УДК 681.51
doi: 10.18101/2304-5728-2017-2-33-39

ГЕНЕРАЦИЯ ТЕСТОВ В СИСТЕМЕ SynGT

© Федорченко Людмила Николаевна
кандидат технических наук, доцент, старший научный сотрудник,
Санкт-Петербургский институт информатики и автоматизации
Российской академии наук (СПИИРАН)
Россия, 199178, Санкт-Петербург, 14 линия В.О., 39
E-mail: LNF@iias.spb.su

В статье представлен алгоритм автоматической генерации тестовых данных при построении синтаксических анализаторов, реализуемый в системе преобразований грамматик SynGT. Используется подход, основанный на генерации прототестов из синтаксической граф-схемы, графического аналога контекстно-свободной грамматики в регулярной форме (КСР-грамматики). В терминах вершин и дуг граф-схемы, нагруженных соответствующей семантической информацией, формулируется критерий тестового покрытия, в соответствии с которым генерируется набор прототестов.
Ключевые слова: синтаксическая граф-схема; КСР-грамматика; прототесты.

Введение

Разработчики программных продуктов не могут гарантировать отсутствие в них ошибок, потому что не в состоянии в этом убедиться. Ошибки программирования систем продолжают оставаться одним из основных неприятных фактов. Цена этих ошибок неимоверно высока, как по своим, иногда катастрофическим последствиям, так и по стоимости их обнаружения и устранения. По данным СМИ, ошибки в программном обеспечении обходятся экономике США в более чем 60 млрд дол. в год. Чтобы оценить убытки в мировом масштабе, достаточно утроить это число, и мы получим сумму, превышающую стоимость ежегодного валового национального продукта многих стран. Анализ стоимости разработки и эксплуатации больших программных систем показывает, что 50% общих расходов делается после того, как она достигает статуса Бета-системы, львиная доля оставшихся расходов затрачивается на модификацию программ и на то, чтобы убедиться в их правильности. Всё это говорит о недостаточно высоком уровне надежности производства программных систем.

Одним из путей изменения ситуации к лучшему является создание подходящих инструментальных средств и широкое использование формальных методов, что позволит кардинально преобразовать практику программной инженерии, двигаясь в направлении создания верифицированного программного обеспечения. В 2005 г. в феврале в Калифорнии был проведен симпозиум по верифицированному программному обеспечению, а в октябре того же года в Цюрихе состоялась конференция IFIP

«Верифицированное программное обеспечение: теории, инструменты и эксперименты», на которых была принята исследовательская программа, рассчитанная на 20–25 лет и направленная на формальную проверку в соответствии со стандартами строгости и точности большого объёма программного кода, используемого в системах критической безопасности. Эту программу инициировал известный британский ученый Энтони Хоар (Charles Antony Richard Hoare), специалист в области информатики и создатель логики Хоара (Hoare Logic), применяемой при конструировании корректных программ. Он обратил внимание на необходимость разработки автоматизированного набора инструментов для проверки корректности программного обеспечения, и в том числе того, которое наиболее интенсивно эксплуатируется – трансляторов для языков программирования и языковых процессоров для конструирования встраиваемого программного кода.

Теории разработки программного обеспечения, широко изучавшиеся как формальные методы программной инженерии, опираются на фундаментальные результаты в теории вычислений, анализе алгоритмов и языках программирования. Основополагающими работами в этой области считаются работы Н. Хомского и А. М. Тьюринга, заложившие основы теории языков и автоматов, ставших прообразами современных компиляторов языков программирования.

Тестирование программных систем является важным компонентом всех проектов, связанных с разработкой общесистемного программного обеспечения и необходимым этапом при создании репозитория верифицированных программ.

По мере роста сложности программных систем растет трудоемкость тестирования. Решить задачу повышения качества и сокращения расходов на тестирование можно за счет привлечения эффективных средств автоматизации разработки тестов по спецификациям.

Синтаксические анализаторы как основная часть компиляторов являются необходимым инструментом при создании общесистемного программного обеспечения как наиболее интенсивно эксплуатируемого, поэтому их надежность особенно важна.

В статье представлен алгоритм автоматической генерации тестовых данных при построении синтаксических анализаторов, реализуемый в системе преобразований грамматик SynGT [2, 3]. Используется подход, основанный на генерации прототестов из синтаксической граф-схемы, графического аналога контекстно-свободной грамматики в регулярной форме (КСР-грамматики) [4].

1. Тестирование на основе спецификаций реализуемого языка

Основные идеи данного подхода заключаются в следующем:

1. реализуемый язык программирования представляется *синтаксической граф-схемой* [2], которая определяет множество программ языка;

2. в терминах вершин и дуг, нагруженных соответствующей семантической информацией, формулируется критерий тестового покрытия;

3. в соответствии с выбранным критерием генерируется набор прототестов.

Программа генерации прототестов — это компонента системы эквивалентных преобразований грамматик SynGT, которая выполняет следующие задачи:

- обходит синтаксическую граф-схему грамматики, пройдя хотя бы единожды по всем дугам графов для нетерминалов [2] исходной грамматики;
- последовательно накапливает информацию, содержащуюся на дугах по всем возможным маршрутам;
- создает набор прототестов языка.

Прототест — это конечная совокупность синтаксических форм для всех конструкций входного языка, задаваемого контекстно-свободной грамматикой в регулярной форме (КСР-грамматикой). Эти формы являются лишь своего рода «заготовками» тестовых вариантов реализуемого языка, так как могут содержать нетерминалы. Чтобы превратить их в тестовые варианты, надо заменить каждое вхождение нетерминала на его терминальное порождение, взятое из той же совокупности синтаксических форм. Такого рода преобразования выполняет редактор тестов.

Синтаксическая граф-схема, представляющая КСР-грамматику языка, является совокупностью графов для правил КСР-грамматики, правые части которой — регулярные выражения над объединенным алфавитом КСР-грамматики. Каждому правилу соответствует один граф с одной входной и одной выходной вершинами. Поскольку регулярные выражения содержат бинарные операции объединения, конкатенации и итерации, соответствующие им графы содержат циклы.

2. Алгоритм генерации прототестов

Вход. Описание исходной грамматики, заданное в специальной форме [4].

Выход. Текстовый файл списка прототестов, по которым затем генерируется матрица тестов.

На основе входных данных о грамматике строится её синтаксическая граф-схема (многокомпонентный граф), а затем специальный промежуточный граф (точнее, множество графов). Вершинами этого промежуточного графа являются точки ветвления (а также начальная и конечная вершины графа для каждого нетерминала граф-схемы, которые не участвуют в поиске), а дугами — пути, соединяющие точки ветвления (дуга может входить в ту же вершину, из которой вышла). Из каждой вершины графа выходит ровно 2 дуги (одна из которых может быть циклической), которые мы будем называть «левая» и «правая».

Для каждой дуги графа вводится счетчик прохождений через эту дугу. При прохождении по дуге счетчик увеличивается на единицу. (Проход по

циклической дуге увеличивает счетчик на число, лежащее в диапазоне больших чисел. Это гарантия того, что циклическая дуга будет пройдена только один раз).

Для каждой вершины графа ищем расстояние от начальной вершины и кратчайший путь. Первый путь выбирается следующим образом.

Идем по левой дуге, за исключением следующих случаев:

- правая дуга еще не проходилась и ведет в ту же вершину (является циклической);
- количество проходов по левой дуге больше, чем количество проходов по правой дуге.

Напомним, что проход по циклической дуге увеличивает счетчик на большое число. Это гарантирует, что циклическая дуга будет пройдена один раз.

Затем в графе выбирается вершина с минимальным расстоянием от начала, по крайней мере, одна из дуг которой не пройдена, и затем строится путь из нее, используя тот же алгоритм, что и для первого пути. При следующем проходе по графу счетчики количества прохождений не сбрасываются в 0. В качестве начала пути используется минимальный путь от начала графа.

Затем процесс повторяется снова и снова, пока все дуги не будут пройдены.

3. Оценка сложности

Если обозначить общую длину всех путей через L, то

$$L \le \frac{n*(n+a*n+b)}{2},$$

где n — количество вершин (исключая начальную и конечную вершины), a,b — константы.

Оценка не может быть улучшена (независимо от алгоритма).

После этого строится последовательность прототестов. Каждой дуге соответствует последовательность термов так, что данное преобразование не представляет проблем.

В качестве примера возьмем фрагмент грамматики языка Си [1] для конструкции «оператор» <statement> и преобразуем его с помощью SynGT[2] в эквивалентный фрагмент — КСР-правило для нетерминала <statement>.

4. Пример

```
statement : *( $label '<label>' ':' ;  $case1 'case' expression ':'
   $case2  ; $default 'default' ':' )
    (      compound ;
        'if' $cond '(' expression $then ')' statement
               [ $else 'else' statement ] $if ;
        'switch' $sw1 '(' expression $sw2 ')' statement $sw3 ;
        'while' $cond '(' expression $wl1 ')' statement $wl2 ;
   $do1 'do' statement 'while' $cond '(' expression ')' $do2 ;
        'for' $for1 '(' [ expression ] $for2 ';' [ expression ]
         $for3 ';' [ expression ] $for4 ')' statement $for5 ;
( 'goto' $goto '<tag>' ; $cont 'continue' ; $brk 'break' ;
  $ret1 'return' [ expression ] $ret2  ;
        expression $expr ; $null ) ';'
   ) .
```

Рис. 1. КСР-правило для нетерминала statement

Для нетерминала statement строим граф в соответствии с определением [2] для построения графа синтаксической граф-схемы.

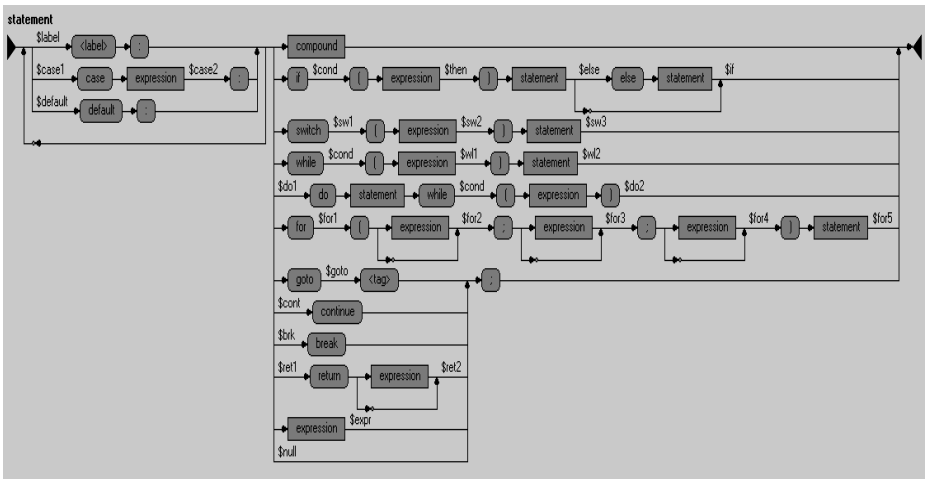

Рис. 2. Граф нетерминала statement

Программа генерации прототестов на выходе выдаёт текстовый файл списка прототестов следующего вида:

1. <statement>-1: <label> : while (<expression>) <statement>
2. <statement>-2: do <statement> while (<expression>)
3. <statement>-3: for (; ;) <statement>
4. <statement>-4: goto <tag> ;
5. <statement>-5: continue ;
6. <statement>-6: break ;
7. <statement>-7: return ;
8. <statement>-8: <expression> ;
9. <statement>-9: ;
10. <statement>-10: case <expression> : <statement> { <declaration> }
11. <statement>-11: case <expression> : <statement> if (<expression>) <statement>
12. <statement>-12: case <expression> : <statement> switch (<expression> <statement>
13. <statement>-13: case <expression> : <statement> while (<expression>) <statement>
14. <statement>-14: case <expression> : <statement> do <statement> while (<expression>)
15.<statement>-15: case <expression> : <statement> for (; ;) <statement>
16.<statement>-16: case <expression> : <statement> goto <tag> ;
17 <statement>-17: case <expression> : <statement> continue ;
18.<statement>-18: case <expression> : <statement> break ;
19.<statement>-19: case <expression> : <statement> return ;
20.<statement>-20: case <expression> : <statement> <expression> ;
21.<statement>-21: case <expression> : <statement> ;
22.<statement>-22: default : <statement> { <statement> }
23.<statement>-23: default : <statement> if (<expression>) <statement>
24.<statement>-24: default : <statement> switch (<expression>) <statement>
25.<statement>-25: default : <statement> while (<expression>) <statement>
26.<statement>-26: default : <statement> do <statement> while (<expression>)
27.<statement>-27: default : <statement> for (; ;) <statement>
28.<statement>-28: default : <statement> goto <tag> ;
29.<statement>-29: default : <statement> continue ;
30.<statement>-30: default : <statement> break ;
31.<statement>-31: default : <statement> return ;
32.<statement>-32: default : <statement> <expression> ;
33.<statement>-33: default : <statement> ;
34.<statement>-34: { }
35.<statement>-35: if (<expression>) <statement>
36.<statement>-36: switch (<expression>) <statement>
37.<statement>-37: while (<expression>) <statement>
38.<statement>-38: <label> : do <statement> while (<expression>)
39.<statement>-39: <label> : for (<expression> ; <expression> ; <expression) <statement>
40.<statement>-40: <label> : goto <tag> ;
41.<statement>-41: <label> : continue ;
42.<statement>-42: <label> : break ;
43.<statement>-43: <label> : return <expression> ;
44.<statement>-44: <label> : <expression> ;
45.<statement>-45: <label> : ;
46.<statement>-46: <label> : switch (<expression>) <statement>
47.<statement>-47: <label> : if (<expression>) <statement> else <statement>
48.<statement>-48: <label> : { <declaration> <declaration> <statement> <statement> }
49.<statement>-49: <label> : <label> : do <statement> while (<expression>)
50.<statement>-50: <label> : <label> : for (<expression> ; <expression> ; <expression>) <statement>
51.<statement>-51: <label> : <label> : goto <tag> ;
52.<statement>-52: <label> : <label> : continue ;
53.<statement>-53: <label> : <label> : break ;
54.<statement>-54: <label> : <label> : return <expression> ;
55.<statement>-55: <label> : <label> : <expression> ;
56.<statement>-56: <label> : <label> : ;
57.<statement>-57: <label> : <label> : switch (<expression>) <statement>
58.<statement>-58: <label> : <label> : if (<expression>) <statement> else <statement>
59 <statement>-59: <label> : <label> : { <declaration> <declaration> <statement>

Рис. 3. Список прототестов

Заключение

Предложенная техника автоматической генерации тестовых данных может быть востребована в первую очередь, в таких областях разработки программного обеспечения, как телекоммуникационные приложения, в частности, интернет-приложения; приложения, работающие над базами данных и при создании языковых процессоров.

Литература

1. B. W. Kernigan, D. M. Ritchie. The C Programming Language. Second Edition. Prentice Hall, 1988.

2. Fedorchenko L. Regularization of Context-Free Grammars. LAP LAMBERT Academic Publishing, Saarbrucken. 2011.

3. Федорченко Л. Н. Извлечение крайней рекурсии из КСР-грамматики в системе SynGT // Труды СПИИРАН. 2002. Вып.1, т. 1. С. 350–359.

4. Open C Compiler. SYNTAX ANALYZER. GRAMMAR REQUIREMENTS SPECIFICATIONS. File GR_SRS.txt Совместный СПИИРАН и INRIA проект– 1995.

GENERATION OF TESTS IN THE SYNGT SYSTEM

Ludmila N. Fedorchenko
Cand. Sci. (Engineering), A/Prof., Senior Researcher,
St Petersburg Institute for Informatics and Automation,
Russian Academy of Sciences (SPIIRAS),
39, 14-th V.O. Line, St Petersburg 199178, Russia
E-mail: lnf@iias.spb.su

The article presents an algorithm for automatic generation of test data in the construction of parsers, which is implemented in the system SynGT for conversion of grammars. We have used an approach based on generation of prototests from the syntax graph-scheme — a graphic analogue of context-free grammar in regular form (CFR-grammar). In terms of vertices and arcs of the syntactic graph-scheme loaded with the appropriate semantic information, a test coverage criterion have been formulated, according to which a set of prototypes is generated.
Keywords: syntax graph-scheme; CFR-grammar; prototests.