Тестирование трансляторов: проблема построения оракула для генератора кода

В.В. Гингина, С.В. Зеленов, С.А. Зеленова

Аннотация. В работе описан общий подход к построению автоматического оракула для тестирования генераторов кода в трансляторах текстов на формальных языках, а также предложена инструментальная поддержка для практического использования этого подхода. Приводятся результаты практического применения описанного подхода к тестированию генератора кода транслятора описаний схем баз данных на языке SQL.

1. Введение

Развитие компьютерной индустрии и информационных технологий привело к тому, что во все большем числе областей бизнеса обработка документов производится автоматически с помощью компьютеров. Чтобы быть пригодными для компьютерной обработки, данные оформляются в виде формального текста, т.е. текста на некотором формальном языке.

Транслятор — это программа, осуществляющая обработку формального текста и перевод его в некоторое другое представление. Примерами трансляторов являются компиляторы и интерпретаторы языков программирования, XML-процессоры, браузеры HTML-страниц, системы поддержки специфицирования и моделирования, текстовые процессоры и издательские системы, серверы запросов СУБД и проч. Трансляторы используются при разработке ПО, при организации работы сетевых приложений, при создании разного рода распределенных информационных систем и т.д. Широкое распространение трансляторов, их повсеместное как прямое, так и косвенное использование в различных областях компьютерной индустрии, включая критические, обуславливает высокие требования к качеству трансляторов.

Существует много разных подходов к достижению качества программного обеспечения (ПО). Одним из подходов является проведение аналитического доказательства корректности ПО. Вопросам теоретического обоснования корректности поведения компиляторов на основе использования различных логических исчислений посвящены, например, работы [3], [15], [16]. Однако,

при разработке программных систем высокой сложности, какими являются реальные трансляторы, более практичным подходом является тестирование [1].

При тестировании встают две основные проблемы:

1. Проблема систематического создания тестовых данных для испытания работы тестируемой системы во всех различных ситуациях.
2. Проблема построения оракула для вынесения вердикта о корректности работы тестируемой системы.

Ввиду сложности и насыщенности современных формальных языков, для которых разрабатываются трансляторы, очень велики объемы добротных наборов тестов и сложность анализа результатов тестирования. Поэтому очень актуальна проблема автоматизации как процесса создания тестовых данных, так и процедуры вынесения вердикта. Наиболее многообещающим в плане автоматизации подходом является тестирование на основе формальных спецификаций и моделей [13].

В трансляторах традиционно выделяют следующие функциональные блоки: front-end, осуществляющий синтаксический разбор и семантический анализ входного текста и построение его внутреннего представления, и back-end, осуществляющий дополнительный анализ входных данных, возможно, некоторые оптимизирующие преобразования, а также собственно генерацию выходного представления (см. Рис. 1).

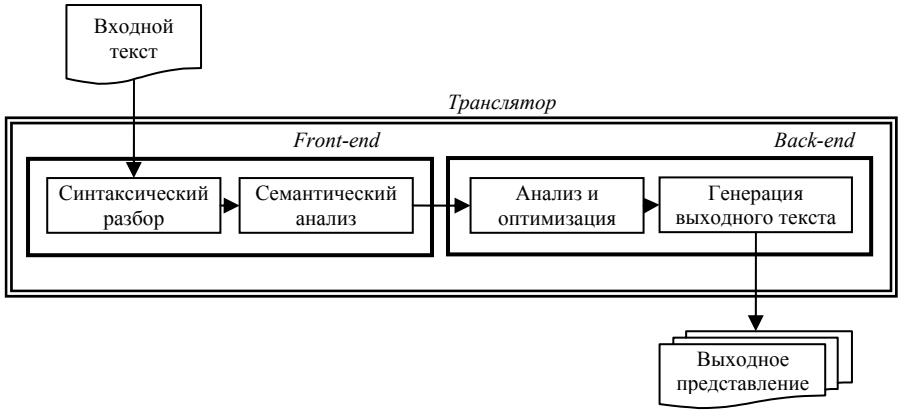

Рис. 1. Функциональные блоки транслятора

Тестирование сложных программных систем нужно проводить как на системном уровне, так и на уровне отдельных модулей и блоков и на уровне проверки отдельных аспектов функциональности. В настоящее время известно

большое количество исследовательских работ, посвященных тестированию отдельных функциональных блоков трансляторов (см. раздел 5).

Можно утверждать, что для блоков, входящих во front-end транслятора, если рассматривать эти блоки как булевские функции, существуют приемлемые подходы к автоматизации их тестирования — систематической генерации тестов и анализу результатов работы тестируемой системы (см. [2], [4], [9], [14], [17], [21]). В вопросе проверки качества работы back-end'а можно утверждать, что в настоящее время существуют удовлетворительные подходы к решению проблемы автоматического построения тестовых данных (см. [6], [7], [22], [23]). Что же касается проблемы оракула при тестировании back-end'а, то существующие практически применимые подходы основаны на идее анализа наблюдаемого поведения выполнения результата трансляции, и таким образом применимы лишь к компиляторам и интерпретаторам (см. [5], [6], [7], [10], [11], [12], [22]).

Однако существует большой класс трансляторов, которые продуцируют «неисполнимые» тексты, а значит, к ним не применимы имеющиеся подходы к решению проблемы оракула. В настоящей статье предлагается подход к автоматическому построению тестов для трансляторов, обеспечивающий решение проблемы оракула для генератора выходного представления.

Статья состоит из введения, пяти разделов и списка литературы. В первом разделе приводятся предварительные сведения, касающиеся подхода к генерации тестов для оптимизаторов на основе моделей. Во втором разделе дается описание предлагаемого в настоящей статье подхода. В третьем разделе рассматривается пример применения предлагаемого подхода на практике. В четвертом разделе приводится обзор близких работ. В заключение работы обсуждаются достоинства и недостатки, а также направления дальнейшего развития предлагаемого подхода.

2. Предварительные сведения

В работе [22] предложен подход к автоматическому тестированию оптимизирующих компиляторов на основе моделей. Модель строится на основе абстрактного описания алгоритма заданной оптимизации.

Алгоритм оптимизации формулируется с использованием *терминов*, обозначающих сущности некоторого подходящего абстрактного представления программы, такого как граф потока управления, граф потока данных, таблица символов и пр. Оптимизатор для осуществления своих трансформаций ищет сочетания сущностей абстрактного представления программы, которые удовлетворяют некоторым *шаблонам* (например, наличие в программе циклов, наличие в теле цикла конструкций с определенными свойствами, наличие в процедуре общих подвыражений, наличие между инструкциями зависимости данных некоторого вида и пр.).

После анализа алгоритма строится модель: каждому термину соответствует свой элемент модели, причем элементы модели могут путем связывания друг с другом образовывать структуры, соответствующие шаблонам.

Процесс генерации тестов осуществляется следующим образом. Перебираются (*итерируются*) различные модельные структуры, которые потом отображаются (*меппируются*) в соответствующие программы на входном языке компилятора (см. Рис. 2).

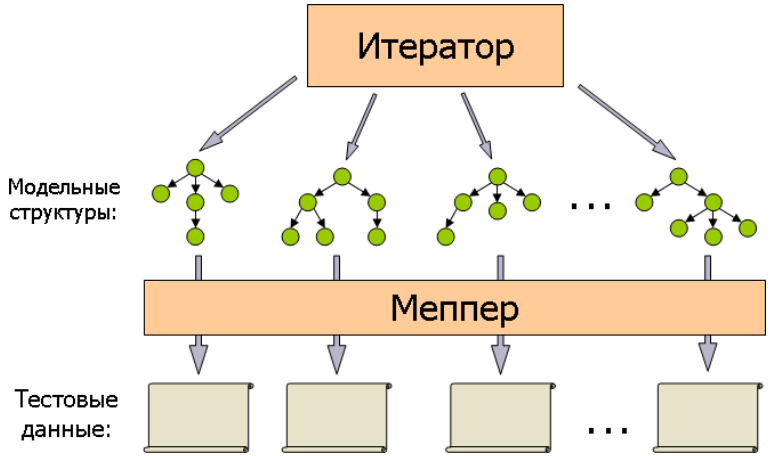

Рис. 2. Общая схема работы генератора тестов

Такой генератор дает возможность получать тесты на исходном языке, в которых присутствуют конструкции, соответствующие выделенным терминам, в различных комбинациях при минимальном содержании других конструкций языка. Таким образом, генератор оказывается «нацелен» на создание тестов для тестирования именно заданного оптимизатора.

В рамках этого подхода оракул проверяет только сохранение семантики выполнения программы во время оптимизации. Для этого в качестве тестовых воздействий на оптимизатор генерируются такие программы, семантика выполнения которых полностью представляется их трассой. Такое свойство тестов позволяет свести задачу проверки сохранения семантики к сравнению трассы выполнения откомпилированного теста с некоторой эталонной трассой.

Описанный подход успешно применялся для генерации тестов не только для тестирования компиляторов языков программирования, но также для тестирования оптимизаторов графических моделей [23], реализации протокола IPv6, подсистемы построения отчетов в биллинговой системе, обработчиков XML-логов [18].

3. Построение оракула для генератора кода

В подходе к построению оракула для тестированию генераторов кода в трансляторах, предлагаемом в настоящей работе, реализуется классическая идея сравнения реального результата работы тестируемой системы с ожидаемым результатом [1].

В качестве основы для организации тестирования применяется модельный подход, описанный выше в предварительных сведениях. Однако в нашем случае из каждой модельной структуры генератор тестов кроме собственно входных тестовых данных должен создавать также и ожидаемый эталонный результат трансляции — либо полностью, либо лишь некоторую абстрактную выжимку, своего рода систему показателей, характеризующих ключевые свойства выходных данных транслятора. Для этого генератор тестов должен содержать два разных меппера: один для отображения модельной структуры во входные тестовые данные, другой для ее отображения в эталонную систему показателей выходных данных (см. Рис. 3).

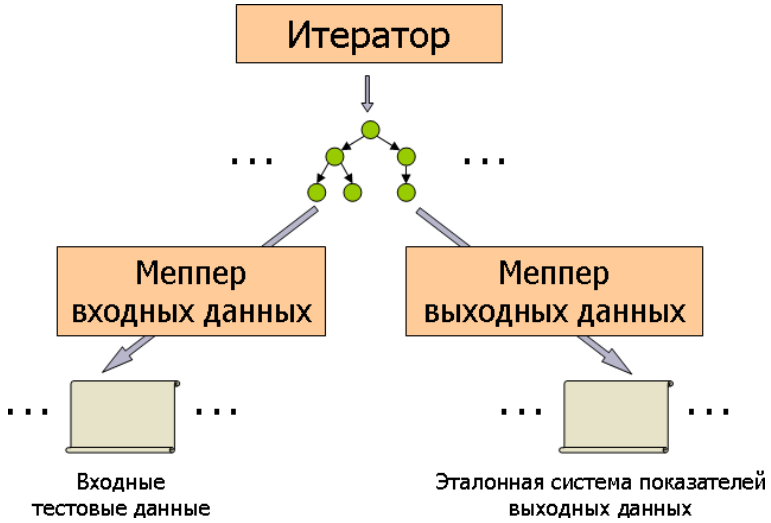

Рис. 3. Схема генерации входных данных и эталонного результата

Для того чтобы было возможно организовать два разных отображения из одной модельной структуры, модель должна строиться с учетом информации не только о входном представлении данных (см. разд. 1), но и об их выходном представлении.

В простейшем случае, когда транслятор лишь «переводит» термины входного представления в какие-то соответствующие термины выходного

представления, оказывается, что одна и та же модель пригодна для моделирования как входных, так и выходных данных.

В случае же, когда транслятор сначала осуществляет некоторый анализ входных данных, а лишь затем генерирует выходные данные, содержащие результаты проведенного анализа, модель следует обогатить дополнительными элементами, которые представляют собой соответствующие результаты анализа. При этом итерацию модельных структур следует организовать так, чтобы элементы модели, соответствующие «результатам анализа», конструировались непосредственно в процессе итерации — это возможно, например, если организовать итерацию одних элементов модели в зависимости от уже созданного значения другого элемента модели.

Далее мы будем говорить о предложенной схеме генерации тестовых данных в контексте тестирования не всего генератора кода в целом, а лишь некоторого ограниченного его аспекта, поскольку это существенно упрощает модель и облегчает разработку итераторов и мепперов.

Итак, генератор тестов вместе с входными данными генерирует также соответствующую эталонную систему показателей выходных данных. При использовании такой схемы генерации тестовых данных анализ правильности работы генератора кода в трансляторе состоит в проверке результата работы транслятора на соответствие созданной генератором тестов эталонной системе показателей (см. Рис. 4).

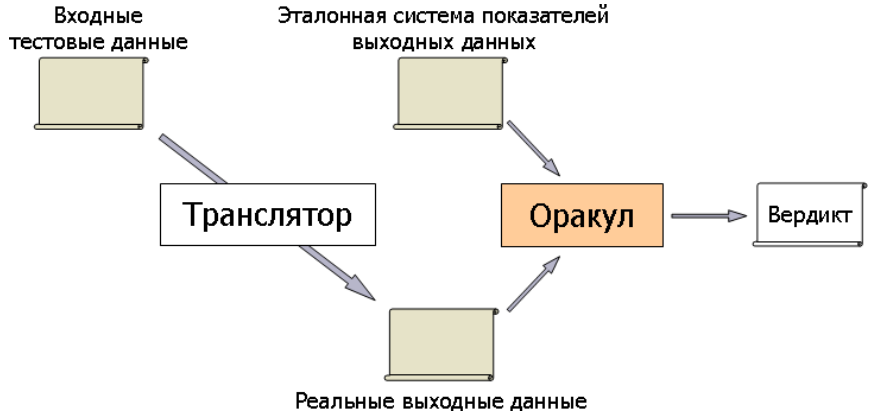

Рис. 4. Схема работы оракула

Таким образом, общий процесс тестирования состоит из следующих шагов:

1. Анализируется документация на генератор кода в трансляторе, и выделяются те аспекты его работы, которые требуется протестировать;

2. Для каждого аспекта выделяются термины и шаблоны входных данных, строится модель;
3. В рамках данного аспекта анализируется алгоритм генерации кода, модель обогащается дополнительной информацией для моделирования результатов трансляции;
4. Разрабатывается итератор модельных структур;
5. Разрабатываются мепперы:
   - для отображения модельных структур в предложения входного языка;
   - для отображения модельных структур в эталонную систему показателей выходных данных;
6. Производится автоматическая генерация тестовых данных:
   - входные данные транслятора;
   - эталонная система показателей ожидаемых выходных данных транслятора;
7. Для каждой сгенерированной пары тестовых данных в автоматическом режиме производится оценка правильности работы генератора кода в трансляторе: входные тестовые данные подаются на вход транслятору, полученный результат работы транслятора сравнивается на соответствие эталонной системе показателей.

4. Пример

В качестве примера применения предложенного подхода рассмотрим тестирование генератора кода в трансляторе[1], который отображает описание схемы БД на языке SQL в описание структуры таблиц на языке RelaxNG [24] и в описание ограничений, наложенных на эту структуру, на языке XML.

Ниже описаны модели для трех аспектов работы данного генератора кода:

1. Трансляция основных конструкций структуры БД (таблицы, столбцы, типы данных) без каких-либо ограничений;
2. Трансляция ссылочных ограничений;
3. Трансляция ограничений вида «CHECK».

Модели структур входных данных задаются в виде формальных описаний на языке TreeDL [19]. Генераторы тестов разрабатывались на языке

[1] Этот транслятор разрабатывался в ИСП РАН в рамках проекта по созданию инструментальной поддержки для автоматической генерации тестовых данных сложной структуры [20].

программирования Java с использованием инструмента OTK (см. [8], [22]), который предоставляет поддержку описанного метода разработки тестов.

В **первой модели**, описывающей структуру БД без ограничений, моделируются понятия: таблицы, столбцы, типы данных SQL и их параметры. UML-диаграмма для этой модели приведена в приложении (см. Рис. 5).

В качестве эталонной системы показателей для этого аспекта использовалось описание структуры БД на языке RelaxNG в том виде, в котором его должен сгенерировать транслятор.

В данном случае одна и та же модель подходит для моделирования как входных, так и выходных данных: транслятор просто «переводит» такие понятия, как «таблица», «столбец» и т.д., в описание элементов с соответствующими тегами на языке RelaxNG.

Работа оракула заключается в текстуальном сравнении RelaxNG-файла – результата работы транслятора с эталонным RelaxNG-файлом.

Во **второй модели**, описывающей ссылочные ограничения, моделируются понятия: таблица, ссылка между двумя таблицами. UML-диаграмма для этой модели приведена в приложении (см. Рис. 6).

В качестве эталонной системы показателей для этого аспекта использовалось описание ограничений на языке XML в том виде, в котором его должен сгенерировать транслятор.

Для обеспечения корректной генерации входных тестовых данных модель была дополнена информацией о столбцах, а также был введен объемлющий модельный элемент, моделирующий схему БД в целом.

При итерации структур зависимостей между таблицами в генераторе тестов применялась вспомогательная абстрактная модель, которая моделирует ациклические графы. Эти графы использовались в качестве «макета» для построения конкретных модельных структур.

Работа оракула заключается в текстуальном сравнении XML-файла – результата работы транслятора с эталонным XML-файлом.

Третья модель, описывающая ограничения вида «CHECK», моделирует следующие понятия: логические выражения, SQL-предикаты (такие, как «LIKE», «BETWEEN» и т.д.). UML-диаграмма для этой модели приведена в приложении (см. Приложение А).

В качестве эталонной системы показателей для этого аспекта использовалось описание ограничений на языке XML в том виде, в котором его должен сгенерировать транслятор.

Для обеспечения корректной генерации входных тестовых данных модель была дополнена информацией о столбцах, таблицах и объемлющей схеме, а также, для обеспечения корректности применения конкретных SQL-предикатов к данным, были добавлены типы данных.

Работа оракула заключается в текстуальном сравнении XML-файла – результата работы транслятора с эталонным XML-файлом.

Отметим, что та часть третьей модели, которая отвечает за моделирование типов данных, была целиком переиспользована из первой модели. Более того, были переиспользованы и соответствующие компоненты генератора тестов.

В Таб. 1 приведены размеры формальных описаний разработанных моделей, а также размер кода разработанных компонентов генераторов тестов. Отметим, что подавляющее большинство компонентов генераторов тестов было сгенерировано автоматически.

Модель	Количество элементов модели	Размер модели (байт / строк)	Количество классов		Размер классов (Кбайт / строк)	
			Всего	Вручную	Всего	Вручную
Структура БД	36	4519 / 314	48	2	95 / 2669	8 / 271
Ссылочные ограничения	6	1165 / 65	16	7	25 / 847	10 / 378
Ограничения вида «CHECK»	51	6456 / 444	73	7	148 / 4039	20 / 679

Таб. 1. Размер кода разработанных моделей и генераторов тестов.

В Таб. 2 приведены размеры переиспользования формальных описаний разработанных моделей, а также размер переиспользованного кода разработанных компонентов генераторов тестов. Как видно из этой таблицы, в модели, описывающей ограничения вида «CHECK», около 60% кода описания модели и около 40% кода компонентов генератора тестов переиспользуется из других моделей.

Модель	Количество элементов модели	Размер модели (байт / строк)	Количество классов		Размер классов (байт / строк)	
			Всего	Вручную	Всего	Вручную
Ограничения вида «CHECK»	33	3816 / 276	37	2	60 / 1717	8 / 271

Таб. 2. Размер переиспользованного кода разработанных моделей и генераторов тестов.

В Таб. 3 приведены некоторые характеристики сгенерированных тестов для рассмотренных моделей. Как видно из этой таблицы, объем разработанного вручную кода компонентов генераторов тестов примерно на два порядка меньше объема тестов, сгенерированных этими генераторами.

Модель	Количество тестов	Общий объем (Кбайт)	Средний размер теста (байт / строк)
Структура БД	151	826	5471 / 140
Ссылочные ограничения	38	353	9297 / 235
Ограничения вида «CHECK»	46	915	19891 / 455

Таб. 3. Некоторые характеристики сгенерированных тестов.

В результате тестирования рассмотренного в этом примере транслятора с помощью предложенного в данной статье подхода на тестах, сгенерированных на основе представленных в здесь моделей, было обнаружено несколько несоответствий между документацией и реализацией генератора кода в трансляторе. Эти несоответствия связаны с представлением в выходных данных транслятора параметров типов данных и параметров ограничений.

5. Близкие работы

Вопросу автоматического тестирования блока синтаксического разбора посвящены работы [4], [9], [14], [21]. Автоматическое тестирование блока анализа статической семантики рассматривалось в [2], [17]. Для этих блоков, понимаемых как булевские функции, в перечисленных работах предлагаются вполне удовлетворительные подходы к автоматизации их тестирования — как систематической генерации тестов, так и анализу результатов работы.

Для решения задачи автоматической генерации тестов для back-end'а предлагаются следующие подходы. Один — это генерация различных синтаксически корректных предложений целевого языка с последующим отбором корректных программ методом проверки сгенерированного предложения на соответствие ASM спецификации (см. [6], [7]). Достоинство этого подхода состоит в использовании формальной спецификации для описания класса корректных программ. Однако недостатком подхода является несистематичность и неэффективность процесса построения тестов. Другой подход состоит в генерации тестов на основе модели входных данных back-end'а в соответствии с некоторым критерием тестового покрытия, формулируемым в терминах модели (см. [22], [23]). Достоинствами подхода

являются использование формальных моделей данных и эффективная генерация тестов. В качестве недостатка можно указать необходимость ручной разработки некоторых компонентов генератора тестов.

Для решения задачи построения автоматического оракула для back-end'а предлагаются следующие подходы. Ряд исследователей рассматривает способы построения оракулов для тестирования некоторых специальных видов оптимизаторов, основанные на анализе абстрактного внутреннего представления программы (см. [11], [12]). Их достоинство состоит в четкой нацеленности на проверку оптимизатора. Однако недостатком подобных подходов является то, что, как правило, при тестировании реального компилятора внутреннее представление программы недоступно извне. Попытка избавиться от этого недостатка основана на идее инструментирования объектного кода компилятором [5]. Однако здесь проблема заключается в том, что для реализации этой идеи требуется вмешательство непосредственно в работу компилятора, что обычно невозможно сделать при тестировании реального коммерческого компилятора.

Принципиально другим подходом к построению автоматического оракула для back-end'а является предлагаемый в ряде работ оракул, который проверяет сохранение семантики программы в процессе компиляции путем сравнения поведения откомпилированной программы с некоторым эталоном. Имеются исследования, в которых в качестве эталона рассматривается поведение той же программы, но откомпилированной другим «эталонным» компилятором [10]. Близкий подход состоит в сравнении трассы выполнения теста с эталонной трассой [22]. В других исследованиях в качестве эталона рассматривается поведение той же программы, интерпретируемой ASM машиной (см. [6], [7]). Эти подходы являются вполне приемлемыми, однако они годятся лишь для случая тестирования компиляторов и интерпретаторов.

6. Заключение

В статье предложен общий подход к построению автоматического оракула для тестирования генераторов кода в трансляторах текстов на формальных языках, а также предложена инструментальная поддержка для практического использования этого подхода. Архитектура оракула основана на идее одновременной генерации как тестовых входных данных для транслятора, так и соответствующей эталонной системы показателей, характеризующих ключевые свойства выходных данных транслятора.

Несмотря на то, что в ходе разработки необходимых компонентов генератора тестов фактически требуется разработка своего рода «эталонного генератора кода», количество требуемых для этого ресурсов, как правило, существенно меньше того количества ресурсов, которое требуется на разработку реального

генератора кода в тестируемом трансляторе. Экономия достигается за счет следующих факторов:

- модель, для которой требуется разработать «эталонный генератор кода», является абстракцией исходного языка выходных данных транслятора, для которого разрабатывается реальный генератор кода, и поэтому, как правило, модель существенно меньше и проще реального языка;
- разрабатываемый «эталонный генератор кода» должен, на самом деле, генерировать не все выходные данные, а лишь абстрактную систему показателей, объем данных которой, как правило, гораздо меньше;
- при разработке генераторов тестов для нескольких аспектов тестирования одного транслятора одни и те же термины часто присутствуют в моделях для нескольких аспектов тестирования, что обуславливает высокий процент переиспользования кода компонентами генераторов тестов.

Направлениями дальнейшего развития предложенного подхода являются:

- проработка методов построения модели на основе анализа входных и выходных данных транслятора;
- разработка методов выделения эталонных систем показателей выходных данных;
- разработка методов сравнения реального результата работы транслятора на соответствие эталонной системе показателей выходных данных для различных практически используемых частных видов языков выходных данных трансляторов.

Литература

[1] B. Beizer. Software Testing Techniques. // van Nostrand Reinhold, 1990.

[2] A.G. Duncan, J.S.Hutchison. Using Attributed Grammars to Test Designs and Implementation // In Proceedings of the 5th international conference on Software engineering. Piscataway, NJ, USA: IEEE Press, 1981. 170–178.

[3] J. Hannan, F. Pfenning. Compiler Verification in LF. // Proc. 7th Annual IEEE Symposium on Logic in Computer Science (1992) 407–418.

[4] J. Harm, R. Lämmel. *Two-dimensional Approximation Coverage* // Informatica Journal. 24. 2000. № 3.

[5] C. Jaramillo, R. Gupta, M.L. Soffa. Comparison Checking: An Approach to Avoid Debugging of Optimized Code. // ACM SIGSOFT 7th Symposium on Foundations of Software Engineering and European Software Engineering Conference, LNCS, 1687 (1999) 268–284.

[6] A. Kalinov, A. Kossatchev, M. Posypkin, V. Shishkov. Using ASM Specification for automatic test suite generation for mpC parallel programming language compiler. // Proc. Fourth International Workshop on Action Semantic, AS'2002, BRICS note series NS-02-8 (2002) 99–109.

[7] A.S. Kossatchev, P. Kutter, M.A Posypkin. Automated Generation of Strictly Conforming Tests Based on Formal Specification of Dynamic Semantics of the

Programming Language // Programming and Computing Software. July 2004. Volume 30. Issue 4. 218–229.
[8] V. Kuliamin, A. Petrenko. Applying Model Based Testing in Different Contexts. // Proceedings of seminar on Perspectives of Model Based Testing, Dagstuhl, Germany, September 2004.
[9] R. Lämmel. *Grammar testing* // In Proc. of Fundamental Approaches Software Engineering. 2029. 2001. 201–216.
[10] W. McKeeman. Differential testing for software. // Digital Technical Journal, 10(1):100–107, 1998.
[11] T.S. McNerney. Verifying the Correctness of Compiler Transformations on Basic Blocks using Abstract Interpretation. // In Symposium on Partial Evaluation and Semantics-Based Program Manipulation (1991) 106–115.
[12] G. Necula. Translation Validation for an Optimizing Compiler. // Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (2000) 83–95.
[13] A.K. Petrenko. *Specification Based Testing: Towards Practice* // LNCS 2244. 2001. 287–300.
[14] P. Purdom. *A Sentence Generator For Testing Parsers* // BIT. 1972. № 2. 336–375.
[15] M. Wand, Zh. Wang. Conditional Lambda-Theories and the Verification of Static Properties of Programs. // Proc. 5th IEEE Symposium on Logic in Computer Science (1990) 321–332.
[16] M. Wand. Compiler Correctness for Parallel Languages. // Conference on Functional Programming Languages and Computer Architecture (FPCA) (1995) 120–134.
[17] М.В. Архипова. Генерация тестов для семантических анализаторов // Вычислительные методы и программирование. 2006, том.7, раздел 2, 55–70.
[18] 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, Москва, 2004, т. 5, 121–156.
[19] А.В. Демаков. TreeDL: язык описания графовых структур данных и операций над ними. // Вычислительные методы и программирование. 2006, том 7, раздел 2, 117–127.
[20] 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.
[21] С.В. Зеленов, С.А. Зеленова. Генерация позитивных и негативных тестов для парсеров // Программирование, том. 31, №6, 2005, 25–40.
[22] С.В. Зеленов, С.А. Зеленова, А.С. Косачев, А.К. Петренко. *Применение модельного подхода для автоматического тестирования оптимизирующих компиляторов* // `http://www.citforum.ru/SE/testing/compilers/`
[23] С.В. Зеленов, Д.В. Силаков. Автоматическая генерация тестовых данных для оптимизаторов графических моделей. // Труды ИСП РАН, Москва, 2006, т. 9, 129–141.
[24] RelaxNG. // `http://relaxng.org/`

Приложение A. UML-диаграммы моделей

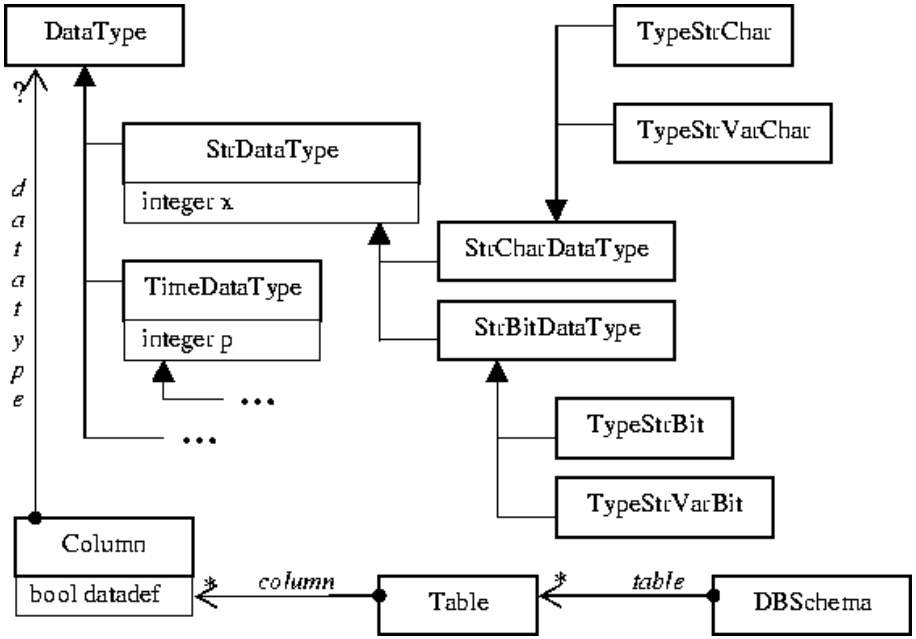

Рис. 5. Модель структуры БД (без ограничений)

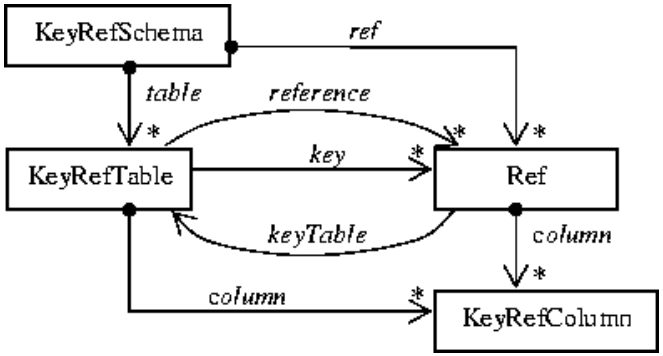

Рис. 6. Модель ссылочных ограничений в SQL