сложности организации и ресурсоемкости выполнения полноценного СА, обеспечивающего анализ всего множества конструкций реального языка программирования для обнаружения широкого набора дефектов.

Рассмотрим основные проблемы обнаружения дефектов с помощью методов СА в программах на языке С. Для обнаружения дефектов в реальных программных системах необходимо выполнить:

— интервальный анализ объектов;
— анализ указателей с учетом возможности перекрытия объектов в памяти, цепочек указателей и циклических ссылок;

— анализ указателей на функции и вызовов функций через указатели;

— анализ объектов, имеющих сложный тип;

— контекстно-чувствительный межпроцедурный анализ с учетом возможных рекурсий.

В настоящее время известны алгоритмы анализа, обеспечивающие решение отдельных задач с тем или иным уровнем качества [4, 5]. При совместном использовании нескольких алгоритмов анализа необходимо учитывать наличие зависимостей между отдельными алгоритмами. Так, например, при проведении интервального анализа используется граф потока управления. Построение полного графа потока управления, в свою очередь, требует анализа выражений в операторах ветвления, для чего необходимы результаты интервального анализа. В общем случае каждый из алгоритмов анализа использует результаты остальных алгоритмов.

Одним из важных свойств любого алгоритма СА является *полнота* получаемого решения. Полное решение включает в себя всю информацию, возможную для анализируемой программы, что позволяет обнаружить все дефекты. Другим важным свойством алгоритмов СА является *точность*. Точное решение не содержит избыточной информации, а его использование для обнаружения дефектов гарантирует отсутствие ложных обнаружений.

В данной статье предлагается способ организации *комплексного статического анализа*, предназначенного для обнаружения дефектов в программах на языке С. Данный подход должен обеспечивать совместное выполнение алгоритмов анализа с учетом взаимных зависимостей между отдельными алгоритмами. Полученное решение должно быть полным, но может не быть точным.

Рассмотрим этапы алгоритма комплексного анализа:

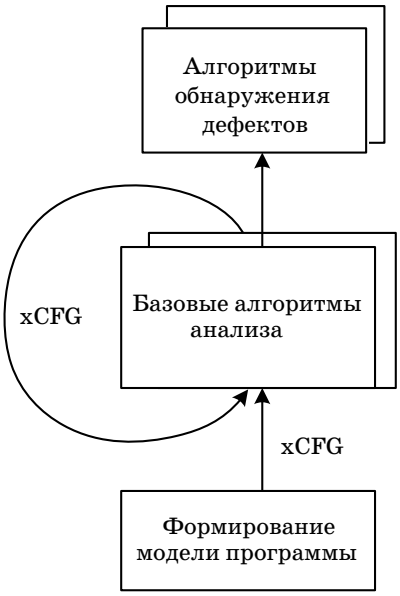
— формирование модели программы;

— извлечение необходимой информации об объектах программы;

— проверка выполнения условий наличия дефекта.

Будем называть алгоритмы, обеспечивающие извлечение необходимой информации об объектах программы, *базовыми алгоритмами анализа*. В качестве базовых выделим алгоритмы анализа указателей на функции, анализа указателей на объекты и интервального анализа.

Для совместного использования базовых алгоритмов предлагается *итеративный алгоритм*, представляющий собой многократное выполнение базовых алгоритмов анализа с уточнением получаемых результатов на каждой итерации. Порядок выполнения базовых алгоритмов на каждой итерации будет соответствовать порядку, в котором перечислены эти алгоритмы.



■ ***Рис. 1.*** *Структура алгоритма комплексного анализа*

Структура предлагаемого алгоритма комплексного анализа представлена на рис. 1.

Формирование модели программы

Для выполнения СА используется одно из представлений исходного кода программы. Обзор существующих представлений приведен в работе [6].

В качестве исходных данных для базовых алгоритмов анализа предлагается использовать расширенный граф потока управления — *extended Control Flow Graph* (xCFG). Основой для построения xCFG является SSA-представление (*Static Single Assignment*, статическое однократное присваивание) программы, дополненное информацией об областях видимости переменных и типах данных.

Рассмотрим основные свойства xCFG. Обычный CFG представляет собой граф, вершинами которого являются операторы программы, дуги соединяют вершины CFG, определяя возможные последовательности выполнения операторов в вершинах CFG. Вершинами xCFG являются конструкции SSA, которые содержат следующую дополнительную информацию: зависимости по данным (версии переменных в SSA), ϕ-функции SSA в местах объединения нескольких дуг, другую информацию.

При построении xCFG выполняется формирование вершин и дуг для конструкций SSA, а также производится замена вызовов функций на

тела функций. Замена осуществляется только для явных вызовов функций, вызовы функций через указатель раскрываются на следующем этапе алгоритма. В месте вызова функции фактические параметры сохраняются во временных переменных, значения которых присваиваются формальным параметрам функции. Добавляется дуга от вызова функции к первой конструкции в теле вызываемой функции. При выходе из функции возвращаемое значение сохраняется во временной переменной, которая может использоваться в месте вызова; добавляется дуга к месту вызова функции. Предложенный подход позволяет проводить контекстно-чувствительный межпроцедурный анализ.

В реальных программах вложенность вызываемых функций может оказаться достаточно велика, что приведет к существенному увеличению xCFG и в результате потребует больших вычислительных затрат для получения решения. Дополнительной проблемой является обнаружение и анализ рекурсивных функций. Для решения этих проблем предлагается ограничить вложенность раскрываемых вызовов функций (как рекурсивных, так и не рекурсивных) некоторым значением, при достижении которого вызовы функций не раскрываются.

Результатом этапа формирования модели является xCFG анализируемой программы, который будет уточняться и дополняться необходимой информацией базовыми алгоритмами анализа.

Базовые алгоритмы анализа

Алгоритмы интервального анализа, анализа указателей на объекты и указателей на функции описываются в виде правил для конструкций анализируемой программы. Базовые алгоритмы анализа формируют с помощью этих правил по одному уравнению для каждой вершины xCFG. Неизвестными в составленных уравнениях являются множества кортежей, описывающие состояние программы. Для алгоритма анализа указателей на функции состояние программы X_{FT} содержит кортежи «указатель-функция», для анализа указателей на объекты X_{PT} — кортежи «указатель-объект», для интервального анализа X_{INT} — кортежи «объект-интервал значений». Состояния программы X_{FT}, X_{PT} и X_{INT} можно разделить на подмножества состояний в отдельных вершинах xCFG: $X_{FT} = \{X_{FT\_l}\}_{l = 1...n}$, $X_{PT} = \{X_{PT\_l}\}_{l = 1...n}$ и $X_{INT} = \{X_{INT\_l}\}_{l = 1...n}$. Будем использовать X_l для обозначения состояния в l-й вершине xCFG, $X = \{X_l\}_{l = 1...n}$ — для обозначения состояния всей программы в любом базовом алгоритме анализа.

Из полученных уравнений составляется система уравнений, отдельно для каждого базового алгоритма анализа. Построенные системы уравнений решаются с использованием *теории решеток* [7]. Для каждой системы уравнений используется решетка, состоящая из подмножеств множества всех соответствующих кортежей. Отношением порядка на решетке является отношение включения подмножеств:

$$X \subseteq Y \Leftrightarrow \forall x \in X \Rightarrow x \in Y,$$

где x — кортеж соответствующего базового алгоритма.

Система уравнений S состоит из уравнений вида $X_l = f_l(X_1, ..., X_n)$ и представляется с помощью функции $F(X) = (f_1(X), ..., f_n(X))$. Поиск решения начинается с пустого множества и расширяется до достижения наименьшей неподвижной точки — *Least Fixed Point* (LFP). LFP функции $F(X)$ на некоторой решетке — это наименьший элемент решетки, удовлетворяющий условию $F(LFP) = LFP$. LFP функции $F(X)$ является минимально возможным решением системы уравнений S.

Для получения LFP требуется обеспечить монотонность функции $F(X)$. Функция $F(X)$ является монотонной, если справедливо

$$\forall X, Y : X \subseteq Y \Rightarrow F(X) \subseteq F(Y).$$

Функция $F(X)$ монотонна, если все функции $f_l(X_1, ..., X_n)$ являются монотонными. Монотонность функций $f_l(X_1, ..., X_n)$ обеспечивается правилами базовых алгоритмов анализа. Алгоритмы поиска LFP описаны в работе [4].

Базовые алгоритмы должны обеспечивать нахождение полного решения. Для этого, при наличии неопределенности, необходимо всегда выбирать наибольшее возможное решение. Например, в вершине, соответствующей ϕ-функции, нужно объединять состояния, полученные со всех входных дуг. Использование такого подхода не гарантирует получение точного решения.

Рассмотрим базовые алгоритмы анализа более подробно. Алгоритм анализа указателей на функции определяет возможные значения указателей в каждой точке программы. На основе полученного решения производится модификация xCFG — выполняется добавление и удаление дуг к телам функций в местах их вызова через указатель. Алгоритм анализа указателей на функции использует информацию от интервального анализа и анализа указателей на объекты. Эта информация позволяет анализировать массивы указателей на функции и структуры, поля которых являются указателями на функции. Алгоритм анализа указателей на объекты определяет взаимосвязи между указателями и объектами, на которые они

указывают. Алгоритм анализа указателей использует результаты интервального анализа, что позволяет анализировать объекты сложных типов с точностью до отдельных элементов. Алгоритм интервального анализа определяет возможные интервалы значений объектов. Интервальный анализ использует результаты анализа указателей для идентификации объектов, операции над которыми выполняются через указатели.

Итеративный алгоритм

Первая итерация.

При выполнении алгоритма анализа указателей на функции на первой итерации результаты анализа указателей на объекты и интервального анализа еще не получены. При решении системы уравнений считается, что неизвестные X_{PT} и X_{INT} содержат все возможные кортежи «указатель-объект» и «объект-интервал значений» соответственно. В процессе анализа указателей на функции в вершинах xCFG, где происходит вызов функций через указатели, добавляются соответствующие дуги. Такой подход позволяет существенно сократить xCFG по сравнению с xCFG, в котором для вызовов функции через указатель имеются дуги к телам всех функций с подходящей сигнатурой. Выполнение остальных базовых алгоритмов на сокращенном xCFG позволяет уменьшить ресурсоемкость и повысить точность результатов на первой итерации. Эти соображения явились определяющими при выборе порядка выполнения базовых алгоритмов внутри итерации.

Последующие итерации.

Рассмотрим работу базовых алгоритмов анализа на последующих итерациях. Обозначим решения, получаемые этими алгоритмами на i-й итерации:

$FT(i)$ — результаты алгоритма анализа указателей на функции;

$PT(i)$ — результаты алгоритма анализа указателя на объекты;

$INT(i)$ — результаты алгоритма интервального анализа.

Начиная со второй итерации, алгоритм анализа указателей на функции использует результаты интервального анализа и анализа указателей на объекты с предыдущей итерации, получается уточненное решение $FT(i+1) \subseteq FT(i)$. В результате выполнения алгоритма анализа указателей на функции происходит сокращение xCFG, что позволяет уточнить результаты всех базовых алгоритмов.

Анализ указателей на объекты использует результаты интервального анализа для получения более точного решения: $PT(i+1) \subseteq PT(i)$. На второй и последующих итерациях алгоритм интервального анализа уточняет набор возможных интервалов за счет более точных результатов алгоритма анализа указателей на объекты: $INT(i+1) \subseteq INT(i)$.

Завершение алгоритма.

Признаком завершения итеративного алгоритма является отсутствие изменений результатов всех базовых алгоритмов на очередной итерации. Результат любого базового алгоритма на $(i+1)$-й итерации зависит от результатов всех алгоритмов на предыдущей итерации, например: $FT(i+1) = F(PT(i), INT(i))$. Если на некоторой итерации результаты всех базовых алгоритмов не изменились, то они также не изменятся на последующих итерациях.

Доказательство сходимости.

Будем называть алгоритм, совместно выполняющий правила всех базовых алгоритмов анализа для построения и решения общей системы уравнений, *комбинированным алгоритмом*. При решении общей системы уравнений одновременно определяются неизвестные всех базовых алгоритмов — X_{FT}, X_{PT} и X_{INT}. Результатом решения этой системы уравнений является LFP.

Рассматриваемый итеративный алгоритм завершается за конечное число итераций и позволяет получить решение, включающее в себя LFP комбинированного алгоритма.

Для доказательства этих утверждений воспользуемся методом индукции. Выделим LFP для базовых алгоритмов анализа: $LFP = \{FT_{LFP}, PT_{LFP}, INT_{LFP}\}$. Докажем, что на первой итерации LFP содержится в полученных решениях: $FT_{LFP} \subseteq FT(1)$, $PT_{LFP} \subseteq PT(1)$ и $INT_{LFP} \subseteq INT(1)$. Алгоритм анализа указателей на функции строит систему уравнений, где неизвестными в левой части являются состояния программы X_{FT}. Неизвестными в правой части могут быть также состояния программы X_{PT} и X_{INT}. Значения X_{PT} и X_{INT} на данный момент еще не определены; они заменяются всеми возможными кортежами PT_{ALL} и INT_{ALL}, которые включают в себя LFP: $PT_{LFP} \subseteq PT_{ALL}$ и $INT_{LFP} \subseteq INT_{ALL}$. Уравнения для алгоритма анализа указателей на первой итерации имеют вид $X_{FT} = F(X_{FT}, PT_{ALL}, INT_{ALL})$. В силу монотонности функции $F(X)$ утверждение $FT_{LFP} \subseteq FT(1)$ верно. Таким же образом доказывается утверждение $PT_{LFP} \subseteq PT(1)$; здесь вместо FT_{ALL} используется $FT(1)$, система уравнений имеет вид $X_{PT} = F(FT(1), X_{PT}, INT_{ALL})$. При доказательстве $INT_{LFP} \subseteq INT(1)$ рассматривается система уравнений $X_{INT} = F(FT(1), PT(1), X_{INT})$.

Докажем, что для решений, полученных на второй итерации, выполняются условия: $FT(2) \subseteq FT(1)$, $PT(2) \subseteq PT(1)$ и $INT(2) \subseteq INT(1)$. Рассмотрим системы уравнений для анализа указателей на функции на первой и второй итерациях:

$$X_{FT} = F(X_{FT}, PT_{ALL}, INT_{ALL});$$
$$X_{FT} = F(X_{FT}, PT(1), INT(1)).$$

Отметим, что функция $F(X)$ является монотонной и выполняются условия $PT(1) \subseteq PT_{ALL}$ и $INT(1) \subseteq INT_{ALL}$. Используя эту информацию, можно доказать, что $FT(2) \subseteq FT(1)$. При доказательстве этого факта необходимо учитывать, что функция $F(X)$ может меняться от итерации к итерации за счет сокращения xCFG. Из-за ограниченного объема статьи доказательство не приводится.

Докажем, что если утверждения верны для i-й итерации, то они также верны и для $(i + 1)$-й итерации. Рассмотрим уравнения одного из базовых алгоритмов на $(i + 1)$-й итерации, например уравнения алгоритма анализа указателей на функции:

$$X_{FT} = F(X_{FT}, PT(i), INT(i)).$$

В силу монотонности функции $F(X)$ и условий $PT_{LFP} \subseteq PT(i)$, $INT_{LFP} \subseteq INT(i)$ верно, что $FT_{LFP} \subseteq FT(i + 1)$. Для алгоритмов анализа указателей на объекты и алгоритма интервального анализа утверждение доказывается аналогично.

Доказательство утверждений $FT(i + 1) \subseteq FT(i)$, $PT(i + 1) \subseteq PT(i)$ и $INT(i + 1) \subseteq INT(i)$ выполняется так же, как доказательство для второй итерации, с учетом $FT(i) \subseteq FT(i - 1)$, $PT(i) \subseteq PT(i - 1)$, $INT(i) \subseteq INT(i - 1)$. В силу конечности числа состояний любой реальной программы решения базовых алгоритмов на первой итерации состоят из конечного числа кортежей. На каждой последующей итерации эти решения, по крайней мере, не возрастают. Из этих утверждений следует, что наступит итерация, на которой ни одно из решений не изменится — итеративный алгоритм завершится. Доказательство закончено.

Сравнение итеративного алгоритма с другими способами комплексного анализа

Будем рассматривать свойства итеративного алгоритма, сравнивая его с другими способами реализации комплексного анализа. В качестве критериев сравнения используем полноту и точность получаемого решения, а также вычислительную сложность алгоритма анализа.

Одной из возможных реализаций алгоритма комплексного анализа является раздельное выполнение базовых алгоритмов с объединением полученных результатов. Этот подход отличается простотой, но не учитывает взаимное влияние алгоритмов анализа друг на друга. Нужно отметить, что экспериментальные исследования ряда существующих средств обнаружения дефектов позволяют сделать вывод об использовании такого подхода. Решение, полученное при раздельном выполнении базовых алгоритмов, будет менее точным, чем решение итеративного алгоритма при выполнении одной итерации. Вычислительная сложность этих алгоритмов примерно одинакова.

Другим вариантом реализации комплексного анализа является комбинированный алгоритм, рассмотренный выше. Поиск решения для комбинированного алгоритма начинается с пустого множества и выполняется до нахождения LFP. Этот процесс не может быть прерван, так как любое промежуточное решение не гарантирует никаких определенных свойств. Преимуществом итеративного алгоритма является то, что решение, полученное на любой итерации, является полным (включает LFP) и может быть использовано для обнаружения дефектов. Недостатком итеративного алгоритма является более низкая точность результатов — решение, получаемое на любой итерации, по крайней мере, не меньше, чем LFP комбинированного алгоритма.

Применение итеративного алгоритма позволяет контролировать вычислительную сложность анализа за счет ограничения числа итераций. Существует класс программ, для которых верно, что большая часть уравнений каждого базового алгоритма не зависит от неизвестных других алгоритмов. Применение итеративного алгоритма для таких программ является эффективным с точки зрения вычислительной сложности за счет независимого решения трех слабосвязанных систем уравнений. При анализе таких программ вычислительная сложность итеративного алгоритма соизмерима со сложностью комбинированного алгоритма, использующего *Strong Components* алгоритм [4] для поиска LFP.

Дополнительным преимуществом итеративного алгоритма является возможность использования эвристик. Под эвристиками будем понимать различные способы увеличения точности и уменьшения вычислительной сложности итеративного алгоритма. Применение эвристик возможно на всех итерациях между выполнением базовых алгоритмов. Для реализации некоторых эвристик необходимо выполнять сложные действия, которые описываются с помощью нетривиальных алгоритмов. Представление алгоритмических аспектов в виде уравнений для решетки не является естественным и может оказаться затруднительным или нереализуемым в составе комбинированного алгоритма. Рассмотрим примеры эвристик для уточнения итеративного алгоритма: *алгоритм уточнения xCFG* и *алгоритм восстановления путей*.

Способы улучшения итеративного алгоритма

В основе предлагаемых способов уточнения итеративного алгоритма лежит анализ возможных дуг xCFG и анализ отдельных путей выполнения программы. При использовании этих эвристик решение, полученное итеративным алгоритмом, может быть более точным, чем решение комбинированного алгоритма. Применение предлагаемых эвристик не нарушает свойство сходимости итеративного алгоритма за конечное число итераций.

Алгоритм уточнения xCFG

Дуги xCFG соединяют между собой пары вершин. Каждая дуга описывает возможную последовательность выполнения конструкций программы. Может существовать один или несколько путей выполнения программы, проходящих через эту дугу. Дуги, для которых нет ни одного пути выполнения программы, будем называть *нереализуемыми переходами*. Алгоритм уточнения xCFG выполняет поиск и удаление нереализуемых переходов, использует результаты базовых алгоритмов анализа.

Определение возможных путей производится с помощью анализа условных выражений в операторах ветвления. Для каждой исходящей дуги оператора ветвления определяется подмножество кортежей из входного состояния, для которых выполняется условие перехода по этой дуге. В случае, если подмножество кортежей для одной из дуг пусто, такая дуга считается нереализуемым переходом и удаляется из xCFG.

Использование рассмотренного алгоритма обеспечивает получение сокращенного xCFG, что, в свою очередь, позволяет повысить общую точность итеративного алгоритма.

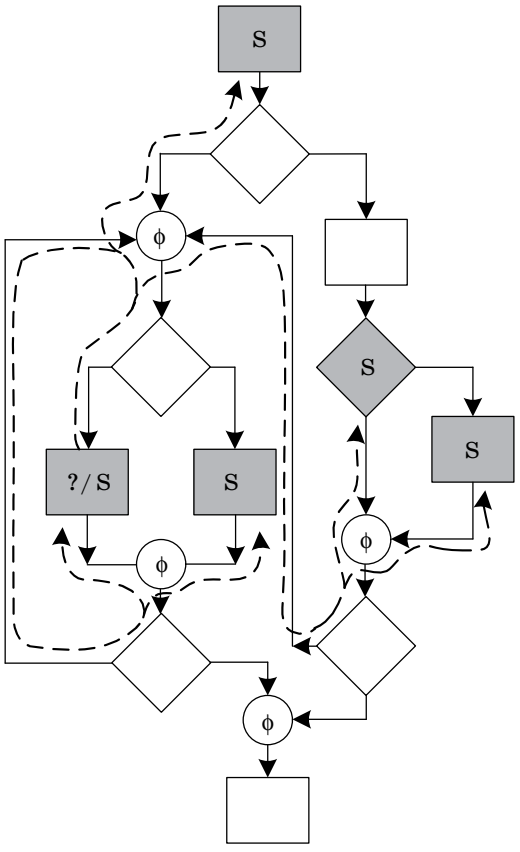
Алгоритм восстановления путей

Базовые алгоритмы анализа не учитывают отдельные пути выполнения программы. Производится объединение путей после операторов ветвления и в вершинах, на которые есть безусловный переход, что приводит к потере точности анализа. Такая реализация базовых алгоритмов обусловлена тем, что отдельный анализ всех возможных путей выполнения для больших программных систем обладает неприемлемой ресурсоемкостью [3]. Для уточнения получаемых результатов предлагается учитывать отдельные пути выполнения только для некоторых частей программы. Выбор анализируемой конструкции, для которой выполняется восстановление путей, осуществляется с учетом влияния решения, получаемого в этой конструкции, на точность общего решения. Исходными данными для алгоритма восстановления путей являются результаты базовых алгоритмов анализа.

Алгоритм состоит из двух стадий — на первой стадии производится построение обратных путей из анализируемой конструкции до всех достижимых точек в рассматриваемой части программы с учетом *ограничивающих условий*. Ограничивающие условия определяют момент завершения стадии построения путей. Примерами таких условий являются: ограничение общей длины путей, ограничение на длину каждого пути, построение путей только внутри анализируемой функции.

Упрощенный xCFG программы, для которой уточняется решение в анализируемой конструкции, помеченной «?», представлен на рис. 2. Строятся все обратные пути от этой конструкции длиной не более 5. Достигнутые вершины помечены «S» — будем называть их начальными вершинами. В данном примере анализируемая конструкция за счет наличия цикла одновременно является начальной вершиной.

На второй стадии от каждой начальной вершины восстанавливаются пути до анализируемой конструкции. Алгоритм восстановления путей начинается с решения, имеющегося в началь-



■ ***Рис. 2.*** *Поиск начальных вершин*

ной вершине, и уточняет это решение, проходя по конструкциям программы. Для каждой конструкции выполняются те же правила, что и в базовых алгоритмах анализа. При прохождении пути через вершину с ϕ-функцией множество решений не меняется.

В результате для целевой вершины вместо одного общего решения X_l получается несколько отдельных решений $X_{l,i}$:

$$X_l = \bigcup_i X_{l,i}.$$

Использование отдельных решений позволяет повысить точность итеративного алгоритма.

Заключение

В данной статье рассмотрены вопросы организации комплексного анализа для обнаружения дефектов в исходном коде программ на языке С. Предложен итеративный алгоритм, который многократно выполняет базовые алгоритмы анализа с уточнением результатов на каждой итерации. Выделены базовые алгоритмы анализа; предъявлены требования к этим алгоритмам. Доказано, что итеративный алгоритм завершается за конечное число итераций, а полученное решение является полным.

Среди достоинств итеративного алгоритма отметим возможность управлять точностью и вычислительной сложностью анализа за счет ограничения числа итераций. Показано, что решение, полученное на любой итерации алгоритма, также является полным и может использоваться для обнаружения дефектов.

Рассмотрены способы улучшения свойств итеративного алгоритма за счет использования дополнительных эвристик. Приводятся примеры эвристик, уточняющих решение итеративного алгоритма за счет удаления нереализуемых переходов и восстановления путей выполнения программы.

Предложенный подход к организации комплексного анализа применим для различных языков программирования. Представленный итеративный алгоритм является основой разрабатываемой системы автоматического обнаружения дефектов в программах на языках С/С++.

Исследование выполнено в рамках работ по государственному контракту № 02.514.11.4081 «Исследование и разработка системы автоматического обнаружения дефектов в исходном коде программного обеспечения» Федерального агентства по науке и инновациям.

Литература

1. **Blanchet B., Cousot P., Cousot R. et al.** A Static Analyzer for Large Safety Critical Software // Proc. of the ACM SIGPLAN 2003 conf. on Programming language design and implementation. 2003. P. 196–207.
2. **Steensgaard B.** Points-to Analysis in Almost Linear Time // Proc. of the 23rd ACM SIGPLAN-SIGACT symp. on Principles of programming languages. 1996. P. 32–41.
3. **Venet A.** A Practical Approach to Formal Software Verification by Static Analysis // ACM SIGAda Ada Letters. 2008. Vol. XXVIII. Issue 1. P. 92–95.
4. **Nielson F., Nielson H. R., Hankin C.** Principles of Program Analysis. Corr. 2nd printing. Berlin: Springer, 2005. 452 p.
5. **Schwartzbach M.** Lecture Notes on Static Analysis. 58 p. www.brics.dk/~mis/static.pdf
6. **Ицыксон В. М., Глухих М. И., Зозуля А. В., Власовских А. С.** Исследование средств построения моделей исходного кода на языках С и С++ // Научно-технические ведомости СПбГПУ. 2009. № 1. С. 122–130.
7. **Биркгоф Г.** Теория решеток. М.: Наука, 1984. 568 с.