электронный журнал
МОЛОДЕЖНЫЙ НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК
Издатель ФГБОУ ВПО "МГТУ им. Н.Э. Баумана". Эл No. ФС77-51038.

УДК 004.421

МЕСТО ПРИЁМОВ ДЕДУКТИВНОЙ ВЕРИФИКАЦИИ В АВТОМАТИЧЕСКОМ СИНТЕЗЕ АЛГОРИТМОВ

***Минашин Г.А.**, студент*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедр систем обработки информации и управления

Научный руководитель: Гапанюк Ю. Е., к.т.н., доцент
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана
bauman@bmstu.ru

1 Введение

Важной составляющей научно-технического прогресса всегда являлась автоматизация человеческого труда. В идеале помимо увеличения производительности автоматизация позволяет делегировать часть низкоквалифицированного труда машинам; при этом планка, отделяющая высококвалифицированный труд от низкоквалифицированного, также перемещается вверх.

Информационные технологии (далее – ИТ) в аспекте автоматизации труда можно рассматривать с двух сторон.

Во-первых, эти технологии являются средством автоматизации. Например, научно-инженерная программа выполняет решение математической задачи численными методами в миллионы раз быстрее, чем пришлось бы человеку, а автоматизированные системы управления производством, основанные на информационных технологиях, самостоятельно проводят работы по организации и поддержке документооборота.

Во-вторых, сами ИТ могут стать объектом автоматизации. В настоящее время, к примеру, наблюдается существенный интерес к средствам моделирования на крайне высоком уровне абстракции, порождающим исполняемые программы в автоматическом режиме.

Существует большое число направлений автоматизации работ в области ИТ. Среди прочего следует отметить переход от императивного стиля программирования к декларативному. Чёткой границы, разделяющей эти два стиля, выделить почти невозможно, но можно с уверенностью говорить, что со временем общий уровень «декларативности» неуклонно повышается.

Известно, что на заре использования электронно-вычислительных машин программирование велось на машинных кодах целевых платформ. Появление первых языков программирования, таких, как «Фортран», «Алгол» и «Лисп», уже привело к появлению

понятия «декларативного»: теперь программист уже не видел своими глазами машинный код, он объявлял, *что* хочет сделать, а решение о том, *как* это сделать частично зависело от транслятора.

С тех пор множество задач, выполняемых транслятором, а вместе с ним и множество преобразований, отделяющих исходный код от машинного, неуклонно увеличивалось. Понятия декларативного и императивного подхода размывались и пересматривались; сегодня наиболее яркими представителями лагеря декларативных языков программирования можно считать «SQL» и «HTML».

Можно отметить также, что переходу от императивного стиля к декларативному в некотором роде соответствует повышение уровня абстракции, т. е. получение возможности концентрироваться на задачах предметной области вместо задач оптимизации алгоритма относительно конкретной платформы.

По всей видимости, сама императивная природа машинного кода привела к тому, что на сегодня императивные языки программирования имеют колоссальное преимущество перед декларативными по таким параметрам, как распространённость и универсальность. При этом, как уже было сказано, декларативный подход является естественным спутником прогресса и автоматизации. Таким образом, становится актуальной задача создания набора промышленных декларативных языков программирования, имеющих возможность заменить современные императивные.

Для решения этой задачи можно воспользоваться методами формальной верификации, в которых выделяют дедуктивную верификацию, сравнение эквивалентности и проверку модели [1, стр. 27]. Будем использовать приёмы дедуктивной верификации, так как именно она подходит для описания трансформационных систем (программ) [1, стр. 58], а кроме того обладает мощным математическим аппаратом, позволяющим устанавливать истинность компьютерных программ. Хотя принципы дедуктивной верификации были разработаны достаточно давно, на практике они были применены в сравнительно небольшом объёме. Большинству программистов, возможно, этот аппарат знаком в виде контрактного программирования.

В самом простом виде контрактное программирование можно представить в виде контрольных точек, расставленных в исходном коде. В этих контрольных точках программист указывает выражения, которые при правильном выполнении программы должны быть истинны. Разумеется, истинность каждого выражения зависит от переменных, с которыми работает программа. На этапе трансляции происходит анализ программы и контрольных точек и, если обнаруживается противоречие, выдаётся соответствующее сообщение.

Родственным и одновременно модным современным приёмом является модульное тестирование.

2 Постановка задачи

Теперь посмотрим, что получится, если интенсифицировать применение методов дедуктивной верификации и использовать их не для анализа уже готовой программы, а для синтеза.

Такой приём широко применяется для обучения «хорошим манерам» программирования; также при помощи этого приёма можно провести доказательство того, что программирование вообще имеет смысл и может быть использовано для решения конкретной задачи.

В промышленных масштабах, тем не менее такой строгий подход к проектированию алгоритмов почти не применяется. Вместо этого программисты обычно полагаются на интуицию, аналогии и шаблоны, что, разумеется, приводит к увеличению скорости разработки программного обеспечения и одновременно к увеличению потерь ресурсов на отладку и тестирование.

Для понимания причин, из-за которых применение формального подхода к проектированию алгоритмов мало распространено, рассмотрим классическую задачу построения программы при помощи инварианта [2, лекция 5]. Программа должна найти наибольший общий делитель (далее – НОД) двух целых чисел m и n, хотя бы одно из которых не равно нулю. Будем использовать для решения широко известный алгоритм Евклида.

В качестве локальных переменных, используемых в программе выберем некоторые целочисленные переменные a и b. Используем следующий инвариант: НОД текущей пары чисел равен НОД исходной пары чисел (P: НОД(a, b) = НОД(m, n)). Также обозначим через r остаток от деления значения переменной a на значение переменной b: $a = i \cdot b + r$, $|r| < |b|$.

Можно показать [3, стр. 9], что НОД(b, r) = НОД(a, b). Таким образом НОД не меняется при переходе от пары (a, b) к паре (b, r), а, следовательно, остаётся истинным и инвариант (при условии его истинности до перехода). Такой переход и будет составлять тело будущего цикла. Для завершения разработки алгоритма нужно определить критерий, отвечающий за то, будет ли выполняться ещё одна итерация или произойдёт выход из цикла.

Нужно отметить, что выполнение условия выхода из цикла должно обеспечивать решение задачи. Также известно, что НОД(a, 0) = a, т. е. если b = 0, НОД можно вычислить очень легко. Это условие (b = 0) и следует выбрать в качестве условия выхода, тогда условие продолжение цикла будет $b \neq 0$.

Все эти рассуждения приводят к формированию программы, псевдокод которой

представлен на рисунке.

```
1 int GCD(int m, int n) {
2     int a = m, b = n; // P
3     while(b ≠ 0) {    // P
4         r = a % b;
5         a = b;
6         b = r;        // P
7     }
8     return a;         // P
9 }
```

Псевдокод программы, вычисляющей НОД двух целых чисел по алгоритму Евклида

Особое внимание следует обратить на строки, помеченные буквой P в комментарии (это строки №№ 2, 3, 6 и 8). После выполнения инструкций, содержащихся в этих строках, необходимо будет истинен инвариант P: НОД(a, b) = НОД(m, n). В строке №8 также будет истинен предикат b = 0.

Ещё раз взглянув на рассуждения, проведённые во время построения алгоритма программы, можно сказать, что хотя составление программы и обрело теоретическую обоснованность и осмысленность, эта обоснованность базируется на относительно неформальных умозаключениях.

Например, в начале рассуждения был выбран инвариант P: НОД(a, b) = НОД(m, n). Причина, по которой был выбран именно такой инвариант, заключается в том, что явно или неявно был проанализирован используемый для решения алгоритм Евклида. Корректность этого алгоритма вытекает из того, что НОД(a, b) = НОД(b, r) и НОД(a, 0) = a.

Далее в качестве тела цикла был выбран набор инструкций, заменяющий значения переменных a и b на b и r соответственно. Интересно, что из бесчисленного множества возможных преобразований было выделено именно это. Очевидно, в этом случае опять имел место некоторый анализ, на этот раз – инварианта. Одновременно был сделан вывод о том, что выбираемое преобразование (замена (a, b) на (b, r)) приближает к решению задачи.

Наконец, было неявно отмечено, что при выполнении программы будет когда-нибудь достигнуто состояние, в котором b = 0. В таком состоянии дать ответ будет элементарно, поэтому такое условие (b = 0) и было сделано условием выхода из цикла.

Таким образом, во время разработки алгоритма был сделан целый ряд неформальных логических выводов. Безусловно, все они следовали из простейших математических законов, а их истинность доказывается весьма строго. Тем не менее выбор конкретного математического закона и, главное, следствия из него опирался на интуицию и аналогию.

Более того, человек, не имеющий опыта программирования и не знакомый непосредственно с алгоритмом Евклида, наверняка потратил бы на составление программы очень большое время, в то время как для этого нужно знать лишь основные математические законы, указанные выше. Кажущаяся простота синтеза алгоритма наводит на мысль делегации этой задачи машине. При этом необходимо иметь некоторые чёткие рекомендации по выбору основных элементов программы.

Под основными элементами программы могут пониматься подготовительные и заключительные инструкции, условия выхода из цикла, инструкции тела цикла. Именно в этом направлении было создано множество принципов в рамках теории формальной верификации.

В данной статье рассмотрим задачу автоматического построения алгоритма в самом общем виде при помощи методов дедуктивной верификации. Также будем использовать конкретный пример для проверки рассуждений на практике.

3 Решение задачи

Традиционно в дедуктивной верификации применяются два основных понятия: это инвариант и ограничивающая функция. Инвариант представляет собой предикат, сохраняющий истинность во всех интересующих программиста местах программы. В контексте циклов это значит, что инвариант должен сохранять истинность до первой итерации, в начале каждой итерации и после завершения цикла.

Ограничивающая функция также зависит от переменных программы и уменьшается с каждой итерацией (или некоторым другим элементарным набором инструкций). Она может служить для примерной оценки времени, оставшегося до завершения работы.

Процесс синтеза алгоритма может быть разбит на две части, следующих одна за другой. Во-первых, на основании прикладной задачи, которая должна быть решена при помощи синтезируемого алгоритма, формируются инвариант и ограничивающая функция. Во-вторых, на основании полученных инварианта и ограничивающей функции формируются основные элементы программы.

Для первой части необходимо формализовать прикладную задачу. Обычно это делается в терминах постусловий и предусловий. Постусловие (консеквент) – это состояние, которое должно быть установлено в результате работы программы, а предусловие (антецедент) – это состояние, которое должно быть истинно перед началом работы программы.

После спецификации прикладной задачи постусловием и предусловием производится получение инварианта и ограничивающей функции. В общем случае это нетривиальная

задача; для её решения могут быть предложены различные методы. Например, выделяются четыре основных метода получения инварианта на основе постусловия [4, стр. 198]: удаление конъюнктивного члена, замена константы переменной, расширение области допустимых значений переменной и добавление дизъюнктивного члена. Все они применяются в разных условиях, а последний метод почти не может быть использован на практике.

Полученный инвариант представим в виде предиката, содержащего некоторые переменные. В качестве ограничивающей функции имеет смысл выбирать функцию, зависящую от этих переменных. На каждой итерации цикла переменные будут изменяться, влияя на значение ограничивающей функции, но при этом оставаться в таком отношении, чтобы инвариант оставался верен.

Хотя методы получения инварианта и ограничивающей функции существуют, можно увидеть, что в большинстве случаев они также недостаточно формализованы и опираются опять же на интуицию и опыт.

Попробуем рассмотреть ещё один пример синтеза алгоритма на основе теоретического подхода. Теперь постараемся максимально строго обосновать рассуждения, приводящие к появлению тех или иных элементов алгоритма. На этот раз будем разбирать более простую, но обладающую основными стандартными компонентами задачу.

3.1 Пример целенаправленного и формализованного построения алгоритма

Пусть дан массив целых чисел A с натуральной нумерацией и длиной N (то есть элементы массива имеют индексы от 1 до N включительно). Необходимо записать в некоторую переменную s сумму элементов этого массива.

Прежде всего формализуем задачу: она заключается в установке постусловия $R: s=\sum_{i=1}^{N} A_i$ при неизменности A и N. Также отметим, что задача связана с обходом массива и, очевидно, будет решена при помощи цикла.

Проанализируем постусловие и построим инвариант. Постусловие содержит постоянные N и A и переменную s (переменная i является связанной и имеет смысл только в квантифицированном выражении). Заменим какую-либо постоянную переменной.

Постоянная A является массивом. Попробуем найти более простой объект, возможно, скаляр. Такой объект есть: это постоянная N. Заменим её переменной j. Тогда получим инвариант $P(j): s=\sum_{i=1}^{j} A_i$, который имеет следующий смысл: переменная s содержит сумму j первых элементов массива A.

Найдём начальные условия. Пусть j=1. Инвариант: $P(1): s=\sum_{i=1}^{1} A_i = A_1$. Установка начальных условий заключается в применении инструкции (s ← 1) и требует одного присваивания и вычисления одного выражения, состоящего из одной переменной (A_1). Такие начальные условия подходят, они могут быть легко получены из предусловия, указанного в задаче.

Попробуем выбрать ограничивающую функцию. Выберем в качестве ограничивающей функции разность между постоянной, заменённой переменной (N), и самой этой переменной (j): t(j) = N - j. Заметим, что можно было бы выбрать в качестве ограничивающей функции и просто t(j) = - j; поправка на N обеспечит неотрицательные значения функции, которые можно будет интерпретировать как меру числа шагов, оставшихся до завершения.

Построим инструкцию, уменьшающую ограничивающую функцию. Ограничивающая функция зависит только от одной переменной – j. Выберем инструкцию (j ← j + 1), тогда t(j + 1) - t(j) = N - j - 1 - N + j = -1, то есть функция, как и ожидалось, уменьшается.

Проверим инвариант после применения инструкции.

До применения инструкции $P(j): s=\sum_{i=1}^{j} A_i = A_1 + A_2 + \ldots + A_j$.

После применения инструкции $P(j+1): s'=\sum_{i=1}^{j+1} A_i = A_1 + A_2 + \ldots + A_j + A_{j+1}$.

Тогда разность $d=s'-s=A_{j+1} \neq 0$ в общем случае. Добавим инструкцию (s ← s + d), что приведёт к восстановлению инварианта.

Рассмотрим область изменения переменной цикла и построим охрану. Переменная цикла изменяется от значения, использованного для установки начальных условий (1) до заменённой постоянной (N). Изменение переменной происходит «снизу вверх», следовательно охраной цикла будет являться j ≠ N.

Покажем, что после завершения цикла будет установлено требуемое постусловие. В конце цикла охрана будет ложной, то есть j = N. Одновременно будет верен инвариант, следовательно $j=N \wedge s=\sum_{i=1}^{j} A_i \Rightarrow s=\sum_{i=1}^{N} A_i = R$.

Итак, были пройдены главные этапы построения алгоритма решения прикладной задачи. Теперь постараемся на основании этого примера в некоторой степени обобщить и, главное, формализовать проведённые рассуждения. В результате должна получиться некоторая методика формирования алгоритма.

3.2 Частный случай методики автоматического синтеза алгоритма методами дедуктивной верификации

Прежде всего обозначим класс задач, для которых будет работать разрабатываемая методика. Будем рассматривать автоматическое построение алгоритма для решения элементарных задач, явно связанных с перебором элементов массива, с нахождением инварианта через замену константы переменной.

Исходными данными является постусловие R, представляющий собой предикат, содержащий объекты (переменные и постоянные) программы. Будем считать также, что все объекты постусловия объявлены и инициализированы.

Очевидно, что решение задачи также зависит и от предусловия. Мы будем предполагать, что решаем задачу в самом общем виде, то есть так, чтобы её решение вообще не зависело от предусловий (формально такое предусловие может быть обозначено, как R: истина). Если же окажется, что это невозможно, будет истребована установка какого-либо конкретного предусловия S. Существование и конкретизация предусловия S может существенно упростить или решение, или доказательство того, что решение невозможно.

Решение задачи заключается в установлении такого состояния, в котором было бы верно постусловие R. Методика построения алгоритма для решения этой задачи сама выражается в виде алгоритма, который описывается далее.

Задача связана с перебором элементов некоторого массива и будет решаться при помощи программного цикла. Таким образом, необходимыми компонентами методики являются: а) построение инварианта цикла P, б) построение ограничивающей функции t, в) выбор инструкции и охраны цикла.

Сначала рассмотрим построение инварианта при помощи замены константы переменной. Выделим множество C всех объектов, от которых зависит постусловие (то, что требуется установить в конце работы программы). Далее исключим из C все объекты, изменение которых разрешено программистом (то есть переменные). В C должны остаться только константы – объекты, изменение которых недопустимо во время выполнения. Среди этих констант нужно будет выделить одну ту, которая и будет заменена переменной. Успешное применение методики, разумеется, возможно только в случае непустого множества C.

Следует стараться прежде всего найти наиболее «простую» константу. Это значит, что скаляры имеют явный приоритет над массивами, а массивы меньшего размера -- над массивами большего размера. Итак, самая «простая» константа заменяется переменной, которую будем называть переменной цикла j; инвариант P тождественен постусловию R, в котором выбранная константа заменена переменной цикла j.

Второй шаг – установка состояния, в котором был бы верен инвариант P. Так как инвариант безусловно зависит от переменной цикла j, необходимо одновременно найти такое её значение, при котором установка инварианта была бы наименее затруднительна. Для оценки трудности можно использовать различные подходы, например, подсчитывать количество инструкций в коде среды исполнения.

Таким образом, нужно перебирать возможные значения переменной цикла j. Для каждого значения происходит по возможности генерация инструкций, которые приводят к установке инварианта P с этим значением. Может случиться, что вообще не найдётся такого значения переменной цикла, при котором установка инварианта была бы приемлемо быстрой. Это значит, что или 1) нужно вернуться назад и попробовать выбрать другую константу для замены, или 2) постусловие слишком нетривиально для анализа и требуется его упрощение, или 3) программист должен самостоятельно добавить инструкции так, чтобы предусловие было более слабым. Последний вариант на самом деле предполагает установку либо конкретизацию предусловия S.

Относительно этого шага можно сделать ещё одно замечание. Оно касается сложности выбора области определения переменной цикла j. Для правильного выбора нужно использовать смысл заменённой константы. Например, если был заменён размер массива, то становится ясно, что переменная цикла не должна принимать значения, которые могли бы привести в выходу за границы массива.

На третьем шаге найдём ограничивающую функцию t. Предполагается, что ограничивающая функция должна «приближать» программу к завершению и предоставлять некоторую (возможно, неточную) оценку числа шагов, оставшихся до завершения. Во-первых, это значит, что ограничивающая функция t должна зависеть от переменной цикла j. Во-вторых, ограничивающая функция убывает, так как с каждым шагом программа должна приближаться к завершению и, следовательно, после каждого шага число оставшихся шагов уменьшается. В случае, если j принимает числовые значения, это неформальное утверждение можно формализовать так: $\forall\ j : t(j_{next}) < t(j)$, где j_{next} – значение переменной цикла на следующем шаге программы.

Значение на переменной цикла на следующем шаге выводится из условия приближения к заменённой константе. Приближение же к заменённой константе необходимо, так как инвариант P, где вместо j поставлена заменённая константа, превращается в постусловие R.

Четвёртый шаг заключается в нахождении инструкции, которая бы уменьшила ограничивающую функцию. В случае числовых значений переменной цикла опять же возможна формализация в виде $j_{next} = j + const$.

http://sntbul.bmstu.ru/doc/640887.html

На пятом шаге выясним, остался ли верен инвариант после изменения переменной цикла. В нетривиальном случае это будет не так (инвариант будет нарушен). Тогда нужно создать инструкцию, которая восстанавливала бы инвариант. Для этого следует пытаться изменять переменные (кроме переменной цикла), которые входят в инвариант, до тех пор, пока не будет найдена оптимальная последовательность инструкций, восстанавливающих инвариант.

Используя третий и четвёртый шаги алгоритма нужно убедиться, что переменная цикла, изменяясь, приближается к заменённой константе. После этого в качестве охраны можно выбрать неравенство переменной цикла и заменённой константы (например, $j \neq N$).

Для рассматриваемого небольшого класса задача корректность обеспечивается нарушением охраны и одновременной истинностью инварианта. Как было отмечено в предыдущем пункте, в качестве охраны выбирается неравенство заменённой константы и переменной цикла. Следовательно, наоборот, их равенство обеспечит выполнение постусловия.

4 Выводы

Таким образом, были формализованы многие рассуждения, неявно проводимые программистом при проектировании алгоритма будущей программы. Эта формализация была выполнена для небольшого класса задач, она открывает возможность автоматизировать процесс построения программы. В некоторых случаях появляется возможность построить программу, имея сведения только о состоянии исполняющей среды, которое нужно установить.

Одновременно выполненная формализация открывает целый ряд новых вопросов. Можно заметить, что часто при автоматической генерации алгоритма был необходим перебор нескольких вариантов. В случае большого числа таких вариантов все выгоды от применения этой методики будут нивелированы высокой вычислительной сложностью. Следовательно, необходимы некоторые рецепты, позволяющие оптимизировать перебор и сократить число возможных вариантов.

Например, при выборе константы, которую необходимо заменить переменной, было предложено искать наиболее «простую» константу (скаляр в общем случае проще массива, а массив меньшей длины проще массива большей длины). Аналогичные оптимизации требуются и для других шагов, рассматриваемых в методике.

Наконец, можно сказать, что приёмы дедуктивной верификации являются достаточно фундаментальными и универсальными. Они могут найти применение не только в проверке правильности уже готовых программ, но и во многих других областях. В данной работе,

например, рассматривается использование приёмов дедуктивной верификации не для анализа алгоритма, а для обратного процесса – его синтеза.

Список литературы

1. Карпов Ю. Г. Верификация параллельных и распределённых программных систем. – СПб.: БХВ-Петербург, 2010. – 560 с.: ил. + CD-ROM.
2. Борисенко В. В. Основы программирования // INTUIT.ru: Национальный Открытый Университет «ИНТУИТ». 2005. URL: http://www.intuit.ru/department/se/pbmsu/ (дата обращения: 01.02.2013г.).
3. Виноградов И. М. Основы теории чисел. М.: Государственное издательство технико-теоретической литературы, 1952.
4. Грис Д. Наука программирования: Пер. с англ. - М.: Мир, 1984. - 416 с., ил.