Генератор сложных данных Pinery: реализация новых возможностей UniTESK

А. В. Демаков, С. В. Зеленов, С. А. Зеленова
{demakov, zelenov, sophia}@ispras.ru

Аннотация. В статье рассказывается о подходе к автоматической генерации тестовых данных сложной структуры, основанном на использовании формального описания данных в виде грамматик. Подход реализован в генераторе Pinery, который является базой для создания переиспользуемых библиотек абстрактных моделей и итераторов для генерации специфических данных. В работе описывается метод разработки подобных расширений, позволяющий в результате получать небольшие множества тестовых данных, нацеленных на тестирование заданных аспектов функциональности программного обеспечения.

1. Введение

Тестирование программного обеспечения (ПО) с целью контроля его качества является одним из важнейших этапов разработки ПО. При тестировании ПО, обрабатывающего данные сложной структуры — к такого рода данным относятся, например, другие программы, сообщения телекоммуникационных протоколов, XML-документы, содержимое баз данных и пр. — построение представительного множества тестов становится чрезвычайно трудоемкой задачей. Наиболее привлекательным здесь является использование тех или иных автоматических генераторов тестовых данных.

Разработанная в ИСП РАН технология автоматизированного тестирования UniTESK [2,7,30] нацелена на тестирование функциональности целевого ПО и основана на использовании формальных моделей ПО. Тестовые воздействия в UniTESK строятся путем перебора вызовов интерфейсных операций целевого ПО и итерации наборов аргументов для каждой операции. Инструменты, поддерживающие UniTESK, предоставляют разработчикам тестов библиотеки базовых итераторов значений простых типов, которые могут быть непосредственно использованы для генерации тестовых воздействий, а могут быть скомпонованы в более сложные генераторы.

Технология UniTESK оказалась удобной для тестирования в том числе и программных систем, обрабатывающих данные сложной структуры:

реализаций телекоммуникационных протоколов [6], моделей аппаратного обеспечения [5]. Однако, для итерации аргументов операций в этих проектах библиотечные итераторы UniTESK оказались недостаточными, и использовались более подходящие сторонние генераторы. При этом каждый раз приходилось решать рутинные технические задачи по адаптации генераторов тестовых данных к используемым инструментам UniTESK.

В настоящей статье мы описываем унифицированный подход к итерации тестовых данных сложной структуры в UniTESK. Подход основан на полученном в ИСП РАН опыте разработки тестовых данных для различных областей ПО и опыте разработки автоматических генераторов тестовых данных на основе грамматик и языковых моделей [1,4,8,20,22,33,34]. Подход реализован в генераторе тестовых данных Pinery, который может использоваться как в качестве итератора в инструментах UniTESK, так и в виде самостоятельного инструмента генерации.

2. Близкие работы

В настоящее время имеется ряд инструментов для автоматической генерации разного рода тестовых данных сложной структуры: для тестирования приложений над базами данных [10,19], для тестирования обработчиков XML-документов [31,32] и др. Существенным недостатком этих инструментов является то, что в них предоставляется слишком бедный набор возможностей для настройки процесса генерации. В результате для достижения приемлемого качества тестирования, приходится генерировать огромные множества тестовых данных, что ведет к чрезмерным затратам ресурсов как на этапе генерации, так и на этапе запуска тестов.

В подходах, представленных в работах [11,14,28], используются грамматики в виде расширенной BNF, снабженные специальными фрагментами программного кода, которые содержат информацию семантического характера: разного рода вспомогательные вычисления и проверки условий. Такое представление семантики в виде произвольного программного кода, разбросанного во многих разных местах описания, приводит к высокой трудоемкости сопровождения входных данных генератора.

В статье [24] описан основанный на грамматиках генератор DGL, который в первую очередь предназначен для генерации тестов на случайной основе. Генерация регулируется путем присвоения продукциям некоторых значений весов. Кроме того, имеется возможность вводить вспомогательные вычисления, а также устраивать систематическую итерацию атрибутов и альтернатив.

Все перечисленные выше подходы нацелены на то, чтобы сгенерировать какие-нибудь тестовые данные, удовлетворяющие предоставленному описанию. Однако в них не рассматриваются какие бы то ни было критерии полноты получаемых наборов тестовых данных.

В статьях [16,17,18] описываются подходы к автоматической генерации тестов для семантических анализаторов, основанные на спецификации семантики в виде атрибутных грамматик [26]. Другой подход [20,21] к этой задаче использует спецификации в виде ASM [15]. Авторы этих подходов рассматривают различные критерии тестового покрытия для семантических анализаторов и описывают соответствующие автоматические генераторы тестов. В общих чертах эти генераторы работают так: сначала генератор создает множество синтаксически корректных предложений, которые затем проверяются на семантическую корректность с помощью интерпретатора атрибутной грамматики или ASM, и все семантически некорректные предложения отбрасываются. Такие генераторы оказываются чрезвычайно ресурсоемкими: как показывает практика, для генерации 3 000 семантически корректных тестов требуется предварительно сгенерировать порядка 3 000 000 синтаксически корректных тестов.

В работе [9] описан генератор тестов Korat, который использует спецификацию тестовых данных в виде Java-метода, проверяющего корректность структуры данных. Кроме того, генератору требуется предоставить набор параметров, отвечающих за то, чтобы множество генерируемых данных было конечным.

В статье [23] описан основанный на грамматиках генератор тестов Geno. Этот генератор имеет ряд простых механизмов настройки генерации, таких как ограничение глубины дерева, ограничение глубины рекурсии, задание комбинатора и т.п.

Общим недостатком всех перечисленных выше подходов является то, что в них отсутствует возможность тонкой настройки генератора для построения тестов, нацеленных на тестирование некоторого заданного аспекта функциональности целевого ПО. Имеющиеся возможности позволяют лишь тем или иным образом уменьшить количество генерируемых тестов и избежать комбинаторного взрыва.

В работе [11] представлен метод для автоматического тестирования ПО, осуществляющего рефакторинг программного кода. Для генерации тестов, нацеленных на тестирование определенного вида рефакторинга, требуется разработать соответствующий генератор с использованием библиотеки ASTGen, созданной авторами метода.

В статьях [22,34] описан метод OTK для тестирования оптимизирующих компиляторов на основе моделей. Метод описывает способ построения модели по абстрактному описанию алгоритма оптимизации, выраженного на естественном языке. Для генерации тестов на основе построенной модели требуется разработать определенные компоненты генератора. Метод OTK снабжен одноименным инструментом, который предоставляет поддержку для формального описания модели данных, а также для разработки всех требующихся компонентов генератора тестов.

Главным достоинством этих двух подходов является следующее: поскольку генераторы тестовых данных фактически разрабатываются вручную, их довольно легко нацелить на генерацию данных для тестирования требуемого аспекта функциональности целевого ПО (например, определенного вида рефакторинга или определенного оптимизатора). Однако платой за это является высокая трудоемкость сопровождения каждого конкретного генератора: часто для модернизации имеющегося генератора с тем, чтобы генерируемые данные обладали некоторым дополнительным свойством, требуется заново разработать порядка половины кода генератора.

Итак, основными недостатками существующих в настоящее время подходов и генераторов тестовых данных являются следующие (и/или):

- бедный набор возможностей для настройки процесса генерации;
- высокая трудоемкость сопровождения входных данных генератора;
- использование массовой генерации тестов-кандидатов с последующей фильтрацией неподходящих кандидатов, что является причиной чрезмерных затрат ресурсов;
- плохая нацеливаемость на тестирование некоторого заданного аспекта функциональности целевого ПО;
- отсутствие критериев оценки качества совокупности генерируемых данных.

3. Модельное представление тестовых данных

Будем исходить из того, что данные сложной структуры составлены из частей, которые организованы в некоторую иерархию, например:

- программа – функция – инструкция – аргумент;
- документ – раздел – параграф – предложение.

При этом некоторые части иерархии могут быть как-то связаны между собой. Например, используемый в программе на языке со строгой типизацией идентификатор должен быть где-то объявлен. Такие связи мы будем называть *горизонтальными*.

Описание возможных иерархических структур часто задается в виде грамматики: схемы XML-документов описываются на языках DTD или XMLSchema, структуры реляционных баз данных часто задаются в виде скрипта создания базы данных на языке SQL, синтаксис формальных языков (например, языков программирования) традиционно описывается в форме Бэкуса-Наура (BNF). Поскольку в грамматиках в основном определяются лишь связи вида «предок–потомок»[1], для задания горизонтальных связей как правило дополнительно к грамматике описывается соответствующий набор ограничений (см. ниже).

[1] Такие языки, как XMLSchema и SQL позволяют задавать также горизонтальные связи некоторых видов.

В предлагаемом здесь подходе модель конкретной иерархической структуры представляется в виде атрибутированного гетерогенного дерева.

Пример.

Рассмотрим следующую BNF-грамматику арифметических выражений (для простоты, в этих выражениях используется только сложение чисел и скобки).

E ::= **F** (“+” **F**)* ;
F ::= <*N*> | **P** ;
P ::= “(” **E** “)” ;

В качестве атрибутированного дерева, соответствующего некоторому выражению, мы рассматриваем его дерево абстрактного синтаксиса. Для данной грамматики оно состоит из вершин следующих типов.

- Тип **E**, вершины которого имеют дочерние вершины:
  - ребенок типа **F**, соответствующий «головному» слагаемому;
  - список из нуля или более детей типа **F**, соответствующий «хвостовым» слагаемым.
- Тип **F**, у которого имеется два подтипа:
  - тип $\mathbf{F}_N$, вершины которого содержат атрибут, представляющий число;
  - тип $\mathbf{F}_P$, вершины которого имеют ребенка типа **P**, соответствующего подвыражению, взятому в скобки.
- Тип **P**, вершины которого имеют ребенка типа **E**, соответствующего подвыражению, которое содержится внутри скобок. ►

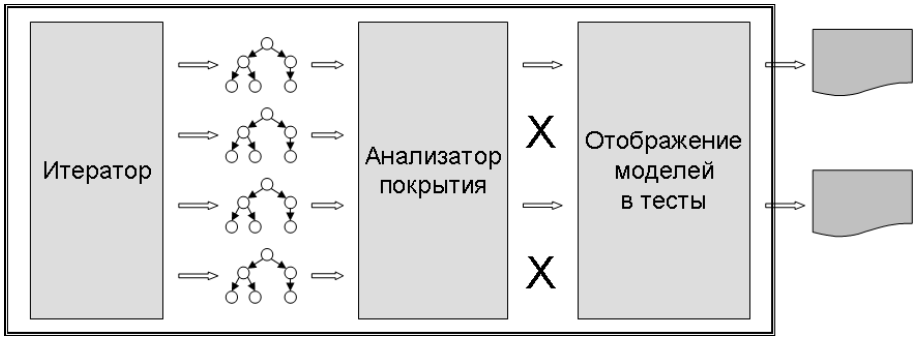

Рис. 1. Архитектура генератора Pinery

4. Архитектура генератора Pinery

Генератор Pinery состоит из следующих основных компонентов (Рис. 1):

- итератор атрибутированных деревьев;
- анализатор достигнутого покрытия;
- транслятор для отображения моделей в тесты.

В общих чертах генерация тестов происходит так (схематичная иллюстрация процесса генерации показана на Рис. 11, псевдокод алгоритма генерации приведен на Рис. 2). Итератор атрибутированных деревьев генерирует деревья последовательно по одному. Если очередное сгенерированное дерево увеличивает уровень достигнутого покрытия, то дерево отображается в тестовые данные, которые могут использоваться для тестирования целевой системы. При этом собственно тестовые данные могут представлять собой как текст на некотором формальном языке (например, тексты программ для тестирования компилятора), так и сложную структуру объектов в памяти ЭВМ (например, для модульного тестирования программных интерфейсов систем, работающих с такого рода объектами).

```
void run( Config c ) {
    Iterator it = createIterator( c );
    for( it.init(); it.has(); it.next() )
                                {
        if( isCoverageIncreased( it.value() ) )
        {
            mapToTest( it.value() );
        }
    }
}
```

Рис. 2. Псевдокод алгоритма генерации Pinery

Процесс итерации управляется набором ограничений, природа которых бывает двух видов:

- необходимые ограничения, выражающие семантику генерируемых данных (например, ограничения, описывающие горизонтальные связи);
- дополнительные ограничения, служащие для снижения количества генерируемых тестов до приемлемого уровня (например, ограничения на длины генерируемых списков, на глубину генерируемых деревьев и пр.).

Каждое ограничение представляет собой предикат, зависящий от структуры некоторого поддерева. Ограничение работает в итераторе как фильтр, который отсеивает все те построенные поддеревья, которые не удовлетворяют данному ограничению, т.е. на которых соответствующий предикат не выполняется.

Вся информация о наложенных ограничениях, о выбранном критерии покрытия, о способе отображения моделей в тесты и о всех прочих настройках передаются в генератор через специальный конфигурационный файл.

Подробнее алгоритм итерации атрибутированных деревьев Pinery изложен в работе авторов [3].

Далее мы рассмотрим некоторые общие вопросы предлагаемого подхода.

5. Цель генерации

Пусть на множестве всех входных данных тестируемой программной системы задан некоторый критерий тестового покрытия. Тогда целью генерации атрибутированных деревьев является получение такого набора тестовых данных, на котором достигается заданный уровень покрытия по этому критерию.

В тех случаях, когда уровень достигнутого тестового покрытия удается вычислять непосредственно в процессе генерации данных, генератор можно снабдить соответствующим анализатором покрытия и останавливать процесс генерации при достижении заданного уровня покрытия.

Пример.

В работе [27] Пардом предложил следующий способ задания элементов покрытия синтаксически корректных предложений для тестирования парсеров: в один элемент покрытия входят те предложения, в дереве вывода которых использована определенная продукция грамматики данного языка. Таким образом, некоторая данная грамматика определяет столько элементов покрытия, сколько в ней содержится продукций.

Для грамматики аддитивных выражений (см. выше) этот способ определяет три элемента покрытия: для продукций **E**, **F** и **P**. ►

6. Настройка эффективной итерации деревьев

Использование ограничений как предикатов для выбора конечного подмножества итерируемых деревьев не является эффективным, поскольку использование фильтра не влияет на скорость генерации всего фильтруемого поддерева.

В Pinery имеются следующие базовые средства настройки эффективной итерации деревьев:

- задание специфического итератора;
- задание комбинатора;
- использование контекста генерации дерева.

Специфический итератор

По умолчанию для перебора значений данного поддерева в Pinery создается некоторый стандартный итератор. Если на данное поддерево наложено какое-то ограничение, то соответствующий ему предикат будет применяться для фильтрации значений именно этого стандартного итератора.

Простейшим средством настройки эффективной итерации данного поддерева является задание для использования в качестве итератора его значений некоторого специфического итератора, который будет заведомо выдавать поддеревья, удовлетворяющие данному ограничению. На практике

использовать это средство без чрезмерных трудозатрат удается в основном лишь для значений скалярных атрибутов. Однако, даже такое узкое использование специфических итераторов дает существенный выигрыш во времени генерации и в качестве получаемых результатов по сравнению с использованием итераторов по умолчанию.

Комбинатор

Другим средством настройки эффективной итерации является задание комбинатора итерируемых значений полей некоторого узла. По умолчанию в качестве комбинатора используется итератор декартова произведения, перебирающий все возможные комбинации значений полей. Простейшим комбинатором, позволяющим существенно снизить количество перебираемых комбинаций, является так называемый итератор диагонали, перебирающий такое множество S кортежей, что $\forall i \in \{1,\text{K},n\}\ \ \forall s \in S_i\ \ \exists(s_1,\text{K},s_n) \in S$: $s = s_i$, где S_i – это множество значений, итерируемое i-м подчиненным итератором.

Выбор того или иного комбинатора в каждом конкретном случае обычно бывает продиктован структурой выбранного покрытия.

Контекст генерации дерева

Еще одним средством настройки эффективной итерации является использование контекста, т.е. информации об уже построенной части дерева.

Рассмотрим, например, следующее семантическое ограничение: имя используемой переменной в программе может использоваться только после объявления этой переменной. Можно так настроить итератор Pinery, чтобы в процессе итерации для имени используемой переменной перебирались только имена переменных, которые имеются в контексте.

Другим примером является настройка эффективной итерации деревьев, удовлетворяющих заданному ограничению на глубину получающихся деревьев. В этом случае можно использовать имеющуюся в контексте информацию о текущей глубине вновь создаваемого узла дерева. Аналогично можно организовать ограничение глубины рекурсии относительно вершин данного типа.

7. Эффективность генератора

Как было сказано выше, основной целью генерации тестовых данных является достижение заданного критерия покрытия. При этом из практических соображений требуется, чтобы генератор работал некоторое достаточно недолгое время.

Время работы генератора, а также уровень покрытия, достигаемый на построенных тестах, очень сильно зависят от того, какие были заданы настройки итерации.

Пример.

Рассмотрим пример с аддитивными выражениями. Предположим, что в качестве цели генерации используется покрытие продукционных правил в дереве вывода[2] с критерием покрытия 100%.

Пусть для итерации используются следующие настройки:

- I_{len}: итератор значений длины списка «хвостовых» слагаемых выражения;
- I_N: итератор значений атрибута-числа <*N*>;
- R: глубина рекурсии относительно вершин типа **E**.

Пусть при переборе значений слагаемых сперва используются генераторы $G(\mathbf{F}_N)$ вершин типа $\mathbf{F}_N$ с комбинатором декартова произведения, и лишь после этого используются генераторы $G(\mathbf{F_P})$ вершин типа $\mathbf{F_P}$.

При значениях настроек $I_{len} = \{0\}$, $I_N = \{2\}$, $R = 1$, будет построено единственное дерево, соответствующее выражению «2». При этом будет достигнуто покрытие 67%: покрыто два правило из трех (**E** и **F**).

При значениях настроек $I_{len} = \{0, 1\}$, $I_N = \{2, 3\}$, $R = 2$, будет построено семь деревьев, соответствующих выражениям «2», «3», «2+2», «2+3», «3+2», «3+3», «(2)». Из этих семи деревьев анализатор покрытия пропустит только два: «2» и «(2)». Остальные деревья будут отброшены, поскольку не увеличивают покрытия: на первом же дереве покрыты два правила из трех (**E** и **F**), а третье правило (**P**) покрыто лишь на седьмом дереве. Итак, достигнуто покрытие 100%, однако генератор работал не эффективно, поскольку из семи сгенерированных деревьев пять было отброшено анализатором покрытия.

При значениях настроек $I_{len} = \{0\}$, $I_N = \{2\}$, $R = 2$, будет построено два дерева, соответствующих выражениям «2», «(2)». Оба дерева будут пропущены анализатором покрытия, и при этом будет достигнуто покрытие 100%. ►

Очевидно, что эффективнее всего генератор будет работать тогда, когда ни одно из сгенерированных деревьев не будет отброшено анализатором покрытия. Следует, однако, учитывать, что для реальной задачи, скорее всего, будет очень нелегко подобрать такие точные настройки, при которых заданный критерий покрытия будет достигаться совсем без фильтрации. Вероятнее всего, будет дешевле задать некоторые более грубые настройки генерации, при которых какая-то приемлемая часть сгенерированных деревьев все-таки будет отброшена, но зато заданный критерий покрытия будет

[2] Выбор такой цели генерации здесь является искусственным и направлен на то, чтобы в простой ситуации проиллюстрировать зависимость эффективности генератора от настроек итерации. В реальности же более адекватной целью генерации для данного примера является покрытие различных длин списка «хвостовых» слагаемых и различных достигнутых глубин рекурсии.

достигнут при затрате приемлемого времени работы тестировщика на настройку генератора.

8. Задачи тестировщика

Итак, основной задачей тестировщика при использовании генератора Pinery является следующее.

1. Определить адекватную цель генерации и соответствующий ей анализатор покрытия.
2. Задать такие настройки генерации, чтобы,
   a. потратив на это приемлемое время,
   b. получить покрытие 100%,
   c. при приемлемой доле отброшенных при фильтрации данных.

Как показывает практика, для результативной генерации тестовых данных, нацеленных на тестирование данного ПО, описанных выше базовых средств оказывается недостаточно. Эти средства в основном позволяют лишь в некоторой степени уменьшить количество генерируемых данных. Однако, с их помощью очень трудно настроить генератор на построение данных, имеющих некоторую специфическую структуру (см. пример ниже). Кроме того, огромное количество настроек для объемных моделей вызывает большие трудности при сопровождении конфигурации генератора.

В предлагаемом здесь подходе для решения этой проблемы используются модели разного уровня абстракции, которые позволяют моделировать различные аспекты генерируемых данных в виде отдельных абстрактных моделей, выраженных в некоторых подходящих математических терминах. Экземпляры абстрактных моделей генерируются независимо и затем отображаются в общую конкретную модель тестовых данных. В результате это позволяет генерировать небольшие множества тестовых данных, нацеленных на особенности тестируемого ПО.

Таким образом, для получения нацеленных тестовых данных, тестировщик должен предварительно решить следующие задачи:

- выделить целевые аспекты тестовых данных и выразить их в виде подходящих абстрактных моделей;
- описать отображение абстрактных моделей в общую конкретную модель.

При использовании абстрактных моделей конфигурация генератора получается намного проще, понятнее и управляемее ввиду фактического разделения ее на модули, соответствующие разным абстрактных моделям.

9. Пример абстрактной модели

Предположим, что мы генерируем тесты для компилятора языка C для тестирования в нем оптимизатора, осуществляющего удаление недостижимого кода (Unreachable-Code Elimination, см. [25]). Алгоритм этой оптимизации в основном работает со структурой линейных участков программы и переходов между ними.

Пусть имеется некоторая модель программ на языке C, в которой линейные участки моделируются помеченными блоками, каждый из которых может завершаться переходом `goto`:

```
label_A:
  {
     ... // инструкции
     goto label_B;
  }
```

Будем генерировать каждый тест в виде функции, состоящей из последовательности помеченных блоков. При этом каждый тест должен успешно компилироваться и выполняться. Чтобы гарантировать завершение выполнения теста за конечное время, можно генерировать функции с ациклической структурой переходов `goto` (например, чтобы каждый переход `goto` вел на метку одного из последующих блоков)[3].

Для моделирования структуры переходов будем использовать абстрактную модель, выраженную в виде ациклического графа. Для построения одного теста при этом потребуется отобразить структуру очередного построенного ациклического графа в структуру переходов `goto`. Различные структуры ациклических графов при этом порождают тесты с различной структурой переходов `goto`. Таким образом, использование такой абстрактной модели позволяет получать тесты со следующими свойствами:

a. все тесты различны и при этом «интересны» в плане их нацеленности на тестируемую функциональность;
b. каждый тест при выполнении гарантированно не зацикливается.

Кроме того, количество неизоморфных ациклических графов сравнительно невелико. Итак, в результате мы получим небольшое множество тестов, нацеленных на тестирование данного оптимизатора. Получить множество тестов с похожими свойствами, используя лишь базовые средства настройки генератора, было бы гораздо сложнее.

[3] Если для тестирования требуется наличие в функции циклической структуры, следует объявить цикл отдельным элементом модели и генерировать циклы, которые выполняются конечное количество раз.

10. Использование абстрактных моделей в Pinery

Как уже было сказано, сложные структуры данных мы моделируем в виде атрибутированных деревьев. При этом горизонтальные связи (см. на Рис. 3.a пример связи, описывающей переход на метку) моделируются путем использования пары атрибутов, один из которых представляет идентификатор, а другой — ссылку на этот идентификатор (см. соответствующие атрибуты `label` и `goto` на Рис. 3.b). Связь считается установленной, если значение атрибута-ссылки равно значению атрибута-идентификатора.

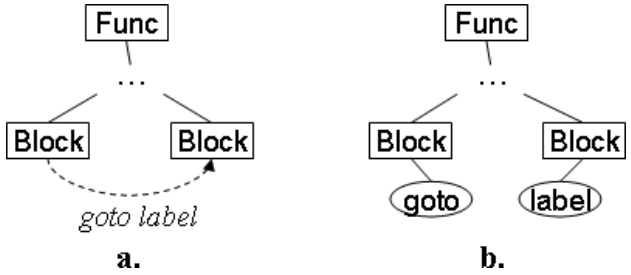

Рис. 3. Моделирование горизонтальной связи путем использования пары атрибутов «идентификатор» (label) и «ссылка» (goto).

В примере, обсуждавшемся в предыдущем разделе, ациклический граф моделирует структуру переходов `goto` в модели программ на языке C. Таким образом, исходную модель программ на C можно назвать *конкретной моделью*, а ациклический граф – *абстрактной моделью*.

Для данного модельного дерева *M* (см. на Рис. 4.a модельное дерево для обсуждавшегося выше примера) можно построить его абстракцию (Рис. 4.c), путем игнорирования некоторых его «несущественных» узлов (Рис. 4.b). После этого полученную абстракцию можно представить в виде соответствующей абстрактной модели *N*, выраженной в подходящих математических терминах (например, ациклический граф на Рис. 4.d). Процедура построения абстракции для моделей, описанных в виде грамматик, подробно обсуждается в работе [4].

Пусть в рассмотренном выше примере соответствующая часть BNF грамматики для исходной модели *M* выглядит так[4]:

[4] Здесь константные терминалы опущены, атрибуты представлены в угловых скобках, фигурные скобки могут использоваться для именования подвыражений BNF.

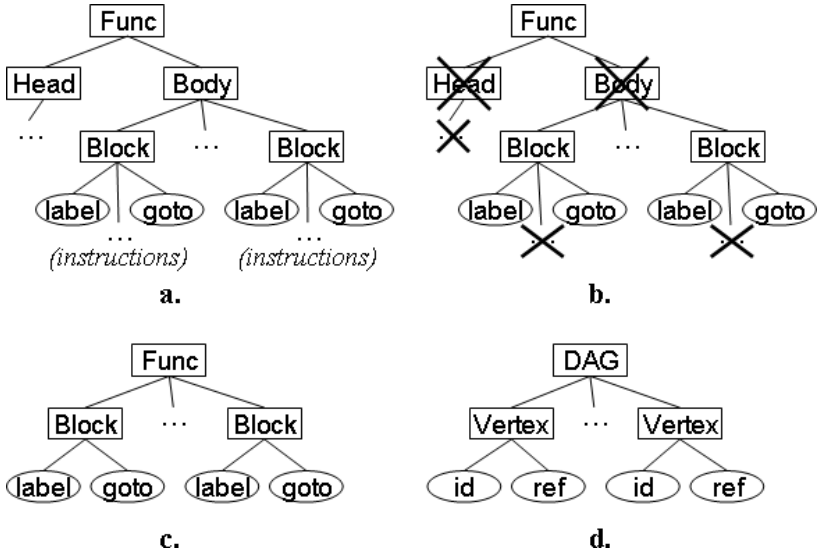

Рис. 4. Построение абстрактной модели N для модельного дерева M.
a. Исходное модельное дерево M; b. Игнорирование несущественных узлов;
c. Абстракция дерева M; d. Соответствующая абстрактная модель N.

```
Func  ::= Head Body;
Head  ::= ...
Body  ::= {blocks:Block*};
Block ::= <label:ID> Insn* <goto:ID>?;
Insn  ::= ...
 ...
```

Тогда соответствующая BNF грамматика для абстрактной модели *N* такова:

```
DAG    ::= {vertices:Vertex*};
Vertex ::= <id:ID> <ref:ID>?;
```

При построении теста сначала создается экземпляр абстрактной модели, а затем в соответствии с ней создается экземпляр конкретной модели. Для определения отображения абстрактной модели в конкретную модель требуется описать соответствующие правила в терминах грамматик для абстрактной и конкретной модели. Набор правил для приведенных здесь грамматик выглядит так:

```
DAG:vertices -> Body:blocks;
Vertex:id    -> Block:label;
Vertex:ref   -> Block:goto;
```

Каждое правило имеет вид


```
<abstr_context>:<abstr_elem> ->
 <concr_context>:<concr_elem>,
```

где `<abstr_elem>` – имя элемента-прообраза в контексте `<abstr_context>` абстрактной модели, а `<concr_elem>` – имя соответствующего элемента-образа в контексте `<concr_context>` конкретной модели.

В ходе построения экземпляра конкретной модели по данному экземпляру абстрактной модели, генератор Pinery ищет потенциальные пары «элемент-прообраз—элемент-образ», которые удовлетворяют некоторому правилу. Если найденный потенциальный элемент-образ является атрибутом, то в качестве его значения устанавливается значение соответствующего найденного элемента-прообраза. Иначе поиск пар продолжается в контексте найденных поддеревьев. Значения тех элементов конкретной модели, для которых не нашлось элемента-образа в абстрактной модели, достраиваются соответствующими стандартными или заданными в конфигурации специфическими итераторами.

Отображение определено корректно, если в данном наборе правил каждый атрибут-образ встречается не более одного раза.

11. Использование нескольких абстрактных моделей

Для данной модели *M* абстрактная модель *N* отражает некоторый аспект из *M*. В ряде случаев бывает целесообразно отразить несколько аспектов из *M* в нескольких отдельных абстрактных моделях $N_1, ..., N_k$.

Пример.
В представленном выше примере используется одна абстрактная модель, которая отражает структуру переходов `goto` в одной функции. Предположим теперь, что мы хотим генерировать каждый тест в виде нескольких функций, которые вызывают друг друга. Чтобы гарантировать выполнение теста за конечное время, мы можем генерировать ациклическую структуру вызовов функций. Таким образом, здесь будет использовано две абстрактные модели: одна из них отражает структуру переходов `goto` в одной функции, а другая – структуру вызовов функций. ►

Для данной модели *M* использование нескольких абстрактных моделей $N_1, ..., N_k$ корректно, если наборы правил отображения в модель *M* из различных абстрактных моделей описывают попарно непересекающиеся множества атрибутов-образов.

Кроме использования нескольких абстрактных моделей для одной конкретной модели, иногда бывает целесообразно рассмотреть уже используемую абстрактную модель в качестве конкретной и строить ее абстракции

аналогичным образом. Таким образом может получиться многоуровневая иерархия абстрактных моделей.

12. Библиотеки абстрактных моделей и итераторов

Использование простых абстрактных моделей, выраженных в математических терминах, дает возможность эффективно итерировать экземпляры абстрактных моделей с помощью широко известных методов перечислительной комбинаторики (см. например [13,29]).

Если абстрактная модель *N* отражает некоторый аспект конкретной модели, то можно легко изменять свойства тестов, касающиеся этого аспекта, путем смены итератора экземпляром модели *N*. Так, если в обсуждавшемся выше примере потребуется генерировать некоторую другую структуру переходов `goto`, то для этого нужно будет использовать соответствующий итератор абстрактных графов. Замечательно, что такая модификация не потребует изменения других частей конфигурации генератора, в том числе описаний используемых моделей и правил отображений абстрактных моделей в конкретную модель. Таким образом, можно независимо управлять свойствами, относящимися к различным аспектам генерируемых тестов, если эти аспекты отражены в разных абстрактных моделях.

Кроме того, абстрактные модели удается легко переиспользовать в различных проектах. Чтобы использовать абстрактную модель *N* для генерации экземпляров данной конкретной модели *M*, требуется лишь описать соответствующие правила отображения из *N* в *M*.

Для создания абстрактной модели для генератора Pinery требуется:

- описать синтаксическую структуру абстрактной модели;
- создать конфигурацию генератора, которая позволит строить экземпляры абстрактной модели, обладающие заданными свойствами;
- разработать на языке Java специфические итераторы, которые используются в конфигурации генератора.

13. Заключение

В работе представлен подход к автоматической генерации тестовых данных сложной структуры, использующий формальные описания тестовых данных в виде грамматик и встраиваемый в технологию автоматизированного тестирования UniTESK, которая основана на формальных спецификациях и моделях и поддерживает весь цикл тестирования начиная от определения требований к целевой системе и заканчивая анализом результатов тестирования.

Для практического использования предложенного подхода разработан генератор тестовых данных Pinery, который:

- позволяет генерировать данные в соответствии с заданными критериями тестового покрытия;
- предоставляет достаточный для большинства практических нужд набор средств для настройки процесса генерации;
- обеспечивает генерацию тестовых данных как в виде текста на некотором формальном языке, так и в виде сложной структуры объектов в памяти ЭВМ.

Генератор Pinery можно рассматривать как базовую среду, которая позволяет создавать специализированные расширения путем разработки библиотек абстрактных моделей и итераторов для решения разнообразных задач генерации специфических данных. В работе описывается метод для разработки подобных специализированных генераторов, который:

- позволяет генерировать сравнительно не большие множества тестовых данных, нацеленных на тестирование заданных аспектов функциональности ПО;
- обеспечивает модульность конфигурации генератора и переиспользуемость отдельных модулей.

Как показала практика, наибольший эффект при разработке тестовых данных в рамках представленного подхода достигается в случае активного использования абстрактных моделей данных.

Литература

[1] М.В. Архипова. Генерация тестов для семантических анализаторов. Вычислительные методы и программирование, том 7, раздел 2, 55–70, 2006.

[2] А.В. Баранцев, И.Б. Бурдонов, А.В. Демаков, С.В. Зеленов, А.С. Косачев, В.В. Кулямин, В.А. Омельченко, Н.В. Пакулин, А.К. Петренко, А.В. Хорошилов. Подход UniTesK к разработке тестов: достижения и перспективы. Труды ИСП РАН, 5:121–156, Москва, 2004.

[3] А.В. Демаков, С.В. Зеленов, С.А. Зеленова. Генерация тестовых данных сложной структуры с учетом контекстных ограничений.Труды ИСП РАН, Москва, 2006, т. 9, 83–96.

[4] С.В. Зеленов, С.А. Зеленова, А.С. Косачев, А.К. Петренко. Генерация тестов для компиляторов и других текстовых процессоров. Программирование, 29(2):59–69, 2003.

[5] В.П. Иванников, А.С. Камкин, В.В. Кулямин, А.К. Петренко. Применение технологии UniTesK для функционального тестирования моделей аппаратного обеспечения. Препринт 8 ИСП РАН, Москва, 2005.

[6] Г.В. Ключников, А.К. Косачев, Н.В. Пакулин, А.К. Петренко, В.З. Шнитман. Применение формальных методов для тестирования реализации IPv6. Труды ИСП РАН, 4:121–140, 2003, Москва.

[7] В.В. Кулямин, А.К. Петренко, А.С. Косачев, И.Б. Бурдонов. Подход UniTesK к разработке тестов. Программирование, 29(6):25–43, 2003.

[8] А.К. Петренко и др. Тестирование компиляторов на основе формальной модели языка. Препринт института прикладной математики им. М.В Келдыша, № 45, 1992.
[9] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated Testing Based on Java Predicates. Proc. of ISSTA 2002, Rome, Italy. July 2002.
[10] Canam Software Turbodata. http://www.turbodata.ca/.
[11] B. Daniel, D. Dig, K. Garcia, D. Marinov. Automated Testing of Refactoring Engines. ESEC/FSE'07, 185–194, 2007.
[12] A. Duncan, J. Hutchison. Using attributed grammars to test designs and implementation. In Proceedings of the 5th international conference on Software engineering, 170–178, 1981.
[13] I.P. Goulden, D.M. Jackson. Combinatorial Enumeration. Wiley, 1983.
[14] R.F. Guilmette. TGGS: A flexible system for generating efficient test case generators, 1995.
[15] Y. Gurevich. Abstract state machines: An overview of the project. LNCS, 2942, 6–13, 2004.
[16] J. Harm. Automatic test program generation from formal language specifications. Rostocker Informatik-Berishte, 20, 33–56, 1997.
[17] J. Harm, R. Lämmel. Testing attribute grammars. In Proceedings of Third Workshop on Attrite Grammars and their Applications, 79–98, 2000.
[18] J. Harm, R. Lämmel. Two-dimensional approximation coverage. Informatica, 24(3), 2000.
[19] IBM DB2 Database Test Generator. http://www-306.ibm.com/software/data/db2imstools/db2tools/db2tdbg/
[20] A. Kalinov, A. Kossatchev, A. Petrenko, M. Posypkin, V. Shishkov. Using ASM specifications for compiler testing. LNCS 2589:415, 2003.
[21] A. Kalinov, A. Kossatchev, M. Posypkin, V. Shishkov. Using ASM specification for automatic test suite generation for mpC parallel programming language compiler. In Proceedings of Fourth International Workshop on Action Semantic, AS'2002, BRICS note series NS-02-8, 99–109, 2002.
[22] A. Kossatchev, A. Petrenko, S. Zelenov, S. Zelenova. Application of model-based approach for automated testing of optimizing compilers. In Proceedings of the International Workshop on Program Understanding, Novosibirsk, 81–88, 2003.
[23] R. Lämmel, W. Schulte. Controllable combinatorial coverage in grammar-based testing. In TestCom, LNCS 3964:19–38, 2006.
[24] P.M. Maurer. Generating test data with enhanced context-free grammars. IEEE Software, 7(4):50–55, 1990.
[25] S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, 1997.
[26] J. Paakki. Attribute grammar paradigms – a high-level methodology in language implementation. ACM Computing Surveys, 27(2):196–255, 1995.
[27] P. Purdom. A sentence generator for testing parsers. Behavior and Information Technology, 12(3):366–375, 1972.
[28] E.G. Sirer, B.N. Bershad. Using production grammars in software testing. In Second Conference on Domain-Specific Languages, 1–13, 1999.
[29] R.P. Stanley. Enumerative Combinatorics, 2 Vols. Cambridge University Press, 1996/1999.
[30] UniTESK. http://www.unitesk.ru/.
[31] XML Generator. http://www.stylusstudio.com/xml_generator.html.

[32] XML-XIG. http://sourceforge.net/projects/xml-xig.
[33] S. Zelenov, S. Zelenova. Automated generation of positive and negative tests for parsers. LNCS 3997:187–202, 2006.
[34] S. Zelenov, S. Zelenova. Model-based testing of optimizing compilers. In Proc. of the 19th IFIP TC6/WG6.1 International Conference on Testing of Software and Communicating Systems — 7th International Workshop on Formal Approaches to Testing of Software (TestCom/FATES), LNCS 4581:365–377, 2007.