УДК 004.415.532.3

Быстрый алгоритм учета зависимостей данных при анализе и тестировании программного обеспечения СБИС

А.С. Щербаков

Университет Мельбурна, Австралия

andreas@softwareengineer.pro

***Аннотация* — При динамическом тестировании программ учет выявленных зависимостей данных затруднен из-за необходимости распространения информации о зависимостях по графу управления. Это приводит к весьма высокой сложности (квадратичной и более по отношению к размеру программы). В работе предложен алгоритм, использующий специальное статическое структурирование блоков программы, информацию о текущем стеке вызовов и представление потока данных в векторном виде с постоянной размерностью. При использовании алгоритма средняя сложность принятия решения о целесообразности выбора предполагаемой траектории очередного выполнения тестируемой программы составляет не более квадрата логарифма размера программы.**

***Ключевые слова* — тестирование программ, зависимость данных, граф управления, гамак, динамическая верификация, верификация, верификация ПО, альтернация, распространение в графе, анализ сценариев.**

I. Введение

Тестирование аппаратно-программных комплексов представляет собой сложную, но крайне важную задачу. В цене готового продукта расходы на тестирование зачастую превышают расходы на собственно разработку изделия, особенно в последнее десятилетие, когда накоплен весьма значительный объем готовых IP-блоков, но не сформирована универсальная система тестирования их свойств в составе сложного проекта. В большинстве случаев тестирование представляет собой многократное исполнение программы по некоторым сценариям, запланированным исходя из имеющейся информации о возможных вариациях путей выполнения, о способах достижения таких вариаций и о зависимостях между операторами или блоками программы, которые могут повлиять на путь ее выполнения и, соответственно, на конечный результат. Как правило, анализ графа управления программы и манипулирование путями ее выполнения (например, с помощью символьного анализа условий переходов [1]) не вызывает принципиальных затруднений. Однако статический анализ зависимостей данных в программе обычно, напротив, затруднен или недоступен, что делает инструментальное наблюдение за тестовым исполнением программы основным источником информации о таких зависимостях. Простейшим примером такого рода является присвоение элемента какого-либо массива с динамически вычисляемым индексом и последующее использование значения данного элемента в некотором условии перехода. За исключением простых случаев, когда адреса обращений к элементу вычисляются одинаковым образом, такая зависимость между исполнением блока, содержащего присваивание, и направлением условного перехода не поддается достаточно простому анализу. Однако в системах тестирования, отслеживающих доступ к адресам оперативной памяти, выявление таких зависимостей легко реализуется в динамическом режиме. Для этого достаточно добавлять направленное ребро от оператора, присвоившего значение элементу памяти, к оператору, использующему значение. При этом формируется граф зависимостей, отдельный от графа управления. Поскольку пути выполнения программы планируются на основе ее графа управления, для реализации учета графа зависимостей требуется его “наложение” на граф управления так, чтобы траектории использования данных были связаны с ребрами графа управления [2]. Такая задача при традиционном подходе (распространение сообщений в произвольном графе) имеет сложность до $O(\boldsymbol{M}\cdot\boldsymbol{W})$, где $\boldsymbol{M}$ – длина программы, $\boldsymbol{W}$ – число распространяемых зависимостей, т.е. в грубом приближении квадратичную относительно размера программы. Для реальных программ такая сложность является неприемлемой. Этот факт затрудняет применение подобных методов и заставляет ограничиваться анализом поведения потока управления программы [3], который существенно менее информативен.

В данной работе предлагается альтернативный алгоритм распространения информации о зависимостях данных по графу управления, снижающий ожидаемую сложность до порядка $O(\boldsymbol{W}\cdot\log^{\alpha}\boldsymbol{M})$, где $\alpha\lesssim 2$. Хотя дальнейшее изложение в основном оперирует понятиями, связанными с ПО, представленный алгоритм применим к широкому классу задач управления и тестирования, в которых зависимости динамически обнаруживаются в процессе выполнения или построения траекторий либо известны статически [4, 5].

II. Формализация задачи

A. *Цели и префиксные узлы*

В используемой модели предполагается, что:

1) известно множество **A** локальных целей планирования траектории. Состав **A** может изменяться во времени. Планирование в каждый момент направлено на достижение хотя бы одной цели **a**∈**A**.

2) С каждой целью **a** связано множество ее *префиксных узлов* **P(a)**, порядок прохождения которых потоком управления влияет на возможность достижения цели. Варьирование этого порядка и является приемом достижения цели.

Такая модель является одним из вариантов подхода к направленному динамическому тестированию. Для примера рассмотрим тестирование программ. Тогда целью может быть прохождение потоком управления еще не покрытого тестами участка кода, а средством ее достижения – изменение выражение условия определенного ветвления. Последовательность присваиваний блоков памяти, участвующих в вычислении выражения, однозначно определяет символическое значение этого выражения, то есть его формулу в фиксированном базисе внешних переменных. При этом предполагается воспроизводимость траекторий выполнения. Следовательно, рассмотрение выражений можно заменить рассмотрением цепочек присваивания и использования содержимого участвующих промежуточных переменных (которые в программе соответствуют участкам оперативной памяти) [6]. Далее, поскольку такие присваивания и использования соответствуют операторам в конкретных узлах графа управления, можно рассматривать последовательности прохождения потоком управления таких узлов как однозначно определяющие значение интересующего выражения.

Рассмотрим для примера короткий фрагмент программы, при этом ограничимся достижением выполнения оператора *действие_1* в качестве цели.

x:=1; z:=0; y:=0
ЕСЛИ *условие_1* ТО y:=x
ЕСЛИ условие_2 ТО z:=y
ЕСЛИ *z*=1 ТО *действие_1*

На рис.1 показан граф управления и возможный порядок траекторий тестирования. Собственно *действие_1* на рисунке опущено, так как целью является изменение выражения условия последнего условного перехода, т.е. выражения для *z*. Целевое условие отмечено диагональной штриховкой, а его префиксные узлы – светлой заливкой. Жирными стрелками показана траектория выполнения, непосредственно предшествующая изображенному состоянию. Ниже перечислены изменения в «знании» о целевом условии и префиксных узлах для диаграмм на рис. 4, *а-г*:

а) обнаружена зависимость целевого условия от начального присваивания *z*:=0;

б) обнаружена его зависимость от присваивания *z*:=*y*; установлено наблюдение *y*;

в) обнаружена зависимость *y* от присваивания *y*:=*x*;

г) цель достигнута и исключена из рассмотрения.

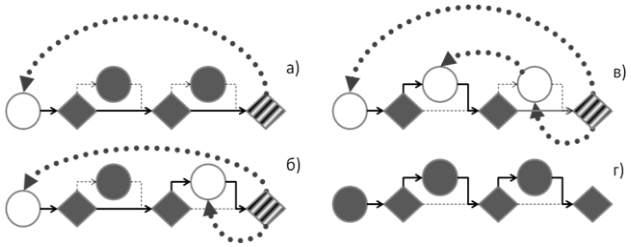

Рис. 1. Пример эволюции цели и ее префиксных узлов

Замечание. На рис. 1 линиями с круглыми штрихами показано дерево зависимостей префиксных узлов. В данной работе используется упрощенное представление совокупности префиксных узлов цели в виде неупорядоченного множества, что может приводить к избыточности рассматриваемых траекторий присваивания. Однако эта избыточность для типичной программы не превышает 10%, поэтому существенное увеличения сложности алгоритма с целью ее уменьшения выглядит неоправданным.

Заметим, что как включаемые в рассмотрение, так и исключаемые из него префиксные узлы не обязательно должны лежать на текущей траектории.

B. Хэш-векторы целей

В процессе планирования путей в программе может возникать большое число целей, сравнимое с размером программы. По мере выполнения целей или отказа от них по иным соображениям стратегии планирования, их число сокращается до малого. Поэтому в средней и финальной стадии тестирования вполне достаточно иметь список ограниченного размера для перечисления целей. В начальной же стадии объем такого списка может оказаться весьма большим. Однако в этом случае возможна его замена избыточной аппроксимацией без существенного снижения производительности, так как подавляющее большинство путей ведут к какой-либо цели и поэтому могут быть приняты к рассмотрению. Исходя из таких соображений, предполагается, что каждая цель идентифицируется числом ξ в диапазоне 1..K, присваиваемым в псевдослучайном порядке или в результате вычисления хэш-функции ее параметров. Вектор булевых значений размера K с единственным истинным значением для координаты с номером ξ мы называем хэш-вектором цели, а покоординатную дизъюнкцию векторов целей, входящих в некоторое множество – хэш-вектором **H** этого множества. Предполагается, что для каждого узла графа управления хранится и обновляется хэш-вектор множества целей, префиксным узлом которых он является.

C. Единичные альтернации

Задачу планирования пути мы сводим к рассмотрению элементарного акта альтернации. Под *альтернацией* в нашем случае понимается направленное изменение траектории выполнения программы путем выбора альтернативной ветви заданного условного перехода в одной из ранее наблюдаемых траекторий. Предполагается, что имеется возможность целенаправленно влиять на такой выбор, как, например, в направленном

тестировании с решением уравнений относительно входных переменных [7].

D. Ψ– функции

Определение 1. Пусть n – узел направленного графа **G**, **Y** – некоторое подмножество множества его узлов. Тогда множеством первичных доступных из n узлов множества **Y** мы называем множество узлов **Y**, доступных из n по путям, не проходящим через другие узлы множества **Y**:

$$\mathbf{\Theta}(\mathrm{n},\mathbf{Y}) = \cup|_{i=0..\infty}([\{x_j\} \rightarrow I(\{x_j : x_j \notin \mathbf{Y}\})\ ]^i(\{n\})\ ) \cap \mathbf{Y},$$

где I – функция, ставящая в соответствие множеству узлов графа объединение списков смежности его членов.

В момент выбора направления перехода в данном узле графа достаточно рассмотреть первичные префиксные узлы цели в возможных продолжениях траекторий. Нет необходимости рассматривать все возможные последовательности прохождения дальнейших префиксных узлов, т.к. это может быть сделано при необходимости в последующих альтернациях траекторий. Еще более важно то, что каждый такой первичный узел взаимно однозначно соответствует некоторой группе возможных последовательностей прохода префиксных узлов. Это позволяет использовать сравнение множеств первичных узлов вместо сравнения множеств путей для качественного анализа.

Определение 2. Дифференциальным множеством первичных доступных префиксов для пары узлов (n^+,n^-) и множества **Y** называется дополнение $\mathbf{\Theta}(n^+,\mathbf{Y})$ до $\mathbf{\Theta}(n^-,\mathbf{Y})$, т.е. множество первичных префиксов, доступных из n^+, но недоступных из n^-.

Чтобы формализовать задачу оценки целесообразности альтернации, вводится дифференциальная функция доступности префиксов целей, которую мы обозначаем как $\mathbf{\Psi}$.

Определение 3. Дифференциальная функция доступности для пары узлов (n^+,n^-) определяется как множество целей, дифференциальное множество первичных доступных префиксов которых не пустое:

$$\mathbf{\Psi}(n^+,n^-) = \{a \in \mathbf{A} : \mathbf{\Theta}(n^+, \mathbf{P}(a)) \setminus \mathbf{\Theta}(n^-, \mathbf{P}(a)) \neq \varnothing\}.$$

Определение 4. Дифференциальная функция доступности определяется как функция, возвращающая хэш-вектор для $\mathbf{\Psi}$:

$$\mathbf{\Psi}^{\mathrm{v}}(n^+,n^-) = \mathbf{H}(\mathbf{\Psi}(n^+,n^-)) = \ldots$$

$$\ldots = \vee\ (a \in \mathbf{A} : \mathbf{\Theta}(n^+,\mathbf{P}(a)) \setminus \mathbf{\Theta}(n^-, \mathbf{P}(a)) \neq \varnothing \quad 2^{\mathrm{a}}),$$

где ξ и **H** – идентификационное число и хэш-вектор соответственно, введенные выше в разделе *B*. Именно быстрое вычисление $\mathbf{\Psi}^{\mathrm{v}}$ является задачей алгоритма, представленного в последующих главах.

Пусть рассматривается вопрос о целесообразности альтернации в узле n, а в ролях n^+ и n^- выступают узлы, следующие непосредственно за n в предполагаемой и базовой траекториях соответственно. Тогда $\mathbf{\Psi}^{\mathrm{v}}(n^+,n^-)$ будет возвращать хэш-вектор целей, для которых возникают новые пути прохождения префиксных узлов. Чем больше логических единиц окажется в векторе, тем более приоритетной можно считать предполагаемую альтернацию. Соответственно, если вектор окажется нулевым, данная альтернация вовсе не представляет интереса.

III. Структурирование графа управления

Для оптимальной работы алгоритма вычисления $\mathbf{\Psi}^{\mathrm{v}}$ граф управления разбивается на блоки, которые образуют иерархическую структуру. За основу при выделении блоков принимается исходная структура программы. Для улучшения сбалансированности структуры желательно добиться, чтобы число узлов в блоке было небольшим, а число блоков в циклических участках графа – минимальным. Несколько приемов, описанных ниже, используются для такой оптимизации в типичных случаях.

В практике анализа и компиляции программ широко применяется представление участков потока управления программы в виде так называемых «гамаков» [8]. Такое представление является идеальным для вычисления $\mathbf{\Psi}^{\mathrm{v}}$, так как множество узлов, участвующих в его вычислении, ограничивается «гамаком», содержащим n^+ и n^-, а вложенные в него «гамаки» либо требуют только простого поиска целей входящих префиксных узлов, либо игнорируются. К сожалению, в реальных программах истинные «гамаки» встречаются достаточно редко. Причинами этого являются исключения, переходы, выход из программы, обработка сигналов и т. п. Перечисленные виды передачи управления встречаются практически повсеместно в современном ПО (за исключением специально спроектированных критических секций кода). В результате вероятность соответствия критерию «гамака» для блока исходного кода достаточно мала. Кроме того, в динамически изменяющемся графе управления может возникнуть ситуация, когда ранее обнаруженный «гамак» перестает удовлетворять критериям и должен быть исключен из рассмотрения. С целью расширения применимости «гамаков» при сохранении их полезных свойств мы рассматриваем блок как «квази-гамак», который имеет один *основной вход* и один или более выходов. Предполагается, что число выходов поддерживается по мере возможности минимально необходимым в рамках заданного подхода к оптимизации графа. Мы не рассматриваем общую задачу поиска оптимального множества «квази-гамаков» в графе, т.к. ее сложность выглядит слишком высокой; преобразования ограничиваются модификацией исходных конструкций кода. В случае если во время тестировании выявлено новое ребро графа управления, нарушающее структуру «гамаков», мы добавляем необходимые выходы к имеющимся блокам, не изменяя их общую схему. Дополнительные *входы* блоков учитывать не обязательно (хотя и возможно для дальнейшей оптимизации). Несложно показать, что все основные конструкции кода – ветвление, цикл, обработчик прерывания [9], последовательность операторов, вызов функции/процедуры и т.п. – представляются каждая в виде «квази-гамака».

На рис. 2 приведен для примера общий вид блочной структуры для программы, содержащей обработчики исключений, условные переходы, цикл и вызов функции.

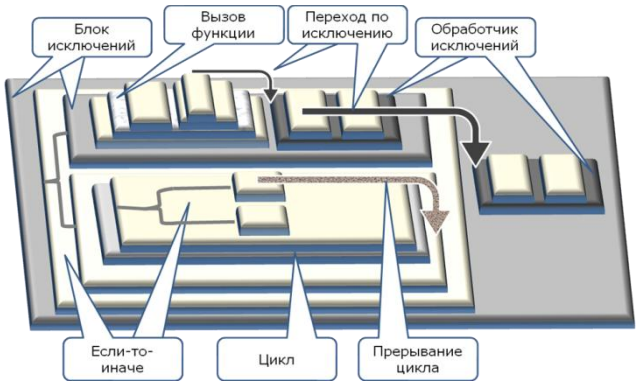

Рис. 2. Пример блочной структуры программы

Последовательности операторов могут представлять собой блоки произвольного и зачастую большого размера (до сотен элементов). Такие последовательности преобразуются в деревья вложенных последовательностей малого размера (рис. 3).

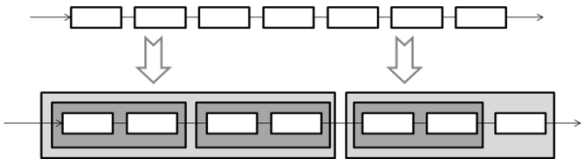

Рис. 3. Преобразование последовательности блоков

Вложенность блоков в целом отражает иерархическую структуру кода, при этом, например, в случае вызова функции, один и тот же блок (тело функции) может быть вложен в различные блоки, ее вызывающие. Поэтому вложенность блоков рассматривается в контексте текущего стека программы. Информация о стеке может быть получена из записи траектории выполнения, относительно которой планируется альтернация. Важным случаем является обработка исключений, когда происходит переход управления из точки генерации исключения на один из входов блока его обработки. Блок обработки при этом выбирается путем поиска ближайшего пригодного в текущем стеке программы. Динамическая привязка ребер перехода по исключениям требует поддержки карты узлов-получателей для стека и добавляет соответствующую сложность, которая учитывалась при моделировании (в предположении ее реализации в виде сбалансированного дерева).

Более сложные схемы. В ряде случаев структура графа может выходить за рамки описанной выше основной схемы. Например, программа может содержать операторы безусловного перехода, теоретически способные придать графу управления программы произвольный вид. Однако в большинстве рекомендованных случаев [10] использование безусловных переходов удовлетворяет небольшому числу шаблонов, легко обнаруживаемых при инструментировании кода. Большая часть этих шаблонов сводится к уже упомянутым блокам – циклам, обработчикам исключений, ветвлению. Однако если, например, участок кода с безусловными переходами описывает конечную машину состояний, то граф действительно может иметь произвольный сложный вид. В предлагаемом подходе такие сложные участки выделяются в особые (нестандартные) блоки, которые, в свою очередь, включают вложенные блоки, структура которых не была нарушена, в неизменном виде. Пример такой реконфигурации показан на рис. 4. Хотя такие блоки могут иметь большой размер, что снижает эффективность предлагаемой оптимизации поиска путей в графе, на практике их доля в общем размере графа мала и существенно не влияет на оценку средней сложности блока.

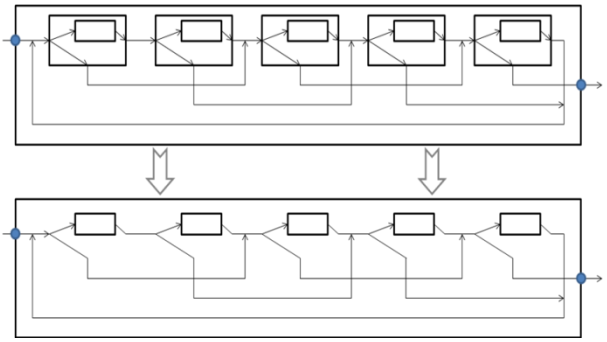

Рис. 4. Реконфигурирование блоков, связанных сложным подграфом переходов

IV. Хранение интерфейсов блоков

Ключевым моментом оптимизации алгоритма вычисления $\boldsymbol{\Psi}^{v}$ является повторное использование ранее вычисленных данных для блоков, позволяющее избежать повторного обхода подграфа, содержащегося в блоке. Такие данные мы называем интерфейсом блока. Конкретный состав интерфейса может быть различным в зависимости от требований к эффективности оптимизации и количеству ресурсов. Мы рассматриваем базовый вариант, обеспечивающий ожидаемый *порядок* сложности. Для основного входа блока b вычисляется и сохраняется хэш-вектор множества целей доступных префиксных узлов. Мы обозначаем его $Y[b]$. В дальнейшем в случаях, когда блок b можно рассматривать как изолированный подграф с единственной точкой входа (т. е. кроме тех случаев, когда поиск начинается внутри b или входящих в него блоков), мы рассматриваем b как коллективный узел, а $Y[b]$ – как хэш-вектор множества целей этого узла. Кроме того, для каждого выхода x блока b вычисляется и хранится функция его доступности из основного входа – вектор $T[b][x]$. Установленный бит в его координате ξ соответствует доступности выхода x по путям, не содержащим префиксных узлов целей с хэш-функцией ξ. Необходим также флаг валидности интерфейса, т.к. интерфейс может стать неактуальным из-за изменения состава целей и (или) их префиксных узлов. Заметим, что в зависимости от среды могут возникать и другие

факторы, влияющие на актуальность интерфейса, например, вызов различных реализаций виртуальной функции. Мы не рассматриваем подробно такие моменты в данной работе, предполагая, что сброс флагов валидности происходит по внешнему сигналу.

Алгоритм вычисления интерфейса достаточно прост и представляет собой распространение в графе.

```
ПРОЦЕДУРА inject (q, a, n, r+, r-)
ЕСЛИ (r+ ∧ ~ s+[n]) ≠ 0 ИЛИ (r- ∧ ~ s-[n]) ≠ 0 ТО
      s+[n] := s+[n] ∨ r+
      s-[n] := s-[n] ∨ r-
      Добавить узел n в очередь q
      Добавить узел n к множеству a
КОНЕЦ_ПРОЦЕДУРЫ

ПРОЦЕДУРА interface (b, n)
q := пустая очередь
a := ∅
inject (q, a, n, ~0)
Y[b] := 0
ПОКА q не пуста ЦИКЛ
  Переместить очередной элемент из q в n
  Y[b] := Y[b] ∨ y[n]
  ЕСЛИ n – основной вход вложенного блока b′ ТО
      ЕСЛИ интерфейс b′ не валиден ТО
            interface (b′, n)
      КОНЕЦ_ЕСЛИ
      ДЛЯ (x ∈выходы b′) ЦИКЛ
        inject (q, a, x, T[b′][x] ∧ s[n])
      КОНЕЦ_ЦИКЛА
  ИНАЧЕ
    ДЛЯ (x ∈ L(n)) ЦИКЛ
      ЕСЛИ x ∈ b ТО inject (q, a, x, s[n] ∧~y[n])
    КОНЕЦ_ЦИКЛА
  КОНЕЦ_ЕСЛИ
КОНЕЦ_ЦИКЛА
ДЛЯ (n ∈ выходы b) ЦИКЛ
      T[b][n] = s[n]
КОНЕЦ_ЦИКЛА
ДЛЯ (n ∈ a) ЦИКЛ
      s[n] := 0
КОНЕЦ_ЦИКЛА
Установить флаг валидности b
КОНЕЦ ПРОЦЕДУРЫ
```

В приведенном алгоритме y[n] – хэш-вектор целей узла n; $L(n)$ – его список смежности.

Особенность учета валидности. Чтобы избежать в дальнейшем необходимости обхода иерархии вложенных блоков в поиске неактуального интерфейса, мы распространяем отмену валидности вниз по стеку вызовов программы, чтобы гарантировать пересчет интерфейса независимо от вида последующих траекторий выполнения. Такое распространение должно охватывать элементы любого возможного стека вызовов, имеющего блок, валидность которого была отменена, в качестве вершины. Такое распространение может выходить за рамки *текущего* стека. Например, одна и та же функция может вызываться из разных мест кода, в результате чего возникает разветвление путей распространения по предполагаемому стеку [11]. Это приводит к тому, что число итерируемых блоков может значительно превосходить число блоков в стеке; однако такой эффект не оказывает принципиального влияния на порядок суммарного времени обработки отмены валидности.

V. Алгоритм вычисления функции Ψ^v

Методом вычисления Ψ^v является непосредственное распространение пары хэш-векторов (s^+,s^-) в графе управления, начиная от узлов (n^+,n^-). Установленный бит в разряде ξ вектора $s^+[n]$, $s^-[n]$ означает, что узел n доступен из n^+, n^- соответственно по путям, не проходящим через узлы-префиксы целей с идентификационным числом ξ. Там, где это возможно, используются сохраненные интерфейсы блоков. Заметим, что при входе в блоки, находящиеся в текущем стеке, сохраненные интерфейсы игнорируются, так как этих блоках уже происходит распространение и вход в них как во вложенные блоки, по сути, является повторным. В подобном случае множественность точек входа приводит к необходимости явного распространения значений в подграфе внутри блоков для правильного присвоения (s^+,s^-) в каждом его узле.

По окончании распространения те разряды хэш-векторов узлов n, для которых соответствующий бит $s^+[n]$ установлен, а $s^-[n]$ – нет, составляют с помощью дизъюнкции вектор Ψ^v, при этом блоки b, для которых был использован сохраненный интерфейс, рассматриваются как целое с известным целевым вектором Y(b). Ниже приводится алгоритм вычисления Ψ^v.

```
ПРОЦЕДУРА inject2(q, a, n, r+, r-)
ЕСЛИ (r+ ∧ ~ s+[n]) ≠ 0 ИЛИ (r- ∧ ~ s-[n]) ≠ 0 ТО
      s+[n] := s+[n] ∨ r+
      s-[n] := s-[n] ∨ r-
      Добавить узел n в очередь q
      Добавить узел n к множеству a
КОНЕЦ_ПРОЦЕДУРЫ
ФУНКЦИЯ Ψv (n+, n-)
q := пустая очередь
a := ∅
inject2 (q, a , n+, ~0, 0)
inject2 (q, a, n- , 0, ~0)
Ψv := 0
  Переместить очередной элемент из q в n
  ЕСЛИ n – основной вход некоторого блока b
  И b не лежит в текущем стеке программы ТО
      ЕСЛИ интерфейс b не валиден ТО
```

```
            interface (b, n)
        КОНЕЦ_ЕСЛИ
        ДЛЯ (x ∈выходы b) ЦИКЛ
          inject2 (q, a, x, T[b][x] ∧ s+[n], T[b][x] ∧ s-[n])
        КОНЕЦ_ЦИКЛА
      ИНАЧЕ
        ДЛЯ (x ∈ L(n)) ЦИКЛ
          inject2 (q, a, x, s+[n] ∧~y[n], s-[n] ∧~y[n])
        КОНЕЦ_ЦИКЛА
      КОНЕЦ_ЕСЛИ
    КОНЕЦ_ЦИКЛА
    ДЛЯ (n ∈ a) ЦИКЛ
        ЕСЛИ n – основной вход некоторого блока b
        И b не лежит в текущем стеке программы ТО
                Ψv := Ψv ∨ ( s+[n] ∧ ~s-[n] ∧ Y[b])
        ИНАЧЕ
                Ψv := Ψv ∨ ( s+[n] ∧ ~s-[n] ∧ y[n])
        КОНЕЦ_ЕСЛИ
        s[n] := 0
    КОНЕЦ_ЦИКЛА
    КОНЕЦ ФУНКЦИИ
```

Здесь y[n] – хэш-вектор целей узла n, $L(n)$ – его список смежности. Очередь q реализуется с приоритетами так, чтобы узлы в блоках, находящихся выше в стеке, имели больший приоритет. Интерфейсы основного входа T[b] и Y[b] описаны в предыдущей главе. Логические операции '∧', '∨' и '~' над векторами выполняются покоординатно.

VI. Моделирующий эксперимент

Для подтверждения гипотезы об ограниченности сложности одного вычисления $\mathbf{\Psi}^v$ (т.е. одного решения при планировании пути) порядком $O(\log^2 \mathbf{N})$ был проведен моделирующий эксперимент с коллекцией приложений с открытым исходным кодом. Для эксперимента использовались коды для программирования систем на кристаллах – главным образом (1) коды инфраструктуры и приложений для систем распознавания и перевода естественного языка на базе GPU; (2) графические ускорители и (3) встроенное ПО устройств беспроводного доступа и роутеров. Суть эксперимента состояла в следующем:

1) с помощью инструментирующего компилятора CLANG [12] код программы преобразовывался в абстрактное синтаксическое дерево. Затем с помощью специально созданного компилятора синтаксическое дерево переформатировалось в структурированный список блоков согласно представленному в данной статье описанию (в частности, с учетом **goto**-переходов, нарушающих границы «гамаков»). Были составлены карты блоков для каждой функции или процедуры (далее – функции).

2) При работе приложения в среде отладчика в случайные моменты времени производилось считывание стека вызовов функций.

3) С помощью карты блоков для функций, вызываемых в стеке, рассчитывалось число узлов, участвующих в распространении при вычислении $\mathbf{\Psi}^v$ согласно описанному в настоящей работе алгоритму, включая (логарифмическую) сложность обработки слияний интерфейсов блоков.

Рис. 5. Средняя сложность распространения составляющих $\mathbf{\Psi}^v$ для одной функции в сравнении с общей сложностью модуля в моделирующем эксперименте

Было обнаружено, что сложность обработки распространения составляющих $\mathbf{\Psi}^v$ составляет в среднем порядка логарифма размера программы и определяется примерно в равных долях размером ее стека вызовов и сложностью обработки каждой функции.

Для большинства функций сложность распространения выражается числом не более 20 базовых единиц, где одна единица соответствует сложности для функции, не содержащей ветвлений и переходов в теле. Однако многие функции не могут быть полностью или даже частично представлены в виде вложенных блоков малого размера, описанных в главе III. Поэтому их сложность сохраняется при структурировании (рис. 5). Тем не менее средняя сложность обработки функции модуля демонстрирует весьма умеренную динамику роста с увеличением размера программного модуля, близкую к логарифмической – в среднем около 6 единиц дополнительной сложности соответствуют увеличении размера в 10 раз (рис. 6).

Порядок сложности обновления интерфейсов блоков оказался ниже $\log^2(\boldsymbol{M})$, где M – размер программы, следовательно, его можно не учитывать в оценке порядка суммарной сложности обработки зависимостей данных. Однако численно сложность обновления может давать основной вклад. Поэтому в практических реализациях рекомендуется по возможности уменьшить частоту обновления интерфейсов, например, делегировать данную функциональность параллельному процессу с более низким приоритетом.

Рис. 6. Сложность распространения составляющих Ψ^v для функций программ в зависимости от наличия предлагаемых преобразований структуры

VII. Заключение

В работе проведена формализация задачи планирования путей выполнения программ с учетом зависимостей данных в рамках концепции единичных альтернаций. Представлен эффективный алгоритм для принятия решений о целесообразности альтернаций, обладающий на практике сложностью около $O(\log^{\alpha} \boldsymbol{M})$ относительно размера программы, где $\alpha \lesssim 2$. В результате становится возможным непосредственный учет множественных зависимостей данных без существенных для практической применимости метода дополнительных затрат вычислительных ресурсов. Моделирование предлагаемого алгоритма для коллекции встраиваемых программ для систем на кристалле подтверждает его высокую эффективность в качестве средства анализа и тестирования ПО СБИС.

Литература

[1] Godefroid P. Compositional dynamic test generation // Acm Sigplan Notices. 2007. V. 42. № 1. P. 47-54.

[2] Khedker U., Sanyal A., & Sathe B. Data Flow Analysis: Theory and Practice. // CRC Press. 2009. 385p.

[3] Majumdar, R., Xu, R. G. Reducing test generation with information partitions // Lecture Notes in Computer Science, 2009. V. 5643/2009. P. 555-569.

[4] Личаргин Д.В., Яровая Д.С., Трушакова А.И., Кравченко М.В. Численная оценка предпочтительности генерируемых траекторий обучения иностранному языку на основе мьультиметодического подхода // Современные проблемы науки и образования. 2015. № 1-1.

[5] Damaggio, E., Deutch, A., Vianu, V. Artifact systems with data dependencies and arithmetic. // ACM Transactions on Database Systems (TODS). 2012. V. 37. № 3. p. 22.

[6] Щербаков А.С. Ускорение направленного тестирования встроенного и внешнего ПО СБИС путем учета потока данных при ограничении вариативности траекторий выполнения // Проблемы разработки перспективных микро- и наноэлектронных систем - 2014. Сборник трудов / под общ. ред. академика РАН А.Л. Стемпковского. М.: ИППМ РАН, 2014. Часть II. С. 15-21.

[7] Godefroid P., Klarlund N., Sen K. DART: Directed Automated Random // Proceedings of PLDI'2005 (ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation),. Chicago, June 2005. P. 213-223.

[8] Zhang F., D'Hollander E.H. Using hammock graphs to structure programs // IEEE Transactions on Software Engineering. 2004. V. 30. № 4. P. 231-245.

[9] Grove D., DeFouw G., Dean J., Chambers C. Call graph construction in object-oriented languages // SIGPLAN Notes, V. 32. № 10. Oct. 1997. P. 108-124.

[10] URL: https://en.wikipedia.org/wiki/Goto (дата обращения: 19.05.2016)

[11] Callahan D., Carle A.; Hall M.W., Kennedy K. Constructing the procedure call multigraph // IEEE Transactions on Software Engineering. Apr 1990. V. 16. №.4. P. 483–487.

[12] Lattner C. LLVM and Clang: Next generation compiler technology // The BSD Conference. 2008. P. 1-2.

A fast algorithm for data dependency tracking in Software and Firmware analysis and testing

A.S. Scherbakov

The University of Melbourne

andreas@softwareengineer.pro

***Keywords* — Software testing, data dependency, data flow, control flow graph, CFG, hammock, dynamic verification, verification, software verification, alternation, propagation in a graph, scenario analysis.**

Abstract

Data dependency analysis plays an important role in Software and Firmware testing and, generally, in scenario exploration and analysis. Data dependency usually may be easily traced dynamically by watching write and read access to the same data aggregates (variables) from different pieces of code; however, application of such knowledge to further test trajectory planning arouses the necessity to propagate dependency information through process control flow graph (CFG). This means at least quadratic complexity w. r. t. the program size by itself. As planner's interest to

particular data may change dynamically, one should implement distinction between active and outdated targets that means even higher complexity. Those facts make such an approach infeasible in practice.

As preliminaries, we choose a formalization reducing the task of trajectory planning to atomic acts of pre-observed trajectory nodes alternation (choosing another branch at a given node). We also use hash vectors for representing collections of trajectory planning targets. As well, we reduce the data dependency tracking to node subsequence tracking.

Within that framework, we present an algorithm providing a reasonable decision on whether it worth to alternate a given trajectory node based on data dependencies found in all successive CFG nodes. The algorithm demonstrates low mean case complexity (about $O(\log^2 M)$, where M is program size) that was obtained in two major steps. (1) We use a hierarchical partitioning of control flow into a number of nesting small size blocks. Examples of such blocks are hammock branching, loop, function or procedure call, exception handler and so on. More complex custom blocks covering unusual code patterns are discouraged but allowed. As for plain sequences of statements, we split each one into trees of nested short sequences. A lightweight dynamic linking between blocks according to the current call stack is employed when needed. (2) We avoid recursive traversal of an already visited nested block's hierarchy each time when we consider entering it by the control flow from inside an embracing block; instead, we use a stored table of hash vectors representing target prefixes previously found throughout that hierarchy.

REFERENCES

[1] Godefroid P. Compositional dynamic test generation // Acm Sigplan Notices. 2007. V. 42. No. 1. P. 47-54.

[2] Khedker U., Sanyal A., & Sathe B. Data Flow Analysis: Theory and Practice. // CRC Press. 2009. 385 p.

[3] Majumdar, R., Xu, R. G. Reducing test generation with information partitions // Lecture Notes in Computer Science, 2009. V. 5643/2009. P. 555-569.

[4] Lichargin D.V., Jarovaja D.S., Trushakova A.I., Kravchenko M.V. Cislennaja ocenka predpochtitel'nosti generiruemyh traektorij obuchenija inostrannomu jazyku na osnove m'ul'timetodicheskogo podhoda. Sovremennye problemy nauki i obrazovanija – *Modern problems of Science and Education*, 2015, no. 1-1 (in Russian).

[5] Damaggio, E., Deutch, A., Vianu, V. Artifact systems with data dependencies and arithmetic. // ACM Transactions on Database Systems (TODS). 2012. V. 37. No. 3. p. 22.

[6] Shherbakov A.S. Uskorenie napravlennogo testirovanija vstroennogo i vneshnego PO SBIS putem ucheta potoka dannyh pri ogranichenii variativnosti traektorij vypolnenija. Problemy razrabotki perspektivnyh mikro- i nanojelektronnyh sistem - 2014. Sbornik trudov pod obshh. red. akademika RAN A.L. Stempkovskogo. M.: IPPM RAN – *Proc. of "Problems of Advanced Micro- and Nanoelectronic Systems Development" Conf.*, 2014, vol. II. P. 15-21 (in Russian).

[7] Godefroid P., Klarlund N., Sen K. DART: Directed Automated Random // Proceedings of PLDI'2005 (ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation),. Chicago, June 2005. P. 213-223.

[8] Zhang F., D'Hollander E.H. Using hammock graphs to structure programs // IEEE Transactions on Software Engineering. 2004. V. 30. No. 4. P. 231-245.

[9] Grove D., DeFouw G., Dean J., Chambers C. Call graph construction in object-oriented languages // SIGPLAN Notes, V. 32. No. 10. Oct. 1997. P. 108-124.

[10] URL: https://en.wikipedia.org/wiki/Goto (дата обращения: 19.05.2016)

[11] Callahan D., Carle A.; Hall M.W., Kennedy K. Constructing the procedure call multigraph // IEEE Transactions on Software Engineering. Apr 1990. V. 16. No. 4. P. 483–487.

[12] Lattner C. LLVM and Clang: Next generation compiler technology // The BSD Conference. 2008. P. 1-2.