Нацеленная генерация данных для тестирования приложений над базами данных

Е. А. Костычев, В. А. Омельченко, С.В. Зеленов
{kostychev, vitaly, zelenov}@ispras.ru

Аннотация. Приложения, обрабатывающие большие массивы данных, являются широко распространенным типом программного обеспечения. В частности, такие приложения решают задачи интеграции данных в области интеграции корпоративных приложений. При решении таких задач используются специальные инструментальные среды, поддерживающие разработку, выполнение и мониторинг приложений, реализующих шаблон извлечения, трансформации и загрузки данных (Extract, Transform and Load - ETL). Специфика функционального тестирования таких приложений заключается в большом количестве возможных комбинаций входных данных. Подходы, реализованные в инструментах генерации данных для функциональной проверки приложений над базами данных, в лучших случаях основываются на схемах баз данных и запросах на SQL, используемых в тестируемых приложениях. Гарантированное покрытие функциональности тестируемого приложения при таких подходах может быть достигнуто только полным перебором возможных комбинаций исходных данных. При помощи предложенного в статье подхода к генерации данных возможно достижение покрытия функциональности приложения с близким к оптимальному объему данных (один набор данных для одной функциональной ветви приложения).

Ключевые слова: генерация тестовых данных; приложения над базами данных; функциональное тестирование; интеграция данных.

1. Введение

Постоянно растущая доступность все больших объемов для хранения информации в электронном виде и все больших вычислительных ресурсов для их обработки приводит к все большей распространенности приложений над все большими объемами данных в различных областях. Так в начале 90х годов прошлого века появился британский национальный корпус – специальным образом размеченный и обработанный большой массив, содержащий 100 миллионов слов в виде коллекции письменных и устных текстов за более чем двадцатилетний промежуток [1]. С тех пор такие национальные корпуса появились и продолжают появляться и пополняться для многих языков. Такие

корпуса языков не только делают более тривиальными многие традиционные задачи лингвистики, но и позволяют ставить и решать ранее невыполнимые задачи, прежде всего связанные с обработкой больших объемов данных [2]. С обработкой больших объемов данных так же связаны задачи в таких областях как статистика, социология, георазведка, некоторые разделы физики, молекулярная генетика, климатология, метеорология.

Хранение и обработка больших объемов данных является критической задачей и для современного корпоративного бизнеса. Корпоративные системы содержат огромные объемы взаимосвязанных данных о потребителях и заказчиках, поставщиках, партнерах, различных бизнес-транзакциях, об оборудовании и персонале, финансовых потоках. Большие объемы корпоративных данных, как правило, хранятся в структурированном виде в одной или нескольких базах данных под управлением одной или нескольких СУБД. Для обеспечения поддержки решений различных задач бизнеса зачастую требуется функциональность, которая может быть реализована только путем интеграции нескольких подсистем корпоративной системы, а иногда и интеграции с подсистемами внешних корпоративных систем. Часто в таких случаях возникает проблема доступа к большим массивам данных из разных хранилищ и имеющих разные форматы. Поэтому приложения репликации, обновления и синхронизации больших объемов данных являются широко распространенным частным случаем интеграционных корпоративных приложений.

Задачи таких приложений могут быть элементарными, но не всегда простыми в реализации, как, например, выборка полных или частичных данных, касающихся физического или юридического лица. Они могут быть и достаточно сложными, как по условиям выборки и агрегации входных данных, так и по формированию конечного результата, как, например, выставление счетов пользователям в соответствии с объемом реально потребленных услуг, тарифными планами и с учетом предоставляемых скидок. Высокая частота возникновения таких задач при решении проблемы интеграции корпоративных приложений привела к появлению специализированных платформ, предоставляющих универсальную, относительно платформ хранения данных, среду выполнения и инструменты для разработки приложений, эффективно решающих задачи извлечения, трансформации и загрузки больших массивов данных [3]. Далее такие приложения будем называть ETL-приложениями (от англ. Extract, Transform and Loading).

2. Постановка задачи

После разработки любого нового приложения или доработки уже существующего требуется проверка корректности его поведения на соответствие функциональным требованиям, реализующим необходимую

бизнес-логику. Бизнес-логику приложений над большими объемами данных, в том числе и ETL-приложений, можно детализировать следующим образом:

- извлечение из источника тех и только тех данных, которые удовлетворяют соответствующим условиям, определенным в требованиях;
- изменение в источнике значений тех и только тех данных, которые удовлетворяют соответствующим условиям, определенным в требованиях;
- трансформация (изменение значений, формата представления) входных данных в соответствии с требованиями;
- загрузка в приемник тех и только тех данных, которые должны быть загружены в соответствии с требованиями;
- изменение в приемнике значений тех и только тех данных, которые удовлетворяют соответствующим условиям, определенным в требованиях.

Проверка корректности поведения реализации осуществляется, как правило, при помощи функционального тестирования, которое состоит из выполнения приложения на специально подготовленных входных данных и сравнения полученных выходных данных с ожидаемыми результатами. В данной статье рассматривается задача генерации входных данных для функционального тестирования приложений над большими объемами данных (далее тестовые данные). В качестве примеров таких приложений рассматриваются ETL-приложения, содержащие все этапы, которые могут встречаться в приложениях над большими объемами данных: извлечение данных, их преобразование и загрузка.

Критерием качества функционального тестирования является полнота покрытия функциональных требований. Чем больше функциональных ветвей алгоритма, реализующего функциональные требования к тестируемому приложению, покрыто при выполнении тестов, тем качественнее тестирование. Из сформулированных выше подзадач функциональности ETL-приложений следует, что для покрытия функциональности таких приложений в массиве тестовых данных должны присутствовать такие данные, что при выполнении приложения:

- выборка данных из источника и их фильтрация происходит по всем возможным комбинациям условий, определенных в требованиях;
- при трансформации данных покрываются все возможные функциональные ветви алгоритма трансформации, соответствующего требованиям.
- обеспечивается проверка отсутствия лишних изменений данных в приемнике и источнике;

- обеспечивается проверка обработки специальных данных (пустые поля, строки ограниченной длины и т.д.).

Множество всех возможных комбинаций значений входных данных заведомо содержит комбинации необходимые для покрытия функциональности тестируемого приложения. Однако такой подход в реальных ситуациях очень быстро вызывает комбинаторный взрыв и приводит к неприемлемым затратам по времени генерации и по суммарному объему полученных тестов. Кроме того, в подавляющем большинстве случаев при проведении функционального тестирования полученное таким способом множество тестов заведомо является чрезмерным: очень многие тесты отличаются лишь такими значениями полей, которые не различимы с точки зрения функциональности. Это создает проблемы при анализе результатов тестирования: одна и та же ошибка будет проявляться на тысячах подобных «неразличимых» тестов, и все эти ситуации потребуют трудоемкого анализа.

Оптимальными с точки зрения покрытия функциональных требований являются такие данные, которые обеспечивают на каждое ветвление функциональности по одной возможной комбинации булевских выражений. В статье рассматривается метод получения тестовых данных, по объемам близких к оптимальному набору.

3. Обзор существующих инструментов

Большинство существующих инструментов генерации тестовых данных (DTM Data Generator [4], Turbo Data [5], DBMonster [6]) обеспечивают поддержку заполнения таблиц БД большим количеством синтаксически корректных данных и предоставляют следующие возможности:

- генерация случайных данных с возможностью задать интервал для числовых типов, длину для строк и формат генерируемых данных;
- генерация данных из списка, с возможностью указать процентное соотношение каждой строки из списка ко всем сгенерированным строкам;
- генерация данных путем их выбора из другой таблицы;
- генерация данных путем их выбора из файла;
- генерация автоувеличивающихся данных, с указанием начального значения и шага;
- генерация данных из существующих библиотек;
- генерация случайных данных по маске;
- генерация данных на основе возвращаемого результата SQL – выражений;
- генерация данных для зависимых таблиц;

- генерация данных на основе внешних процедур генерации, пользователь имеет возможность создавать свои процедуры.

Некоторые инструменты (AGENDA [7], HTDGen [8]) поддерживают генерацию данных не только на основе ограничений, заданных пользователем или схемой, но и на основе SQL-запросов тестируемого приложения. В этом случае гарантируется, как минимум, что SQL-запросы, используемые в реализации приложения, будут возвращать не пустой результат. Однако при генерации данных на основе реализации нет гарантии покрытия всех требуемых функциональных ветвей, так как реализация может содержать ошибочные ветви или не содержать требуемые. Так же такой подход не обеспечивает покрытия всех требуемых функциональных ветвей трансформации и фильтрации данных.

В общем случае, используя доступные инструменты, гарантировать полное покрытие всех функциональных ветвей тестируемого приложения возможно только полным перебором комбинаций значений полей, в зависимости от которых происходят ветвления при выборке, трансформации и фильтрации данных в приложении. Как было сказано выше, полный перебор вызывает комбинаторный взрыв и приводит к неприемлемым затратам по времени генерации и по суммарному объему полученных тестов.

Часто, с точки зрения покрытия функциональности, значения нескольких полей бывают связаны некоторой семантикой, которая выражается в том, что значение одного поля зависит от значений каких-то других полей. При этом, как правило, перебор комбинаций значений, удовлетворяющих этой семантике, сильно снижает общее количество вариантов. Например, пусть имеются два поля A и B, принимающие целые значения в диапазоне [1 .. 30], причем значение поля B зависит от значения поля A следующим образом (см. Рис. 1):

- если $1 \le A \le 10$, то $B = 1$;
- если $11 \le A \le 20$, то $11 \le B \le 20$;
- если $21 \le A \le 30$, то $B = A$.

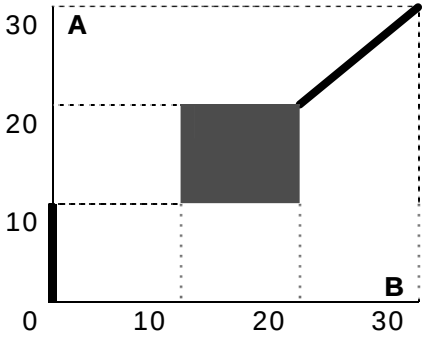

Рис. 1. Пример зависимости значений полей.

Если перебирать не все возможные комбинации полей, а только удовлетворяющие перечисленным условиям, то количество получаемых в результате комбинаций снизится практически на порядок (с 900 до 120).

Кроме того, функциональность тестируемого приложения может разбиваться на несколько независимых друг от друга аспектов. В этом случае существенно снизить количество перебираемых комбинаций значений полей позволяет так называемый диагональный комбинатор, перебирающий такое множество S кортежей, что для любого i от 1 до n и для любого s из S_i существует такой кортеж $(s_1, \ldots, s_n)$ в S, что $s = s_i$, где S_i – это множество значений, итерируемое i-м подчиненным итератором. Такой метод гарантирует, что в полученном множестве тестов будет содержаться каждое значение каждого поля. При этом общее количество тестов будет существенно меньше, чем в случае использования прямого произведения: вместо произведения мощностей множеств значений всех полей получится лишь максимальное из значений этих мощностей.

Инструмент Pinery [9], [10], [11], разработанный в ИСП РАН, предназначен для генерации тестов, обладающих сложной структурой. Генерация тестов с данной синтаксической структурой настраивается путем задания в терминах этой структуры ограничений на фрагменты тестов. При этом ограничения могут быть условными, т.е. зависеть от значений каких-то полей.

При задании ограничения на значения поля в Pinery можно пользоваться, в частности, следующими средствами:

- указать непосредственные значения;
- задать функцию вычисления в зависимости от значений других полей;
- указать принадлежность некоторому множеству, например:
  - отрезку целых чисел;
  - множеству значений другого поля в данном тесте (например, в случае задания значений для внешнего ключа).

Важным ограничением другого рода в Pinery является указание способа комбинирования полей. В частности, наряду с использованием прямого произведения в Pinery есть возможность использовать диагональный комбинатор, а также указать наличие зависимости между значениями полей, причем сами эти комбинаторы могут образовывать иерархию.

Эти возможности позволяют в Pinery более тонко, по сравнению с другими инструментами, настраивать процесс генерации и получать тестовые данные по объемам близкие к оптимальным.

4. Описание метода

Метод нацеленной генерации тестовых данных реализует предложенную в ИСП РАН технологию разработки тестов UniTESK [12], [13], [14], основанную на использовании формальной модели тестируемой системы и

общей схемы, представленной на Рис. 2. Краеугольным камнем этой схемы является формализация требований [15] к тестируемой системе.

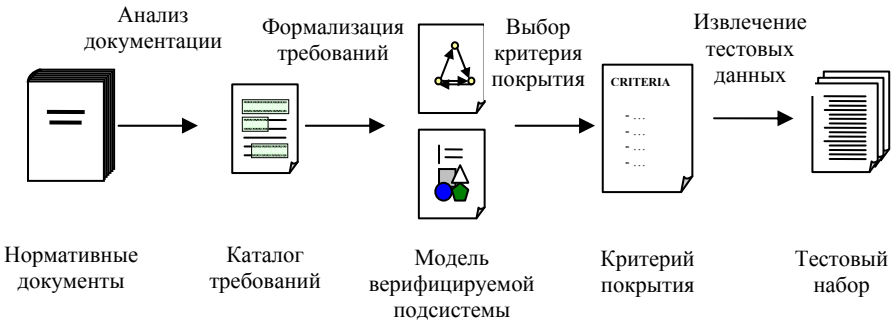

Рис. 2. Общая схема нацеленной генерации тестов.

Эта схема состоит из нескольких этапов. На первом этапе требования к тестируемой системе извлекаются из нормативных документов и систематизируются. В результате получается каталог требований, в котором требования сформулированы максимально однозначно, они классифицированы, и, возможно, установлены связи между отдельными требованиями. Каталог требований используется на последующих этапах.

Второй этап нацелен на представление требований в формальном виде. Требования из каталога записываются с использованием того или иного математического формализма. Такая запись требований называется формальной моделью.

На третьем этапе в терминах построенной модели формулируется критерий покрытия. Для этого в множестве входных данных выделяется конечное число подмножеств (возможно, пересекающихся между собой), объединение которых совпадает со всем множеством, так чтобы было единообразно функциональное поведение тестируемой системы на входных данных из одного подмножества. Отправной точкой для выделения подмножеств входных данных с единообразным поведением приложения являются условия, содержащиеся в требованиях каталога. Однако, кроме того, конкретная семантика этих условий может явиться почвой для формулирования дополнительных гипотез о работе тестируемого приложения, которые индуцируют некоторое подразбиение входных данных на более мелкие подмножества.

На четвертом этапе на основе построенной модели и в соответствии со сформулированным критерием покрытия автоматически генерируются тестовые данные. В нашем подходе автоматизированная генерация

проводится с использованием инструмента Pinery. Для запуска генератора ему требуется предоставить формальное описание модели тестовых данных, а также настройки генерации, нацеленные на достижение критерия покрытия.

Вопросы, связанные с прогоном тестов, анализом результатов, обнаружением дефектов и их исправлением на тестируемой системе, выходят за рамки данной статьи.

5. Пример

Проиллюстрируем, как применяется предложенный метод, на примере тестирования приложения, работающего в рамках упрощенной системы банковского кредитования.

Нормативными документами здесь являются неформальное описание системы кредитования и работы приложения:

- Клиент с определенным типом регистрируется в базе, получая идентификатор.
- Клиент получает в кредит определенную сумму денег, которую необходимо выплатить в течение несколько месяцев.
- Клиент ежемесячно должен вносить платеж, сумма которого вычисляется исходя из оставшегося долга, деленного на количество месяцев, оставшихся до окончания срока действия кредита.
- На клиента может быть наложен штраф, если сумма очередного внесенного взноса меньше суммы необходимого ежемесячного платежа, а также клиенту может быть начислена премия, если сумма внесенного взноса больше суммы ежемесячного платежа. Размер штрафов и премий зависит от типа клиента и от типа кредита.
- Если клиент желает ежемесячно получать уведомления о текущем состоянии кредита, он может оставить свой адрес электронной почты.
- Приложение запускается в конце месяца и осуществляет следующие действия:
    A. рассчитывает штраф/премию клиента и списывает с долга внесенную сумму и рассчитанный штраф/премию по следующим правилам:
        I. Если взнос, внесенный клиентом, меньше суммы ежемесячного платежа, то на абонента налагается штраф:
            1. Если абонент имеет статус «VIP», то штраф за просроченный платеж равен 0.
            2. Если абонент имеет статус «USUAL», то он облагается штрафом за просроченный платеж с коэффициентом 0.03.

II. Если взнос, внесенный клиентом, не меньше, чем сумма ежемесячного платежа, и не больше общей суммы долга, то абонент получает премию:
    1. Если абонент имеет статус «VIP» и тип кредита «B» или имеет статус «USUAL» и тип кредита «A», то коэффициент при расчете премии равен 0.02.
    2. Если абонент имеет статус «VIP» и тип кредита “A”, то коэффициент при расчете премии равен 0.05.
    3. Если абонент имеет статус «USUAL» и тип кредита “B”, то коэффициент при расчете премии равен 0.01.
III. Если взнос, внесенный клиентом, больше общей суммы долга, то приложение должно сигнализировать о необходимости возврата лишних денег клиенту.

B. Если для клиента определен адрес электронной почты, то после вычисления оставшегося долга приложение отправляет клиенту уведомление о текущем состоянии кредита.

Короткое имя	Полное имя	Тип	Описание
MP	MONTH_PAYMENT	NUMBER	Ежемесячный взнос
D	DEBT	NUMBER	Общая сумма оплаты
P	PAYMENT	NUMBER	Внесенная сумма
CL	CLIENT_TYPE	CHAR	Тип клиента: “V” – VIP клиент “U” – обычный клиент (USUAL)
CR	CREDIT_TYPE	CHAR	Тип кредита: “A” – тип “A” “B” – тип “B”
E	EMAIL	STRING	Адрес электронной почты

Табл. 1. Параметры работы системы кредитования.

На **первом этапе** из нормативной документации извлекается каталог требований к тестируемой системе. В нашем случае каталогом требований являются пункты правил поведения тестируемого приложения (A с подпунктами и B). В целях более удобного восприятия, требования к поведению приложения могут быть полуформально представлены в виде следующей схемы, приведенной на Рис. 3.

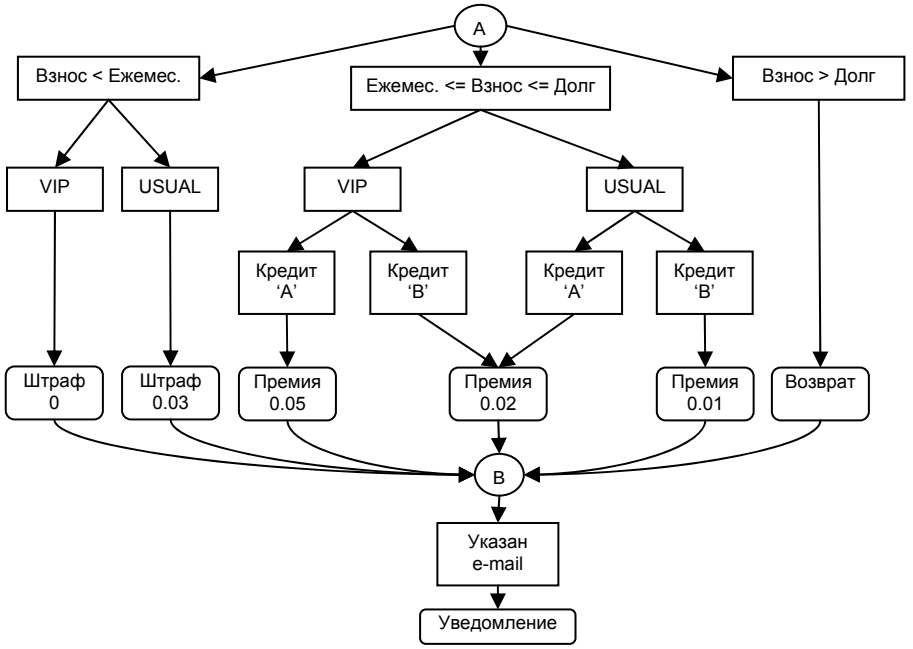

Рис. 3. Схематичное изображение требований к системе кредитования.

Приступим ко **второму этапу** – построению формальной модели. Параметры работы тестируемого приложения представлены в Табл. 1.

Условие консистентности данных в терминах введенных параметров таково: MP <= D. Требования к поведению приложения формально описываются так:

A.I.1. P < MP **&&** CL = “V” **=>** Штраф 0;

A.I.2. P < MP **&&** CL = “U” **=>** Штраф -0.03;

A.II.1. MP <= P <= D **&&** (CL = “V” **&&** CR = “B” || CL = “U” **&&** CR = “A”)

=> Премия 0.02;

A.II.2. MP <= P <= D **&&** CL = “V” **&&** CR = “A” **=>** Премия 0.05;

A.II.3. MP <= P <= D **&&** CL = “U” **&&** CR = “B” **=>** Премия 0.01;

A.III. P > D **=>** Возврат.

B. E ≠ “” **=>** Уведомление.

Перейдем к **третьему этапу** – формулировке критерия покрытия. Для этого, помимо условий, содержащихся в требованиях, мы введем дополнительно следующие гипотезы относительно поведения тестируемого приложения:

1. При выполнении условия MP <= P <= D приложение может вести себя по-разному в следующих случаях:
   - MP = P = D;
   - MP = P < D;
   - MP < P < D;
   - MP < P = D.
2. При выполнении условий требования A.II.1 приложение может вести себя по-разному в следующих случаях:
   - CL = “V” **&&** CR = “B”;
   - CL = “U” **&&** CR = “A”.

Формулировать правила разбиения области входных данных на подмножества с единообразным поведением тестируемого приложения мы будем последовательно для каждого из параметров. Начнем с параметров MP, D, P, CL и CR, относящихся к требованиям из группы A.

Исходя из гипотезы 1, область входных данных в части параметров MP и D разбивается на два подмножества:

- MP = D;
- MP < D.

Исходя из условий требований, в случае MP = D область входных данных в части параметра P разбивается на три подмножества:

- P < MP;
- MP = P = D;
- P > D.

Исходя из условий требований и из гипотезы 1, в случае MP < D область входных данных в части параметра P разбивается на пять подмножеств:

- P < MP;
- MP = P < D;
- MP < P < D;
- MP < P = D;
- P > D.

Исходя из условий требований группы A.I, в случае P < MP область входных данных в части параметров CL и CR разбивается на два подмножества, соответствующих всем возможным значениям параметра CL.

Исходя из условий требований группы A.II и из гипотезы 2, в случае MP <= P <= D область входных данных в части параметров CL и CR разбивается на

четыре подмножества, соответствующих всем возможным сочетаниям значений этих параметров.

Исходя из условия требования A.III, в случае P > D область входных данных в части параметров CL и CR составляет единственное подмножество.

Перейдем к параметру E, относящемуся к требованию B. Исходя из условия требования B, область входных данных в части параметра E разбивается на два подмножества: с пустым и непустым значением этого параметра.

Разбиение области входных данных в части параметров, относящихся к требованиям из группы A, индуцируется прямым произведением (с учетом зависимостей) разбиений областей данных каждого из этих параметров.

Поскольку функциональность приложения разбивается на два независимых аспекта A и B, то разбиение всей области входных данных индуцируется диагональным комбинированием разбиений областей данных, соответствующих каждой из этих групп.

Перейдем к **четвертому этапу** – автоматизированной генерации тестовых данных с использованием инструмента Pinery.

Для генерации тестовых данных на Pinery прежде всего необходимо задать формальное описание схемы базы данных (например, с использованием DDL-подмножества языка SQL). После этого в терминах заданной схемы БД описываются ограничения на генерируемые данные. В рассматриваемом примере база данных содержит одну таблицу CREDITS, состоящую из пяти полей: MONTH_PAYMENT, DEBT, PAYMENT, CLIENT_TYPE, CREDIT_TYPE. Далее мы будем ссылаться на эти поля по их коротким именам.

Ограничения в этом примере бывают двух видов:

- ограничение на значения конкретного поля;
- ограничение на способ комбинирования нескольких полей.

Начнем с описаний ограничений первого вида.

Для получения ситуаций MP = D и MP < D достаточно положить, например, MP = 6, а D = 6 и D = 30. Для получения ситуаций, когда значение поля E пусто и непусто, достаточно положить, например, E = “” и E = “…@…” (некоторый адрес). Для задания этих значений в Pinery требуется описать следующие ограничения, задающие список конкретных значений полей[1]:

```
MP = { 6 };
D = { 6, 30 };
E = { “”, “…@…” };
```

[1] В Pinery ограничения записываются в XML-форме. Здесь, в целях экономии места, а также для повышения понятности записи, мы приводим ограничения, записанные на полу-формальном псевдо-коде.

Это примеры так называемых *безусловных* ограничений – значения этих полей *во всех случаях* являются такими, как это перечислено в соответствующем ограничении. Однако, не всегда можно обойтись лишь безусловными ограничениями. Например, множество значений поля P зависит от значений полей MP и D. Таким образом, нам потребуются два условных ограничения: для случаев MP = D и MP < D.

Для получения ситуации P < MP достаточно положить P = MP – 1; для получения ситуации P > D достаточно положить P = D + 1; для получения ситуации MP < P < D при MP < D достаточно положить P = (MP + D)/2. В результате, ограничения для поля P запишутся так:

```
P[ MP<D ] = { MP - 1, MP, (MP + D)/2, D, D + 1 };
P[ MP=D ] = { MP - 1, D, D + 1 };
```

Множество значений полей CL и CR зависит от значений полей P, MP и D:

```
CL[ P<MP ] = { “V”, “U” };
CR[ P<MP ] = { “A” };
CL[ MP<=P && P<=D ] = { “V”, “U” };
CR[ MP<=P && P<=D ] = { “A”, “B” };
CL[ P>D ] = { “V” };
CR[ P>D ] = { “A” };
```

Перейдем к рассмотрению того, как задается комбинатор значений полей для генерации кортежей таблицы CREDITS. Начнем с того, что множества значений полей CL и CR во всех случаях можно комбинировать методом прямого произведения:

```
Product( CL, CR )
```

Аналогично, множества значений полей MP и D можно комбинировать методом прямого произведения:

```
Product( MP, D )
```

Поскольку значения полей CL и CR зависит от значений полей P, MP и D, а значение поля P в свою очередь также зависит от значений полей MP и D, то наличие таких зависимостей выражается использованием специального «зависимого» комбинатора, который задает комбинации полей, относящихся к требованиям из группы A:


```
Depend(  Product( MP, D )
      => P
      => Product( CL, CR )
      );
```

Как было указано выше, комбинации полей, относящихся к требованиям из групп A и B, производятся методом диагонали, который задается комбинатором значений полей для генерации кортежей таблицы CREDITS:

```
combinator( CREDITS ) =
    Diagonal( Depend(
Product( MP, D )
                    => P
                    =>
Product( CL, CR )
                    )
            , E
            );
```

Итоговые тестовые данные представлены в следующей Табл. 2.

D	6	6	6	6	6	6	6	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30
MP	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
P	5	5	6	6	6	6	7	5	5	6	6	6	6	18	18	18	18	30	30	30	30	31
CL	V	U	V	V	U	U	V	V	U	V	V	U	U	V	V	U	U	V	V	U	U	V
CR	A	A	A	B	A	B	A	A	A	A	B	A	B	A	B	A	B	A	B	A	B	A
E		@		@		@		@		@		@		@		@		@		@		@

Табл. 2. Тестовые данные для системы кредитования.

Заметим, что применение зависимого комбинирования позволило задать единую конфигурацию генератора вместо двух (для случаев MP = D и MP < D). А применение зависимого и диагонального комбинирования сократило количество тестовых данных более, чем на 65%, по сравнению с общим прямым произведением: в нашем случае получилось 22 кортежа, в то время как прямое произведение дает 64 (3*2*2*2 = 24 для случая MP = D, плюс 5*2*2*2 = 40 для случая MP < D).

Экономия увеличивается при увеличении количества возможных значений полей. Так, например, в случае добавления еще одного типа клиента и одного типа кредита описанный подход дает уже экономию в 70% (3*3*3*2+

5*3*3*2 = 144 для прямого произведения против (3 + 1*3*3 + 1) + (3 + 3*3*3 + 1) = 44 в нашем подходе).

Еще больше заметно увеличение экономии при добавлении полей, относящихся к другим независимым аспектам работы приложения. Пусть, например, для взноса указывается тип оплаты с двумя возможными значениями: 'O' – оплата по безналичному расчету и 'E' – оплата наличными. Тогда количество записей в нашем подходе не увеличивается (поскольку применяется диагональный комбинатор), а при использовании прямого произведения количество вариантов удваивается.

6. Заключение

В статье предложен способ генерации данных для функционального тестирования приложений над большими объемами данных, нацеленных на покрытие функциональных ветвей приложения. Отличие предложенного подхода от реализованных в существующих инструментах состоит в том, что полученные тестовые данные, обеспечивая покрытие функциональности тестируемого приложения, являются более оптимальными относительно временных затрат на тестирование, а также временных и трудозатрат на анализ результатов тестирования.

Генерация данных при помощи декартова комбинирования всех полей обеспечивает полное покрытие, но практически всегда является избыточной, причем избыточность стремительно нарастает при увеличении количества комбинируемых полей и подмножеств их значений, от которых зависит поведение приложения.

Предлагаемый подход основан на формализации и анализе требований к тестируемой системе. Использование наряду с прямым произведением комбинаторов зависимых множеств и диагональных комбинаторов позволяет очень существенно сократить тестовый набор без потери качества тестирования и в результате получить тестовые данные, по объемам близкие к оптимальным.

Подход инструментально поддерживается генератором данных сложной структуры Pinery.

Литература

[1] British National Corpus (BNC), 2009, http://www.natcorp.ox.ac.uk/corpus/index.xml
[2] . N. Oostdijk and P. Haan, Corpus-Based Research into Language. In honour of Jan Aarts, Amsterdam/Atlanta, GA, 1994, VII.
[3] M. L. Songini, QuickStudy: Extract, Transform and Load (ETL), 2004, Computerworld, http://www.computerworld.com/s/article/89534/QuickStudy_ETL
[4] DTM Data Generator, test data generator for database testing purposes, 2004, http://www.sqledit.com/dg/
[5] Test Data Generator - TurboData - "Out of the Box", 2009, http://www.turbodata.ca/
[6] DBMonster - The dbMonster home page – About, 2003, http://dbmonster.kernelpanic.pl/

[7] D. Chays, Y. Deng, P.G. Frankl, E.J. Weyuker, An AGENDA for testing relational database applications, Software testing, verification and reliability, 2004, VOL 14; PART 1, pages 17-44
[8] C. Binnig, D. Kossmann, E. Lo, Testing database applications, 2006, Proceedings of the 25th ACM SIGMOD international conference on management of data / Principles of database systems, Chicago
[9] А.В. Демаков, С.В. Зеленов, С.А. Зеленова. Генерация тестовых данных сложной структуры с учетом контекстных ограничений. // Труды ИСП РАН, 2006, том 9, 83–96.
[10] А.В. Демаков, С.В. Зеленов, С.А. Зеленова. Генератор сложных данных Pinery: реализация новых возможностей UniTESK // Труды ИСП РАН, Москва, 2008, т. 14, часть 1, 119–136.
[11] А.В. Демаков, С.В. Зеленов, С.А. Зеленова. Использование абстрактных моделей для генерации тестовых данных сложной структуры // Программирование, Москва, 2008, том. 34, № 6, 341–350.
[12] В. В. Кулямин, А. К. Петренко, А. С. Косачев, И. Б. Бурдонов. Подход UniTesK к разработке тестов. Программирование, 29(6):25-43, 2003.
[13] 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, 2004, том. 5, 121–156.
[14] UniTESK, индустриальная технология надежного тестирования, 2006, http://unitesk.ru/
[15] В. В. Кулямин, Н. В. Пакулин, О. Л. Петренко, А. А. Сортов, А. В. Хорошилов. Формализация требований на практике. // Препринт ИСП РАН. М.: ИСП РАН, 2006.