УДК 51.681.3

А.Н. Максимец

ВЕРИФИКАЦИЯ ПРОГРАММ: СОСТОЯНИЕ, ПРОБЛЕМЫ, ЭКСПЕРИМЕНТАЛЬНЫЕ РЕЗУЛЬТАТЫ. I

Представлен углубленный обзор проблем верификации программного обеспечения. Рассмотрены методы верификации реактивных и функциональных систем. Даны основные определения для представления и анализа программ. Приведено краткое описание методов верхней и нижней аппроксимации.

Введение

Тематика данной статьи, как видно из названия, касается проблемы верификации программного обеспечения. Верификация программ – один из самых трудных (если не самый трудный) этапов разработки программного обеспечения. Трудность данного этапа состоит в том, что от разработчика, кроме знаний программистского характера, требуются знание и владение методами современной алгебры, логики, комбинаторики, теории чисел и других смежных областей. Кроме данных субъективных факторов имеются и объективные факторы, связанные с тем, что в настоящее время имеющиеся методы верификации не находятся на достаточном уровне развития, позволяющим верифицировать системы индустриальных размеров. Сложность программного обеспечения постоянно существенно возрастает, а методы его анализа отстают.

Когда речь идет о методах верификации и обоснования программного обеспечения, то имеются в виду прежде всего формальные методы. К данным методам в последнее время наблюдается повышенный интерес, поскольку опыт разработчиков программного обеспечения показывает, что из 5-ти, 6-ти инвестируемых проектов на рынок попадает один, максимум два программных продукта. Причем характерной особенностью данных продуктов является то, что они построены на хорошей теоретической основе с обоснованием каждого шага разработки. Последнее обстоятельство позволяет видеть перспективы дальнейшего развития данного программного продукта или всей системы в целом, направления этого развития, цели и задачи, решаемые с его помощью.

Программные системы на сегодняшний день делят на два класса: класс функциональных систем и класс реактивных систем. К первому классу относятся системы, которые должны работать в конечном времени и имеющие вход и выход (т. е. реализуют вычисление некоторой функции), а ко второму – системы, которые должны работать потенциально бесконечное время, реагируя в процессе своего выполнения на внешние и внутренние раздражители. К сожалению, методы верификации свойств первых систем совершенно не применимы к верификации свойств вторых систем, что привело к тому, что для обеих классов разрабатываются разные методы.

Исследования в области формальных методов верификации идут в основном в двух направлениях:

1) дедуктивная верификация или доказательство теорем [1–3];

2) алгоритмическая верификация с помощью разрешающих процедур, подобных процедурам проверки выполнимости логических формул на модели [4, 5], символического моделирования [6] или символическое выполнение траекторий [7].

Краткий обзор методов верификации

Методы верификации реактивных систем. Одним из наиболее важных и перспективных методов верификации реактивных систем является метод проверки на

модели. Суть этого метода состоит в том, что верифицируемая система представляется в виде математической модели, а ее ожидаемые свойства в виде формул подходящего формального логического языка. Процесс верификации сводится к проверке формул спецификации на модели системы. Если процесс проверки прошел успешно, то на данной модели ошибки в реальной системе на обнаружено (это не означает, что они в ней отсутствуют), в противном случае обнаруживается ошибка и, что очень важно, система верификации генерирует путь, ведущий к данной ошибке. Важным преимуществом этого метода является его автоматическое выполнение, что привело к тому, что многие этапы верификации для многих формальных логических языков превратились в стандартные процедуры и успешно используются в фирмах, которые разрабатывают не только программное обеспечение, но и микропроцессорную технику (например, AMD, IBM, Intel) [8]. Кроме того, данный метод применим к системам по размерам близким к индустриальным, благодаря манипуляциям с моделью, уменьшающих число ее состояний.

В качестве математической модели выбирается модель конечного автомата, работающего со словами бесконечной длины. Как правило это конечные автоматы Бюхи или обобщенные автоматы Бюхи (автоматы Мюлера). А в качестве формального логического языка выбираются языки линейной темпоральной логики [9] или более общие языки (язык ветвящегося времени CTL, или его обобщение CTL^* и т. п.). Выбор автоматной модели объясняется тем, что как сама модель, так и спецификация реальной системы представляются в виде конечного автомата Бюхи. Для языков, распознающих этими автоматами, разрешимы многие важные свойства, в частности, свойство включения одного языка в другой и свойство пустоты языка. А эти свойства позволяют эффективно выполнять верификацию специфицированных свойств на модели системы. Благодаря разрешимости проблем включения языков и пустоты языка для конечных автоматов над деревьями, метод верификации на модели расширяется на спецификации в языках логик ветвящегося времени (CTL, CTL^*). Кроме чисто автоматных методов с успехом используются и методы базирующиеся на свойствах частичных порядков, а также на символическом выполнении [7].

Описанию данной техники посвящены монографии [10, 11].

Второй широко используемой математической моделью являются сети Петри (СП) и их модификации: цветные, временные, гибридные СП. Эти модели наиболее адекватно представляют параллельные и распределенные системы. На тему СП имеется обширная литература, среди которой отметим наиболее известные монографии [11–13].

Методы верификации функциональных систем. Если верификация реактивных систем сводится к разрешимым свойствам теории конечных автоматов, то верификация функциональных программ сводится к проблеме поиска доказательства теорем в языках программных динамических логик или языке предикатов первого порядка. Одним из первых языков программных логик был язык, предложенный Хоаром [14] и названный в его честь логикой Хоара. Основу метода Хоара составляют методы поиска и генерации инвариантов программных циклов с последующим использованием дедуктивных методов доказательства утверждений (теорем) о свойствах программы. Этот метод является и на сегодняшний день основным при верификации программных систем функционального типа. Под проблемой (частичной) верификации систем функционального типа понимается проблема, состоящая в том, чтобы по заданным высказываниям φ и ψ о программе P доказать истинность высказывания ψ на значениях выходных переменных программы P при условии, что значения ее входных переменных удовлетворяю т высказыванию φ.

Решение этой проблемы достаточно сложно и требует тщательного анализа программы, однако оно может быть суще-

ственно облегчено, если для заданного состояния программы известны ее инварианты, т. е. такие утверждения, которые истинны всякий раз при прохождении процесса вычислений через это состояние. Использование инвариантов при верификации сводится к проблеме поиска инвариантных соотношений для заданной конструкции программы (скажем, инварианты цикла), а затем доказать, что из построенного множества соотношений следует нужный предикат (постусловие), наличие которого гарантировало бы правильность работы программы. За методами решения данной проблемы закрепилось название методов потокового анализа программ. Возникновение этих методов связано с практической деятельностью в программировании и прежде всего с созданием качественного и надежного программного обеспечения современных вычислительных систем.

Исторически первым методом потокового анализа считается интервальный метод, предложенный Коком, Шварцем [15] и развитый впоследствии Аллен [16] и другими авторами. Суть данного метода сводится к представлению управляющего графа программы множеством интервалов, при котором с каждым интервалом ассоциируется некоторая локальная информация. По множеству интервалов строится новый производный граф, который снова представляется своим множеством интервалов и т.д. до тех пор, пока он не стягивается к одной единственной вершине (в этом случае исходный граф называется сводимым). Информация, сопоставленная этой единственной вершине, представляет собой глобальную информацию о программе в целом. Затем эта информация распространяется на интервалы с помощью обратного процесса.

В начале 70-х годов определился новый подход к анализу потоков данных, предложенный независимо Летичевским [17] и Килдаллом [18]. Килдалл рассмотрел задачу поиска в программах инвариантных соотношений вида $r = 0$, где r – переменная, а 0 – константа. В итеративном методе Килдалла инварианты и их последовательные приближения рассматриваются в виде элементов полурешеток, а поиск инвариантов для простейших фрагментов программы – линейных участков – сводится к вычислению значения функции, называемой функцией поиска инвариантов. Эта функция по линейному участку и множеству соотношений на его входе строит некоторое множество соотношений на выходе данного линейного участка. Килдалл детально исследовал случай, когда функция поиска инвариантов является дистрибутивной относительно основной операции полурешетки, предложил алгоритм и алгебраическую трактовку поиска такого рода соотношений.

Вскоре после выхода в свет работы Килдалла, Кам и Ульман [19], критически рассмотрев подход Килдалла и его алгоритм, заметили что этот алгоритм дает полные множества соотношений только лишь в абсолютно свободных алгебрах, т. е. когда не учитываются никакие свойства операций алгебры данных программы. Было показано, что генерация полных систем соотношений вида $r = c$, где c – константа, невозможна, если заменить требование дистрибутивности требованием монотонности функции поиска инвариантов. Если не предполагать алгебру данных абсолютно свободной, то дистрибутивность функции поиска вариантов, вообще говоря, не имеет места и множества инвариантов, получаемые с помощью алгоритма Килдалла, не являются полными. Кам и Ульман детально исследовали свойства алгоритма Килдалла в случае монотонной функции, установили неразрешимость задачи построения полной системы инвариантов в общем случае и предложили некоторое усовершенствование алгоритма Килдалла. Также доказано, что как алгоритм Килдалла, так и предложенный ими алгоритм дает единственное, хотя и отличное от полного множество инвариантов.

В работах как Килдалла, так и Кама, Ульмана рассматривались задачи одностороннего анализа потока данных, т. е. задачи, в которых свойства состояния программы определяются исключительно свойствами его предшественников (прямая

задача потокового анализа) или, наоборот, только свойствами его наследников (обратная задача потокового анализа). В.Н. Касьянов [20] рассмотрел более общую задачу потокового анализа, в которой имеются зависимости свойств состояний и их наследников друг от друга.

Более общая постановка задачи поиска инвариантных соотношений сделана А.А. Летичевским [21], где предлагалось генерировать инвариантные соотношения типа равенств, неравенств, квазиравенств и квазинеравенств с учетом свойств алгебры данных, над которой работает рассматриваемая программа. Позже были разработаны методы верхней и нижней аппроксимации для генерации инвариантных соотношений, а также и соответствующие данным методам алгоритмы. Эти алгоритмы дают полные множества инвариантов типа равенств $t = t'$, где t, t' – выражения составленные из переменных, констант и операций алгебры данных, для программ над такими алгебрами данных как векторные пространства, свободные абелевы группы и коммутативные полугруппы, а также и для абсолютно свободных алгебр. Последняя из алгебр тесно связана с понятием логико-термальной эквивалентности схем программ над памятью. Детальное исследование методов поиска инвариантов для программ над абсолютно свободной алгеброй данных выполнено С.Л. Кривым [22] и В.К. Сабельфельдом [23]. В результате, на базе методов потокового анализа, построены полные системы преобразований для такого класса программ относительно логико-термальной эквивалентности, введенной В.Э. Иткиным [24].

Важный практический и теоретический шаг в развитии методов поиска инвариантов внесли Халбвош и Куазо [25], рассмотревшие задачу поиска инвариантов типа линейных равенств и неравенств. Они, используя методы линейной алгебры и линейного программирования, предложили алгоритмы генерации инвариантных соотношений такого типа. Эти алгоритмы генерировали хотя и неполные, но достаточно богатые множества инвариантов, чем существенно расширили сферу применения методов потокового анализа программ. Дальнейшее развитие данный подход получил в методе абстрактных интерпретаций программ с последующим применением методов математического программирования (в частности, линейного программирования) [26].

Дальнейшее развитие методы потокового анализа нашли при анализе программ, алгебры данных, являющихся коммутативными кольцами или полями. Разработка алгоритмов поиска инвариантов для таких программ дает возможность оптимизировать и частично верифицировать многие программы вычислительной математики. Основная трудность – это вычислительная сложность алгоритмов решения основных задач поиска инвариантов в программах над такими алгебрами.

Для того, чтобы получать более или менее практичные алгоритмы, необходимо ограничивать степень полиномов (или степень инвариантов). Эти ограничения, вообще говоря, хотя и существенные, но на практике вполне удовлетворительны, так как в большинстве случаев достаточно ограничиться инвариантами малых степеней. Так задача генерации инвариантов в виде полиномов частично была решена Мюллером и Сайдлом [27] для ограниченных степеней инвариантов.

Отметим одно интересное использование методов потокового анализа в сочетании с методами верификации программ путем символического выполнения [28]. Они использовались в индуктивном методе диалогового синтеза инвариантных утверждений в заданных точках программы, суть которого состоит в последовательном объединении и ослаблении предикатов, генерируемых при прохождении символического выполнения через эти точки.

В настоящее время в области развития методов потокового анализа имеется заметный прогресс, связанный с получением эффективных алгоритмов, обобщающих как алгоритмы Килдалла, Кама и Ульмана, так и другие [29].

Как уже отмечалось, важность развития методов потокового анализа программ заключается прежде всего в их практической ценности. На базе алгоритмов потокового анализа разработаны многие алгоритмы оптимизации. Среди них, в первую очередь, стоит отметить алгоритмы поиска и удаления избыточных выражений, алгоритмы оптимального распределения регистров, редукции программ и алгоритмов с учетом дополнительной информации (задача смешанных вычислений [30]), оптимизации процедур с учетом входа и выхода, введение дополнительных точек входа в процедуры и т. д.

Можно рассматривать также и обратную задачу: по имеющимся входному предикату (предусловию) и выходному предикату (постусловию) синтезировать программу, входные данные которой удовлетворяют предусловию, а выходные данные – постусловию. Такую постановку задачи и некоторые подходы к ее решению можно найти в книгах Абрамова С.А. [31] и Е. Дейкстры [32].

Необходимые понятия и определения

***U*-*Y*-схема программы над памятью.** Понятие схемы программы, рассматриваемое в этом разделе, представляет собой наиболее распространенную математическую модель программы, которая получается, если абстрагироваться от конкретной информационной среды, над которой она работает.

Пусть D – некоторое множество, на котором определены операции сигнатуры Ω, предикаты сигнатуры Π и в котором принимают свои значения переменные из конечного множества $R = \{r_1,\ldots,r_m\}$ (память). Пара (D, Ω) представляет собой универсальную Ω-алгебру, которая будет называться алгеброй данных, а частичное, вообще говоря, отображение вида $b : R \to D$ – состоянием памяти. Множество всех состояний памяти B будем называть информационной средой.

Обозначим $T(\Omega, R)$ – Ω-алгебру термов над R. Отображение b можно продолжить на все множество $T(\Omega, R)$, если положить

$$b(t(r_1,\ldots,r_m)) = t(b(r_1),\ldots,b(r_m)).$$

При этом, если $b(r_i)\ (i = 1,2,\ldots,m)$ неопределенно, то левая и правая части считаются неопределенными, а равенство двух выражений истинно тогда и только тогда, когда обе части принимают одинаковые значения или обе не определены.

Рассмотрим множество выражений вида $u(t_1,\ldots,t_n)$, где u – символ n-арного предиката из Π, а $t_1,\ldots,t_n \in T(\Omega, R)$. Каждое такое выражение определяет предикат на множестве B. Значение этого предиката при заданном состоянии памяти $b \in B$ определяется как $u(t_1,\ldots,t_n)(b) = u(b(t_1),\ldots,b(t_n))$. Пусть $U(R, \Omega, \Pi)$ означает множество всех таких выражений, $\overline{U}$ – множество пропозициональных булевых функций от выражений из $U(R, \Omega, \Pi)$. Элементы множества $U(R, \Omega, \Pi)$ будем называть базовыми условиями над памятью R, а элементы множества $\overline{U}$ – элементарными условиями над памятью R.

Оператором присваивания называется выражение вида:

$$y = (r := t(r)) = (r_1 := t_1(r),\ldots,r_m := t_m(r)),$$

где $r = (r_1,\ldots,r_m)$, а $t_1(r),\ldots,t_m(r) \in T(\Omega, R)$, $r_1,\ldots,r_m \in R$. Каждый оператор присваивания y задает некоторое преобразование на множестве B. Преобразование, выполняемое y на B, определяется равенством $y(b) = b'$, где

$$b' = y(b) = \begin{cases} b(t_i), & \text{если } r_i \in \{r_1,\ldots,r_m\} \\ b(r_i), & \text{если } r_i \notin \{r_1,\ldots,r_m\} \end{cases},$$

где $i = 1,2,\ldots,m$.

Если $g = y_1 \cdot y_2 \cdot \ldots \cdot y_n$ – произведение операторов присваивания, то ему соответствует преобразование y_g, равное последовательной суперпозиции преобразований, соответствующих операторам

$y_1, y_2, \ldots, y_n$. Пусть $Y(R,\Omega)$ означает множество всех операторов присваивания. Элементы данного множества будем называть базовыми операторами.

Пара $(U(R, \Omega, \Pi), Y(R, \Omega))$ называется стандартным базисом над памятью R, определенным сигнатурой (Ω, Π). Если $U \subseteq U(R, \Omega, \Pi)$, $Y \subseteq Y(R, \Omega)$, то стандартной U-Y-схемой программы над памятью R (или просто U-Y-схемой программы) называется множество состояний A схемы программы вместе с множеством переходов $S \subseteq A \times U \times F(Y) \times A$ и двумя выделенными состояниями: начальным a_0 и заключительным a^*, где $F(Y)$ означает множество всех слов в алфавите Y. Слова в алфавите Y, т. е. композиции базовых операторов, будем называть элементарными операторами.

Пусть (a,u,y,b), $(d,u_1,y_1,c) \in S$. Если $a = d$, а $b \neq c$ или $y \neq y_1$, то состояние a называют *ветвлением*, если же $b = c$, а $a \neq d$ или $y \neq y_1$, то состояние b называют *слиянием*.

Если $(a,u,y,b) \in S$, то говорят, что данный переход ведет из состояния a в состояние b. Состояние a_0 характеризуется тем, что в S нет ни одного перехода, ведущего в a_0. Путем $l(a,b)$, ведущим из состояния a в состояние b, длины n называется последовательность состояний $a_1 = a, a_2, \ldots, a_{n+1} = b$, связанных переходами $(a_i, u_i, y_i, a_{i+1}), i = 1,2,\ldots,n$. При этом путь из состояния a в состояние b называется линейным участком, если для всех $i = 1,2,...,n$ в множестве переходов S имеется единственный переход, ведущий из состояния a_i в a_{i+1}. Состояние a называют входом линейного участка, а состояние b – выходом.

Процесс выполнения U-Y-схемы программы при заданном начальном состоянии памяти $b_0 \in B$ – это последовательность пар вида $(a_1, b_1), (a_2, b_2), \ldots, \ldots, (a_n, b_n)$, где $a_1 = a_0$, $b_1 = b_0$, $a_i \in A$, $b_i \in B$, $i = 1,2,\ldots$ и для всякой пары (a_i, a_{i+1}) состояний U-Y-схемы программы в множестве S есть переход (a_i, u_i, y_i, a_{i+1}) такой, что $u_i(b_i) = 1$ и $b_{i+1} = y_i(b_i)$. Процесс выполнения U-Y-схемы программы называется терминальным, если он конечен и последняя пара имеет вид (a^*, b), где a^* –заключительное состояние.

U-Y-схема программы вместе с интерпретацией предикатов и заданным начальным состоянием памяти $b_0 \in B$ называется U-Y-программой.

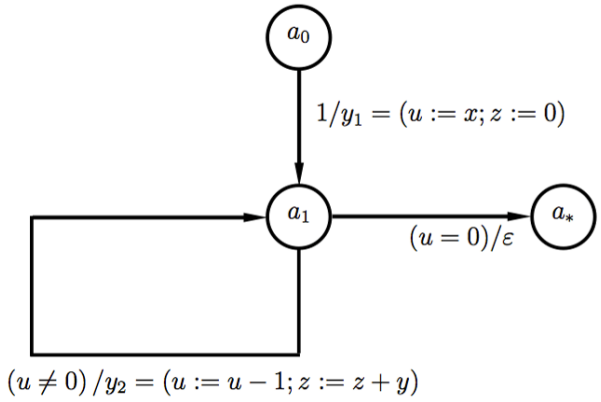
Для представления U-Y-схем программ в памяти вычислительной машины удобно пользоваться управляющим графом или регулярными выражениями алгебры алгоритмов. При представлении U-Y-схем программ над памятью управляющими графами с каждой U-Y-схемой программы связывается орграф $G = (A, E)$, называемый управляющим графом, дуги которого помечены следующим образом: если $(a,u,y,b) \in S$, то $(a,b) \in E$ и помечено парой u/y, а $Pr(a_0) = \varnothing$ и $Ps\ (a^*) = \varnothing$.

Помимо представления U-Y-схем с помощью управляющих графов существуют и другие способы представления.

Пример. U-Y-программа умножения целых положительных чисел x и y посредством сложения – УМН (x, y) с результатом z. Текстовое представление U-Y-программы УМН (x, y) имеет вид:

```
УМН(x, y)
u := x
z := 0
while u ≠ 0 do
        u := u − 1;
        z := z + y;
end while
```

А представление в виде управляющего графа U-Y-программы УМН(x, y) показано далее на рисунке.



Рисунок

Инварианты в состояниях U-Y-программ

Пусть A – U-Y-схема программы над памятью R, интерпретированная на области данных D (U-Y-программа), и L – язык, в котором записываются утверждения о свойствах информационной среды B. Относительно L будем предполагать, что всякое его предложение (называемое также условием) может быть выражено формулой $\Phi(r)$ языка исчисления предикатов первого порядка, содержащей свободные переменные из кортежа $r = (r_1,\ldots,r_m)$ и интерпретированной на области данных D. Сигнатура этого исчисления содержит все символы сигнатур Ω и Π. Пусть $u(r)$ – некоторое условие из L.

Определение. Условия $\Phi(r)$ из L называется инвариантом состояния a U-Y-программы A относительно условия $u(r)$, если оно истинно при каждом прохождении состояния a в процессе выполнения программы A для тех и только тех начальных состояний памяти из B, на которых истинно условие $u(r)$. Условие $u(r)$ называется начальным. Если оно тождественно истинное на D, то $\Phi(r)$ будем называть просто инвариантом состояния a.

Содержательно инвариант $\Phi(r)$ состояния a относительно условия $u(r)$ характеризует множество $B(a) \subseteq B$ состояний информационной среды, достигающих в состоянии a заданной U-Y-программы на допустимых путях, начинающихся в удовлетворяющих начальному условию $u(r)$ состояниях памяти. Схема построения множества $B(a)$ следующая. Строится множество S всех допустимых путей из a_0 в a, реализующихся для начальных состояний из $B_u \subseteq \hat{A}$ (характеристическое множество условия $u(r)$). По каждому пути

$$s = (a_0, b_0),\ldots,(a_k, b_k),(a_{k+1}, b_{k+1}),$$

где $k > 0, a_{k+1} = a^*$, восстанавливается его операторная история – последовательность $y_0,\ldots,y_k$ операторов, отмечающих переходы на данном пути. Рассматривая затем композицию $y_s = y_0 \cdot y_1 \cdot \ldots \cdot y_k$ (а это, как нетрудно видеть, также оператор присваивания над R) строим множество $B = y_s(B_u)$. После чего искомое множество $B(a)$ определяется как B_u. Эта схема, однако, неконструктивна, как в силу нерекурсивности множества S, так и в силу того, что множества вида B_s могут не являться характеристическими для условий языка L. Выходом из этого положения является как аппроксимация множества S некоторым объемлющим множеством путей S', которое хотя и содержит "лишние" (недопустимые) пути, но устроено проще, так и аппроксимация множеств вида B_s их подмножествами $B_{s'}$ – характеристическими для некоторых условий из L. Обе аппроксимации ведут к уменьшению множества $B(a)$, так как уменьшается каждый член пересечения $\bigcap_{s \in S} B_s$ и добавляются новые его члены. С учетом сужения множества $B(a)$ с неконструктивного $\bigcap_{s \in S} B_s$ до некоторого его приближения снизу можно говорить о максимальном инварианте состояния a как об условии $\Phi(r)$ языка L таком, что $B_u \subseteq B(\text{à})$ и для всякого другого инварианта $\Phi'(r)$ имеет место $\Phi(r) \Rightarrow \Phi'(r)$, где $\Rightarrow$ означает отношение "быть следствием".

Построение инвариантов осуще-

ствляется генератором инвариантов в языке L. Понятие генератора включает в себя три компоненты: функцию ef: $L\times U\times Y\to L$ – "эффект оператора", структуру полурешетки на множестве условий из L и описание итерационного алгоритма.

Функция ef по условиям u' и u из L, истинным перед выполнением оператора $y\in Y$, строит условие $ef(u',u,y)$, истинное на преобразованном состоянии памяти. Иногда будем рассматривать более простой вариант функции ef, когда в числе ее аргументов нет условия u (сужение $ef(u',u'y)$ до $ef(u,y)$), обуславливающего переход u/y. Заметим, что из определения функции ef вытекает, в частности, ее монотонность по первому аргументу, если множество условий N является логическим следствием множества N', то $ef(N,u,y)$ – логическое следствие $ef(N',u,y)$.

Операция $\wedge$ взятия нижней грани на множестве условий языка L, вообще говоря, отличается от конъюнкции условий. Если условиться называть множество условий, сопоставленных состояниям, разметкой, то итеративный алгоритм, отталкиваясь от начальной разметки состояний (когда отметка a_0 есть условие $u(r)$, а отметки остальных вершин – тождественно истинные условия), строит последовательность уточняющих разметок вплоть до получения некоторой стабильной разметки, которая и соответствует системе инвариантов.

Рассмотрим как осуществляется итерационный шаг. Итак, пусть задана текущая разметка состояний $U-Y$-программы, $Inv(a)\in L$ – текущая отметка состояния a и $z:V\to\{0,1\}$ – функция, позволяющая отделить состояния с обновленными отметками от остальных. Алгоритм завершает работу, если $(\forall a\in V)(z(a)=0)$. Если же это условие не выполняется, то выбирается a такое, что $z(a)=1$. Затем для каждого перехода из этого состояния (их может быть один или два) строим уточнения. Уточнение для перехода (a,u,y,a'), отмеченного парой u/y, состоит в том, что вычисляется эффект $ef(Inv(a),y)$, а затем нижняя грань $ef(Inv(a),y)\bigwedge Inv(a')$ и, наконец, если эта нижняя грань отлична от $Inv(a')$, то $z(a')$ полагается равным 1. Уточнение переходов, исходящих из состояния a, сопровождается изменением $z(a)$ с 1 на 0.

Описав итерационный шаг, общий для всех алгоритмов рассматриваемого класса, отметим, что в определенных ситуациях в состояниях-ветвлениях уточнение производится не по двум переходам, а лишь по одному. Для этого требуется, чтобы условие u или $\neg u$, определяющее выбор альтернативы в ветвлении, было следствием текущей отметки $Inv(a)$. Завершимость итерационного процесса обычно гарантируют условиями обрыва убывающих цепей в полурешетке языка L, что, может сказываться на качестве функции ef.

Если описанный алгоритм завершился, то функция Inv дает совокупность инвариантов в состояниях U-Y-программы. Эту совокупность будем также обозначать $\{N_a\}$, где $a\in A$.

Перейдем к более традиционной теоретико-множественной терминологии, принятой в теории программных инвариантов. Поскольку N и $ef(N,u,y)$ представляют некоторые предикаты на множестве D, то их можно рассматривать как отношения на D, определяемые этими предикатами. Тогда булеан $B(L)$ удобно представлять в виде решетки относительно теоретико-множественных операций пересечения и объединения, которая содержит нуль $\varnothing$ и единицу L. Выражения $ef(N,u,y)\bigcap(\bigcup)ef(N',u',y')$ в этом случае понимаются как пересечение (объединение) соответствующих отношений на D, а то, что множество $ef(N,u,y)$ является логическим следствием множества формул $ef(N',u,y)$ – как теоретико-множественное включение $ef(N,u,y)=ef(N',u,y)$. В дальнейшем будем следовать данным обозначениям.

Число различных возможных путей в программе (при наличии в ней хотя бы одного цикла) может быть бесконечным и тогда процесс построения условия состояния a тоже может стать бесконечным. Тем не менее, пусть $a_1,\ldots,a_k$ – все состояния U-Y-программы A, связанные переходами (a_i,u_i,y_i,a) с состоянием a и N_i – множество инвариантов состояния a_i относительно некоторого условия. Тогда, очевидно, что $\bigcap_{i=1}^{k} ef(N_i,u_i,y_i)$ будет инвариантом состояния a относительно того же начального условия. Этот простой факт служит отправной точкой для построения двух итеративных методов генерации инвариантов [21, 33].

В первом из них, называемом методом нижней аппроксимации (МНА), итеративный процесс задается рекуррентным соотношением

$$N_a^{(n)} = \bigcap_{(a',u,y,a)\in S} ef(N_{a'}^{(n-1)},u,y), \qquad (1)$$
$$n>0, \quad a,a'\in A,$$

а начальное приближение $\{N_a^{(0)}\}$ – равенствами $N_{a_0}^{(0)}=\{u\}$ и $N_a^{(0)}=\varnothing$ для $a\neq a_0$. Из свойства монотонности функции ef следует, что $N_a^{(0)}\subseteq N_a^{(1)}\subseteq\ldots$ независимо от состояния a. Указанный итеративный процесс может завершиться через конечное число шагов в результате стабилизации последовательностей $N_a^{(i)}$ для всех $a\in A$ или может продолжаться бесконечно, но достоинство этого метода, состоит в том, что, не дожидаясь стабилизации процесса вычислений, их можно прервать на любом шаге генерации, поскольку всякое множество $N_a^{(n)}$ включается в множество инвариантов состояния a.

В другом методе, называемом методом верхней аппроксимации (МВА), итеративный процесс задается рекуррентным соотношением

$$N_a^{(n)} = N_a^{(n-1)} \bigcap \bigcap \left(\bigcap_{(a',u,y,a)\in S} ef\left(N_{a'}^{(n-1)},u,y\right)\right), \qquad (2)$$
$$n>0, \quad a,a'\in A,$$

а начальное приближение определяется равенством $N_{a_0}^{(0)}=\{u\}$ и некоторой совокупностью простых путей, покрывающих все множество состояний U-Y-программы A. Вычисление начального приближения осуществляется вдоль этих путей, начиная с $N_{a_0}^{(0)}$: если для некоторого $a'\in A$ уже известно $N_{a'}^{(0)}$, переход (a,u,y,a') принадлежит одному из путей заданной системы, а $N_a^{(0)}$ еще не известно, то полагаем $N_a^{(0)}=ef(N_{a'}^{(0)},u,y)$. Из соотношения (2) видно, что для всякого $a\in A$ имеют место включения $N_a^{(0)}\supseteq N_a^{(1)}\supseteq\ldots$ и, следовательно, искомая совокупность инвариантов может быть получена только после стабилизации итеративного процесса. Поскольку процесс поиска инвариантов может быть бесконечным, это является недостатком метода МВА, который, однако, в случае результативного завершения генерирует более полные системы инвариантов чем метод МНА.

Исследование методов генерации инвариантов показывает, что конечность процесса поиска инвариантов тесно связана с языком условий L. Наиболее распространенными языками условий являются языки типа равенств и неравенств, так как практически любой язык программирования содержит предикаты равенства и неравенства. Разработка методов поиска инвариантов для этих языков сталкивается со значительными трудностями. Действительно, формулы (1) и (2) предполагают алгоритмическое и, по возможности, эффективное решение задач пересечения множеств соотношений, проверки соотношения на предмет того, что оно является следствием заданного множества соотношений и т. д. и т. п. Решение перечисленных задач уже для этих языков тре-

бует привлечения достаточно сложных методов современной общей алгебры.

Выводы

В данной работе, являющейся первой частью обзора методов верификации и их приложений, представлены краткие сведения истории развития и актуальных задач проблемы верификации программ. В них рассмотрена проблематика верификации реактивных и функциональных систем, где указано место предложенного подхода в формальных методах верификации. Приведены основные понятия, используемые в этой области, а также описываются свойства введенных понятий вместе с обоснованием их правильности. Намечаются дальнейшие пути развития предлагаемых методов.

В следующей статье будут более детально рассмотрены методы верификации и их реализация применительно к конкретным программам над конкретными алгебрами данных.

1. *Kaufmann M., Manolios P., Moore J.S.* Computer Aidede Reasoning: An Approach, // Kluwer Academic Publishers. – 2000. – 212 p.
2. *Nipkow T., Paulson L., Wenzel M.* Isabelle/HOL: A Proof Assistant for Higher Order Logic, // Springer Verlag. – 2002. – LNCS. – Vol. 2283. – P. 3–51.
3. *Oppen D.C., Cook S.A.* Proving Assertion About Programs that Manipulate Data Structures // In Proceed. Of the 7-th Annual ACM Symposium on Theory of Computing (STOC 1975). – Aluquerque. – NM. – ACM Press. – 1975. – P. 107–116.
4. *Clarke E.M., Emerson E.A.* Design and Synthesis of Synchronization Skeleton Using Branching Time Temporal Logic. In D.C. Kozen, edit // Logic of Program Workshop. – Springer Verlag. – 1981. – LNCS. – Vol. 131. – P. 52–71.
5. *Queille J.P., Sifakis J.* Proving Specification and Verification of Concurent Systems in CESAR // In Proceed. of the 5–th Intern. Symposium on Programming. – Springer Verlag. – 1982. – LNCS. – Vol. 137. – P. 373–351.
6. *Jones R. B.* Simbolic Simulation Method for Industrial Formal Verification // Kluwer Academic Publishers. – 2002. – 286 p.
7. *Chou C.* The Mathematical Foundation of Symbolic Trajectory Evaluation. In N. Halbwacha and D. Peled, edit // Proc. of the 11–th International Conference on Computer Aided Verification (CAV 1999). – Springer Verlag. – 1999. – LNCS. – Vol. 1633. – P. 196–207.
8. *Ray. S.* Scalable Techniques for Formal Verificaton // Springer Science+Business Media. – 2010. – LLC. – P. 9–57.
9. *Vardi M.Y., Wolper P.* An Automata–theoretic Approach to Linear Temporal Logic // Logics for Concurrency: Structure versus Automata Springer Verlag. – 1966. – LNCS. – Vol. 1043. – P. 238–266.
10. *Кларк Е.М., Грамберг О., Пелед Д.* Верификация моделей программ: Model Chcking. – М.: МЦНМО, 2000. – 416 с.
11. *Карпов Ю.Г.* MODEL CHECKING. Верификация паралелельных и распределенных программных систем. – Санкт–Петербург: БХВ–Петербург, 2010. – 560 с.
12. *Jensen K., Kristensen L.M.* Colored Petri Nets: Modelling and Validation of Concurent Systsems. Springer Verlag. – 2009. – P. 384.
13. *Penczek W., Polrola A.* Advanced in Verification of Time Petri Nets and Timed Automata. Springer Verlag. – 2006. – P. 256.
14. *Hoare C.A.R.* An axiomatic basis of computer programming // CACM. –1969. – Vol. 12. – P. 576–580.
15. *Cocke J., Schwartz J.T.* Programming Language and their Compilers // Courant Institute of Mathem. Sciences. – New York University. – 1970. – N.Y. – P. 3–21.
16. *Allen F.E.* Interprocedural analysis // ACM SIGPLAN Notices. – 1976. – Vol. 6, N 7. – P. 23 – 31.
17. *Летичевский А.А.* Эквивалентность и оптимизация программ // Труды междунар. симпозиума по теоретическому программированию. – Новосибирск, 1972. – С. 166–180.
18. *Kildall G.A.* A unified approach to program optimization // Conf. Rec. of ACM Symp. on Prince. of Program Languages, Boston, Massachusetts, Oktober 1–3, 1973. – P. 194–206.
19. *Kam J.B., Ullman D.J.* Monotone data flow analysis frame works // Acta Inform. – 1978. – Vol. 3. – P. 305–318.
20. *Касьянов В.Н.* Учет априорной информации при анализе свойств состояний программ // Математическая теория программирования. – Новосибирск: ВЦ СО АН СССР, 1985. – С. 150–157.

21. *Годлевский А.Б., Капитонова Ю.В., Кривой С.Л. и др.* Итеративные методы анализа программ. Равенства и неравенства // Кибернетика. – 1990. – № 3. – С. 1–9.
22. *Кривой С.Л.* О поиске инвариантных соотношений в программах. // Математическая теория проектирования вычислительных машин. – Киев: ИК АН УССР. – 1978. – С. 35–51.
23. *Сабельфельд В.К.* Учет свойств операций при глобальном анализе свойств программ // Математическая теория программирования. – Новосибирск: ВЦ СО АН СССР. – 1985. – С. 132–149.
24. *Иткин В.Э.* Логико–термальная эквивалентность схем программ // Кибернетика. – 1972. – № 1. – С. 5–27.
25. *Cousot P., Halbwochs N.* Discovery of Linear Restraints Among Variables of Program // Conf. Record of the 5–th Annual ACM Symposium on Princieples of Programming Languages. – USA. – 1978. – P. 84–96.
26. *Cousot P.* Abstract Interpretation Based Formal Methods and Future Challenges // Informatics LNCS. – Vol. 2000, 2001. – P. 138–156.
27. *Muller–Olm M., Seidl H.* Computing Interprocedurally Valid Relations in Affine Programs. // Symposium on Principles of Programming Languages, 2004.
28. *Костырко В.С., Бакулин А.В.* Об индуктивном синтезе инвариантных утверждений и функций программ // Кибернетика. – 1986. – № 1. – С. 18–24.
29. *Karr M.* Affine Relationships Among Variables of a Program // Acta Informatica. – 1976. – № 6. – P. 133–151.
30. *Годлевский А.Б., Кривой С.Л.* Применение техники смешанных вычислений к построению эффективных алгоритмов унификации и приведения выражений // Тезисы докл. Всесоюз. конф. "Методы трансляции и конструирования программ". – Новосибирск, 1988. – С. 59–62.
31. *Абрамов С.А.* Элементы анализа программ. – М.: Наука, 1988. – С. 129.
32. *Дейкстра Э.* Дисциплина программирования. – М.: Мир, 1975. – С. 247.
33. *Летичевский А.А.* Об одном подходе к анализу программ // Кибернетика. – 1979. – № 6. – С. 1–8.

Получено 07.03.2013

Об авторе:

Максимец Александр Николаевич,
аспирант.

Место работы автора:

Киевский национальный университет
имени Тараса Шевченко.
Тел: +38 050 9852274.
E-mail: maksymets@gmail.com