К вопросу о генерации начальных данных, обеспечивающих заданную трассу SPMD-программы

С.С. Гайсарян, П.Н. Яковенко

Аннотация. Исследуется проблема автоматизированной генерации входных данных для SPMD-программы на основании ее исходного текста. Актуальность проблемы определяется тем обстоятельством, что оценка производительности, масштабируемости и других динамических свойств SPMD-программы связана с применением различных интерпретаторов и других средств динамического анализа, но любому инструментальному средству динамического анализа необходимы наборы «типовых» входных данных, обеспечивающих критические сценарии работы программы. Дальнейшее развитие связано с уточнением методов, описанных в разделе 3, (межпроцедурный анализ, более тщательный анализ указателей и др.), развитием метода декомпозиции и изучением возможности применения динамических методов для анализа недоступных компонентов анализируемой программы (внешние функции и др.).

1. Введение

Мы исследуем проблему автоматизированной генерации входных данных для SPMD-программы на основании ее исходного текста. Эта проблема формулируется следующим образом: основываясь на анализе исходного текста программы, необходимо в автоматизированном режиме сформировать требуемое количество наборов начальных данных, обеспечив требуемое распределение данных на узлах параллельной вычислительной системы (кластера). Задача генерации входных данных, реализующих выполнение программы по заданной трассе в управляющем графе, является одной из важнейших в области автоматизированной генерации тестов и тестовых покрытий. Однако в такой постановке эта задача здесь не рассматривается. В работе предлагается новый метод генерации входных данных по заданной трассе, основанный на построении набора ограничений, заданных в виде уравнений и неравенств и являющихся условием прохождения управляющего потока программы вдоль заданного пути.

Актуальность проблемы определяется тем обстоятельством, что оптимизация SPMD-программы невозможна без анализа всевозможных сценариев ее работы. Для оценки производительности, масштабируемости и других динамических свойств SPMD-программ разработаны и применяются различные

интерпретаторы и другие средства динамического анализа SPMD-программ, но любое инструментальное средство динамического анализа требует наличия у пользователя наборов «типовых» входных данных, обеспечивающих критические сценарии работы программы. Например, для анализа производительности программы важно иметь в наличии входные данные, на которых система показывает лучшее и худшее время работы. Как правило, наборы «типовых» входных данных создаются пользователем вручную, из-за чего настройка SPMD-программы становится неэффективной и требующей неоправданно высоких затрат труда прикладных программистов.

Задача генерации входных данных, реализующих выполнение программы по заданной трассе в управляющем графе, является одной из важнейших в области автоматизированной генерации тестов и тестовых покрытий. Однако в такой постановке эта задача здесь не рассматривается.

В работе предлагается новый метод генерации входных данных по заданной трассе, основанный на построении по исходному тексту программы набора ограничений, заданных в виде уравнений и неравенств. Эти ограничения описывают условия прохождения управляющего потока программы вдоль заданного пути. На основе построенных ограничений формулируется математическая задача, решение которой является искомым набором входных данных. Вид математической задачи и сложность ее решения определяется вычислительной сложностью программы. В общем случае получается задача решения системы нелинейных уравнений и неравенств.

Исследования проблемы генерации входных данных мы проводим для прикладных программ, написанных на одном из языков Fortran 77, C или Java. Ограничения на допустимые конструкции перечисленных языков рассматриваются далее в работе. Отметим, что указанные ограничения постоянно снимаются по мере развития и обобщения предлагаемого метода.

Статья имеет следующую структуру. В разделе 2 вводятся основные понятия и формулируется задача генерации наборов входных данных. В разделе 3 описывается предлагаемый метод генерации входного набора данных для заданного пути.

2. Основные понятия и постановка задачи

Мы анализируем программу, беря за основу ее граф потока управления. *Граф потока управления,* или *управляющий граф (УГ)* программы ***P*** – это направленный корневой граф $G = <N,E,s,e>$, состоящий из множества вершин N и множества ребер $E = \{(n,m) | n,m \in N\}$, соединяющих вершины. В каждом УГ имеется две выделенные вершины – s (`entry`) и e (`exit`). Каждой вершине УГ соответствует *базовый блок* программы ***P*** [1]. Ребро, соединяющее вершины УГ m и n, указывает на возможность передачи управления от базового блока m к базовому блоку n. Если степень вершины УГ m больше 1, то каждое ребро, выходящее из вершины m, помечается *условием* (*предикатом ветви*).

Процедура построения УГ на основе абстрактного синтаксического дерева описана в [1].

Путь управления (или просто – *путь*) в программе определяется как последовательность вершин в управляющем графе $path = < p_1, p_2, ..., p_{q_p} >$, где для $1 \le i \le q_p\text{-}1$ $(p_i, p_{i+1}) \in E$. Путь из вершины `entry` в вершину `exit` называется *полным*, остальные пути – *неполными*. *Конкатенация* путей $p = < p_1, ..., p_{q_p} >$ и $w = < w_1, ..., w_{q_w} >$ определяется как путь $pw = < p_1, ..., p_{q_p}, w_1, ..., w_{q_w} >$. Если *first(p)* обозначает начало пути *p*, а *last(p)* – конец пути *p*, то говорят, что пути *p* и *w соединяются*, если $(last(p), first(w)) \in E$. Если *p* и *w* – два неполных пути, то путь *pw* называется *собственным*, если пути *p* и *w* соединяются и *несобственным* – в противном случае.

Среди всех объявленных в программе переменных можно выделить подмножество входных переменных. *Переменная является входной*, если она либо расположена в операторе ввода данных, либо является параметром программы, передаваемым, например, посредством аргументов командной строки. Значение, присваиваемое входной переменной в таких операторах, будем называть *входным значением*. Полный набор входных значений – по одному для каждой входной переменной – составляет *набор входных данных* (вкратце *входной набор*) программы.

Каждая входная переменная обладает множеством допустимых значений, для которых поведение программы определено. Входной набор является *допустимым*, если каждое входное значение принадлежит множеству допустимых значений соответствующей входной переменной. Мы рассматриваем программы, которые производят полный контроль входных данных. В теле такой программы присутствует проверка допустимости входного набора. Если данные не проходят проверку, то выполнение программы корректно завершается. Таким образом, допустимость входных данных сводится к анализу управляющего графа. В УГ есть пути, соответствующие допустимым и недопустимым входным данным, но выполнение программы по любому из путей не должно приводить к аварийному останову. Поэтому далее в тексте мы будем считать, что любой набор входных данных является допустимым.

Будем говорить, что набор входных данных ***x*** проходит по заданному пути или, что то же самое, вдоль заданного пути *u*, если на этом наборе входных данных поток управления программы двигается в управляющем графе в точности по пути *u*.

Путь называется *достижимым*, если существует хотя бы один набор входных данных ***x***, который проходит по этому пути, и *недостижимым* в противном случае. *Тестовым набором данных* для пути *u* называется произвольный набор значений входных переменных, проходящий по пути *u*.

Каждому пути можно поставить в соответствие предикат пути. В программах, выполняющих контроль входных данных, истинность предиката является

необходимым условием прохождения управляющего потока по заданному пути. Достаточность этого условия будет рассмотрена далее в этой работе. *Предикат пути* `Pr = bp1 AND bp2 AND … AND bpn` определяется как конъюнкция предикатов ребер, составляющих этот путь. Не ограничивая общности рассуждений, будем считать, что каждый предикат ветви представляет собой операцию сравнения вида `Expr R 0`, где `R` – одна из операций вида `{=, <>, <, <=, >, >=}`, а `Expr` – арифметическое выражение. Основываясь на понятии предиката, сформулируем эквивалентное определение достижимости пути. Путь является *достижимым*, если существует хотя бы один набор входных данных ***x***, который реализует истинность предиката данного пути.

Если предикаты ветвей представляют собой более сложные логические выражения, то, составив из них предикат пути, мы можем преобразовать его в ДНФ при помощи эквивалентных преобразований булевой алгебры. Достаточным условием истинности ДНФ является истинность хотя бы одного ее дизъюнкта. Следовательно, для доказательства достижимости пути нам достаточно построить входной набор, реализующий истинность любого одного дизъюнкта. Если построить такой набор невозможно ни для одного из дизъюнктов, то путь является недостижимым.

Сформулируем задачу автоматизированной генерации наборов входных данных, рассматриваемую в этой статье. По заданной программе ***P*** и пути `u` сгенерировать тестовый набор входных данных ***x***, который реализует истинность предиката пути `u`.

Рассматривая выполнение программы вдоль заданного пути, можно утверждать, что арифметическое выражение в левой части предиката каждой ветви представляет собой функцию, явным или неявным образом зависящую от вход-ных переменных. Предположим, что мы получили представление этой зависи-мости в явном виде для каждого предиката ветви. Тогда предикат пути прини-мает вид `Pr=(F1(I) R 0) AND (F2(I) R 0) AND … AND (Fn(I) R 0)`, где *I*=`(i1,…,ik)` – вектор входных переменных. Задача нахождения входных значений, реализующих истинность предиката *Pr*, эквивалентна задаче нахождения решения системы уравнений и неравенств, составленной из сравнений вида `Fi(I) R 0` в правой части предиката `Pr`.

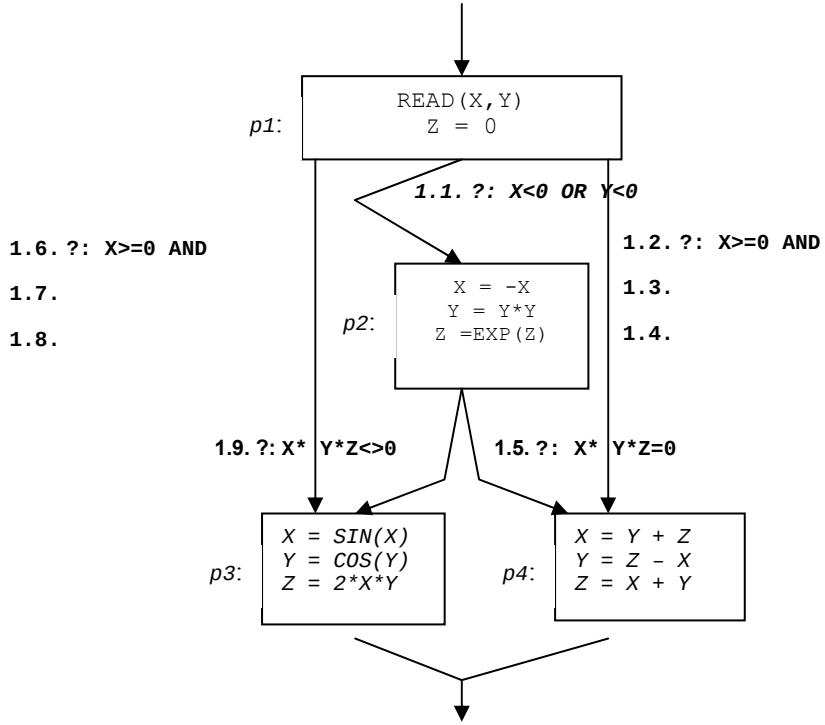
Мы предлагаем новый метод решения задачи автоматизированной генерации теста для заданного пути. В этом методе для исследуемого пути строится математическая задача, эквивалентная исходной. В общем случае – это задача решения системы уравнений и неравенств относительно входных переменных. Представление функциональной зависимости предиката ветви от входных переменных в явном виде формируется при помощи символьной интерпретации программы. Решение построенной системы находится при помощи математических методов решения систем уравнений и неравенств. В частности, если построенная система является системой уравнений, то решение может быть

найдено методом Ньютона.

Рассмотрим пример (*Fortran 77*).

```
1:      READ(X,Y)
2:      Z = 0
3:      IF (X < 0 OR Y < 0) THEN
4:            X = -X
5:            Y = Y*Y
6:            Z = EXP(X+Y)
7:      ENDIF
8:      IF (X * Y * Z = 0) THEN
9:            X = Y + Z
10:           Y = Z - X
11:           Z = X + Y
12:     ELSE
13:           X = SIN(X)
14:           Y = COS(Y)
15:           Z = 2*X*Y
16:     END
```

Построим для программы управляющий граф.



Пусть необходимо построить тест для пути P=<p1,p2,p4>. Предикат Pr для пути P является конъюнкцией предикатов ветвей (p1,p2) и (p2,p4), т.е.

Pr = (X.3<0 OR Y.3<0) AND (X.8*Y.8*Z.8=0). Нотация X.8 означает, что в данном соотношении используется значение переменной X, которое та имеет непосредственно перед выполнением восьмой строки программы.

Преобразуем предикат в дизъюнктивную нормальную форму. Pr = (X.3<0 AND X.8*Y.8*Z.8=0) OR (Y.3<0 AND X.8*Y.8*Z.8=0). В предикате используются значения переменных X,Y и Z, взятые перед выполнением третьей и восьмой строк программы. Чтобы выразить используемые значения через входные значения, выполним символьную интерпретацию программы вдоль пути P.

Входными являются переменные X и Y, т.к. только они присутствуют в операторе чтения данных в первой строке программы. Программа не имеет параметров, следовательно, Z – внутренняя переменная.

Обозначим X0 и Y0 – входные значения программы. В результате символьной интерпретации мы получим следующие символьные выражения для значений переменных программы в каждой точке пути *P*. Ниже выписаны значения только для строк программы, входящих в путь *P*.

```
1: X.1=X0, Y.1=Y0, Z.1=<?>
2: X.2=X0, Y.2=Y0, Z.2=0
3: X.3=X0, Y.3=Y0, Z.3=0
4: X.4=-X0, Y.4=Y0, Z.4=0
5: X.5=-X0, Y.5=Y0* Y0, Z.5=0
6: X.6=-X0, Y.6=Y0* Y0, Z.6=EXP(-X0+ Y0* Y0)
8: X.8=-X0, Y.8=Y0* Y0, Z.8=EXP(-X0+ Y0* Y0)
9: X.9= Y0* Y0+ EXP(-X0+ Y0* Y0), Y.9=Y0* Y0, Z.9=EXP(-X0+ Y0* Y0)
10: X.10= Y0* Y0+ EXP(-X0+ Y0* Y0), Y.10= EXP(-X0+ Y0* Y0)-
      Y0* Y0+ EXP(-X0+ Y0* Y0), Z.10=EXP(-X0+ Y0* Y0)
11: X.11= Y0* Y0+ EXP(-X0+ Y0* Y0), Y.11= EXP(-X0+ Y0* Y0)-
      Y0* Y0+ EXP(-X0+ Y0* Y0), Z.11= Y0* Y0+ EXP(-X0+ Y0*
      Y0)+ EXP(-X0+ Y0* Y0)- Y0* Y0+ EXP(-X0+ Y0* Y0)
```

Подставим полученные соотношения в предикат пути и получим:

```
Pr = (X0<0 AND (-X0)* Y0* Y0* EXP(-X0+ Y0* Y0)=0) OR
     (Y0<0 AND (-X0)* Y0* Y0* EXP(-X0+ Y0* Y0)=0)
```

Предикат пути представляет собой ДНФ, состоящую из двух дизъюнктов. Для того, чтобы предикат *Pr* был истинным, достаточно, чтобы истинным был хотя бы один из дизъюнктов. Рассмотрим первый из них. Он представляет собой систему {X<0; -X*Y*Y*EXP(-X+Y*Y)=0}, составленную из неравенства и равенства. В силу того, что значение экспоненциальной функции всегда больше нуля, а X<0, то, очевидно, что решение существует только при Y=0. Итак, пара входных значений X0=-1, Y0=0 является тестом для заданного пути *u*.

Решение построенной математической задачи разумно (но не обязательно) искать при помощи современных математических пакетов, содержащих в себе

реализации быстрых алгоритмов нахождения решения для многих классов задач. Применение математических пакетов для решения задачи автоматизированной генерации тестов оправдано, поскольку позволяет быстро разработать прототип работающей системы тестирования. Большинство современных алгоритмов решения математических задач реализованы в таких широко используемых пакетах, как *Maxima*, *Mathlab*, *Mathematica*, *Matchcad* и других. Эти программные продукты постоянно совершенствуются и пополняются новыми алгоритмами, поэтому с течением времени круг задач, а значит и программ, для которых можно в автоматизированном режиме построить тестовое покрытие, при помощи предлагаемого метода будет лишь расширяться.

В общем случае математическая задача, получаемая при символьном анализе пути в программе, является задачей решения системы нелинейных уравнений и неравенств. Далее в тексте мы будем говорить о системах уравнений, опуская упоминание о неравенствах, хотя в построенных системах могут также присутствовать и неравенства. Мы будем опираться на тот факт, что неравенство может быть сведено к уравнению путем введения дополнительной независимой переменной, на область значений которой наложено ограничение. Например, неравенство `x+y>7` может быть сведено к уравнению `x+y-7-V=0` путем введения дополнительной переменной `V`, такой что `V>0`. Мы выполняем такое преобразование, потому что применяемый в нашей экспериментальной системе пакет Maxima не умеет решать неравенства, но может решать уравнения с ограничениями на область значений искомых переменных. В Maxima это делается при помощи "`ASSUME`" выражений, в нашем случае `ASSUME(V>0)`.

В этой работе мы будем основываться на следующих предположениях (ограничениях) относительно поведения анализируемой программы:

- Поведение программы детерминировано, т.е. сколько бы раз мы ни запускали программу на одном и том же наборе входных данных, каждый раз мы получаем один и тот же результат. Типичным недетерминизмом в программе является зависимость ее поведения от генератора псевдослучайных чисел или от текущего системного времени;
- Поведение программы обусловлено только теми данными, которые либо были переданы программе перед ее запуском через аргументы командной строки, либо были введены в программу при помощи операций чтения из файла (в т.ч. с консоли);
- Все данные, вводимые в программу извне, подготавливаются до запуска программы и не меняются в ходе ее выполнения, т.е. значения, вводимые из файла и с консоли, могут быть (теоретически) переданы в качестве аргументов командной строки, не меняя семантики программы;
- Программа не использует оператор безусловного перехода *goto*. Все обратные дуги в управляющем графе соответствуют структурному оператору цикла.

3. Метод генерации входного набора данных для заданного пути

Предлагаемый метод генерации теста для заданного пути основан на построении математической задачи, решение которой реализует истинность предиката пути. В случае произвольной программы мы получаем задачу решения системы уравнений и неравенств относительно входных переменных. Для нахождения требуемых входных данных необходимо, во-первых, уметь строить систему уравнений по заданному пути и, во-вторых, находить решение этой системы. В этой работе мы будем исследовать вопрос построения по заданному пути в программе системы уравнений, решение которой является тестом для этого пути. Методы решения систем нелинейных уравнений не рассматриваются в данной работе.

В начале рассмотрим правила построения системы уравнений для программы с простой структурой, а именно для программы, удовлетворяющей следующим ограничениям.

Пусть у нас есть программа (или фрагмент программы), обладающая следующими свойствами:

- Все переменные в программе имеют вещественный тип (`float`, `real`, `double` и т.п.);
- Вычислительная часть программы состоит только из операторов присваивания и условных операторов;
- В правой части оператора присваивания и в условии ветвления присутствуют только арифметические вычисления, использующие четыре стандартных арифметических операции над значениями ранее определенных переменных и числовыми константами.

Будем считать, что среди всех переменных программы выделено непустое подмножество, определяющее входные переменные. Входные переменные обладают тем свойством, что значение любой переменной в каждой точке программы может быть выражено в виде функции, зависящей только от значений входных переменных и констант.

Не ограничивая общности рассуждений, будем считать, что значения входных переменных не меняются на протяжении всего времени выполнения программы. Пользуясь этим свойством, выполним символьную интерпретацию нашей программы вдоль анализируемого пути (символьная интерпретация программы, удовлетворяющей установленным нами ограничениям, не является затруднительной). В результате в каждой точке пути мы для каждой внутренней (т.е. не входной) переменной получим ее значение, представленное в виде символьного выражения, которое зависит только от входных переменных и констант.

Для каждого предиката ветви в исследуемом пути выполним символьную подстановку для всех внутренних переменных путем их замены на соответствующие символьные выражения, полученные в ходе символьной

интерпретации. Все преобразованные условия ветвления выпишем отдельно. В итоге мы получим набор логических выражений (ограничений), зависящих только от входных переменных и констант. Одновременное удовлетворение всех ограничений является необходимым условием прохождения управляющего потока программы вдоль исследуемого пути.

В общем случае мы не можем утверждать, что это условие является также и достаточным по следующей причине. Когда мы строим символьное выражение, мы не учитываем, что вычислительная система обладает ограниченными ресурсами. Операции сложения и умножения могут приводить к переполнению регистров. Во время операции присваивания присваиваемое значение может усекаться с отбрасыванием значимых разрядов. Эта проблема требует дополнительных исследований. На текущий момент мы считаем, что истинность предиката является необходимым и достаточным условием прохождения теста по заданному пути.

Обозначим входные переменные программы через `i1,...,in`, а предикаты ветвей, явным образом выраженные через входные переменные, как `Condi, i=1..m`. Тогда искомый набор значений входных переменных должен реализовывать истинность булевой функции

```
F(i1,...,in) = Cond1 AND Cond2 AND ... AND Condm.
```

Применяя тождественные преобразования булевой алгебры, преобразуем эту функцию в ДНФ (дизъюнктивную нормальную форму)

```
F(i1,...,in) = D1 OR D2 OR ... OR Dk
```

где D_i – дизъюнкт, представляющий собой набор элементов вида `Expr(i1,...,in) R 0`, соединенных операцией конъюнкции (`Expr` – арифметическое выражение, `R` – одна из операций сравнения {=,<>,<,>,<=,>=}).

Функция *F* истинна тогда и только тогда, когда истинен хотя бы один из дизъюнктов `Di`, поэтому для нахождения набора значений входных переменных, реализующих истинность функции `F`, нам достаточно последовательно проанализировать все дизъюнкты `Di` и найти набор значений входных переменных `i1,...,in`, реализующий истинность любого из них. Если же такого набора не найдется, то мы можем утверждать, что путь не является достижимым, т.к. до настоящего момента все наши преобразования были эквивалентными.

Каждый `Di` представляет собой алгебраическую систему уравнений, зависящую, вообще говоря, от всех входных переменных. Решение, удовлетворяющее этой системе, реализует истинность соответствующего дизъюнкта, а следовательно, и всей функции `F`, и, таким образом, представляет собой искомый набор значений, реализующий проход управляющего потока вдоль заданного пути.

Мы свели задачу генерации набора входных данных к задаче решения системы уравнений. Эта система в общем случае является нелинейной и для нее не существует универсальных алгоритмов, дающих точное решение. Однако для многих классов систем, в особенности обладающих свойством выпуклости, известны как аналитические, так и численные алгоритмы, дающие решение с любой заданной точностью. Если все предикаты ветвей представляют собой проверки на равенство, то полученная система является системой уравнений и для нее может быть найдено численное решение методом простых итераций, методом Ньютона или каким-либо другим методом.

Рассмотрим частный случай, когда построенная система является линейной, относительно входных переменных. В этом случае мы можем не преобразовывать систему неравенств в систему уравнений, а решать ее в исходном виде. Для нахождения решения воспользуемся симплекс-методом решения задачи линейного программирования. Все уравнения и/или неравенства системы мы полагаем ограничениями на исходную задачу, а целевую функцию вводим произвольным образом, например, `Z = i1 + i2 +...+ in`. Заметим, что для определения того, является путь достижимым или нет, нам достаточно выяснить, существует ли у построенной задачи линейного программирования опорное решение. В качестве искомого набора ***`i1,...,in`*** мы можем взять найденное опорное решение или любую точку, расположенную на отрезке, соединяющем две вершины симплекса (если только обе точки не принадлежат плоскости, описываемой ограничением со знаком неравенства `{<,>,<>}`). Напомним, что симплекс является выпуклой фигурой, и существуют алгоритмы, находящие координаты всех его вершин.

Анализ программы более сложной структуры требует умения обрабатывать в ходе символьной интерпретации различные языковые конструкции. Среди них функции, определенные в программе, библиотечные функции, массивы, указатели, циклы с фиксированным и переменным числом итераций.

Анализ функции со скалярными аргументами, передаваемыми по значению, у которой мы можем проанализировать исходный текст, не представляет существенных затруднений. Тело такой функции может быть символьно проинтерпретировано вдоль заданного пути отдельно от основной программы. Мы получим символьное выражение для возвращаемого значения относительно формальных параметров функции и предикат этого пути, выраженный через формальные параметры, `y = Fp(I)` при `Prp(I)=true`, где `y` – возвращаемое значение, `I` – вектор формальных параметров, `Fp` – символьное выражение для значения функции вдоль пути `p`, `Prp` – предикат для пути `p`. Анализируя путь в основной программе, мы выполняем подстановку символьного значения для возвращаемого значения вместо каждого обращения к функции. При этом предикат пути для возвращаемого значения объединяется с предикатом пути в основной программе через операцию конъюнкции.

Например, пусть предикат пути в основной программе содержит неравенство `7-F(u)<0`. Исходный текст функции `F` нам доступен, и в нем задан путь `p`, через который должен пройти управляющий поток. Мы проводим символьную интерпретацию функции `F` вдоль пути `p` и получаем символьное выражение для возвращаемого значения `y=Fp(x)=x*x-2*x+1` при `x<0`. Выполняем подстановку `Fp(x)` вместо `F(u)`, а предикат `x<0` для тела функции включаем в предикат для основной программы. В итоге получим `7-u*u-2*u+1<0 AND u<0`.

Путь в теле функции может быть явно не специфицирован, т.е. с точки зрения генерации теста для основной программы нас не интересует движение управляющего потока внутри вызванной функции. Тогда в результате символьной интерпретации тела функции мы получим набор альтернатив $y_1=F_1(x)$ *AND* $Pr_1(x)$ *OR* $y_2=F_2(x)$ *AND* $Pr_2(x)$ *OR … OR* $y_n=F_n(x)$ *AND* $Pr_n(x)$, каждая из которых в равной степени подходит для подстановки в предикат пути основной программы. Мы осуществляем подстановку всех y_i по следующему правилу. Каждый дизъюнкт в основной программе, содержащий вызов функции *F,* преобразуется в *n* идентичных дизъюнктов (т.е. создается *n-1* копия), связанных операцией *OR*. В теле *i*-го дизъюнкта осуществляется подстановка $F(u)->F_i(x)$ и к дизъюнкту добавляется предикат $Pr_i(x)$. Например, если в функции F есть ветка для значений $x>0$ и для нее $y=x*x*x$, то после подстановки предикат основной программы примет вид `7-u*u-2*u+1<0 AND u<0 OR 7-u*u*u<0 AND u>0`.

Передача скалярных параметров по ссылке или по указателю анализируется аналогичным образом. При этом функция *F* обрабатывается как функция, возвращающая несколько значений, и для каждого пути внутри функции мы получим несколько символьных выражений – по одному для каждой переменной, результирующее значение которой доступно в объемлющем блоке (вызывающей функции).

На текущий момент мы не умеем анализировать передачу массивов в качестве фактических параметров функции. Это ограничение связано с общей проблемой символьного анализа массивов, которая будем рассмотрена далее в этой работе.

Символьная интерпретация функции, исходный текст которой недоступен, например, библиотечной функции, невозможен. Мы считаем, что с этим ограничением можно бороться при помощи табличного задания функции. Предположим, что для библиотечной функции у нас есть таблица значений, тогда вместо подстановки символьного выражения для тела функции мы подставляем в точку вызова символьное выражения для таблицы. Таблица для функции $y=F(x)$ представляется в виде ДНФ $y_1=c_1$ *AND* $x_1=a_1$ *OR* $y_2=c_2$ *AND* $x_2=a_2$ *OR … OR* $y_n=c_n$ *AND* $x_n=a_n$. Мы пока не рассматриваем методы получения такой таблицы для произвольной библиотечной функции, однако, зная область допустимых значения каждого из параметров функции,

можно построить сетку и вычислить значение функции (выполнив ее) для каждой точки сетки.

Массивы и указатели относятся к языковым элементам, анализ которых в ходе символьной интерпретации в общем случае является затруднительным. Проблема заключается в том, что в ходе построения символьных выражений нам необходимо знать, какой элемент массива обрабатывается в данной точке программы. Для указателей необходимо знать, на какой объект он указывает в данной точке программы.

Мы рассматриваем программы, в которых массивы удовлетворяют следующим ограничениям:

- Массивы распределяются статически или на стеке, нет динамического выделения памяти для массива. Размер массива известен на стадии компиляции.
- Массивы не используются в качестве формальных и фактических параметров функции.
- Блочные операции с массивами не используются.
- Выражение `Expr`, используемое для адресации ячейки массива `A (A[Expr])`, является константным и может быть вычислено при помощи статического анализа. Исключение составляют циклы, о которых будет рассказано далее в этой работе.

Анализируя программу, удовлетворяющую указанным ограничениям, мы рассматриваем массив как набор отдельных переменных. Каждому элементу массива ставится в соответствие служебное имя, которое является уникальным внутри контекста, в котором объявлен массив. Например, операция доступа к элементу массива `A[7][25]` заменяется на *__%A_7_25__*.

Адрес каждой адресуемой ячейки массива вычисляется на стадии компиляции. В промежуточном представлении программы (абстрактном синтаксическом дереве) указывается соответствующее служебное имя используемой ячейки. Во время символьной интерпретации программы во всех символьных выражениях фигурируют служебные имена ячеек. Системы уравнений, формируемые по предикатам ветвей анализируемого пути, также используют служебные имена элементов массива. Если некоторый массив является входной переменной, то при генерации теста на основе решения системы уравнений производится обратное преобразования имени в элемент массива с соответствующим адресом.

Все элементы массива, которые входят в решение системы уравнений, получают соответствующие значения из решения. Значения для всех остальных элементов выбираются случайным образом из множества значений заполненных ячеек. Например, пусть в программе объявлен массив `A[2][2]`, `A` – входная переменная. Решая систему, мы получили решение, в которое входит `__%A_0_0__=3` и `__%A_1_1__=7`, тогда мы генерируем набор входных данных `A[0][0]=3, A[1][1]=7, A[0][1]=Random{3,7}, A[1][0]=Random{3,7}.`

Статический анализ программ, в которых использование массивов выходит за рамки указанных ограничений, является крайне сложным или вообще невозможным. Мы исследуем возможность применения динамического анализа программы для решения этой проблемы.

Использование указателей в программе также затрудняет ее статический анализ, поэтому мы на текущий момент рассматриваем программы, в которых указатели используются только для передачи скалярного значения в функцию по указателю. Декларация указателя допускается только в списке формальных параметров функции. Любое использование указателя в теле функции ограничивается операцией взятия значения “*”. Например, если в программе есть объявление функции `int F(int* i)`, то в теле функции *F* допускается использование `*i` как в левой, так и в правой части оператора присваивания. Изменение самого указателя не допускается, но он может использоваться как фактический параметр. Например, в теле функции *F* может быть обращение `G(i)`. Такие ограничения позволяют нам на стадии компиляции определить, на какой объект указывает любой из указателей, определенных в программе.

Символьная интерпретация пользовательских функций рассматривалась ранее в этой работе. Отметим особенность построения символьных выражений для таких функций. Если в определении функции присутствует формальный параметр, передаваемый по указателю, то символьные выражения строятся не только для значения, возвращаемого в операторе `return`, но также для всех таких параметров функции. В языке *Fortran 77* все формальные параметры передаются по ссылке, поэтому данное правило символьной интерпретации функций всегда применяется при анализе *Fortran*-программ.

Сложность символьной интерпретации циклов состоит в том, что определить зависимость между входными и выходными данными цикла и выразить ее в виде символьных выражений, не развертывая итерации цикла, практически нереально. Вместе с тем значения внутренних переменных, изменяемых в цикле, могут оказать влияние на дальнейшее поведение программы, а именно на участке от точки выхода из цикла до конца программы.

Мы рассматриваем программы, в которых присутствуют циклы только с фиксированным количеством итераций, т.е. `FOR`-циклы с константными границами. Например, `FOR i=1 TO 10` в Фортране или `for(i=0; i<=10; i++)` в Си. Шаг цикла может отличаться от единицы, но должен быть константным. Цикл с фиксированным числом итераций мы разворачиваем в линейную программу. На каждой операции обращение к значению итератора цикла заменяется соответствующей константой. Как было сказано ранее в этой работе, в циклах допускается использование массивов с переменными адресными выражениями, если адрес ячейки зависит только от итератора цикла. Для циклов ограничение на использование массивов в программе ослабляется, потому что в процессе развертки цикла значение итератора заменяется на константу и, следовательно, использование массивов в

программе, получающейся после развертки цикла, удовлетворят ранее установленным ограничениям.

Рассмотрим пример.

Пусть в программе присутствует нижеприведенный цикл

```
DO 10 I=1, 2
 IF (A[I] .GT. 0) THEN
 S = S + A[I]
 ELSE
 S = S - A[I]
 ENDIF
10 CONTINUE
```

Сначала для каждой итерации создается копия тела цикла, и во всех точках использования итератора он заменяется на фактическое значение – константу.

```
 IF (A[1] .GT. 0) THEN
 S = S + A[1]
 ELSE
 S = S - A[1]
 ENDIF
 IF (A[2] .GT. 0) THEN
 S = S + A[2]
 ELSE
 S = S - A[2]
 ENDIF
```

Затем все обращения к элементам массива заменяются на соответствующие служебные имена.

```
 IF (__%A_1__ .GT. 0) THEN
 S = S + __%A_1__
 ELSE
 S = S - __%A_1__
 ENDIF
 IF (__%A_2__ .GT. 0) THEN
 S = S + __%A_2__
 ELSE
 S = S - __%A_2__
 ENDIF
```

Если в программе содержатся вложенные циклы, то они последовательно разворачиваются, начиная с самого глубокого уровня вложенности. Полученная в итоге развертки циклов программа удовлетворяет установленным ограничениям и допускает символьную интерпретацию.

Рассмотренные методы символьной интерпретации программы, содержащей сложные для статического анализа языковые конструкции, могут значительно увеличить размер получаемых в итоге символьных выражений. Это приводит к построению систем уравнений и неравенств большой размерности, решение

которых может быть затруднительным даже для современных численных методов.

Уменьшение размерности задачи может быть достигнуто путем выделения в программе блоков, каждый из которых анализируется отдельно. Для каждого блока независимо от других выполняется символьная интерпретация, строится система уравнений и находится ее решение. После независимого анализа всех блоков полученные решения объединяются, и находится решение исходной задачи, которое является основой для генерации теста для заданного пути. Основным кандидатом в такие блоки являются пользовательские процедуры и функции, исходный текст которых доступен для статического анализа.

4. Методы упрощения систем уравнений и неравенств

В этом разделе будут рассмотрены основы метода декомпозиции задачи генерации входных данных, применяемого для упрощения систем уравнений и неравенств. Предполагается, что анализируемая программа отвечает ранее установленным ограничениям. Изложенное обоснование метода основывается на том, что мы всегда можем получить аналитическое решение для системы уравнений. Это ограничение является достаточно жестким при анализе больших программ. Мы исследуем возможности модификации метода декомпозиции для применения численных методов решения системы уравнений.

Рассмотрим простой пример.

```
1: READ(X)
2: X = F(X)
3: X = F(X)
4: IF (X > 7) THEN
  …
5: ENDIF
6: FUNCTION F(X)
7: RETURN (X+1)
```

Пусть нам необходимо сгенерировать входные данные, реализующие прохождение управляющего потока через тело условного оператора. Если бы мы применяли изложенный ранее метод, то после символьной интерпретации программы и подстановки тела функции `F` в обеих точках ее вызова мы бы получили неравенство `((X+1)+1)>7`. Решение этого неравенства, например, `X=6`, является входным значением, проходящим вдоль заданного пути.

Рассмотрим, как поставленная задача может быть решена путем декомпозиции программы.

Каждый вызов функции `F` заменяется на уникальную служебную переменную. Если функция вызывается несколько раз с одним набором параметров, то каждый вызов замещается отдельной уникальной переменной. Мы не анализируем, менялись ли значения фактических параметров между вызовами

или нет, передавались ли параметры по значению или по ссылке. После замещения обоих вызовов функции *F* программа примет следующий вид:

```
1: READ(X)
2: X = %Y
3: X = %Z
4: IF (X > 7) THEN
  …
5: ENDIF
```

Где `%Y = F(X.2), %Z = F(X.3).` Выполним символьную интерпретацию программы и тела функции *F*. Получим следующие символьные выражения:

```
X.2 = X0
X.3 = %Y
X.4 = %Z
```

Вызовов функций в преобразованной программе нет, поэтому подстановки тел функций в точках вызова не осуществляются.

Начинаем решать задачу с анализа предиката `%Z > 7`. Это неравенство само по себе является решением, поэтому мы сразу получаем ограничение на область допустимых значений служебной переменной `%Z`. Извлекаем сохраненное представление этой служебной переменной и подставляем его в неравенство `%Z > 7.` Получаем соотношение `F(X.3) > 7`. Используя ранее полученное символьное выражение для `X.3`, имеем `F(%Y) > 7`.

Подставляем символьное выражение для возвращаемого значения функции `F`, т.е. делаем подстановку тела функции, и получаем `%Y+1>7`. Переносим единицу в правую часть неравенства и получаем ограничение на множество допустимых значений служебной переменной `%Y.`

Повторяем для служебной переменной `%Y` все шаги по аналогии с переменной `%Z`. При этом мы выполняем вторую подстановку тела функции `F` и получаем ответ `X>5`. Этот ответ в точности такой же, какой дает анализ программы целиком, не прибегая к ее декомпозиции.

Рассмотрим алгоритм декомпозиции программы в общем виде.

Пусть необходимо сгенерировать входные данные, проходящие вдоль заданного пути *u*. Прежде всего, выполним необходимые преобразования для программы, а именно развернем циклы и произведем необходимые замены наименований переменных в точках обращения к элементам массивов.

Управляющий граф полученной программы не содержит циклов. Вдоль пути *u* произведем замещение вызовов всех функций на обращение к служебным переменным с уникальными именами. Если параметр функции передается по ссылке (по указателю), то все операции доступа к значению фактического параметра, расположенные ниже вызова функции, также заменяются на обращение к уникальной служебной переменной.

Выполним символьную интерпретацию основной программы и всех пользовательских функций.

Рассмотрим путь *u* в основном (головном) блоке программы и построим систему уравнений и неравенств для предикатов ветвей этого пути. Предположим, что мы нашли аналитическое решение этой системы. Найденное решение представляет собой набор ограничений на множество допустимых значений входных переменных и/или служебных переменных.

Рассмотрим все служебные переменные в решении. Эти переменные соответствуют либо значениям, возвращаемым пользовательской функцией, либо выходным значением фактического параметра, передаваемого по ссылке (по указателю).

Пользуясь результатами символьной интерпретации пользовательских функций, выполним обратную подстановку для каждой служебной переменной. Рассмотрим эту операцию для служебной переменной, представляющей собой выходное значение второго параметра `(Y)`, передаваемого по ссылке, функции `F(X,Y)`.

Пусть где-то в исходной программе был вызов `F(U,V)` функции `F` с параметрами `(U,V)`. Пусть выходному значению второго параметра мы сопоставили служебную переменную `%A`. Если, решая на некотором этапе анализа программы систему уравнений, мы получили решение, зависящее от `%A`, то мы выполняем следующие действия.

Возьмем символьное выражение для выходного значения параметра `Y`, которое было вычислено ранее в ходе символьной интерпретации функции `F`. Это выражение представляет собой набор элементов вида `Yout = Expr(Xin,Yin)` при `Cond(Xin,Yin)=true`, соединенных операцией дизъюнкции. Выполним символьную подстановку для *X* и *Y*, заменив их на фактически переданные в функцию значения `U` и `V` соответственно. В результате мы получим набор элементов вида `Yout = Expr(U,V)` при условии истинности ограничения `Cond(U,V)`. Подставляем `Expr(U,V)` в решение вместо *%A* и добавляем в систему уравнений и неравенств ограничение `Cond(U,V)`.

Если внутри функции есть `n` путей, то мы получаем `n` альтернативных символьных выражений для `Yout` – по одному для каждого пути. Каждое из этих выражений может быть подставлено вместо служебной переменной `%A`. Для нахождения искомых входных данных программы нам достаточно, чтобы хотя бы одна из подстановок привела нас к решению. Вместе с тем, нам необходимо проверить все подстановки, чтобы убедиться, что решения исходной задачи не существует и путь является недостижимым.

Описанные выше действия повторяются для каждой служебной переменной до тех пор, пока мы не избавимся в системе ограничений от всех служебных переменных. В конце концов, у нас останутся ограничения только для входных

переменных. Разрешая эти ограничения, мы найдем искомые входные значения, проходящие вдоль пути *u*.

Заметим, что в структурированной программе любой структурный элемент, имеющий один вход и один выход, можно описать в виде отдельной функции. А сам элемент в тексте программы заменить на обращение к этой функции. Таким образом, метод декомпозиции обобщается на произвольный структурный элемент, например, цикл.

5. Близкие работы

В последнее время задаче генерации входных данных, реализующих выполнение программы по заданной трассе в управляющем графе, уделяется значительное внимание. Разработаны различные методы решения этой задачи [2]. Среди всего разнообразия методов можно выделить два основных класса – статические и динамические. Статические методы основываются исключительно на статическом анализе программы и ее символьной интерпретации [3 – 6]. Динамические методы реализуют генерацию входных данных при помощи итеративного выполнения программы на последовательно “улучшаемых” наборах данных до тех пор, пока не будет найден искомый набор. Для корректировки данных после каждой итерации, как правило, применяют различные методы оптимизации [7]. Отметим смешанный статико-динамический подход, основанный на методе последовательной релаксации [8]. Для решения задачи генерации входных данных также применяются генетические алгоритмы [9, 10].

Методы, предложенные в настоящей работе, являются статическими. Они позволяют рассматривать более широкий класс программ, чем методы [3 – 6]. Кроме того, сняты ограничения на размер анализируемых программ. Дальнейшее развитие связано с уточнением методов, описанных в разделе 3 (межпроцедурный анализ, более тщательный анализ указателей и др.), развитием метода декомпозиции и изучением возможности применения динамических методов для анализа недоступных компонентов анализируемой программы (внешние функции и др.).

Литература

1. А. Ахо, Р. Сети, Д. Ульман. *Компиляторы: Принципы, Технологии, Инструменты*. Вильямс, М, 2001.
2. J. Edwardson. *A survey on automatic test data generation*. ECSEL: Proceedings of the 2nd Conference on Computer Science and Engineering in Linkoping, 1999, pp. 21-28.
3. B.J. Choi, A.P. Mathur, R.A. DeMillo, E W. Krauser, R.J. Martin, A.J. Offutt, E.H. Spafford. *The Mothra tool set*. Proceedings of the 22nd Hawaii International Conference on System Sciences, 1989, pp. 275 – 284.
4. *R. A. DeMillo, A. J. Offutt*. Constraint-based automatic test data generation. IEEE Transactions on Software Engineering, 17(9), 1991, pp. 900 – 910.
5. *R. Ferguson, B. Korel*. *The chaining approach for software test data generation*. IEEE Transactions on Software Engineering, 5(1), January 1996, pp. 63 – 86.

6. B. Korel. *Automated software test data generation*. IEEE Transactions on Software Engineering, 16(8), August 1990, pp. 870 – 879.
7. N. Tracey, J. Clark, K. Mander. *Automated program flaw finding using simulated annealing*. Proceedings of ACM SIGSOFT international symposium on Software testing and analysis, volume 23, March 1998, pp. 73 – 81.
8. N. Gupta, A.P. Mathur, M.L. Soffa. *Automated Test Data Generation Using An Iterative Relaxation Method*. SIGSOFT '98, 11/98, Florida, USA.
9. R.P.Pargas, M.J.Harrold, R.R.Peck. *Test-Data Generation Using Genetic Algorithms*. Technical Report, July, 1999.
10. C.Michael, G. McGraw. Automated Software Test Data Generation for Complex Programs. Technical Report.
11. I. Bourdonov, A.V. Demakov, A. Kossatchev, A. Petrenko, D. Galter. *KVEST: Automated Generation of Test Suites from Formal Specifications*. Proceedings of World Congress of Formal Methods, Toulouse, France, LNCS, No. 1708, 1999, pp. 608-621.
12. C. Yan. *Performance Tuning with AIMS - An Automated Instrumentation and Monitoring System for Multicomputers*. Proceedings of the 27th Hawaii international Conference on Systems Sciences, ACM, January 1994.
13. L. DeRose, Y. Zhang, D. Reed. *SvPablo: A multi-language performance analysis system*. In Proceedings of 10th International Conference on Computer Performance Evaluation, September 1998.
14. B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B.Irvin, K.L. Karavanic, K. Kunchithapadam, T. Newhall. *The Paradyn Parallel Performance Measurement Tools*. IEEE Computer 28, 11, November 1995.
15. H. Agrawal, J.R. Horgan. *Dynamic program slicing*. Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices, 25(6), 1990, pp. 246-256.
16. D. Jackson, E.J. Rollins. *A new model of program dependencies for reverse engineering*. Proceedings of the 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM SIGSOFT Software Engineering Notes 19(12), 1994, pp. 2-10.
17. E.W. Dijkstra. *Notes on structural programming*. TH-Report 70-WSK-03, Dept. of Mathematics, Technological University Eindhoven, The Netherlands, 1970.