Генерация тестовых данных сложной структуры с учетом контекстных ограничений

А.В. Демаков, С.В. Зеленов, С.А. Зеленова

1. Введение

Тестирование программных систем является важным компонентом всех проектов, связанных с разработкой программного обеспечения. По мере роста масштабов и сложности программных систем растет трудоемкость тестирования. Решить задачу повышения качества и сокращения расходов на тестирование можно за счет привлечения эффективных средств автоматизации разработки тестов.

Одной из наиболее распространенных задач, возникающей при тестировании сложных программных систем, является генерация тестовых данных сложной структуры. Такие задачи характерны для тестирования систем, использующих интернет-протоколы, XML-технологии, SQL-интерфейсы баз данных, интерпретаторов и трансляторов языков программирования, спецификаций, командных языков, а также многих других видов программных систем.

Существенным недостатком имеющиеся в настоящее время инструментов для генерации тестовых данных сложной структуры для тестирования, например, приложений над базами данных [1–3], XML-приложений [4–6], компиляторов и других обработчиков формальных языков [7–10] является то, что в них очень узок предоставляемый спектр возможностей по управлению процессом генерации тестовых данных. Поэтому для достижения приемлемого качества тестирования приходится генерировать очень большие множества тестовых данных, что сопряжено с большими накладными расходами по использованию ресурсов как на этапе генерации тестовых данных, так и на этапе анализа результатов прогонов тестов.

В настоящей статье представлена технология автоматической генерации тестовых данных сложной структуры, которая предполагает возможность тонкой настройки процесса генерации тестовых данных и оптимизации этого процесса под особенности функциональности конкретного тестируемого приложения.

В основу предлагаемой технологии положен опыт работ ИСП РАН по разработке тестовых данных и созданию автоматических генераторов тестовых данных на основе языковых моделей [11–16].

Предложенный в ИСП РАН подход к работе с данными сложной структуры является частью разработанной в ИСП РАН технологии UniTesK тестирования программного обеспечения на основе формальных спецификаций и моделей [17–19]. Этот подход основан на использовании формального описания данных сложной структуры.

2. Представление тестовых данных

Многие языки формального описания данных сложной структуры, по сути, описывают некоторое атрибутированное дерево. Среди таких языков – BNF грамматики формальных языков, XMLSchema для описания структуры XML-документов, ASN.1 для описания формата данных телекоммуникационных протоколов и многие другие.

Действительно, сама структура записи информации в компьютере предполагает такую форму: всякий объект записывается в памяти как некая последовательность бит, в которой можно выделить подпоследовательности, в них – другие подпоследовательности и т.д. При этом выделенные подпоследовательности могут быть связаны между собой, например, равны друг другу.

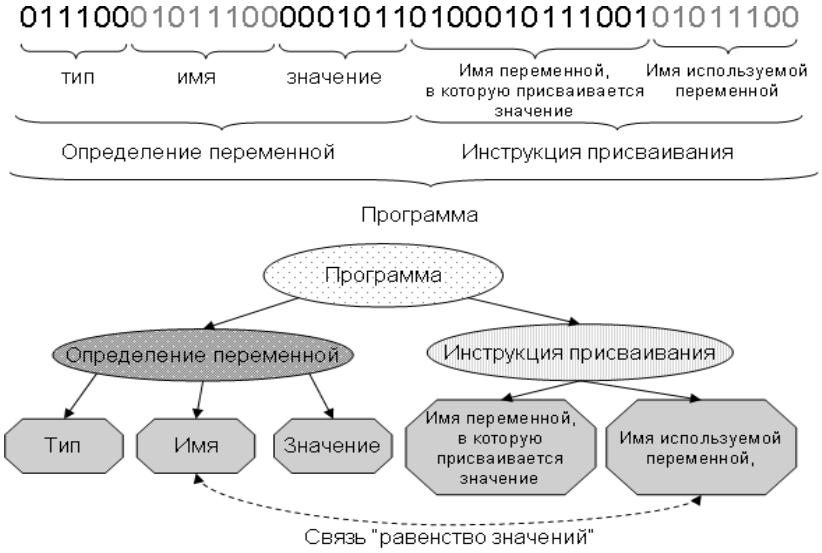

Рис.1. Представление данных сложной структуры в виде последовательности бит и в виде атрибутированного дерева.

Основываясь на этом наблюдении, можно свести генерацию данных сложной структуры к генерации атрибутированных деревьев. При этом необходимо учитывать и горизонтальные связи, возникающие в сложных структурах.

В этой статье мы опишем своеобразный framework для создания генераторов атрибутированных деревьев с учетом горизонтальных связей, или, иными словами, с учетом контекста элементов дерева.

3. Предварительные сведения и понятия

В начале введём понятие атрибутированного дерева.

Как известно, дерево – это граф, в котором нет циклов. Мы будем рассматривать ориентированные деревья, то есть деревья, у которых имеется выделенная вершина – корень, и все ребра ориентированы от более близких к корню вершин к более дальним.

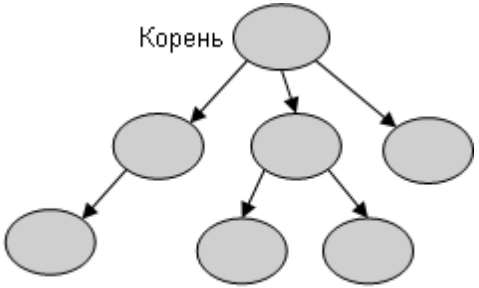

Рис.2.Ориентированное дерево.

Пусть A – некоторая вершина ориентированного дерева. Если она не является корнем, то существует единственная вершина B, из которой в A идет ребро. Вершину B мы будем называть родителем вершины A, а вершину A, соответственно, ребёнком вершины B. В отличие от родителя, детей у вершины может быть много.

Деревья, которые мы рассматриваем, являются атрибутированными, то есть с каждой вершиной могут быть связаны её атрибуты – объекты произвольного типа.

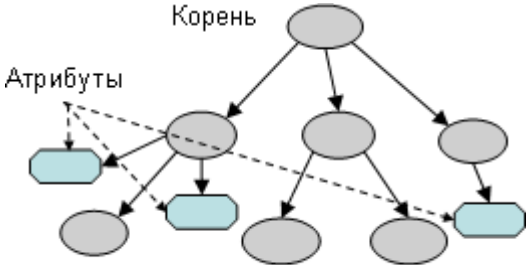

Рис.3. Атрибутированное дерево.

Мы будем рассматривать гетерогенные деревья, то есть деревья, у которых каждая вершина имеет определённый тип. Тип регламентирует, какие дети и атрибуты и какого типа могут быть у вершины данного типа. При этом все дети и атрибуты именуются. Кроме того, в определении типа может

содержаться указание на то, что детей (или атрибутов) данного типа может быть несколько (то есть они образуют список), а также на то, что какой-то ребёнок или атрибут является необязательным, то есть может отсутствовать у вершины данного типа.

Пример. Рассмотрим следующее определение типа вершины: вершины типа A должны содержать одного ребёнка типа B с именем child, необязательный атрибут name типа “строка”, список list детей типа C. Примеры вершин типа A приведены на рис.4.

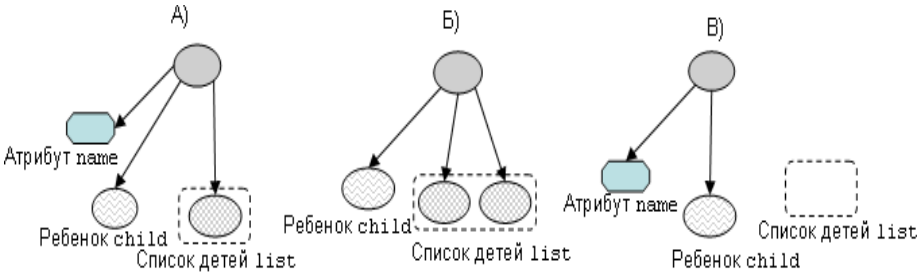

Рис.4. Примеры вершин типа A

Заметим, что на рис. 4 Б) нет атрибута name – это возможно, так как в определении типа этот атрибут заявлен необязательным, в то же время список list присутствует всегда, хотя может и не содержать элементов.

Итак, в определении типа вершины должны быть описаны:

- атрибуты вершины: их имена и типы, а также то, являются ли они обязательными;
- списки атрибутов: их имена и типы элементов этих списков;
- дети вершины: их имена и типы, а также то, являются ли они обязательными;
- списки детей: их имена и типы элементов этих списков.

Детей, атрибуты и списки вершины мы будем называть *полями* этой вершины.

Пусть имеется конечное множество `T` типов вершин, замкнутое относительно использования типов (то есть все типы вершин, на которые есть ссылки в определениях типов из этого множества, определены). Далее мы будем рассматривать множество `D` деревьев, типы вершин которых принадлежат множеству `T`.

4. Идея метода

Множество вершин типа `t` изоморфно декартову произведению множеств объектов, которые могут быть её полями. Поясним это на примере. Пусть вершина типа t содержит:

- список `list` детей типа `p;`

- ребёнка `child` типа `s`;
- необязательного ребёнка `opt_child` типа `r`;
- атрибут `attr` типа `W`;

Пусть M_l – множество списков вершин типа `p`, M_s – множество вершин типа `s`, M_r – множество вершин типа `r`, M_w – множество объектов типа `W`. Тогда множество вершин типа `t` изоморфно декартову произведению множеств M_l, M_s, M_w, $M_r \cup \varepsilon$ (здесь ε – символ отсутствия ребёнка). Действительно, любой кортеж из этого декартого произведения – набор возможных значений полей вершины типа `t`.

Это наблюдение обеспечивает общий метод генерации вершин типа `t`. Итак, для построения (конечного) множества вершин типа `t` необходимо построить конечные множества значений полей такой вершины и выбрать подмножество из декартова произведения этих множеств.

До сих пор мы говорили о ситуации, когда нет никаких связей между атрибутами, а также нет никаких ограничений на значения полей вершины. Если такие связи или ограничения есть, то множества значений полей зависят друг от друга. То есть для того, чтобы определить множество допустимых значений для одного поля, нужно знать значения каких-то других полей. Если в строящемся дереве два атрибута разных вершин зависят друг от друга, то эта зависимость может быть описана через зависимость полей некоторого узла (см. Рис. 5).

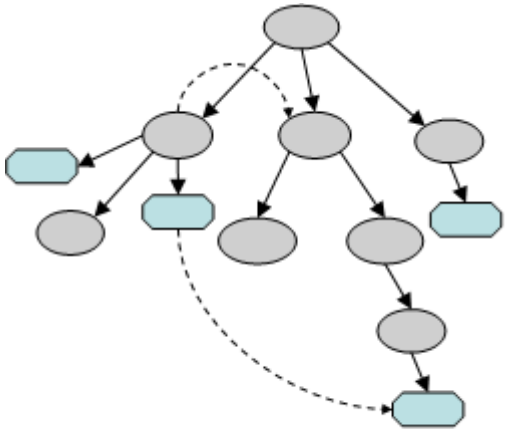

Рис.5. Зависимость атрибутов как зависимость полей одной вершины.

Процесс генерации теперь будет выглядеть несколько по-другому. Сначала нужно определить зависимости между значениями полей. Граф зависимостей должен быть ациклическим, иначе будет невозможно установить порядок построения полей вершины. Когда порядок построения полей определен, строим множество значений для независимого поля, выбираем из него какое-то

значение и устанавливаем его в качестве соответствующего поля вершины. Затем строим множества значений для полей, зависящих от уже установленного поля, выбираем из них значения, устанавливаем выбранные значения в качестве соответствующих полей вершины и т.д. Таким образом, множества значений полей строятся в соответствии с уже построенным контекстом, поэтому результирующая вершина будет удовлетворять всем наложенным ограничениям. Для построения детей вершины можно использовать такой же метод генерации вершин соответствующего типа (см. Рис. 6; стрелками указано направление движения данных от одного генератора к другому).

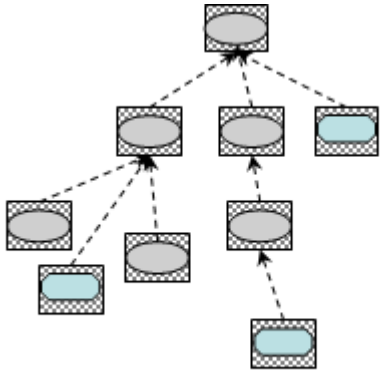

Рис.6. Система генераторов вершин разных типов.

Откуда могут возникать ограничения и связи между элементами дерева? Есть четыре основных источника.

- первый источник – это семантические требования на данные, такие как требование существования определения используемой переменной в языках программирования или требование уникальности значений какого-то атрибута в XML-документе;
- второй источник – это удовлетворение требований определённого критерия покрытия. В большинстве практических случаев множество всевозможных деревьев с вершинами определённых типов – это бесконечное множество, и мы, конечно же, не можем построить его целиком. Но это и не нужно, так как обычно такое множество можно разбить на конечное число классов таким образом, что в одном классе будут находиться элементы, “эквивалентные” с точки зрения целевой задачи. Такое разбиение называется критерием покрытия множества деревьев. Для одного и того же множества деревьев можно формулировать разные критерии покрытия; они будут зависеть от целевой задачи;

- третий источник – это ограниченность ресурсов. Часто даже в пределах одной задачи сформулировать для неё точный критерий покрытия невозможно или же слишком трудоёмко. Поэтому приходится брать более сильный критерий покрытия, разбивающий множество деревьев на более мелкие классы. При этом число классов может весьма сильно возрасти. О том, как можно бороться с этой проблемой, будет сказано немного позже;
- наконец, четвертый источник – это наличие рекурсии во множестве типов вершин `T`. Наличие рекурсии влечет необходимость её ограничения, то есть запрещения перебора деревьев с глубиной рекурсии, большей некоторого числа. При отсутствии ограничений на рекурсию множество допустимых деревьев бесконечно, а это значит, что генератор может войти в бесконечный цикл.

Итак, в самом общем виде идея состоит в том, чтобы создать систему генераторов вершин. Генераторы образуют иерархическую структуру, соответствующую структуре типов вершин. При этом работа генератора поля вершины может зависеть от результата работы генераторов других полей этой же вершины в соответствии с требованиями, накладываемыми на строящееся дерево, то есть работа генераторов зависит от контекста строящейся вершины.

5. Способ описания ограничений

Чтобы как-то учитывать требования при построении дерева, необходимо организовать движение информации по дереву. Иными словами, при построении очередной вершины мы должны знать, в каком контексте находится эта вершина и какие ограничения должны быть наложены на поля вершины в соответствии с этим контекстом.

В каждый момент построения дерева мы имеем какую-то конфигурацию вершин и атрибутов. Для конкретной вершины в этом недостроенном дереве мы можем определить её *состояние* – это то, что окружает вершину в данный момент. Фактически, это вся построенная часть дерева, но с точки зрения данной вершины. Рассмотрим понятие состояния на примере. На рис. 7 показано дерево с тремя вершинами типа `Def` (определение переменной). Чем отличается состояние, например, второй вершины от состояния первой? Состояния этих вершин различны, так как положения их в дереве различны. Из состояния (положения) первого определения видно, что в нём не могут быть использованы ссылки на переменные, так как до этой инструкции нет определённых переменных, а во втором определении можно использовать ссылку на переменную, поскольку к этому моменту уже есть определённая переменная `v1`.

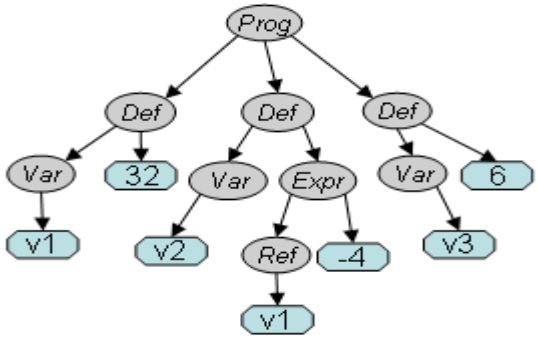

Рис.7. Разные состояния вершин одного типа.

При построении дерева обычно не требуется полностью знать состояние вершины. Нужны некоторые аспекты построенной части дерева, которые могут влиять на построение полей данной вершины. Эти аспекты будем называть *аспектами состояния* вершины. В приведённом выше примере для построения новой инструкциии определения переменной нам нужен только один аспект состояния – список уже определённых переменных. Заметим, что при построении очередного элемента дерева все аспекты состояний вершин, для которых существенен этот элемент, должны измениться.

Итак, множество значений поля вершины может зависеть от

- значений других полей этой вершины;
- некоторых аспектов состояния вершины.

Введём понятие *элементарного ограничения*. Это ограничение накладывается на поле вершины. Чтобы его определить, нужно задать:

- тип вершины `t`;
- поле `f` вершины типа `t`, на которое накладывается ограничение;
- поля вершины f_1, f_2, …, f_k, от которых зависит множество допустимых значений поля `f`;
- аспекты состояния вершины S_1, S_2, … , S_n, от которых зависит множество допустимых значений поля `f`;
- способ вычисления множества допустимых значений поля `f`;
- функцию проверки значения поля `f` на соответствие определяемому ограничению;
- функцию ограничения количества используемых значений поля `f`.

Итак, в каждой вершине строящегося дерева определены:

- аспекты состояния этой вершины, существенные для её построения;
- ограничения на поля вершины.

Систему аспектов состояния вершины и ограничений на её поля будем называть контекстом данной вершины. Контексты детей вершины зависят от контекста вершины. Контекст ребёнка, который зависит от какого-то поля, должен вычисляться с учётом построенного значения этого поля.

6. Итераторы

В определении элементарного ограничения указано, что для него нужно задать способ вычисления множества допустимых значений. Мы будем использовать для этого технику итераторов.

В самом общем виде *итератор* – это объект, имеющий несколько последовательных состояний. Итератор может перевести себя в начальное состояние; находясь в каком-то состоянии, он может перевести себя в следующее состояние, а также ответить на вопрос, имеется ли у него следующее состояние.

Итератором значений называется итератор, связанный с некоторым упорядоченным множеством. Итератор значений можно рассматривать как "указатель" на элемент множества. Для него имеются три операции: встать в начало множества, передвинуться на следующий элемент множества и выдать текущий элемент множества (вместо "выдачи" текущего элемента итератор значений может проделать с ним любую другую операцию).

Помимо итераторов значений, существуют комбинаторы. *Комбинатор* – это итератор, который управляет другими итераторами, то есть указывает им, в каком порядке передвигаться по состояниям. Состоянием комбинатора обычно является кортеж состояний подотчётных ему итераторов.

Пример. Пусть имеются итераторы значений `i1`, `i2`, связанные с множествами `{a,b,c}`,`{d,e}` соответственно. И пусть мы хотим построить итератор значений декартова произведения этих множеств.

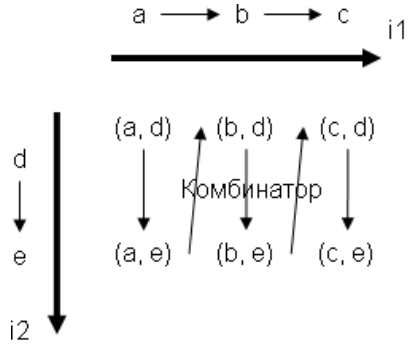

Рис.8. Пример комбинатора итераторов.

Используем для этого такой комбинатор (Рис. 8): его состояния – это тройка состояний итераторов `i1`, `i2`. Начальное состояние – тройка начальных

состояний итераторов `i1`, `i2`. Таким образом, чтобы перейти к начальному состоянию, наш комбинатор переводит итераторы `i1`, `i2` в начальное состояние. Чтобы перейти к следующему состоянию, комбинатор смотрит, в каком положении находится итератор `i2`. Если у этого итератора есть следующее состояние, то итератор в него переводится – получается новая пара состояний итераторов `i1`, `i2`, то есть новое состояние для комбинатора. Если у итератора `i2` нет следующего состояния, то комбинатор смотрит, есть ли следующее состояние у итератора `i1`; если есть, то итератор `i1` переводится в следующее состояние, а итератор `i2` – в начальное, снова получается новая пара состояний итераторов `i1`, `i2`, а значит и новое состояние комбинатора. Если же следующего состояния нет ни у итератора `i1`, ни у итератора `i2`, то это означает, что и комбинатор не имеет следующего состояния.

Другой важный пример – это комбинатор системы зависимых итераторов. Пусть нам нужно проитерировать пары букв и цифр, при этом для каждой буквы есть своё множество цифр, с которыми она может быть в паре. Пусть для буквы `a` – это множество `{0, 1}`, для буквы `b` – множество `{2, 3, 5}`, для буквы `c` – множество `{4}`. Заведём итераторы для множеств цифр и букв: итератор `i1` для множества `{a, b, c}`, итераторы `i2`, `i3`, `i4` – для множеств `{0, 1}`, `{2, 3, 5}`, `{4}` соответственно. Состоянием комбинатора теперь будет пара состояний двух других итераторов: итератора `i1` и итератора, соответствующего состоянию итератора `i1`. Работа комбинатора проиллюстрирована рис. 9. При переходе итератора букв в следующее состояние в зависимости от этого состояния выбирается итератор цифр. При переходе к другой букве итератор цифр будет другим.

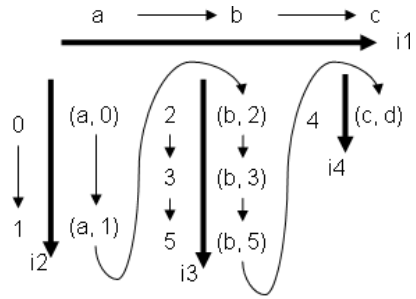

Рис.9. Комбинатор зависимых итераторов.

Можно определить комбинатор системы зависимых итераторов для итерации кортежей длины `n`. Точно так же, как было в примере с парами буква-цифра, итератор первого компонента кортежа последовательно проходит все свои состояния. В зависимости от его состояния выбирается итератор второго компонента кортежа, который также проходит все свои состояния (в это время состояние итератора первой компоненты кортежа менять нельзя). В зависимости от состояний итераторов первого и второго компонентов

выбирается итератор третьего компонента кортежа и т.д. Итераторы, от которых зависит выбор других итераторов, будем называть *осями зависимости*. Так итератор первого компонента кортежа – это первая ось зависимости, итератор второго компонента – это вторая ось зависимости и т.д. В приведенном выше примере одна ось зависимости – итератор букв.

7. Схема генерации

В данном разделе описан один из возможных подходов, использующих определённые ранее понятия. Тот генератор, который мы опишем, является итератором, то есть он строит деревья последовательно, по одному.

Как отмечалось раньше, для каждого типа вершин имеется свой генератор. Все эти генераторы появляются по требованию и зависят от уже построенной части дерева, то есть контекста.

Итак, опишем генератор вершин типа t. Контекст для строящейся вершины типа t – это внешний параметр генератора.

1. Генератор создаёт вершину-заготовку, в которой поля не установлены.
2. По контексту вершины генератор вычисляет граф зависимостей между полями вершины. Для этого он использует информацию из элементарных ограничений.
3. Из графа зависимостей вычисляется порядок построения полей: $f_1 \rightarrow f_2 \rightarrow \ldots \rightarrow f_n$.
4. Далее строится комбинатор зависимых генераторов значений полей, у которого первая ось зависимости – генератор независимого поля f_1, вторая ось зависимости – генератор поля f_2, зависимого от поля f_1, и т.д. При этом генераторы полей при переходе в новое состояние проставляют значение соответствующего поля вершины.
5. Генератор значений конкретного поля f_i создаётся так:
   - если поле – ребёнок c типа s, то генератор строит для него контекст, основываясь на контексте вершины-родителя и значений полей, от которых зависит поле f_i. Для полученного контекста строится генератор вершин типа s, который используется в качестве генератора значений поля;
   - если поле – атрибут, то генератор значений для него строится так:
     - для всех элементарных ограничений, наложенных на данный атрибут, вычисляются множества допустимых значений $M_1, M_2, \ldots, M_k$. Как было сказано ранее, они задаются итераторами;
     - строится итератор, перебирающий значения из объединения множеств $M_1, M_2, \ldots, M_k$, которые удовлетворяют всем элементарным ограничениям, наложенным на данный атрибут. Этот итератор и будет генератором значений данного атрибута;
   - если поле – список, то для него строится комбинатор зависимых итераторов, у которого первая ось зависимости – итератор длины списка, вторая – итератор первого элемента списка, третья – итератор второго элемента списка, и т.д. Этот комбинатор используется в качестве генератора значений списка. Генераторы значений элементов списка строятся так же, как и генераторы значений полей.

Заметим, что при генерации должны своевременно обновляться все аспекты состояний вершин, которые могут использоваться в элементарных ограничениях для вычисления множества допустимых значений.

8. Использование абстрактных моделей

В описании схемы генерации почти ничего не говорилось об использовании аспектов состояний вершин. Между тем, помимо снижения вычислительных затрат, этот механизм даёт весьма важные преимущества при решении задач снижения количества генерируемых данных, достижения критериев покрытия и целенаправленной генерации.

Для начала введём понятие абстрактной модели. *Абстрактная модель* объекта – это фактор-объект, получаемый с помощью абстрагирования от некоторых деталей. В качестве примера можно привести граф наследования классов в языках Java или C++. Граф наследования не содержит никакой информации о методах или полях классов, он является абстрактной моделью системы классов, для которой эта информация несущественна.

Абстрактная модель может иметь разные представления, так, например, граф наследования можно представить в виде списка рёбер, или в виде последовательности чисел, или, графически, в виде дерева. То есть модель эквивалентна некоторой части объекта, но может не совпадать с ней (Рис. 10).

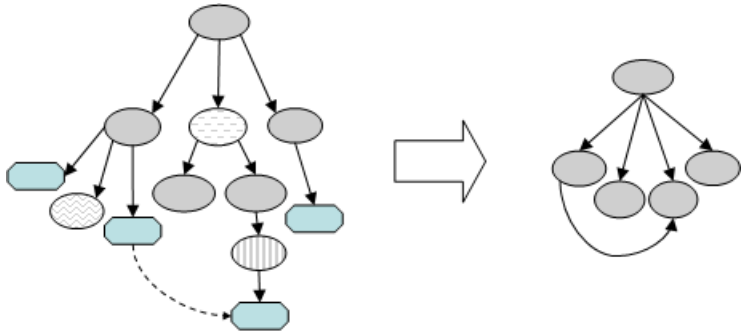

Рис.10. Абстрагирование от деталей.

По абстрактной модели можно построить много различных объектов, *реализующих* эту модель. Для графа наследования классов это можно сделать, меняя состав методов и полей.

Опишем теперь, как можно использовать абстрактные модели для генерации атрибутированных деревьев.

Пусть мы хотим построить группу деревьев, отвечающих некоторой абстрактной модели. Для этого мы немного расширим понятие состояния вершины. Добавим к нему новый аспект, который будет содержать описание абстрактной модели этой вершины. Автоматически в корневой вершине возникает элементарное ограничение на соответствие дерева имеющейся абстрактной модели. При переходе к генерации полей мы должны будем добавлять к их контексту абстрактные модели, моделирующие эти поля (они могут быть получены из абстрактной модели корневой вершины), а также ограничение соответствия значения поля его абстрактной модели (Рис. 11). Таким же образом можно моделировать не только дерево целиком, но и отдельные его части.

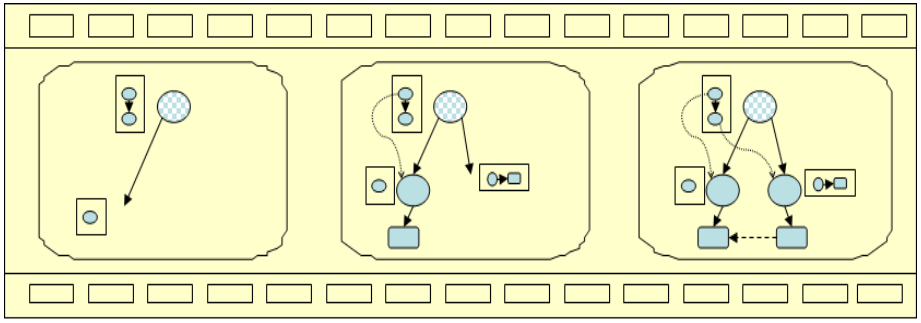

Рис.11. Использование абстрактной модели при генерации.

С помощью техники абстрактных моделей можно эффективно уменьшать количество деревьев, увеличивать целенаправленность генерации. Действительно, при использовании “переборного” способа генерации для построения дерева, отвечающего абстрактной модели, требуется перебор многочисленных деревьев, не соответствующих этой модели, а при применении описанного способа требуемое дерево получается с самого начала. Кроме того, использование абстрактных моделей – это удобный способ ограничения рекурсии.

9. Заключение

В статье представлена технология автоматической генерации тестовых данных сложной структуры, которая предполагает возможность тонкой настройки процесса генерации тестовых данных и оптимизации этого процесса под особенности функциональности конкретного тестируемого приложения. Эта возможность позволяет организовывать целенаправленную генерацию тестовых данных и, как следствие, генерировать множества тестовых данных относительно небольшого размера и с достаточно высоким качеством.

Предложенная технология автоматической генерации тестовых данных сложной структуры будет востребована, в первую очередь, в таких областях

разработки ПО, как телекоммуникационные приложения, в частности, интернет-приложения; приложения, работающие над базами данных; компонентное ПО, использующее XML-интерфейсы; языковые процессоры.

Литература

1. Quest Software Datafactory. http://www.quest.com/datafactory/
2. Canam Software Turbodata. http://www.turbodata.ca/
3. IBM DB2 Database Test Generator. http://www-306.ibm.com/software/data/db2imstools/db2tools/db2tdbg/
4. XML-XIG. http://sourceforge.net/projects/xml-xig
5. Sun XML Instance Generator. http://www.sun.com/software/xml/developers/instancegenerator/
6. XML Generator. http://www.stylusstudio.com/xml_generator.html
7. Paul Purdom. A sentence generator for testing parsers. // Behavior and Information Technology. 12(3):366–375, 1972.
8. B.A. Wichmann, B.Jones. Testing ALGOL 60 compilers. // Software Practice and experience. 6 (1976) 261–270.
9. A. Celentano, S. Crespi Reghezzi, P. Della Vigna, C. Ghezzi, G. Granata, and F. Savoretti. Compiler Testing using a Sentence Generator. // Software – Practice and Experience. 10:897–918, 1980.
10. A.G. Duncan, J.S. Hutchison. Using Attributed Grammars to Test Designs and Implementation. // In Proceedings of the 5th international conference on Software engineering. 170–178, 1981.
11. А.К. Петренко и др. Тестирование компиляторов на основе формальной модели языка // Препринт института прикладной математики им. М.В Келдыша, № 45, 1992.
12. A. Kalinov, A. Kossatchev, A. Petrenko, M. Posypkin, V. Shishkov. Using ASM Specifications for Compiler Testing // Proceedings of Abstract State Machines - Advances in Theory and Applications 10th International Workshop, ASM 2003.
13. С.В. Зеленов, С.А. Зеленова, А.С. Косачев, А.К. Петренко. Генерация тестов для компиляторов и других текстовых процессоров // Программирование, Москва. – 2003. – 29. – № 2. – с. 59-69.
14. С.В. Зеленов, С.А. Зеленова. Генерация позитивных и негативных тестов парсеров // Программирование, том. 31, №6, 2005, 25–40.
15. С.В. Зеленов, С.А. Зеленова, А.С. Косачев, А.К. Петренко. Применение модельного подхода для автоматического тестирования оптимизирующих компиляторов // CIT Forum, 2003. http://www.citforum.ru/SE/testing/compilers/
16. М.В. Архипова. Генерация тестов для модулей проверки статической семантики в компиляторах. // Труды ИСП РАН, т. 8, 2004, с. 59-76.
17. A.K. Petrenko. Specification Based Testing: Towards Practice // LNCS. – 2001. – 2244. – p. 287-300.
18. В.В. Кулямин, А.К. Петренко, А.С. Косачев, И. Б. Бурдонов. Подход UniTesK к разработке тестов. // Программирование, 29(6):25–43, 2003.
19. V. Kuliamin, A. Petrenko, A. Kossatchev, I. Bourdonov. UniTesK: Model Based Testing in Industrial Practice. // Proceedings of the 1-st European Conference on Model-Driven Software Engineering, Nurnberg, December 2003.