УДК 519.876.5

И. В. Рудаков, А. В. Ребриков

ПРОВЕРКА ВЫПОЛНЕНИЯ ФУНКЦИОНАЛЬНЫХ ТРЕБОВАНИЙ К АЛГОРИТМУ НА ОСНОВЕ СТРУКТУРНОЙ ГЕНЕРАЦИИ МОДУЛЬНЫХ ТЕСТОВ

Разработан метод верификации алгоритмов, на базе дедуктивного подхода к проверке выполнения заданных требований. Основой метода является математическая модель описания наблюдаемого поведения алгоритма — элемента рефлексивно-транзитивного замыкания путей в графе выполнения алгоритма. При создании метода предусмотрена возможность управления устранением цикломатической сложности за счет снижения точности метода и использования жадных алгоритмов поиска.

E-mail: irudakov@yandex.ru, rebrikov_a@mail.ru

Ключевые слова: *автоматизированная верификация, дедуктивный метод, структурная генерация тестов, наблюдаемое поведение алгоритмов.*

Применение дедуктивных методов [1] для верификации алгоритмов сопряжено с ограниченностью спектра разрешимых задач с помощью теории [2], заложенной в используемый SMT-решатель [2]. В то же время, путем расширения теории при дедуктивном подходе верифицируемый алгоритм можно задавать на языке его реализации, благодаря чему не требуется создание специализированных моделей, как в системах, основанных на верификации состояний системы [3]. При анализе функционирования сложных алгоритмов для полной, или формальной, верификации [3, 4] требуются большие временные затраты, поэтому с целью их сокращения необходимо иметь возможность проведения неполной верификации [5], посредством которой также можно проводить проверку систем с нецелочисленной логикой, что невозможно сделать при полной верификации.

Управляющим графом алгоритма называется кортеж [6]:

$$G = (N, E, n_0),$$

где N — множество вершин, каждая из которых соответствует оператору алгоритма; E — множество дуг, соотвествующих переходу управления; n_0 — стартовая вершина.

Множество N состоит из двух непересекающихся множеств N_s, N_p, где N_s — операторы, и у каждого $n_s \in N_s$ имеется не более одной вершины-потомка, т.е. $\exists! n_j \in N{:}(n_s, n_j) \in E$; N_p — условные операторы, и у каждого из них существует конечное число потомков. У стартовой вершины n_0 нет предков, и любая вершина графа достижима из нее.

Обозначим все множество переменных алгоритма как V, и множество функций, изменяющих значения переменных как $F{:}2\ D \to D$, где D — домены переменных V.

На определенном графе удобно ввести следующие функции разметки:

$func : N \times V \to \{F \cup \emptyset\}$: каждой паре вершин управляющего графа и изменяемых в них переменных соотвествует набор функций, изменяющих значение переменных. Если паре (n, v) соответствует пустое множество, то значение переменной v в вершине n не меняется. Без ограничения общности можно полагать, что отображение биективно, положив, что значение любой переменной может изменяться только один раз;

$use : N \to 2\,V$: каждой вершине управляющего графа соотвествует набор переменных, используемых в операторе;

$def \subset func : N \to 2\,V$: каждой вершине управляющего графа соответствует набор переменных, определенном в данном операторе. Без ограничения общности можно положить, что $func : def \to \{F \cup \emptyset\}$, т.е. определение переменной совпадает с ее первым и единственным изменением.

При выполнении оператора алгоритма изменяется состояние последнего. Фактически изменение наблюдаемого состояния при выполнении оператора описывает то, как алгоритм порождает наблюдаемое поведение в терминах описанной выше модели.

Уровнем абстракции наблюдаемого поведения алгоритма [7] будет называться отношение:

$$\sigma : N \to 2V \cup \{\phi\}.$$

Если $\sigma(n) = \phi$, то это означает, что оператор n является ненаблюдаемым. На множестве уровней абстракции можно ввести отношение частичного порядка и говорить, что $\sigma_1 \leq \sigma_2$, если для всех операторов алгоритма n верно, что $\sigma_2(n) = \phi \Rightarrow \sigma_1(n) = \phi \vee \sigma_1(n) \subset \sigma_2(n)$.

Для описания измения состояния алгоритма используется структура

$$s = (n, val),$$

где $n \in N$, $val : V \to D$ — значение переменной v, т.е. состояние описывается текущим выполняемым оператором и значениями переменных.

Деревом поведения алгоритма называется ациклический ориентированный граф

$$behaviour = (S, R, s_0),$$

где S — множество состояний алгоритма; R — отношение достижимости на множестве S; s_0 — выделенная корневая вершина (начальное состояние алгоритма).

Дерево поведения процесса может быть бесконечным, но из каждой его вершины исходит конечное число дуг.

Состояние $s = (n, val(V))$ будет наблюдаемым на заданном уровне абстракции σ, если:

а) $\sigma(n) \neq \phi$;

б) $\exists v \in V : v \in \sigma(n) \wedge func(n, v) \neq \emptyset$.

Тогда наблюдаемое поведение алгоритма на уровне абстракции σ есть ни что иное, как транзитивное расширение отношения достижимости R до множества наблюдаемых состояний.

Назовем два алгоритма эквивалентными на уровне абстракции σ, если их наблюдаемое поведение совпадает при одинаковых начальных состояниях.

Достаточный уровень абстракции σ' определяет ту часть алгоритма, которую необходимо выполнить для наблюдения операторов, заданных уровнем абстракции σ. Для выделения необходимых операторов используется механизм зависимостей.

Существуют два основных типа зависимостей в последовательной программе: зависимости по управлению, которые определяются управляющими конструкциями программы, и зависимости по данным, которые определяются переменными, используемыми в алгоритме.

Оператор $n \in N$ зависит по управлению от предиката c, который содержится в операторе условного ветвления $n_c \in N$, если в структуре потока управления алгоритма от выбора пути выполнения, на который потенциально влияет c, зависит, будет ли выполнен оператор n.

Оператор $n \in N$ зависит по данным от оператора n', если данные, определяемые в $n' \in N$, используются в n и потенциально могут достичь n через последовательность присваиваний переменных.

Для достаточного уровня абстракции σ' дадим следующее конструктивное определение. Если оператор n, входит в наблюдаемое поведение алгоритма, т.е. $\sigma(n) \neq \phi$, то множество операторов $S \subset N$, от которых n транзитивно зависит по управлению или данным, войдет в достаточный уровень абстракции вместе с оператором n и теми же наблюдаемыми переменными:

$$\sigma(n) = \phi \Rightarrow sigma'(n) = \phi;$$
$$\sigma(n) \neq \phi \Rightarrow \forall n' \in S : \sigma'(n') = \sigma(n).$$

Докажем, что $\sigma \leq \sigma'$.

Согласно определению частичного порядка на множестве уровней абстракции, надо доказать, что (1) $\forall n \in N \sigma(n) \subset \sigma'(n)$ и (2) $\sigma'(n) = \phi \Rightarrow \sigma(n) = \phi$.

(1) В силу определения достаточного уровня абстракции, если оператор n включается в уровень абстракции σ, то он включается с теми же наблюдаемыми переменными в σ', т.е. $\sigma(n) \subset \sigma'(n)$.

(2) Предположим обратное: $\exists n \in N : \sigma'(n) = \phi \wedge \sigma(n) \neq \phi$. Но это противоречит определению σ', ведь если $\sigma(n) \neq \phi$, то в σ' по крайней мере попадет n.

Из справедливости (1) и (2) согласно определнию частичного порядка следует, что $\sigma \leq \sigma'$, **ч.т.д**.

Отметим, что если два алгоритма G_1 и G_2 эквивалентны на уровне абстракции σ', то они эквивалентны на любом уровне абстракции $\sigma \leq \sigma'$. Действительно, уменьшение уровня абстракции (при прямом исключении рассматриваемых операторов либо косвенным путем за счет уменьшения числа рассматриваемых переменных) приводит к дальнейшему транзитивному расширению в дереве поведения. А раз на текущем уровне они совпадают, то и любые транзитивные операции на одинаковых множествах приведут к одинаковому результату.

Обозначим алгоритм, полученный из алгоритма G на уровне абстракции σ, как G_σ, построение которого аналогично построению наблюдаемого поведения: необходимо лишь провести операцию транзитивного расширения достижимости операторов до множества всех наблюдаемых операторов:

$$G_\sigma = (V_\sigma, E_\sigma, n_0\sigma);$$
$$\forall n \in N : n \in N_\sigma \Leftrightarrow \sigma(n) \neq \phi;$$
$$\forall (n_1, n_2) \in E : (n_1, n_2) \in E_\sigma \Leftrightarrow$$
$$\Leftrightarrow n_1 \in N_\sigma \wedge n_2 \in N_\sigma \wedge (\neg\exists n_3 : n_1 \to *n_3 \wedge n_3 \to *n_2);$$
$$n_0\sigma : \neg\exists n \in N_\sigma : n \to *n_0\sigma.$$

Алгоритмы G и G'_σ эквивалетны на уровне абстракции σ.

Необходимо отметить, что достаточный уровень абстракции является единственным и максимальным: $\forall\sigma\exists!\sigma' : G \Leftrightarrow_\sigma G'_\sigma \wedge \neg\exists\sigma_1 : \sigma' \leq \leq \sigma_1 \wedge \sigma' \neq \sigma_1$.

Понятие уровня абстракции можно расширить следующим образом:

$$\sigma : N \to 2^{P_V D},$$

где $P_V D$ — множество предикатов над переменными алгоритма и их доменами, задающих условия корректности алгоритма.

Достаточный уровень абстракции определяется аналогично: если в наблюдаемый предикат входит переменная $v \in V$, то во множество попадают все операторы, от которых n зависит по переменной v, и все операторы, от которых n зависит по управлению.

Основная идея метода заключается в когерентности работы алгоритма [8–11] его поведение изменяется лишь в операторах ветвления, поэтому логично проверять корректность работы алгоритма на таких наборах данных, при которых его поведение изменяется, а также в

точках, близких к этим наборам. Метод базируется на существующем методе граничных значений за тем исключением, что граничные значения берутся не из артефактов анализа предметной области, а из кода. Кроме того, к граничным значениям необходимо добавить значения переменных из условий корректности (которые, строго говоря, являются артефактами предыдущих этапов разработки).

Алгоритм структурной генерации тестов описывается следующим образом.

Входные данные: граф выполнения программы G, условия корректности σ.

Выходные данные: I — наборы входных параметров исходной программы.

Переменные: S_v — множество операторов, зависящих от операторов, изменяющих значение переменной v; S — множество решаемых систем уравнений; I — множество входных значений алгоритма. Тогда

1) $I = [], S = []$;

2) Получить все пути P из начальной точки алгоритма G в конечную;

3) $\forall p \in P$ получить систему: $s = INPUTGEN(p)$, $\forall n \in p \forall p_v \in \in \sigma(n) \forall v \in p_v : n \in S_v$ добавить в s отрицание предиката p_v, если его еще нет в s, $S \leftarrow s$;

4) $\forall s \in S$ получить, если возможно, решение: $i = SOLVE(s), I \leftarrow i$. Вернуть I.

Процедура INPUTGEN(p)

5) $s = [\,]$;

6) $\forall v \in p$ определить ее тип;

7) если v — выражение, то добавить его в систему уравнений s;

8) если v — условие, и исходящее ребро из него помечено, как истинная, то добавить его предикат в систему уравнений s;

9) если v — условие, и исходящее ребро из него помечено, как ложное, то получить отрицание предиката и добавить его в систему уравнений s;

10) вернуть s.

Ключевым недостатком метода структурной генерации является его неработоспособность при большом числе ветвлений алгоритма, циклов и рекурсий. Безусловно, метод обеспечивает полное покрытие кода и способен обнаруживать нарушение условий корректности, однако затрачиваемые ресурсы (как память, так и время) растут экспоненциально с увеличением числа ветвлений.

Кроме того, предложенный метод масштабирования алгоритмов по механизму зависимостей плохо оптимизирует вычислительные затраты, если контролируемый оператор находится далеко от начальной

вершины. В таком случае вероятность исключения из рассмотрения большого числа кодов заметно уменьшается.

Для устранения указанных недостатков понятие уровня абстракции необходимо дополнить свойством глубины анализа (которое легко расширяется на циклы и операторы вызова). Данное свойство задает максимальную глубину пути в поддереве поведения алгоритма, корень которого лежит в начале ветвления. При превышении этой глубины предикаты ветвлений, лежащих глубже заданного числа, перестают добавляться в решаемую систему уравнений.

При таком подходе возрастает число ошибок первого рода, т.е. пропускаемых ошибок, но этот недостаток частично устраняется путем проведения независимого тестирования участков кода, лежащих глубже заданного уровня. В свою очередь это приводит к росту числа ложных срабатываний, но запуск тестируемого алгоритма на сгенерированных данных исключает ошибки второго рода.

Формализацией глубины абстракции является цикломатическая сложность алгоритма G, определяемая следующим образом:

$$M = |E| - |N| + 2P,$$

где M — цикломатическая сложность кода; P — число компонент связности в графе G; $|E|$, $|N|$ — мощности множеств E, N.

Согласно общепринятому критерию [12], считается, что алгоритм тестируем, если $M \le M_{\max} = 50$. Тогда для обеспечения возможности тестирования необходимо ограничить число анализируемых путей в графе, используя механизм абстракций. Для построения эквивалентного алгоритма (который необходим для исключения ложных срабатываний, а также для минимизации P: после редукции алгоритма до $\sigma' P = 1$) требуется по-прежнему строить алгоритм $G_{\sigma'}$, однако при анализе путей для генерации границ классов эквивалентности пожно ограничиться меньшим уровнем абстракции:

$$\sigma \le \sigma_C \le \sigma',$$

где σ_C — уровень абстракции, обеспечивающий непревышение цикломатического числа $M_{\max}$.

Построение σ_C аналогично построению достаточного уровня абстракции с тем изменением, что операторы добавляются в σ_C только в том случае, если при этом не превышается цикломатическое число $M_{\max}$. Очевидно, что такая процедура построения не является детерминированной, потому положим, что уровень абстракции σ_C дает первый из всех возможных вариантов, обеспечивающих максимальную цикломатическую сложность алгоритма:

$$\sigma_C = \sigma_{Ci},\ i = \min j \forall j : M(G_{\sigma_{C_j}}) = \max M(G_{\sigma C_k}),\ k \in 1..K,$$

где $M(G)$ — цикломатическая сложность алгоритма G; K — число уровней абстракций, при которых цикломатическая сложность наблюдаемого алгоритма меньше $M_{\max}$.

Поскольку цикломатическое число есть верхняя оценка числа путей, то она является показателем эффективности тестирования. При этом при разбиении алгоритма на меньшие части суммарная цикломатическая сложность снижается за счет удаления ребер, ранее связывающих алгоритмы, что и обусловливает потерю точности при таком подходе.

С учетом всего сказанного выше метод структурной генерации описывается следующим образом:

Входные данные: граф выполнения программы G, условия корректности σ.

Выходные данные: наборы входных параметров исходной программы I.

Обозначение переменных: S_v — множество операторов, зависящих от операторов, которые изменяют значение переменной v; S — множество решаемых систем уравнений; I — множество входных значений алгоритма.

1) $I = [], S = [], P = []$.

2) построить алгоритм $G_{\{}\sigma'\}$;

3) получить алгоритм $G_0 = G_{\sigma_C}$ из алгоритма $G_{\sigma'}$ и все части алгоритма $G_{\sigma'}$ G_k, не попавшине в алгоритм G_0 из-за нарушения цикломатической сложности;

4) $\forall i = 0..K : pathes = PATHGEN(G), P \leftarrow pathes$;

5) $\forall p \in P$ получить систему: $s = INPUTGEN(p)$, $\forall n \in p \forall p_v \in \in \sigma(n) \forall v \in p_v : n \in S_v$ добавить в s отрицание предиката p_v, если его еще нет в s, $S \leftarrow s$;

6) $\forall s \in S$ получить, если возможно, решение: $i = SOLVE(s), I \leftarrow i$. Вернуть I.

Устранение цикломатической сложности возможно не только за счет недобавления операторов в уровень абстракции σ_C по причине нарушения цикломатической сложности, но и на основе иных условий. Эти методы можно классифицировать следующим образом [6, 12, 13].

Метод группового исключения. В итоговый уровень абстракции не включаются те переменные и операторы, исключение которых изменяет уровень цикломатической сложности менее заданного порога. Эффективен только в том случае, когда покрытие по данным не коррелиует с покрытием путей, в противном случае точность получаемого набора тестов мала.

Метод итеративного исключения. На каждой итерации отбора в уровень абстракции не включается та переменная, исключение кото-

рой изменяет уровень цикломатической сложности наименьшим образом. Более эффективен с точки зрения качества получаемых тестов, однако более затратный.

Метод комбинированного исключения. Совмещает в себе идеи методов группового и итеративного исключения. Если переменные и операторы слабо влияют друг на друга, данный метод исключает их, как метод группового исключения, в противном случае он исключает их в течение итеративного процесса. Слабость связи исчисляется уровнем связи в графе зависимостей алгоритма.

Жадный конструктивный метод. Пространство поиска представляется как направленный ациклический граф, корнем которого является пустое множество. В пространстве с n оптимизаций корень содержит n детей, представляющих множества размером в один элемент. Каждый последующий узел содержит $2n$ детей, которые представляют добавление отличающейся оптимизации к той, что находится в родительском узле. Конструктивный алгоритм спускается от корня, выбирая на каждом шаге узел с наилучшим значением порождаемого покрытия до тех пор, пока не сконструирует множество заданной длины.

Структурная генерация с масштабированием показала существенное улучшение покрытия кода при одновременном уменьшении общего числа тестов (рис. 1) и лучших временных характеристиках, чем у полной верификации.

На рис. 2 показано время генерации в зависимости от глубины абстракции, а также число обнаруженных ошибок. График числа обнаруженных ошибок позволяет определить примерный уровень насыщения

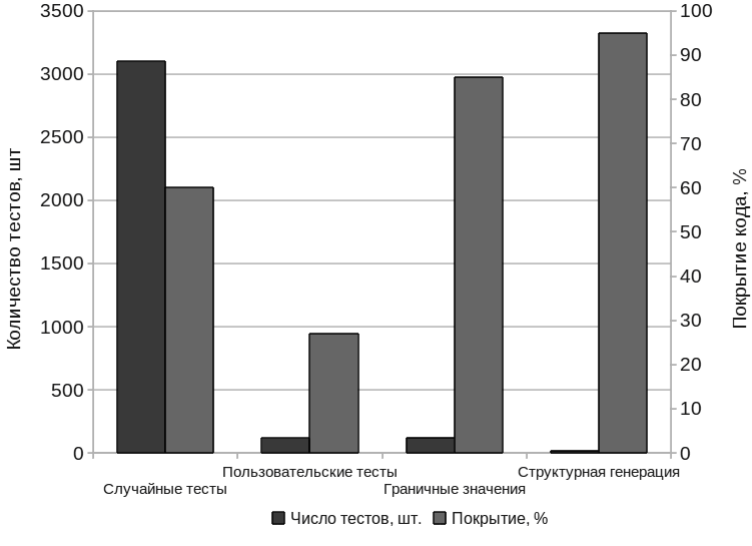

Рис. 1. Сравнение способов генерации тестов по покрытию кода

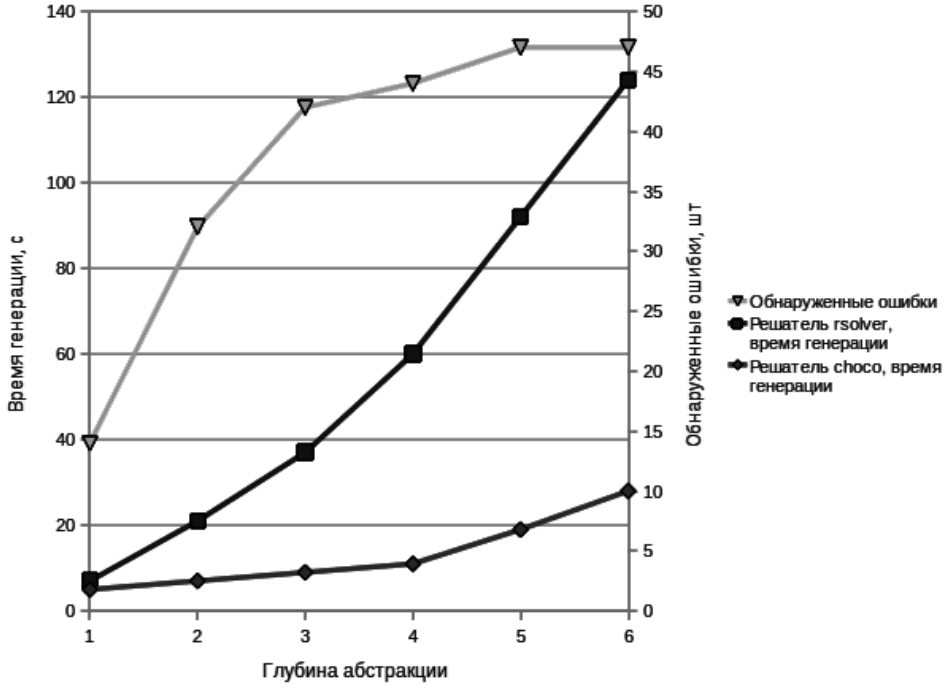

Рис. 2. Влияние глубины абстракции на время генерации и число найденных ошибок

при глубине абстракции 3, после которого не происходит повышения качества тестирования. Данный вывод подверждает устоявшийся цикломатический критерий сложности.

Эксперименты проводились на модулях с открытым исходным кодом репозитория CPAN [14]. Программный комплекс, реализующий данную схему для алгоритмов, которые записаны на языке Perl, зарегистрирован в Объединенном фонде алгоритмов и программ [16].

Ручное составление тестов является наименее эффективным (см. рис. 1) с точки зрения покрытия кода. При этом автоматическая генерация тестов обеспечивает почти полное покрытие (исключение составляют функции, содержащие циклы). В то же время (рис. 3), для ручного составления тестов требуется несравнимо больше времени, чем п составлении автоматическими методами.

Важной характеристикой метода генерации тестов является число тестов, после которого перестает возрастать степень покрытия кода:

$$\eta = \frac{n_s}{n},$$

где n_s — число тестов, после которого перестает возрастать (или незначительно возрастает) покрытие кода; n — общее число тестов. Тесты с номерами до n_s называются полезными тестами. Для случайных тестов этот показатель обычно низкий (в наших тестовых

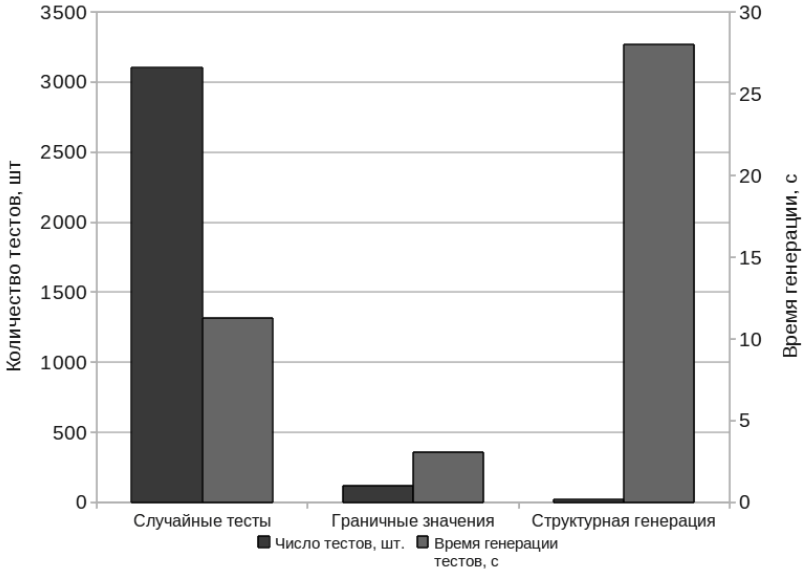

Рис. 3. Сравнение методов генерации тестов по времени затрат

примерах всего 0,051), для граничных значений он составляет 0,734, для тестов, генерируемых автоматически – 1 (что следует из структуры алгоритмов генерации на каждую непокрытую ветвь возвращается только один тестовый случай).

Качество теста можно охарактеризовать следующими величинами:

$$c_0 = \frac{c}{n} \to \max;$$
$$\eta \to \max;$$
$$t_0 = \frac{t}{n} \to \min;$$
$$e \to \max;$$
$$q_1 = \frac{\eta c_0}{t_0} = \frac{\eta c}{t} \to \max;$$
$$q_2 = \frac{\eta e}{t_0} = \frac{\eta e n}{t} \to \max,$$

где c — покрытие кода тестами данного метода генерации; n — число тестов, обеспечивших данное покрытие; η — относительный уровень насыщения; t — время, необходимое для генерации заданного числа тестов; e — число обнаруживаемых мутантов; q_1 — мультипликативный критерий качества, направленный на максимизацию, без учета обнаружения ошибок; q_2 — мультипликативный критерий качества, направленный на максимизацию, без учета покрытия. Введены два разных критерия q_1 и q_2, поскольку существует корреляция между параметрами e и c_0.

Таким образом, автоматическая генерация тестов, выигрывая практически по всем частным критериям, существенно отстает по временным характеристикам, в результате чего оказывается наихудшим из рассмотренных способов генерации тестов по критерию q_2 (табл. 1).

Таблица 1

Результаты сравнения методов генерации тестов

Вид генерации	с, %	t, с	η	e	q_1	q_2
Случайная	60	11,3	0,051	21	0,271	293,814
Граничные значения	85	3,1	0,734	36	20,126	1798,537
Структурная генерация	94	217,97	1	47	0,43	8,409
Общее значение	94	232,37	0,123	47	0,513	85,422

Помимо мультипликативного критерия качества для сравнения способов генерации тестов применяется мутационный критерий. Изначально все тестовые модели приводятся к состоянию, в котором используемые методы генерации входных параметров не обнаруживают ошибок в работе моделей. Затем в программы вносятся ошибки — мутации, для каждой одной мутации строится отдельная программа — мутант. Показателем качества является число обнаруживаемых мутантов (табл. 2).

Таблица 2

Сравнение методов генерации тестов (обнаружение ошибок)

Вид генерации	Ошибок выполнения	Нарушений условий корректности	Всего
Случайная	14	7	21
Граничные значения	25	11	36
Автоматическая генерация	33	14	47

В результате проделанной работы был разработан метод проверки алгоритмов, реализующий программный комплекс. Сравнение разработанных и существующих способов генерации данных показало улучшение времени проверки за счет уменьшения числа проводимых тестов, но при этом обеспечивающего полноту покрытия кода.

Проведенные исследования позволили:

1) установить глубину абстракции, после которой не происходит существенного повышения числа найденных ошибок. Полученный результат используется в методе как первая оценка при поиске достаточного уровня абстракции, что позволяет значительно сократить время структурной генерации;

2) установить значение числа ложных срабатываний — из 47 найденных ошибок 44 были подтверждены составленными вручную тестами;

3) показать, что с помощью разработанного метода можно генерировать тесты, выявляющие больше ошибок, и что данный метод покрывает больше кода, чем традиционные методы ручного и автоматизированного составления тестов. Кроме того, структурная генерация тестов применима к нецелочисленным алгоритмам, которые не могут быть проверены с помощью формальной верификации.

Эксперименты проводились на модулях с открытым исходным кодом репозитория CPAN [14]. Программный комплекс, реализующий данную схему для алгоритмов, коорые записаны на языке Perl, зарегистрирован в Объединенном фонде алгоритмов и программ [15, 16].

СПИСОК ЛИТЕРАТУРЫ

1. Кулямин В. В. Методы верификации программного обеспечения / Всероссийский конкурсный отбор обзорно-аналитических статей по приоритетному направлению “Информационно-телекоммуникационные системы”, 2008. – 117 с.
2. Чень Ч., Ли Р. Метод резолюций // Математическая логика и автоматическое доказательство теорем = Chin-Liang Chang; Richard Char-Tung Lee (1973). Symbolic Logic and Mechanical Theorem Proving. Academic Press. – М.: Наука, 1983. – 358 с.
3. Leue S., Wei W. Counterexample-based refinement for a boundedness test for CFSM languages, International Workshop on Model Checking Software (SPIN): Lecture Notes in Computer Science (LNCS) / Godefroid P. (ed.). – Vol. 3639. San Francisco, CA, USA. August 2005. – Springer-Verlag. – P. 58–74.
4. Symbolic model checking: $10^{2}0$ states and beyond / J Burch, E. Clarke, K. McMillan, D Dill, and L. Hwang // Information and Computation. – 1992. – Vol. 98, No. 2. – P. 142–170.
5. Рудаков И. В., Ребриков А. В. Неполная верификация сложных дискретных систем // Информационные технологии. – 2011. – № 3. – С. 31–34.
6. Podgurski A. and Clarke L. A. A Formal Model of Program Dependences and its Implications for Software Testing, Debugging, and Maintenance // IEEE Trans. Softw. Eng. – 16, 9 (Sep. 1990). – P. 965–979.
7. Савенков К. О. Масштабирование дискретно-событийных имитационных моделей: дис. . . . канд. физ.-мат. наук: 05.13.133. МГУ, 2007.
8. Gupta N., Mathur A. P. and Soffa M. L. Generating Test Data for Branch Coverage // Proc. of the 15th IEEE Int. Conf. on Automated Software. – 2000. – P. 219–227.
9. Рудаков И.В., Ребриков А. В. Масштабирование алгоритмов для автоматической генерации модульных тестов // Вестник МГТУ им. Н.Э. Баумана. Сер. Приборостроение. – 2011. – № 4. – С. 119–126.
10. Ребриков А. В. Редукция операторов алгоритма для оптимизации времени структурной генерации модульных тестов / Материалы четырнадцатого научно-практич. семинара “Новые информационные технологии в автоматизированных системах”. – М.: Моск. гос. ин-т электроники и математики, 2011. – С. 28–34.

11. Рудаков И. В., Ребриков А. В. Неполная верификация систем, представленных в виде вероятностных автоматов с нечеткой функцией переходов // Информатика и системы управления в XXI веке: Сб. тр. молодых ученых, аспирантов и студентов МГТУ им. Н.Э. Баумана. – 2010. – С. 76–78.
12. Harrison. Applying Mccabe's complexity measure to multiple-exit programs: Software: Practice and Experience. October 1984. Vol. 14, Issue 10. – P. 1004–1007.
13. Брусенцов Л. Автоматическая оптимизация при компиляции [Электрон. ресурс]. [Режим доступа свободный] http://www.osp.ru/os/2011/02/13007711/
14. Comprehensive Perl Archive Network [Электрон. ресурс] http://cpan.org/ [Режим доступа свободный].
15. Comprehensive Perl Archive Network, pgriffin's author page [Электрон. ресурс] http://search.cpan.org/∼pgriffin/ [Режим доступа свободный].
16. Ребриков А. В. Автоматический генератор тестов. 2010. 02076881.00425-01.

Статья поступила в редакцию 10.05.2012