ТЕОРЕТИЧЕСКИЙ И ПРИКЛАДНОЙ НАУЧНО-ТЕХНИЧЕСКИЙ ЖУРНАЛ

Программная инженерия

ПрИн

6
2013

Учредитель: Издательство "НОВЫЕ ТЕХНОЛОГИИ"

Издается с сентября 2010 г.

Редакционный совет
Садовничий В.А., акад. РАН (*председатель*)
Бетелин В.Б., акад. РАН
Васильев В.Н., чл.-корр. РАН
Жижченко А.Б., акад. РАН
Макаров В.Л., акад. РАН
Михайленко Б.Г., акад. РАН
Панченко В.Я., акад. РАН
Стемпковский А.Л., акад. РАН
Ухлинов Л.М., д.т.н.
Федоров И.Б., акад. РАН
Четверушкин Б.Н., акад. РАН

Главный редактор
Васенин В.А., д.ф.-м.н.

Редколлегия:
Авдошин С.М., к.т.н.
Антонов Б.И.
Босов А.В., д.т.н
Гаврилов А.В., к.т.н.
Гуриев М.А., д.т.н.
Дзегеленок И.Ю., д.т.н.
Жуков И.Ю., д.т.н.
Корнеев В.В., д.т.н.,
Костюхин К.А., к.ф.-м.н.
Липаев В.В., д.т.н.
Махортов С.Д., д.ф.-м.н.
Назиров Р.Р., д.т.н.
Нечаев В.В., к.т.н.
Новиков Е.С., д.т.н.
Нурминский Е.А., д.ф.-м.н.
Павлов В.Л., д.ф.-м.н.
Пальчунов Д.Е., д.т.н.
Позин Б.А., д.т.н.
Русаков С.Г., чл.-корр. РАН
Рябов Г.Г., чл.-корр. РАН
Сорокин А.В., к.т.н.
Терехов А.Н., д.ф.-м.н.
Трусов Б.Г., д.т.н.
Филимонов Н.Б., д.т.н.
Шундеев А.С., к.ф.-м.н.
Язов Ю.К., д.т.н.

Редакция
Лысенко А.В., Чугунова А.В.

Журнал издается при поддержке Отделения математических наук РАН, Отделения нанотехнологий и информационных технологий РАН, МГУ имени М.В. Ломоносова, МГТУ имени Н.Э. Баумана, ОАО "Концерн "Сириус".

СОДЕРЖАНИЕ

Вьюкова Н. И., Галатенко В. А., Самборский С. В. LLVM как инфраструктура разработки компиляторов для встроенных систем 2

Терехов А. Н., Брыксин Т. А., Литвинов Ю. В. QReal: платформа визуального предметно-ориентированного моделирования 11

Шеенок Д. А. Инструментальное средство проектирования оптимальной архитектуры отказоустойчивых программных систем 20

Баранов Д. В. Компьютерная реализация устранения избыточных правил в условных системах переписывания 27

Харламов А. А., Ермоленко Т. В. Семантическая сеть предметной области как основа для формирования сети переходов при автоматическом распознавании слитной речи 33

Иванова А. В., Адуенко А. А., Стрижов В. В. Алгоритм построения логических правил при разметке текстов 41

Contents 48

Журнал зарегистрирован в Федеральной службе по надзору в сфере связи, информационных технологий и массовых коммуникаций. Свидетельство о регистрации ПИ № ФС77-38590 от 24 декабря 2009 г.

Журнал распространяется по подписке, которую можно оформить в любом почтовом отделении (индексы: по каталогу агентства "Роспечать" — **22765**, по Объединенному каталогу "Пресса России" — **39795**) или непосредственно в редакции.
Тел.: (499) 269-53-97. Факс: (499) 269-55-10.
Http://novtex.ru E-mail: prin@novtex.ru

Журнал включен в систему Российского индекса научного цитирования.
Журнал входит в Перечень научных журналов, в которых по рекомендации ВАК РФ должны быть опубликованы научные результаты диссертаций на соискание ученой степени доктора и кандидата наук.

© Издательство "Новые технологии", "Программная инженерия", 2013

УДК 004.4'418

Н. И. Вьюкова, стар. науч. сотр.,
В. А. Галатенко, д-р физ.-мат. наук, стар. науч. сотр.,
С. В. Самборский, стар. науч. сотр., НИИ системных исследований РАН, г. Москва,
e-mail: niva@niisi.msk.ru

LLVM как инфраструктура разработки компиляторов для встроенных систем

Представлен обзор инфраструктуры LLVM и ее оценка с учетом особенностей разработки компиляторов для микропроцессоров, используемых во встроенных системах. Обсуждаются особенности работы с LLVM с точки зрения пользователя, дано описание общей структуры LLVM и промежуточного представления программы.

Ключевые слова: *компилятор, LLVM, встроенные системы, промежуточное представление*

Введение

Микропроцессоры, применяемые во встроенных системах, и методы создания таких систем имеют ряд особенностей, которые необходимо учитывать при разработке средств компиляции и также при выборе инфраструктуры построения компиляторов.

Встроенные системы, как правило, ориентированы на выполнение определенного класса задач, и разработка аппаратных средств для них зачастую предполагает расширение системы команд инструкциями, специфическими для типичных задач предметной области. При этом может применяться метод встречной оптимизации, т. е. система команд может неоднократно меняться по результатам экспериментальных исследований в целях достижения указанных выше характеристик. Соответственно, должны изменяться и средства компиляции. Поэтому важно, чтобы инфраструктура поддерживала гибкие средства описания системы команд, причем это описание в идеале должно быть единым для компилятора, ассемблера, дизассемблера и других инструментов.

Характерная особенность встроенных систем — довольно жесткие ограничения на использование ресурсов, в частности, памяти. Такие системы также часто работают в режиме реального времени, а значит необходим контроль временных характеристик выполнения. Для гарантированного соблюдения ограничений нужны средства статического анализа, например, для оценки максимального размера стека, используемого потоком выполнения, для оценки времени выполнения в наихудшем случае и т. п. Соответственно, необходима реализация дополнительных проходов анализа. Могут также понадобиться дополнительные проходы оптимизации, для того чтобы автоматически использовать специальные вычислительные команды процессора при генерации кода. Таким образом, важным требованием к инфраструктуре компиляции является простота добавления дополнительных или альтернативных проходов анализа и оптимизации.

Как для анализа, так и для оптимизации существенна возможность обработки всей программы целиком, которая реализуется в инфраструктуре как оптимизация времени компоновки (LTO — *link-time optimization*). Это обеспечивает максимальную полноту анализа и дополнительные возможности для оптимизации за счет интенсивного использования inline-подстановок.

Важное значение имеет также структура внутреннего представления программы; это представление должно достаточно полно отражать семантику входной программы и в то же время (по крайней мере, на поздних стадиях компиляции) быть достаточно близким к машинному представлению, что необходимо для реализации машинно-зависимых оптимизаций и средств анализа. При этом инфраструктура должна обеспечивать простой в использовании интерфейс доступа к элементам представления.

В компиляторах для универсальных процессоров обычно применяют эвристические методы оптимизации кода, что позволяет компилировать быстро и при этом получать высокопроизводительный код. Однако

для встроенных систем эвристические методы далеко не всегда обеспечивают генерацию достаточно эффективного кода. Хотя микропроцессоры, используемые во встроенных системах, обычно имеют высокий потенциал параллелизма на уровне команд, использовать его при компиляции оказывается довольно сложно в силу аппаратных особенностей, таких как сложные ограничения на параллельное исполнение команд, небольшие, узкоспециализированные классы регистров и т. п. Подобные особенности связаны с наличием ограничений на энергопотребление и размеры микросхем. По тем же причинам микропроцессоры этого класса, как правило, не содержат средств, требующих сложных аппаратных решений, таких как аппаратное планирование потока команд, ротация регистрового файла, предикатное исполнение, что также усложняет задачу генерации эффективного кода.

Для максимального использования потенциала быстродействия, предоставляемого процессором, значительная часть программного кода приложений зачастую реализуется на языке ассемблера, что требует больших трудозатрат и высокой квалификации программистов, а также затрудняет портирование кода на другие процессоры. Поэтому целесообразным представляется применение точных методов оптимизации кода [1—3] с использованием методов математического программирования. Хотя такой подход может существенно замедлить процесс компиляции с языка высокого уровня, он может быть вполне оправданной альтернативой ассемблерному программированию.

Работа посвящена анализу инфраструктуры построения компиляторов LLVM[1] [4] с учетом упомянутых выше требований. Рассмотрены преимущества и ограничения текущих версий компилятора Clang/LLVM в сравнении с широко известным свободно-распространяемым компилятором GCC [5]. Clang/LLVM — это компилятор, который использует инфраструктуру LLVM и программу Clang [6], транслирующую входную программу на языке C, C++, Objective C или Objective C++ во внутреннее представление LLVM.

Приведено также описание системы типов и внутреннего представления программ LLVM. Еще одну статью авторы планируют посвятить структуре генератора кода и средствам описания целевых платформ, а также существующим в настоящее время средствам, обеспечивающим интерфейс к внутренним структурам LLVM на языке Python.

Clang/LLVM с точки зрения пользователя

Диагностика. Сильной стороной Clang по сравнению с GCC изначально являлось качество диагностики. Сообщения, выдаваемые Clang, во многих случаях выглядят существенно более содержательными и развернутыми. Такие сообщения включают отображение строки исходного текста с маркерами, показывающими позицию, к которой относится сообщение; если ошибочный код содержит макровызов, то выводится также соответствующее макроопределение. Пример:

```
asm.i:16:10: error: member reference base
type ' int
*' is not a structure or union
      return NAME(p);

            ^~~~~~~

asm.i:12:19: note: expanded from macro 'NAME'
#define NAME(X) X.member-> pname
```

Соответствующее сообщение GCC 4.7.2 выглядит так:

```
asm.i: In function 'foo':
    asm.i:16:3: warning: return makes
    pointer from integer without a cast
```

Справедливости ради нужно отметить, что GCC последних версий по информативности сообщений во многих случаях приблизился к Clang.

Если попробовать скомпилировать при помощи Clang/LLVM не очень аккуратно написанный код (например, какой-нибудь старый проект), то можно увидеть, что Clang по умолчанию выдает существенно больше предупреждений, чем GCC. Например, на одной из программ авторы получили 409 предупреждений против 5, выданных GCC. Это связано с тем, что в Clang по умолчанию включены все виды предупреждений, а в GCC — только наиболее критичные.

Отдельные виды предупреждений Clang можно отключать при помощи опций командной строки; возможно также управлять выдачей предупреждений на уровне строк исходного кода, используя прагму `#pragma GCC diagnostic`, поддержка которой ожидается и в GCC, начиная с версии 4.8.

Поддержка расширений GCC. Clang/LLVM поддерживает большинство расширений языков C/C++, реализованных в GCC, включая атрибуты и ассемблерные вставки (операторы `asm`). Тем не менее переход к использованию Clang/LLVM вместо GCC может оказаться затруднительным в силу того, что не поддерживаются некоторые варианты синтаксиса этих расширений. Например, на одной из программ авторы получили предупреждение о некорректном употреблении атрибута `transparent_union` в конструкциях следующего вида:

```
typedef union __attribute__
((__transparent_union__))

{

...

} foo_ptr_t;
```

[1] Название инфраструктуры LLVM является аббревиатурой, которая расшифровывается как Low-Level Virtual Machine (низкоуровневая виртуальная машина).

Вследствие того что атрибут был проигнорирован, далее в программе возникли ошибки компиляции в тех местах, где использовались свойства прозрачных объединений. На самом деле Clang/LLVM поддерживает этот атрибут, но он должен располагаться в конце декларации, перед завершающим символом ";".

Производительность. На простых тестах производительности для процессоров Intel при использовании Clang/LLVM были получены такие же или даже более высокие показатели производительности в сравнении с GCC-4.4.5. Для других целевых процессоров это может быть не так. Например, для процессора MIPS результаты (при максимальной оптимизации) оказались несколько хуже (до 10...40 %). Измерения скорости компиляции программ на языке C показали, что GCC работает существенно быстрее, чем Clang/LLVM.

Статический и динамический анализ. Clang и LLVM изначально проектировались как совокупность библиотек с соответствующим API, которые могут быть использованы для разработок различных средств программирования, поэтому говоря об их возможностях, нужно рассматривать всю совокупность основанных на них проектов. Инструмент статического анализа Clang Static Analyzer [7] был успешно применен при разработке операционной системы и приложений реального времени. Clang Static Analyzer позволяет обнаруживать дефекты кода различного характера — ошибки обращения по нулевому указателю, некорректные логические выражения, ошибки вычисления счетчика цикла и индекса массива и др.; подробнее об опыте работе с этим инструментом написано в работе [8].

LLVM также поддерживает средства динамического анализа, позволяющие обнаруживать ошибки работы с памятью (AddressSanitizer) и ошибки синхронизации потоков (ThreadSanitizer) во время выполнения программы [9].

Кросс-компиляция. При разработке встроенных систем обычно применяется метод кросс-компиляции, когда компилятор выполняется на одной платформе (например, на процессоре i386 под Linux), а код генерируется для другого типа процессора и/или другой операционной системы. Из исходных текстов GCC (и бинарных утилит), при соответствующем конфигурировании можно получить готовый кросс-компилятор для заданной целевой платформы. LLVM является универсальным кросс-компилятором, который может генерировать (ассемблерный или объектный) код для любой из поддерживаемых архитектур; однако, в отличие от GCC, LLVM не предоставляет драйвера компиляции, который обеспечивает формирование и запуск командных строк компиляции, ассемблирования и компоновки. Существуют проекты ELLCC [10] и EmbToolkit [11], предназначенные для создания инструментов кросс-компиляции на базе Clang/LLVM, однако авторам не удалось получить с их помощью готовое решение для своей конфигурации.

Поддержка генерации кода для MIPS. Авторов интересовало в первую очередь использование LLVM для архитектуры MIPS. Поддержка этой архитектуры появилась в LLVM относительно недавно, тем не менее, генератор кода для нее работает достаточно устойчиво. Реализована генерация кода для наборов команд mips32, mips64, mips32r2, mips64r2 под ОС Linux. Однако на данный момент LLVM существенно проигрывает GCC в отношении таких возможностей, как поддержка индивидуальных процессоров (с учетом специфики набора команд и их латентностей), поддержка различных опций кодогенерации, заполнение гнезд задержки, генерация векторных команд.

В целом, по мнению авторов, Clang/LLVM является динамично развивающимся инструментарием с хорошими перспективами дальнейшего роста. В то же время GCC обладает преимуществами более зрелого проекта, имеющего давнюю историю развития и множество пользователей. К таким преимуществам в первую очередь следует отнести поддержку большего числа входных языков и целевых платформ.

Общий обзор LLVM

Работа над проектом LLVM была начата в 2000 г. в университете Иллинойса. Целью проекта было создание гибкой инфраструктуры, позволяющей разрабатывать различные как экспериментальные, так и промышленные средства анализа, оптимизации, трансформации и компиляции программ [12, 13]. Внутренняя структура проекта GCC, начало которого относится к 80-м гг. прошлого века, к тому времени стала слишком громоздкой, неудобной, трудно поддающейся для введения новых или альтернативных проходов анализа и оптимизации кода. Довольно сложными и преимущественно процедурными являлись средства описания целевых микропроцессорных архитектур в GCC. Хотя разработчики постоянно предпринимали усилия по совершенствованию внутренней организации GCC, многие изначально заложенные подходы затрудняли решение указанных проблем. К числу причин такого положения можно отнести использование языка C, "монолитность" кода, применение большого числа макросов и глобальных переменных, не всегда удачные структуры данных.

В работе [14] приводятся примеры, которые дают представление о степени компактности и простоте понимания программного кода LLVM в сравнении с GCC. В двух примерах, представленных ниже, сравниваются фрагменты кода этих инфраструктур, реализующие одни и те же функции.

- Генерация одной команды внутреннего представления:

LLVM:

```
Value *B = IRB. CreateLoad(IRB.
CreateIntToPtr(A, Ty));
```

GCC:

```
t = build1 (INDIRECT_REF, shadow_type,
build1 (VIEW_CONVERT_EXPR, shadow_ptr_type,
t));
```

```
t = force_gimple_operand (t, &stmts, false,
NULL_TREE);
gimple_seq_add_seq (&seq, stmts);
shadow_value = make_rename_temp
(shadow_type, " ");
g = gimple_build_assign (shadow_value, t);
```

• Установка признака **noreturn** для вызова функции, которая не возвращает управление:

LLVM:

```
Call-> setDoesNotReturn();
```

GCC:

```
TREE_THIS_VOLATILE (call) = 1;
```

В 2010 г. начались подготовительные работы по переходу к C++ как к языку разработки GCC. В 2012 г. было официально объявлено, что для сборки последующих версий GCC будет использован компилятор g++, и разработка новых средств также будет вестись на C++; постепенно также будет осуществлен перенос существующего кода на C++.

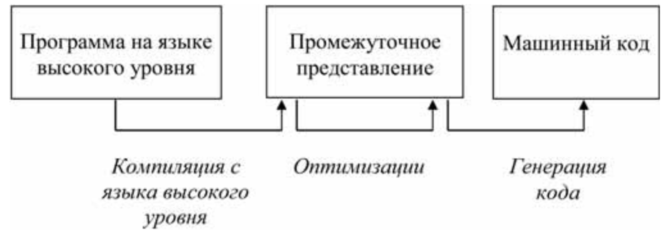
В проекте LLVM, с учетом опыта проекта GCC, а также методов современного компиляторостроения, изначально были заложены перечисленные ниже решения, обеспечивающие хорошо структурированную, гибкую внутреннюю организацию.

• Понятие абстрактной низкоуровневой виртуальной машины и ее язык, составляющие идейную основу, вокруг которой выстроен весь набор инструментов LLVM. Внутреннее представление входной программы — это запись ее семантики на языке виртуальной машины. Отличительной чертой внутреннего языка LLVM является низкоуровневый набор инструкций в сочетании с высокоуровневой системой типов данных, что позволяет в компактной форме отражать семантику программ на различных языках, эффективно проводить анализ и оптимизирующие преобразования программ, генерировать машинный код и отладочную информацию. Для сравнения, промежуточное представление GIMPLE, используемое в GCC, не является самодостаточным, поскольку при генерации отладочной информации формата DWARF компилятор обращается к представлению операндов в форме AST (в виде синтаксического дерева).

• Трехфазная структура компилятора (см. рисунок), обеспечивающая поддержку множества входных языков и множества целевых платформ.

В настоящее время LLVM используется для множества языков, в частности языков, поддерживаемых GCC, а также .NET, Python, Ruby, Scheme, Haskell, D и др. В качестве целевых архитектур поддерживаются X86, X86_64, Sparc, ARM, PowerPC, MIPS, Cell, PTX, XCore, MBlaze.

Хотя трехфазная структура считается классической и описана едва ли не в любом учебнике по построению компиляторов, на практике она далеко не всегда бывает реализована в полной мере. Так, разработчики, пытавшиеся использовать frontend'ы GCC для целей анализа, рефакторинга и т. п., оказывались перед необходимостью включать в приложение практически весь код GCC.



Трехфазная структура компилятора

• Использование языка C++ и разумных стандартов кодирования, что обеспечивает высокий уровень абстракции, компактность и читаемость кода. Достаточно подробная документация, как внешняя, написанная вручную, так и внутренняя, сгенерированная при помощи Doxygen.

• Развитые, преимущественно декларативные (в отличие от GCC), средства описания целевых платформ, которые позволяют задавать информацию как для генерации кода, так и для его ассемблирования и дизассемблирования.

• Представление всех видов анализа и преобразования кода в виде проходов компилятора.

• Модульная инфраструктура оптимизатора, возможность произвольно комбинировать проходы анализа и преобразований.

• Набор средств для отладки и тестирования, включающий графическую визуализацию внутреннего представления, инструмент для генерации минимального тестового кода, на котором воспроизводится ошибка (**bugpoint**) и др.

• Поддержка оптимизации времени компоновки (LTO).

• Организация LLVM в виде набора библиотек, которые могут быть использованы при построении как базового, стандартного набора инструментов, так и специфических инструментов, создаваемых сторонними группами разработчиков. Ниже перечислены основные инструменты, входящие в состав комплекта Clang/LLVM.

- **clang** — компилятор с языков C, C++, Objective C или Objective C++;
- **opt** — оптимизатор внутреннего представления LLVM;
- **llc** — генератор машинного кода;
- **lli** — интерпретатор внутреннего представления;
- **llvm-as** — утилита, преобразующая внутреннее представление из текстового формата в бинарный;
- **llvm-dis** — утилита, преобразующая внутреннее представление из бинарного формата в текстовый;
- **llvm-ld** — утилита, позволяющая скомпоновать файлы внутреннего представления (с под-

ключением библиотек) в один файл внутреннего представления или в выполняемый файл;
- **llvm-prof** — печать данных профилирования;
- **llvm-link** — утилита, позволяющая скомпоновать несколько файлов внутреннего представления в один;
- **llvm-ar** — архиватор файлов внутреннего представления;
- **llvm-diff** — структурное сравнение модулей во внутреннем представлении;
- **llvm-stress** — генератор случайных файлов внутреннего представления (для тестирования);
- **bugpoint** — утилита, позволяющая автоматически сократить тест, на котором происходит ошибка;
- **llvm-extract** — утилита для выборки заданной функции из файла внутреннего представления.

LLVM приобрел признание и популярность как среда для разработки экспериментальных и промышленно используемых инновационных методов анализа и оптимизации, а также инструментальных средств разработки для встроенных систем. Например, в работе [15] представлен пакет средств компиляции, анализа и эмуляции программ для микропроцессора RISCO, разработанного на базе архитектуры MIPS. Этот пакет включает инструмент анализа для оценки времени выполнения программы в наихудшем случае.

Благодаря поддержке многочисленных целевых платформ и универсальности, внутреннее представление программ LLVM было выбрано компанией Khronos Group в качестве основы стандартного портируемого внутреннего представления (SPIR — *Standard Portable Intermediate Representation*) языка OpenCL [16]. Это обстоятельство позволило распространять программное обеспечение в портируемом бинарном виде, не раскрывая исходных текстов.

Внутреннее представление LLVM

Общая характеристика

Внутреннее представление программы LLVM — последовательность меток и низкоуровневых команд над значениями, принадлежащими к высокоуровневому набору типов, включающих элементарные типы (целочисленные, вещественные, метки) и производные типы, такие как указатели, массивы, векторы и структуры, а также функции. Поддерживается RISC-подобный набор команд, в основном трехадресных, имеющих два аргумента и один результат.

Низкоуровневость набора команд внутреннего представления дает такие преимущества, как компактность кода, возможность проводить широкий спектр трансформаций. Применение высокоуровневой информации о типах обеспечивает возможности для глубокого межпроцедурного анализа и интенсивных оптимизаций, в том числе на стадии компоновки, выполнения или реоптимизации в периоды простоев системы.

Внутреннее представление может храниться в трех различных формах: в виде данных в оперативной памяти, в виде биткода на диске и в текстовом виде, пригодном для чтения и анализа программистом; все три формы семантически эквивалентны друг другу.

Структура модуля

Внутреннее представление программы в LLVM состоит из модулей.

Модуль представляет одну единицу трансляции программы на исходном языке. Модуль состоит из определений функций, определений глобальных переменных и деклараций. Рассмотрим примеры определений и деклараций в приведенном ниже внутреннем представлении программы:

```
#include < stdio >
int main () {
  printf ("hello world!");
}
```

Соответствующий модуль состоит из декларации строковой переменной **@str**, определения функции **@main** и декларации функции **@puts**:

```
@str = private unnamed_addr constant
[13 x i8] c"hello world!\00"
define i32 @main() nounwind {
entry:
  %puts = tail call i32 @puts(i8*
getelementptr inbounds ([13 x i8]* @str,
i32 0, 132 0))
  ret i32 0
}

declare i32 @puts(i8* nocapture) nounwind
```

Спецификатор **private** в описании переменной **@str** означает, что она видна только в пределах данного модуля; **constant** соответствует спецификатору **const** языка С. Спецификатор **unnamed_addr** означает, что адрес переменной несуществен, так что определения константных переменных с одинаковыми значениями могут быть слиты. Далее следует тип переменной: [**13 x i8**] (массив из 13 элементов типа **i8** — целое размера 8 бит) и начальное значение, заданное строкой символов в кавычках. Помимо этих элементов, декларация переменной может содержать имя секции, в которую должна быть помещена переменная, выравнивание, а также номер адресного пространства, например:

```
@G = addrspace(5) constant float 1.0,
section "foo", align 4
```

Определение функции **@main** в данном примере содержит тип (**i32** — целое размера 32 бита), список аргументов в скобках (в данном случае пустой), атрибут **nounwind**, который означает, что функция не может завершиться особым образом в результате исключительной ситуации. Другие примеры возможных атрибутов функций: **noinline** (не выполнять inline-

подстановку данной функции), **noreturn** (данная функция не возвращает управление).

Тело функции состоит из двух инструкций — вызова (**tail call** — хвостовой вызов) функции **puts** и инструкции возврата **ret**. Аргумент функции задается как адрес строки **@str**, получаемый при помощи команды внутреннего языка LLVM **getelementptr** (описание этой команды приведено далее).

Программный код, составляющий тело функции, имеет представление SSA (*Single Static Assignment*): последовательность команд внутреннего "машинного" языка LLVM, где каждой внутренней переменной (виртуальному регистру) значение присваивается только один раз. Для слияния переменных, значения которым могли быть присвоены в разных базовых блоках, используется инструкция **phi**.

Последняя строка модуля содержит декларацию стандартной функции **puts** с одним аргументом типа "указатель на **i8**". Аргумент имеет атрибут **nocapture**, который означает, что функция не создает копий указателя, которые могли бы быть использованы где-либо, помимо самой функции **puts**.

Типы данных

Система типов является одним из важнейших свойств внутреннего представления LLVM. Благодаря тому, что с операндами инструкций связаны типы, многие виды трансформаций могут выполняться непосредственно, без предварительных проходов анализа. Типы данных подразделяются на простые и производные.

Простые типы

- *Целочисленные типы* различных размеров, задаваемых в битах: **i1, i2, i3, ..., i64**, ... Максимальный размер 2^{23}-1.
- *Типы данных с плавающей точкой*:

half — 16-битные значения;

float — 32-битные значения;

double — 64-битные значения;

fp128 — 128-битные значения со 112-битной мантиссой;

x86_fp80 — 80-битные значения (X87);

ppc_fp128 — 128-битные значения (PowerPC).

X86mmx — представляет значения, хранимые на регистрах MMX процессора x86.

- *Тип* **void** не представляет какое бы то ни было значение, не имеет размера.
- *Тип* **label** представляет метки в коде программы.
- *Тип* **metadata** представляет встроенные метаданные программы. Этот тип не может быть использован для создания производных типов.

Производные типы

Производный тип может быть построен на основе простых или других производных типов. Поддерживаются следующие производные типы: массивы, функции, структуры, указатели, векторы.

- *Массив* — это набор однотипных значений, хранящихся в памяти последовательно. Тип элементов — любой тип, имеющий размер. Примеры:
  - **[40 x i32]** — одномерный массив из 40 32-битных целых;
  - **[2 x [3 x [4 x i16]]]** — многомерный массив из 2 × 3 × 4 16-битных целых.
- *Функция*. Тип функции представляет сигнатуру функции, он содержит тип результата и типы параметров. Тип результата может быть простейшим (кроме **x86mmx**) или производным (за исключением функции). Тип функции не имеет размера. Примеры:
  - **float (i16, i32 *)** — функция с результатом типа float, имеющая два параметра: **i16** и указатель на **i32**;
  - **i32 (i8*, ...)** — функция с результатом типа **i32**, имеющая переменное число аргументов. Первый аргумент обязателен и имеет тип "указатель на **i8**".
- *Структура*. Тип структуры представляет набор элементов различных типов, расположенных в памяти последовательно. Типы элементов структур должны иметь размер.

Структуры могут быть упакованными и неупакованными. В упакованных структурах элементы располагаются с выравниванием 1 байт. В неупакованных структурах элементы имеют выравнивание, определяемое ABI целевой платформы. Примеры:

  - **{float, i32 (i32)*}** — структура из двух элементов; первый элемент имеет тип **float**, второй — указатель на функцию типа **i32 (i32)**;
  - **< { i8, i32 } >** — упакованная структура из двух элементов размера 5 байт.

Скрытая **(opaque)** структура соответствует декларации именованной структуры без описания элементов. Пример описания скрытой структуры: **%X = type opaque**.

- *Указатель*. Тип указателя служит для описания позиций в памяти. Он используется для доступа к объектам в памяти.

С типом указателя может быть связан номер адресного пространства, в котором находятся указуемые объекты. По умолчанию подразумевается нулевое адресное пространство. Семантика ненулевых адресных пространств зависит от целевой платформы. Примеры:

  - **[4 x i32]*** — указатель на массив из четырех элементов i32;
  - **i32 addrspace (5)*** — указатель на значение **i32**, находящееся в адресном пространстве 5.

В LLVM не допускаются указатели **void*** и указатели на метки **(label*)**; вместо этого следует использовать **i8***.

- *Вектор*. Этот тип представляет векторы элементов простейших типов, используемые как операнды SIMD-инструкций. Его определение включает число и тип элементов. Тип элементов может быть целочисленным типом, типом с плавающей точкой или указателем на эти типы. Примеры:

- **<4 x i32>** — вектор из четырех 32-битных целых;
- **<8 x float>** — вектор из восьми 32-битных значений с плавающей точкой;
- **<4 x i64*>** — вектор из четырех указателей на 64-битные целые.

Инструкции

Множество инструкций внутреннего представления LLVM включает следующие группы: терминирующие инструкции, бинарные арифметические операции, бинарные побитовые операции, векторные операции, агрегатные операции, операции адресации и доступа к объектам в памяти, операции преобразования типов и прочие операции.

В статье приведены лишь краткие описания инструкций, опущены многие детали, такие как модификаторы инструкций. Например, модификатор **exact** в применении к целочисленному делению означает, что результат некорректен, если остаток от деления ненулевой, а в применении к логическому сдвигу вправо — что результат некорректен, если хотя бы один ненулевой бит был удален.

Терминирующие инструкции

Терминирующие инструкции — это инструкции, которыми заканчиваются базовые блоки кода. К ним относятся следующие инструкции:

ret — возврат из функции;

br — условный или безусловный переход на другой базовый блок данной функции;

switch — переключатель, который позволяет разветвить поток управления по более чем двум направлениям;

indirectbr — инструкция косвенного перехода;

invoke — инструкция вызова функции с возможностью возврата по одной из двух меток — для случая нормального возврата и случая возникновения исключительной ситуации;

resume — эта инструкция является частью системы обработки исключительных ситуаций LLVM;

unreachable — указывает оптимизатору, что данный участок кода является недостижимым.

Бинарные арифметические операции над целыми значениями

Все эти операции включают два аргумента и один результат, имеющие одинаковые целочисленные типы — скалярные или векторные:

add — сложение целых значений;

sub — вычитание целых значений;

mul — умножение целых значений;

udiv — беззнаковое деление целых значений;

sdiv — знаковое деление целых значений;

urem — беззнаковая инструкция вычисления остатка от деления;

srem — знаковая инструкция вычисления остатка от деления.

Результат первых трех перечисленных выше инструкций корректен как для знаковых, так и для беззнаковых типов, поскольку для отрицательных значений используется дополнительное представление. Результат имеет тот же размер, что и операнды.

Бинарные арифметические операции над значениями с плавающей точкой

Аргументы и результат этих операций имеют одинаковые типы с плавающей точкой — скалярные или векторные:

fadd — сложение;

fsub — вычитание;

fmul — умножение;

fdiv — деление;

frem — взятие остатка от деления.

Бинарные побитовые операции

Операции этой группы реализуют различные манипуляции над битами значений. Они могут появляться как результат оптимизаций понижения мощности операций (например, деление или умножение целого значения на степень числа 2 реализуются как сдвиги):

shl — сдвиг влево, результат равен **op1 * 2^op2**;

lshr — логический сдвиг вправо, старшие биты результата заполняются нулями;

ashr — арифметический сдвиг вправо;

and — побитовая логическая операция "и";

or — побитовая логическая операция "или";

xor — побитовая логическая операция "исключающее или".

Векторные операции

Векторными операциями являются:

extractelement — извлечение компонента вектора по номеру;

insertelement — запись компонента вектора по номеру;

shufflevector — создание вектора из элементов двух входных операндов-векторов. Эта операция имеет три операнда: два вектора **v1** и **v2** одинакового типа и маску — массив из **m** элементов типа **i32**. Элементы маски задают номера элементов вектора, являющегося конкатенацией **v1** и **v2**, из которых должен быть составлен результат.

Операции над агрегатными значениями

Операция **extractelement** позволяет выбрать элемент из структуры или массива. Первый операнд — значение агрегатного типа — структура или массив. Последующие операнды — индексы, относящиеся к последовательным уровням иерархии агрегатного типа. Например, для выборки элемента массива **B [5] [13]** нужно задать индексы 5, 13. Аналогично используются индексы для доступа к элементам вложенных структур.

Операция **insertelement** проводит запись элемента в структуру или массив. Первый аргумент задает агрегатное значение, второй — записываемое зна-

чение, последующие аргументы задают индексы для доступа к агрегатному значению, так же как в операции **extractelement**.

Операции адресации и доступа к объектам в памяти

В эту группу включены операции выделения памяти, адресации, чтения, записи, а также операции атомарного доступа к памяти и барьеры памяти, близкие по своей семантике к средствам, вошедшим в последние стандарты языков C и C++:

alloca — инструкция, выделяющая область памяти в стеке;

load — инструкция чтения значения из памяти;

load atomic — атомарное чтение значения;

store — запись значения в память;

store atomic — атомарная запись значения в память;

fence — барьер между операциями доступа к памяти;

cmpxchg — атомарная условная модификация значения в памяти;

atomicrmw — атомарная модификация значения в памяти (допустимые варианты этой команды: ***ptr = val** и ***ptr = *ptr op val**, где ***op*** может быть операцией сложения, вычитания, логическим "и", "или", взятием минимума, максимума и др.);

getelementptr — операция вычисления адреса элемента в агрегатной структуре данных; в качестве аргумента задается указатель на значение, имеющее тип структуры, массива или вектора, далее, как в команде **extractelement**, задаются индексы.

Операции преобразования типов

Операции этой группы имеют в качестве аргументов значение и тип, к которому оно должно быть преобразовано. Результатом является преобразованное значение:

trunc — преобразует целое (или вектор целых) к целому (или вектору целых) меньшей ширины;

zext, sext — преобразуют целое (или вектор целых) к целому (или вектору целых) большей ширины;

zext заполняет старшие биты нулями, **sext** — размножает знаковый бит;

fptrunc — преобразует значение с плавающей точкой к заданному типу с плавающей точкой меньшего размера;

fpext — преобразует значение с плавающей точкой к заданному типу с плавающей точкой большего размера (например, **float** к **double**);

fptoui, fptosi — преобразуют значение (или вектор значений) с плавающей точкой к ближайшему беззнаковому или знаковому целому, с округлением в сторону нуля;

uitopf, sitofp — преобразуют целое (беззнаковое или знаковое) значение (или вектор целых значений) к значению (или вектору значений) с плавающей точкой;

ptrtoint — преобразует указатель (или вектор указателей) к целому (или вектору целых) заданного типа;

inttoptr — преобразует целое (или вектор целых) к указателю (или вектору указателей) заданного типа;

bitcast — преобразует входное значение к заданному типу, не меняя битовое представление значения; тип результата должен иметь тот же размер, что и входное значение (примеры преобразований, которые могут осуществляться при помощи **bitcast**: **float** к **i32**, указатель к указателю другого типа, вектор целых к вектору целых другого типа, имеющему тот же размер, например **<2 x i32>** к **<8 x i8>**)).

Прочие операции

К этой группе относят следующие операции.

icmp — возвращает битовое значение (или вектор битов), соответствующий результату сравнения целых, указателей (или векторов целых или указателей).

fcmp — возвращает битовое значение (или вектор битов), соответствующий результату сравнения значений (или векторов значений) с плавающей точкой.

phi — реализует узел phi в SSA-представлении функции, в качестве аргументов задается последовательность пар (массивов) вида [*значение*, *метка*], где *метка* определяет базовый блок. Пример:

```
Loop: ; бесконечный цикл...

   %indvar = phi i32 [0, %LoopHeader],
[%nextindvar, %Loop]
   %nextindvar=add i32 %indvar,1
   br label %Loop
```

Здесь команда **phi** сообщает оптимизатору, что значение переменной **%indvar** равно либо 0, либо **%nextindvar**, а также метки базовых блоков, в которых переменная могла получить значение.

select — условный выбор одного из двух входных значений. Пример:

```
%X = select i1 %cond, i8 17, i8 42
```

Переменная **%X** получает значение **17**, если **%cond = = 1**, и **42** в противном случае.

call — вызов функции.

va_arg — служит для доступа к аргументам, переданным функции с переменным числом аргументов.

landingpad — является частью системы обработки исключений LLVM.

Внутренние функции

LLVM поддерживает внутренние (*intrinsic*) функции, аналогичные встроенным (*builtin*) функциям GCC. Вызовы внутренних функций преобразуются в последовательность команд, реализующую семантику функции. Определены, в частности, внутренние функции, соответствующие стандартным функциям языка C: (**memcpy, memmove** и др.). Предусмотрен механизм определения внутренних функций, специфических для целевой архитектуры (например, через

внутренние функции реализована поддержка библиотеки Altivec для архитектуры PowerPC).

Заключение

LLVM представляет собой развитую, хорошо структурированную инфраструктуру для создания компиляторов и других средств разработки программ. Компилятор Clang/LLVM в настоящее время поддерживает языки C, C++, Objective-C, Objective-C++ большинством расширений, введенных в GCC, и для платформы i386 в целом не уступает GCC по производительности генерируемого кода. Для других целевых процессоров по поддержке различных опций генерации кода и по его производительности LLVM может уступать GCC.

Внутренний язык LLVM достаточен для представления семантики программ на современных языках программирования, включая такие механизмы как обработка исключений и поддержка атомарных операций. Наличие высокоуровневой информации о типах, развитая система команд внутреннего языка позволяют эффективно проводить различные оптимизации и обеспечивают потребности генерации кода. Механизм внутренних функций дает возможность программистам использовать специфические команды процессора в программах на языках высокого уровня.

Недостатком представляется отсутствие комплексных типов и, соответственно, операций над данными этих типов. При генерации кода для процессоров, поддерживающих операции над комплексными значениями, оказывается невозможным использовать эти средства автоматически. Для того чтобы использовать команды, работающие с комплексными данными, программист должен вместо стандартных средств языка высокого уровня применять вызовы внутренних функций, что приводит к снижению мобильности программного обеспечения.

В целом, тем не менее, LLVM обладает рядом свойств, которые представляются важными и полезными при разработке и реализации промышленных и экспериментальных средств оптимизации и генерации кода:

- внутреннее представление, позволяющее эффективно проводить анализ и трансформации программы;
- гибкий механизм сборки приложений, простота добавления новых компонентов, библиотек;
- простые средства встраивания дополнительных или альтернативных проходов оптимизации;
- внутреннее представление на стадии кодогенерации (DAG), позволяющее применять различные методы выбора команд;
- развитые средства описания целевых процессоров и внутреннего представления этих описаний;
- разнообразные инструменты отладки и тестирования, включая дампы промежуточного представления на каждом проходе, отладочные дампы, визуализацию графа выбора, генерацию случайных тестов.

Все это делает LLVM привлекательным инструментом для разработки экспериментальных и промышленно используемых инновационных методов анализа и оптимизации, а также инструментальных средств разработки для встроенных систем.

Список литературы

1. **Вьюкова Н. И., Галатенко В. А., Самборский С. В.** Совместное решение задач выбора и планирования команд в условиях дефицита регистров // Программная инженерия. 2012. № 2. С. 35—40.
2. **Самборский С. В.** Формулировка задачи планирования линейных и циклических участков кода // Программные продукты и системы. 2007. Т. 79. № 3. С. 12—16.
3. **Вьюкова Н. И., Самборский С. В.** Многомерная конвейеризация циклов // Программные продукты и системы. 2008. № 4. С. 8—13.
4. **The LLVM Compiler** Infrastructure. URL: http://llvm.org.
5. **GCC**, the GNU Compiler Collection. URL: http://gcc.gnu.org.
6. **Clang:** a C language family frontend for LLVM. URL: http://clang.llvm.org.
7. **Clang** Static Analyzer. URL: http://clang-analyzer.llvm.org
8. **Галатенко В. А., Костюхин К. А., Шмырев Н. В., Аристов М. С.** Использование свободно распространяемых средств статического анализа исходных текстов программ в процессе разработки программ для операционных систем реального времени // Программная инженерия. 2012. № 5. С. 2—6.
9. **Clang** 3.3 Documentation. URL:http://clang.llvm.org/docs.
10. **ELLCC** — The Embedded LLVM Compiler Collection. URL:http://ellcc.org.
11. **EmbToolkit.** URL: http://www.embtoolkit.org/index.php/Main_Page.
12. **Lattner C.** LLVM: An Infrastructure for Multi-Stage Optimization. Computer Science Dept., University of Illinois at Urbana-Champaign. Masters Thesis. 2002.
13. **Lattner C. and Adve V.** LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation // Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO'04). Palo Alto, California, Mar. 2004. URL: http://www.cgo.org/cgo2004/papers/06_76_lattner_c.pdf.
14. **Serebryany K., Vyukov D.** Finding races and memory errors with compiler instrumentation. AddressSanitizer, ThreadSanitizer. GNU Tools Cauldron, 10 July 2012. URL: http://gcc.gnu.org/wiki/cauldron2012?action=AttachFile&do=get&target=kcc.pdf.
15. **Giuliano de Souza Vilela Cid.** A LLVM based development environment for embedded systems software targeting the RISCO processor. Universidade Federal do Rio Grande do Norte, 2010. URL: http://risco-llvm.googlecode.com/files/paper.pdf.
16. **SPIR 1.0** Specification for OpenCL. Khronos Group — OpenCL Working Group — SPIR subgroup, 2012-08-24. URL: http://www.khronos.org/registry/cl/specs/spir_spec-1.0-provisional.pdf.

УДК 004.4'22

А. Н. Терехов, д-р физ.-мат. наук, проф., зав. каф.,
e-mail: Andrey.Terekhov@lanit-tercom.com,
Т. А. Брыксин, стар. преп.,
Ю. В. Литвинов, стар. преп.,
Санкт-Петербургский государственный университет

QReal: платформа визуального предметно-ориентированного моделирования

Описан подход к разработке программного обеспечения, основанный на применении специализированных графических языков. Рассмотрены существующие на настоящее время платформы для создания специализированных визуальных сред разработки, описана платформа QReal, разрабатываемая на кафедре системного программирования СПбГУ. Дано описание общей архитектуры системы, основных ее функциональных особенностей, приведены примеры ее практического применения.

Ключевые слова: *предметно-ориентированное моделирование, визуальное программирование, DSM-платформы, MetaCASE-системы*

Введение

Разработка больших программных систем — непростая задача. Связано это, в частности, с тем, что такое программное обеспечение сложно воспринимаемо, его трудно представить себе визуально, а держать в голове сотни тысяч строк программного кода одновременно невозможно. Одним из подходов, который призван решать эту задачу, является визуальное моделирование. В нем программа представляется в виде набора моделей на графических языках, описывающих систему с различных точек зрения, от требований до описания деталей реализации. Визуальное моделирование активно применяется сейчас при анализе и проектировании. Однако в подавляющем большинстве случаев визуальные модели используют только как часть технической документации и как средства передачи информации программистам, тогда как было бы полезнее автоматически генерировать код системы по визуальным моделям.

В случае с визуальными языками общего назначения[1] полноценное создание системы с использованием только графических языков чрезвычайно сложно по ряду причин, основной из которых является семантический разрыв между моделями и кодом [1]. Однако зачастую оказывается, что можно создать свой визуальный язык под конкретную предметную область или даже конкретную задачу. Более того, используя знания о предметной области при разработке инструментария для этого языка, можно добиться полной генерации работающего исходного кода системы по визуальным моделям. Сами модели при этом могут оставаться достаточно близкими к предметной области, для того чтобы их могли создавать и использовать да-

[1] Визуальный язык, так же как и текстовый, может быть определен как совокупность артефактов, синтаксических правил, выражающих допустимые способы связи артефактов, и семантики, выражающей значение комбинации артефактов. В текстовых языках в роли артефактов выступают текстовые символы, образующие цепочки, в визуальных языках — графические символы, расположенные в двухмерном пространстве.

же непрограммисты. Такой подход называется предметно-ориентированным визуальным моделированием (*Domain-Specific Modelling*, DSM). Результаты ряда исследований [2—4] указывают на то, что данный подход оказывается весьма эффективным, в том числе отмечается повышение производительности труда программиста в 3—10 раз.

Разумеется, создание визуального языка, редакторов, генераторов и других инструментальных средств для этого языка "с нуля" под каждую конкретную задачу было бы неоправданно трудоемким. По этой причине существуют инструментальные средства, позволяющие в большой степени автоматизировать этот процесс, такие инструменты называют DSM-платформами или metaCASE-системами. Такие системы позволяют создавать визуальную технологию (называемую DSM-решением) за время порядка нескольких дней, что делает это экономически оправданным даже для небольших проектов. Несмотря на то что существуют зрелые коммерческие и исследовательские DSM-платформы (например, MetaEdit+, Eclipse GMP), предметно-ориентированное моделирование применяется все еще довольно редко. Такое положение связано, в частности, с недостатками существующих DSM-платформ (подробнее о которых будет сказано далее) и отсутствием методологической базы для их применения, что свидетельствует о необходимости проведения дальнейших исследований в этой области.

Один из исследовательских проектов в области предметно-ориентированного моделирования, именуемый QReal [5, 6], реализуется на кафедре системного программирования Санкт-Петербургского государственного университета с 2007 г. и базируется на более чем двадцатилетнем опыте коллектива кафедры в области разработки графических языков [7—12]. Проект ставит перед собой цель создания DSM-платформы, достаточно простой в использовании, чтобы свой визуальный язык в течение нескольких часов мог создать даже человек, который применяет ее впервые. Платформа при этом должна быть достаточно функциональной, чтобы с ее помощью можно было разрабатывать большие и сложные визуальные технологии[2]. В настоящей статье представлены текущие результаты выполнения этого проекта.

Существующие DSM-платформы

На данный момент существует несколько известных DSM-платформ, таких как Eclipse Graphical Modeling Project, MetaEdit+, Visual Studio Visualization and Modeling SDK (ранее называвшийся Microsoft DSL Tools) и большое число академических разработок.

Graphical Modeling Project[3, 4] (GMP) — набор проектов, разрабатываемых в рамках проекта с открытым исходным кодом Eclipse, связанных с разработкой редакторов визуальных языков. Наиболее зрелыми проектами из этого направления являются библиотеки Graphical Editing Framework (GEF) и Eclipse Modeling Framework (EMF). Средство GEF предназначено для создания векторных графических редакторов, обладающих широкими возможностями взаимодействия с пользователем. Средство EMF используют для генерации классов модели и инструментальной поддержки для визуальных языков по метамоделям[5], представленным в различных форматах (XMI[6], аннотированных Java-классов и др.). Цель проекта GMP — объединить эти и ряд других технологий в единую DSM-платформу на базе среды разработки Eclipse.

На данный момент GMP стал де-факто стандартом для научных исследований, связанных с визуальным моделированием и метамоделированием. Однако этот проект плохо подходит для промышленного использования потому, что на данный момент находится в стадии активного развития. Постоянно вносятся изменения, документация либо отсутствует, либо устарела. Кроме того, GMP состоит из довольно слабо связанных между собой компонентов, развивающихся независимо. Целостной технологии, поддерживающей весь цикл разработки визуального языка, пока не создано. Это обстоятельство приводит к довольно большой сложности разработки в рамках GMP и высокому порогу освоения технологий. Для создания даже простого языка требуется строить несколько моделей (модели абстрактного и конкретного синтаксисов языка[7], модель инструментов, где описываются пункты меню и подобные вещи, модель соответствия между этими моделями и т. д.). Как следствие, потребность в создании исследовательских DSM-платформ "с нуля" все-таки есть. Такие платформы существуют и продолжают появляться новые, несмотря на наличие GMP. Описания нескольких таких платформ можно найти в работах [14—16].

DSM-платформа MetaEdit+[8] является на настоящее время самой зрелой, она активно используется организациями-разработчиками программного обес-

---

[2] Домашняя страница проекта QReal на GitHub: URL: https://github.com/qreal/qreal

[3] Более подробный обзор на русском языке см. в работе [13].

[4] Graphical Modeling Project homepage: URL: http://www.eclipse.org/modeling/gmp/

[5] Метамодель — модель, описывающая синтаксис визуального языка. Этот способ описания синтаксиса несколько отличается от принятого для текстовых языков (форм Бэкуса-Наура и их аналогов), но точно так же описывает множество корректных программ. Метамодель перечисляет сущности визуального языка, их свойства, допустимые связи между сущностями.

[6] XML Metafata Intercharge, стандарт хранения визуальных моделей, см. URL: http://www.omg.org/spec/XMI/2.4.1/

[7] Абстрактный синтаксис отвечает за логическую структуру модели, конкретный — за ее внешний вид. Определение этих и других базовых понятий метамоделирования дано в работе [1].

[8] DSM-платформа MetaEdit+: URL: http://www.metacase.com/

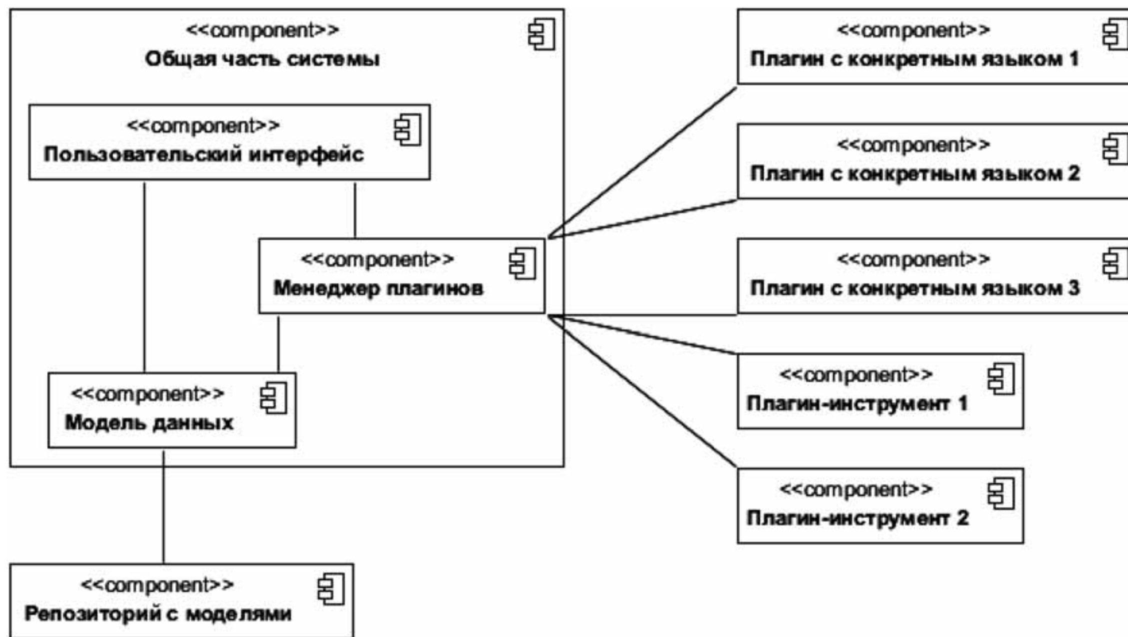


Рис. 1. Общая архитектура DSM-решений на основе QReal

печения. Платформа получила развитие из академической разработки MetaEdit, превратившись в коммерчески успешный проект. Средства MetaEdit+ позволяют задавать метамодели визуальных языков с использованием визуального метаязыка GOPPRR (аббревиатура, образованная основными сущностями метаязыка — *graph*, *object*, *property*, *port*, *role*, *relationship*). В состав платформы входит графический редактор для задания внешнего вида сущностей создаваемого языка, а также набор средств для описания правил генерации кода по визуальным моделям. Сами правила задаются в текстовом виде на специальном языке, существует возможность вызывать одни правила из других, связь между правилами задается в графическом виде.

Несмотря на то что авторы MetaEdit+ активно публикуют свои научные результаты (см., например, книгу [17]), исходные коды системы закрыты, а сама система стоит весьма дорого. Это обстоятельство не позволяет в полной мере использовать опыт, методы и средства, полученные в ходе выполнения проекта MetaEdit+ и реализовывать на его основе новые функциональные возможности.

Система Visual Studio Visualization and Modeling SDK (VMSDK)[9], ранее Microsoft DSL Tools, — расширение для среды разработки Microsoft Visual Studio, позволяющее генерировать редакторы визуальных языков в виде подключаемых модулей для Visual Studio. Метамодель языка задается в графическом виде, с ней связывается внешний вид элементов языка. Внешний вид элемента можно выбирать из фиксированного числа геометрических фигур, либо задать произвольную иконку. Если требуется использовать какое-то нестандартное отображение элемента (например, не прямоугольник, а ромб), необходимо ручное кодирование на языке C#. Несмотря на то что система VMSDK распространяется бесплатно, ни она сама, ни сгенерированные с ее помощью редакторы не могут работать вне среды Visual Studio, которая стоит весьма дорого и имеет закрытые исходные коды.

Таким образом, существует необходимость разработки DSM-платформ для исследовательских целей, проект QReal — одна из таких разработок.

Проект QReal

Изначально проект QReal был посвящен созданию среды моделирования, включающей в себя набор редакторов UML 2.0, однако требуемые редакторы, обладающие необходимыми функциями, были слишком похожи друг на друга, чтобы создавать их друг за другом кодированием "вручную". Возникла потребность в средствах быстрого создания визуальных языков и инструментальной поддержки для них.

Для автоматизации процесса создания таких редакторов метамодели соответствующих языков (внешнего вида и свойств элементов языка и связей между этими элементами) должны быть описаны формально с помощью некоторого языка[10], текстового или графического (или даже набора языков). Далее эти описания могут быть использованы двумя различными способами, приведенными ниже.

[9] Visual Studio Visualization and Modeling SDK (бывший DSL SDK): URL: http://archive.msdn.microsoft.com/vsvmsdk

[10] В случае текстовых языков для этого чаще всего используют формы Бэкуса-Наура.

- По описанию метамодели генерируется программный код, который тем или иным образом дополняет код самого DSM-решения, привнося в него особенности данного языка. Такой подход к созданию редактора назовем генеративным.
- В состав DSM-решения входит механизм, который по запросу читает нужные данные прямо из описания метамодели и обрабатывает их нужным образом (например, считывает количество и тип свойств элементов и отображает их в редакторе свойств). Таким образом, во время работы с редактором происходит интерпретация метамодели языка. Такой подход назовем интерпретативным.

Первый подход более прост в реализации и обычно дает определенный выигрыш в скорости работы. Однако второй подход позволяет создавать более гибкие инструменты, например, дает возможность менять метамодель языка динамически в процессе работы с редактором этого языка.

Исторически в QReal первым был реализован генеративный подход, что сделало архитектуру инструментария более модульной (рис. 1). Конкретные DSM-решения на основе QReal получают дополнением и параметризацией кода самой DSM-платформы. Абстрактная функциональность редакторов (например, такая, как способность двигать и масштабировать элементы на диаграммах, соединять их связями, хранить в репозитории значения их свойств и т. п.) максимально абстрагируется и формирует так называемое "ядро" системы, в то время как специфика каждого конкретного визуального языка оформляется в виде подключаемого модуля-плагина. Код каждого такого модуля генерируется автоматически по описанию метамодели языка, компилируется в плагин и не зависит от каких-либо других частей QReal. Плагины загружаются диспетчером модулей, который, в свою очередь, предоставляет интерфейс остальным частям QReal для доступа к информации, содержащейся в плагинах редакторов — именам и типам элементов и связей между ними, изображениям элементов, числу и типу их свойств, правилам соединения элементов связями и т. п.

Помимо плагинов редакторов в QReal существуют также плагины инструментальных средств. В них выносятся функциональные возможности различных средств CASE-пакета, которые могут быть полезны проектировщику. К их числу относятся: механизм версионирования моделей; генераторы кода; отладчики и интерпретаторы; механизм задания и проверки ограничений на создаваемые модели; механизм задания и проведения рефакторинга моделей, а также ряд других. Это позволяет повторно использовать инструменты в создаваемых на базе QReal DSM-решениях — стоит добавить какую-нибудь функциональность в "ядро" системы или оформить ее в отдельный подключаемый модуль, и все DSM-решения при желании получат эту функциональность автоматически.

Средства метамоделирования

Для описания метамодели разрабатываемого языка в QReal применяется два описанных ниже подхода.

- *Графический*: метамодель задается разработчиком языка графически в специальном визуальном редакторе, предназначенном для редактирования метамоделей (называемом "метаредактор") с помощью простого визуального языка, являющего аналогом MOF[11]. Для описания представлений элементов языка на диаграммах используется графический редактор форм, позволяющий либо создавать новые векторные изображения из набора примитивов, либо загружать уже готовые растровые.
- *Текстовый*: метамодель описывается с помощью XML-формата, графические изображения элементов задаются на разработанном авторами языке WTF (*widget template format*), являющимся расширением языка описания векторной графики SVG.

Следует отметить, что в QReal эти два подхода являются взаимозаменяемыми, поскольку XML-описание метамодели языка может быть как загружено в метаредактор, так и сгенерировано из него. Исторически в QReal текстовый способ задания метамоделей появился раньше, однако в настоящее время графический метаредактор используют чаще как более выразительное средство.

Типичный процесс создания нового языка в QReal проходит следующим образом. Используя метаредактор, автор языка создает модель, описывающую желаемый язык — определяет сущности этого визуального языка и связи между ними (в метаязыке им соответствуют блоки "Элемент" и "Связь"). С помощью метаредактора можно также задавать отношения наследования между элементами на диаграмме и отношения допустимой вложенности одних элементов в другие как в контейнеры. Эти отношения на диаграмме указываются направленными ассоциациями. Кроме этого существует возможность задавать некоторые дополнительные свойства создаваемого графического редактора, поддержка которых осуществлена в "ядре" QReal (способность "вытягивать" из элементов определенные связи, сортировать вложенные элементы и уметь их скрывать для элементов-контейнеров и некоторые другие).

Помимо абстрактного синтаксиса важно уметь задавать, как будут внешне выглядеть элементы разрабатываемого языка. Для этого в QReal реализован редактор формы фигур, который, по сути, представляет собой векторный графический редактор, но обладает рядом особенностей, отражающих специфику его использования. Например, есть возможность задания положения элементов управления, отображаемых на фигуре, позволяющих изменять свойства элемента.

[11] MetaObject Facility: URL: http://www.omg.org/mof/

Это делает возможным отображение и редактирование логических свойств прямо на диаграмме при моделировании с помощью этого языка — например, отредактировать имя элемента прямо на диаграмме или выбрать значение свойства перечислимого типа из выпадающего списка.

В QReal для разработанного метаязыка была создана инструментальная поддержка и соответствующая инфраструктура, обеспечивающие сквозной процесс создания графических редакторов: разработчик может спроектировать новый визуальный язык, скомпилировать подключаемый модуль соответствующего графического редактора и подключить его к QReal, не выходя из системы.

Также существует набор перечисленных далее дополнительных инструментов, позволяющих расширить функциональность создаваемых DSM-решений.

- Средства описания исполнимой семантики позволяют задавать операционную семантику для элементов динамических языков, что позволяет автоматически создавать для них в QReal визуальные интерпретаторы и отладчики. Про то, как по описанию операционной семантики могут быть созданы отладчики соответствующих языков (на примере платформы Eclipse), см. в работе [18].
- Средства задания ограничений позволяют задавать условия, которые должны выполняться применительно к элементу или к группе элементов на диаграмме. Например, для элемента "Начало" диаграммы деятельности UML можно было бы задать, что в него не должно входить никаких связей, а исходить должна только одна. При использовании получившегося редактора данные ограничения будут проверяться автоматически на создаваемых моделях, при необходимости выдавая предупреждения и сообщения об ошибках.
- Средства задания рефакторинга позволяют описывать трансформации графов моделей, которые в дальнейшем можно выполнить при моделировании с помощью данного языка. Это могут быть общие для всех языков элементы рефакторинга (например, выделение части диаграммы в поддиаграмму) или характерные элементы именно для данного языка (например, "Выделение интерфейса" для диаграммы классов UML).

Кроме того, в QReal реализована технология быстрого прототипирования визуального языка прямо в процессе моделирования, так называемое "метамоделирование на лету". При первоначальном анализе предметной области, когда визуального языка еще нет и непонятно, какие в нем могут быть сущности и связи, диаграммы обычно рисуют в виде набросков на бумаге. "Метамоделирование на лету" позволяет рисовать такие наброски в CASE-системе, уточняя и формализуя визуальный язык в процессе рисования. Среда QReal в режиме "метамоделирования на лету" позволяет в процессе рисования диаграммы добавлять и удалять элементы в палитре, менять их свойства и внешний вид без метаредактора. С точки зрения пользователя процесс создания визуального языка выглядит как добавление нового узла в палитру, задание его внешнего вида (если надо, иначе по умолчанию он будет прямоугольником, что для первых набросков оказывается достаточно), определение некоторых его свойств, использование его на диаграмме, редактирование элемента и его свойств по результатам использования. Получившийся метаязык можно сохранить и затем открыть в метаредакторе, чтобы доопределить для него дополнительные функциональные возможности, недоступные в режиме "метамоделирования на лету", например, ограничения, правила рефакторинга, наследование и отношения вложенности между узлами. Таким образом, первые этапы разработки визуального языка в значительной степени автоматизируются. Их результаты не теряются, постепенно уточняются, становятся полноценным визуальным языком.

Средства моделирования

Получаемое типовое предметно-ориентированное решение на базе QReal включает в себя перечисленные далее инструментальные средства.

- **Графический пользовательский интерфейс**, включающий в себя обозреватель модели, редактор свойств элементов, рабочую область редактора диаграмм, палитру доступных для моделирования элементов, окно для отображения предупреждений и сообщений об ошибках, а также набор меню и панелей главного окна для управления остальными инструментами.
- **Репозиторий**, представляющий собой простую объектно-ориентированную базу данных, хранящий все создаваемые в QReal модели. Репозиторий представлен в виде отдельной динамически загружаемой библиотеки, в результате чего он может быть использован сторонними программами (например, генераторами кода или анализаторами моделей) через специальный программный интерфейс.
- **Средства для работы с логическими и графическими моделями**. Логическая модель системы отражает ее внутреннюю структуру, и это то, с чем работают генераторы и другие инструменты. Графическая модель системы — это набор диаграмм, отображающих ее логическую модель, с ней работает пользователь.
- **Средства многопользовательской работы**, основанные на создаваемых новых версиях моделей. Реализованы традиционные операции систем контроля версий (сохранение и получение данных с удаленного сервера, переключения между версиями), а также возможность визуального сравнения версий одной и той же модели.
- **Пошаговые отладчики и интерпретаторы**, созданные либо с помощью описания семантики элемен-

тов языка при метамоделировании, либо закодированные "вручную" разработчиком языка.

- **Генераторы кода на текстовых языках и других артефактов по визуальным моделям.** QReal предоставляет библиотеки, позволяющие достаточно удобно реализовать генератор в любой целевой текстовый язык, требуемый в конкретном предметно-ориентированном решении. Пока в QReal отсутствуют более высокоуровневые средства автоматизированного создания генераторов кода, поэтому генераторы реализуются с использованием соответствующих библиотек кодированием на языке C++.
- **Механизм реализации рефакторинга**, позволяющий осуществлять преобразования моделей, заданные разработчиком языка на фазе метамоделирования, а также общие для всех языков (например, переименование элементов по шаблону или выделение в поддиаграмму).
- **Механизм проверки ограничений**, который автоматически проверяет правила, заданные на экземпляры элементов на диаграммах (например, проверка, что значение определенного свойства входит в определенный диапазон, или что из элемента не исходит больше связей, чем задано).
- **Механизм распознавания жестов мышью** [19], используемый для быстрого создания элементов на диаграммах и связей между ними. Так, при создании редактора каждому элементу ставится в соответствие отдельный жест мышью, и если в процессе моделирования разработчик зажмет правую кнопку мыши и сделает похожий жест, на диаграмме появится соответствующий элемент. Связь между элементами можно создать, осуществив жест мышью, начинающийся и заканчивающийся на интересующих элементах на диаграмме.

Все представленные выше инструментальные средства реализованы в виде подключаемых модулей, поэтому по необходимости выбранное DSM-решение может быть сконфигурировано произвольным их числом. Это бывает полезно, поскольку если целью стоит создание легковесного редактора (например, для целей обучения), отладчики или рефакторинг моделей его только усложнят.

В последнее время в проекте QReal довольно много внимания уделяется вопросам удобства использования средств визуального моделирования и удобству реализации собственно процесса моделирования. Одни из самых частых операций при моделировании — это создание и удаление элементов и связей на диаграммах, а также "красивое" расположение их относительно друг друга. Автоматизируя или упрощая эти операции, можно сократить время и число рутинных действий проектировщика, облегчив его работу. В этом направлении кроме рассмотренного распознавания жестов мышью был реализован еще один способ быстрого создания связей. Он заключается в том, что при выделении объекта на диаграмме вокруг него появляются несколько вспомогательных графических элементов (в форме разноцветных кружков), каждый из которых ассоциирован с видами связей, которые из данного элемента могут исходить. При нажатии на них из элемента можно "вытащить" связь. Если пользователь при этом отпускает кнопку мыши на свободном пространстве диаграммы, ему предлагается список элементов, которые можно соединить с выбранным данным типом связи. Если кнопка мыши отпускается на существующем элементе, то система проверяет, можно ли соединить эти элементы выбранным типом связи, и решает, стоит ли создавать данную ассоциацию или нет.

Кроме перечисленных выше, было сделано много мелких улучшений, таких как средства автоматической раскладки элементов на диаграмме, режим "перпендикулярных связей" (связи задаются ломаными линиями, участки которых соединяются между собой под прямым углом), а также ряд эвристик языка ДРАКОН [20] — создание групп элементов, возможность вставки элемента "внутрь" связей и некоторые другие.

Апробация

Самой успешной на данный момент визуальной предметно-ориентированной технологией, созданной с помощью QReal, является QReal:Robots [21] — средство программирования роботов Lego Mindstorms NXT 2.0[12]. Ситуация, в которой появился этот проект, весьма типична для предметно-ориентированного моделирования. Есть достаточно узкая предметная область, потребность в нетривиальных средствах программирования в этой области и DSM-платформа, позволяющая быстро создать нужные средства программирования.

Постановка задачи. Изучение информатики в школах базируется на понятии "исполнителя" — некоторого объекта, который выполняет команды, описанные в программе, в некотором окружении. В качестве такого исполнителя до сих пор активно применяется "черепашка" LOGO[13], но она постепенно вытесняется робототехническими конструкторами, в частности, Lego Mindstorms NXT, которые имеют ряд преимуществ. Прежде всего, они материальны, кроме того, они имеют датчики, с помощью которых могут "видеть" внешнюю среду, к тому же детям интересно собирать из конструктора робот, который они потом будут программировать. Существуют методические пособия по преподаванию информатики с использованием этого конструктора в школах, например, работа [22].

---

[12] LEGO Mindstorms homepage: URL: http:// mindstorms.lego.com/en-us/ Default.aspx

[13] См., например, MyRobot, язык программирования Лого: URL: http://myrobot.ru/logo/aboutlogo.php

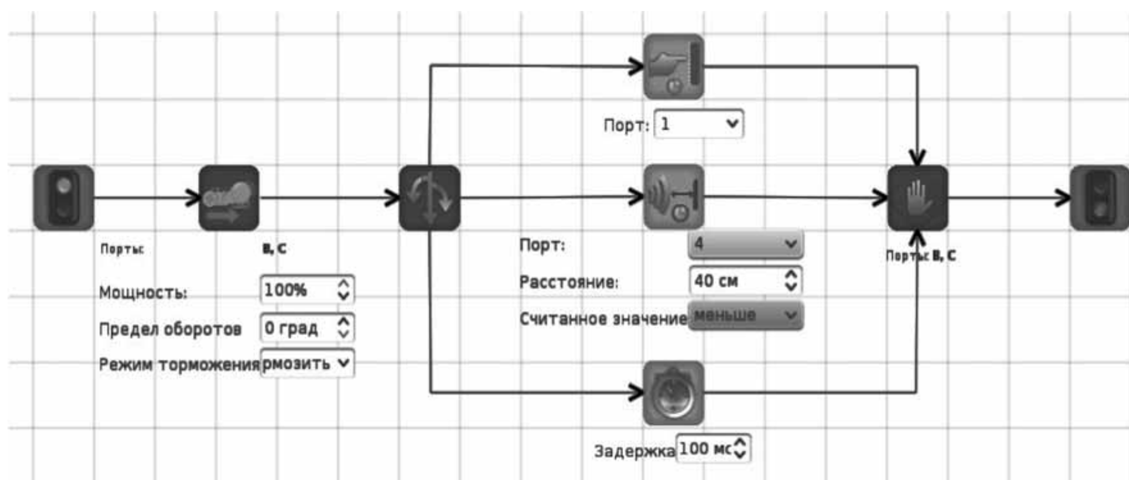


Рис. 2. Пример программы в QReal:Robots

Однако программировать такие роботы сложно потому, что из одного набора деталей можно собрать самые разные конструкции, так что в базовом наборе команд будут не команды вида "вперед на 20 шагов", "влево на 90 градусов", как в "черепашке", а низкоуровневые команды вида "мотор, подключенный к порту А, включить с мощностью 70 % на три полных оборота". По этой причине для обучения школьников 5—6 классов программированию обычно используют визуальные среды Robolab [23] и NXT-G[14], где программа является набором блоков, каждый из которых представляет элементарную команду роботу. Такие программы гораздо нагляднее кода на языках типа С.

Существующие визуальные среды имеют ряд недостатков: плохая поддержка сложных математических выражений, отсутствие средств отладки, устаревший пользовательский интерфейс, неполная русификация, высокая стоимость. С точки зрения школьных преподавателей они настолько существенны, что есть насущная потребность в создании новой визуальной среды. Такой средой и должна стать система QReal:Robots. С научной точки зрения она интересна тем, что создавалась с помощью QReal и является хорошим примером визуальной технологии, созданной с помощью DSM-платформы, которая дошла до фазы практического внедрения.

Функциональные возможности системы QReal:Robots. Концептуально программа на языке QReal:Robots представляется в виде набора элементарных команд роботу, связанных стрелками, означающими передачу управления между блоками. Пример программы представлен на рис. 2.

Созданную на визуальном языке программу можно исполнить прямо на компьютере, посылая команды роботу через Bluetooth или USB. При этом текущий исполняемый блок будет подсвечиваться, будут выводиться показания сенсоров и значения переменных и появится возможность остановить или даже изменить программу в процессе выполнения. По диаграмме также можно сгенерировать код на языке С для операционной системы nxtOSEK[15], загрузить его на робот и исполнить на роботе. При этом отладочная информация выводиться не будет, но робот будет способен действовать автономно, без связи с компьютером. Кроме того, программу можно исполнить вообще без робота, на двухмерной модели, запущенной внутри среды QReal:Robots. Двухмерная модель реализует фиксированную конфигурацию робота (трехколесную тележку, типичную для задач, требующих перемещения робота в пространстве, например, движения по линии или игре в робофутбол). Однако есть возможность задать конфигурацию и положение сенсоров, элементы внешнего мира, включая стены, цветные линии и области на полу.

Визуальный язык состоит из блоков (порядка двадцати) и одного вида связи, поддерживает математические выражения с арифметическими операциями и тригонометрическими функциями, использование переменных, доступ к текущим значениям сенсоров прямо из выражения. Присутствует поддержка параллельно исполняемых фрагментов программы, условного оператора и циклов.

Использование DSM-платформы QReal при разработке QReal:Robots. Наиболее полезной DSM-платформа оказалась при разработке визуального языка и редактора для него. Первый прототип языка был создан в течение нескольких часов, причем большую часть времени занял поиск подходящих иконок для блоков. Визуальный язык создавался в метаредакторе

[14] LEGO web site, NXT-G download page: URL: http://service.lego.com/en-us/helptopics/?questionid=2655

[15] nxtOSEK — операционная система реального времени, работающая на роботах NXT и совместимая со стандартной прошивкой. Является самой быстрой на данный момент ОС для NXT. Более подробно см. URL: http://lejos-osek.sourceforge.net/

QReal. Метамодель языка активно использовала такие возможности метаязыка, как наследование (например, в иерархии *"абстрактный блок" — "блок управления двигателем" — "блок управления мощностью" — "моторы вперед"/"моторы назад"*), задание перечислений, использование нескольких образов одного логического элемента метамодели на диаграмме. Отметим, что узел *"абстрактный блок"* был использован в метамодели дважды, поскольку от него наследуются остальные блоки и обилие входящих в него связей типа *"наследование"* загромождало бы диаграмму. Первый работающий прототип системы был представлен кибернетикам через неделю после начала разработки. Он включал в себя язык и интерпретатор, управлявший роботом по Bluetooth.

Инструментарий QReal не смог помочь при написании интерпретатора диаграмм, код интерпретатора пришлось писать вручную. Среда QReal использовалась только в виде репозитория, откуда была взята информация о диаграммах. Сам интерпретатор был написан вручную на C++. Это вполне закономерно, поскольку интерпретатор включает в себя код, сильно связанный с предметной областью — реализацию взаимодействия с роботом по Bluetooth, систему команд робота, особенности инициализации датчиков и т. д., что не имеет смысла обобщать и выносить на уровень метатехнологии. Единственное, что, как представляется, имеет смысл сделать — формализовать класс языков, "похожих на блок-схемы", и вынести в метатехнологию общие части, касающиеся организации процесса вычисления. Семантика таких языков обычно основывается на понятии токена исполнения и формализуется с помощью сетей Петри. Эти языки встречаются достаточно часто, поэтому такое обобщение представляется целесообразным (например, диаграммы активностей UML 2 имеют именно такую семантику). При этом знания, отражающие специфику предметной области (например, посылка конкретной команды на робота) могут быть вынесены в скрипты, выполняющиеся при попадании токена исполнения в блок. Работы над таким обобщением в QReal на данный момент уже ведутся.

При разработке генератора QReal использовался как библиотека классов, которую использовал генератор, написанный вручную на C++. Генератор принципиально содержит в себе много знаний о предметной области, однако также содержит много кода, общего для всех предметных областей. По этой причине степень повторного использования при разработке генератора существенно выше, чем в случае интерпретатора. В проекте QReal были попытки полностью избежать ручного кодирования при разработке генератора с помощью описания правил генерации на специальном языке, для генератора QReal:Robots даже была выполнена новая реализация на такой системе. Однако практика показала, что такой подход оказывается менее удобным, даже чем ручное кодирование с использованием библиотек разработки генераторов. Связано это с тем обстоятельством, что генератор все-таки содержит много знаний о предметной области, и технология способна упростить лишь и без того несложные вещи, такие как обход модели. Цена, которую надо заплатить за использование технологии — знание еще одного языка, языка описания правил генерации, — в данном случае оказалась выше, чем получаемая от нее выгода.

Заключение

На основе полученного опыта авторы полагают, что при наличии мощной инструментальной поддержки использование предметно-ориентированных визуальных языков реализует принципиально новый подход к созданию сложных систем с довольно низким порогом вхождения для новичков и многократным увеличением производительности профессионалов. Для этого в распоряжении проектировщиков должен быть обширный набор программных средств, начиная от визуальных редакторов и репозитория и заканчивая средствами отладки контроля версий и процессов рефакторинга создаваемых моделей. Платформы DSM помогают ускорить и автоматизировать процесс создания подобных инструментальных средств, позволяя применять данный подход к разработке программного обеспечения для различных предметных областей и наборов задач. При этом DSM-решение создает, как правило, один очень опытный разработчик (или небольшая команда опытных разработчиков), а пользуется этим решением большое число людей, которые иногда даже могут не обладать навыками программирования.

Существующие на данный момент DSM-платформы либо коммерческие, либо активно развиваются и крайне сложны в использовании. Описываемый в статье проект QReal ставит своей целью исследование, реализацию и апробацию таких инструментов с упором на удобство использования, в том числе и непрофессионалами.

На данный момент среда QReal использовалась для создания целого ряда визуальных технологий, самой зрелой из которых стала система QReal:Robots. Команда, использовавшая QReal:Robots как основное средство программирования, выступала на нескольких соревнованиях по робототехнике и показала достойные результаты; с использованием среды проводились занятия со школьниками. Существуют и другие технологии, созданные с помощью QReal. В их числе следующие технологии: программирования мобильных телефонов; моделирования генераторов банковских отчетов; моделирования бизнес-процессов; моделирования структуры баз данных, а также другие, для решения более мелких задач. Подход доказал

свою применимость, но существует еще большое число открытых проблем, требующих исследования, например, поддержка эволюции визуальных языков, переиспользование частей визуальных языков, автоматизация процесса создания генераторов, создание и апробация методологий разработки предметно-ориентированных решений.

Список литературы

1. **Кознов Д. В.** Основы визуального моделирования. М.: БИНОМ, 2008.

2. **Weiss P., Lai C. T. R.** Software Product-line Engineering. Addison Wesley Longman, 1999.

3. **Kelly S., Tolvanen J.-P.** Visual domain-specific modelig: benefits and experiences of using metaCASE tools // Proceedings of International workshop on Model Engineering, ECOOP, 2000 / J. Bezivin, J. Ernst. eds. URL: http://www.dsmforum.org/papers/visual_domain-specific_modelling.pdf

4. **Kieburtz R.** et al. A software engineering experiment in software component generation // Proceedings of 18th International Conference on Software Engineering. March, 1996. Berlin. IEEE Computer Society Press, 1996. P. 542—552.

5. **Терехов А. Н., Брыксин Т. А., Литвинов Ю. В., Смирнов К. К., Никандров Г. А., Иванов В. Ю., Такун Е. И.** Архитектура среды визуального моделирования QReal // Системное программирование. 2000. Т. 4. С. 171—196.

6. **Кузенкова А. С., Дерипаска А. О., Таран К. С., Подкопаев А. В., Литвинов Ю. В., Брыксин Т. А.** Средства быстрой разработки предметно-ориентированных решений в metaCASE-средстве QReal // Научно-технические ведомости СПбГПУ, Информатика, телекоммуникации, управление. 2011. Вып. 4 (128). С. 142—145.

7. **Долгов П., Иванов А., Кознов Д., Лебедев А., Мурашева Т., Парфенов В., Терехов А.** Объектно-ориентированное расширение технологии RTST // Записки семинара кафедры системного программирования "CASE-средства RTST++". СПб.: Изд-во С.-Пб университета, 1998. Вып. 1. С. 17—36.

8. **Иванов А. Н., Кознов Д. В., Мурашова Т. С.** Поведенческая модель RTST // Записки семинара кафедры системного программирования "CASE-средства RTST++". 1998. СПб.: Изд-во СПб. С. 37.

9. **Терехов А. Н.** RTST — технология программирования встроенных систем реального времени // Записки семинара Кафедры системного программирования "CASE-средства RTST++". 1998. СПб.: Изд-во СПб. университета. Вып. 1. С. 3.

10. **Терехов А. Н., Кияев В. И., Комаров С. Н.** Принципы информатизации системы управления в Санкт-Петербургском Государственном Университете // Вестник Санкт-Петербургского университета, Серия 8: Менеджмент. 2004. № 2. С. 151—200.

11. **Парфенов В. В., Терехов А. Н.** RTST — технология программирования встроенных систем реального времени // Системная информатика. 1997. № 5. С. 228—256.

12. **Terekhov A. N., Romanovskii K. Yu., Koznov D. V., Dolgov P. S., Ivanov A. N.** RTST++: Methodology and CASE tool for the development of information systems and software for real-time systems // Programming and Computer Software. 1999. Vol. 25. № 5. P. 276—281.

13. **Сорокин А. В., Кознов Д. В.** Обзор Eclipse Modeling Project // Системное программирование. 2010. Т. 5. № 1. С. 6—32.

14. **De Lara J., Vangheluwe H.** AToM3: A Tool for Multi-formalism and Meta-modelling. // Proc. of Conference "Fundamental approaches to software engineering". Berlin: Springer, 2002. P. 174—188.

15. **Sukhov A. O., Lyadova L. N.** Metalanguage: a Tool for Creating Visual Domain-Specific Modeling Languages // In Proceedings of the 6th Spring/Summer Young Researchers' Colloquium on Software Engineering, SYRCoSE. 2012. P. 42—53.

16. **Nianping Z., Grundy J., Hosking J., Na Liu, Shuping Cao, Mehra A.** Pounamu: A meta-tool for exploratory domain-specific visual language tool development // Journal of Systems and Software. 2007. N 8. P. 1390—1407.

17. **Kelly S., Tolvanen J.** Domain-Specific Modeling: Enabling Full Code Generation. Hoboken, New Jersey: Wiley-IEEE Computer Society Press, 2008. 448 p.

18. **Bandener N.** Visual interpreter and debugger for dynamic models based on the Eclipse platform. Diploma Thesis, 2009. URL: http://is.uni-paderborn.de/uploads/tx_dsorexams/Diploma_Thesis_Nils_Bandener.pdf

19. **Осечкина М. С., Брыксин Т. А., Литвинов Ю. В., Кириленко Я. А.** Поддержка жестов мышью в мета-CASE-системах // Системное программирование. Вып. 5: Сб. статей / Под ред. А. Н. Терехова, Д. Ю. Булычева. СПб.: Изд-во СПбГУ, 2010. С. 52—75.

20. **Паронджанов В. Д.** Дружелюбные алгоритмы, понятные каждому. Как улучшить работу ума без лишних хлопот. М.: ДМК-пресс, 2010. 464 с.

21. **Брыксин Т. А., Литвинов Ю. В.** Среда визуального программирования роботов QReal:Robots // Материалы международной конференции "Информационные технологии в образовании и науке". Самара. 2011. С. 332—334.

22. **Филиппов С. А.** Робототехника для детей и родителей. М.: Наука, 2011. 264 с.

23. **Portsmore M.** ROBOLAB: Intuitive Robotic Programming Software to Support Life Long Learning, APPLE Learning Technology Review, Spring/Summer 1999.

ИНФОРМАЦИЯ

8—9 ноября 2013 г. в г. Львов пройдет Четырнадцатая Международная конференция в области обеспечения качества ПО "Software Quality Assurance Days"

Конференция будет посвящена вопросам, связанным с тестированием и обеспечением качества программного обеспечения.

Сайт конференции: http://sqadays.com/article.sdf/sqadays/sqa_days14/about

УДК 681.3

Д. А. Шеенок, аспирант,
КрИЖТ ИрГУПС,
e-mail: dmitryshkras@rambler.ru

Инструментальное средство проектирования оптимальной архитектуры отказоустойчивых программных систем

Рассмотрена задача планирования затрат на достижение необходимого уровня надежности компонентов отказоустойчивого программного обеспечения. Обоснована необходимость применения специализированных алгоритмов оптимизации для решения данной задачи в силу большого числа альтернативных вариантов построения программной архитектуры. Описаны алгоритмы программной системы для решения этой задачи оптимизации с помощью полного перебора возможных вариантов, либо с использованием специализированного генетического алгоритма. Предложен алгоритм применения инструментального средства в жизненном цикле программного обеспечения.

***Ключевые слова:** проектирование программной архитектуры, оптимизация, генетический алгоритм*

Дизайн архитектуры в жизненном цикле программного обеспечения

В любой современной методологии разработки программного обеспечения выделяется этап разработки программной архитектуры. Уже на этапе формирования требований к программному продукту начинается проектирование его архитектуры. Системный архитектор определяет общую структуру каждого архитектурного представления, декомпозицию представлений и интерфейсы взаимодействия элементов. Таким образом, происходит разбиение большой системы на более мелкие части (модули) в соответствии с определенным уровнем абстракции. Архитектурный компонент при этом может быть определен по-разному в зависимости от архитектурного подхода и степени подробности описания архитектуры.

Модульный подход к построению программной системы, как правило, приводит к использованию иерархической структуры взаимодействия ее модулей. Иерархическая схема, отражая функции модулей, одновременно показывает структуру связей между ними. Иерархические структуры системы характеризуются, с одной стороны, вертикальным управлением, когда модули верхнего уровня имеют право вмешательства и координирования работы модулей нижнего уровня. С другой стороны, действия модулей верхнего уровня зависят от информации, полученной в результате функционирования модулей нижних иерархических уровней. Таким образом, сверху вниз идут в основном управляющие воздействия, а снизу вверх — информация о соответствующих решениях и переменных. Число архитектурных уровней в модели архитектуры программного обеспечения (ПО) зависит от особенностей отдельного проекта системы.

Известно, что чем позже будет обнаружена ошибка проектирования, тем дороже обойдется ее исправление разработчику программного продукта. Поэтому для соблюдения требований к надежности разрабатываемой программной системы необходимо уже на стадии дизайна архитектуры тщательно прорабатывать связи между компонентами и устанавливать иерархию их взаимодействия. Компоненты, наиболее часто используемые или архитектурно связанные с множеством других компонентов, оказывают наибольшее

влияние на надежность системы. Зависимости компонентов позволяют неисправности распространяться из компонента в котором она происходит к другим компонентам. Обнаружение отказов во время разработки программного компонента зависит от эффективности процесса его тестирования. Для снижения вероятности сбоя в таких компонентах применяется также введение программной избыточности, которое заключается в использовании разнообразных методов или технологий разработки [1].

При выборе того или иного варианта построения архитектуры отказоустойчивой программной системы руководствуются критериями надежности и затрат на реализацию системы с заданной надежностью. Очевидно, что положения этих критериев противоречат друг другу, так как система с большей надежностью требует больших затрат ресурсов.

Показателем надежности системы является прогнозируемое значение коэффициента готовности системы. Основным ресурсом при разработке являются трудозатраты специалистов. Для расчета этих показателей используется модель архитектуры ПО, параметрами которой являются следующие [2]:

M — число архитектурных уровней в архитектуре ПО;

N_j — число компонентов на уровне j, $j \in \{1, ..., M\}$;

D_{ij} — множество индексов компонентов, зависящих от компонента i на уровне j, $i \in \{1, ..., N_j\}$, $j \in \{1, ..., M\}$;

E_{ij} — множество индексов компонентов, от которых зависит компонент i на уровне j, $i \in \{1, ..., N_j\}$, $j \in \{1, ..., M\}$;

F_{ij} — событие сбоя, произошедшее в компоненте i на уровне j, $i \in \{1, ..., N_j\}$, $j \in \{1, ..., M\}$;

PU_{ij} — вероятность использования компонента i на уровне j, $i \in \{1, ..., N_j\}$, $j \in \{1, ..., M\}$;

PF_{ij} — вероятность появления сбоя в компоненте i на уровне j, $i \in \{1, ..., N_j\}$, $j \in \{1, ..., M\}$;

R_{ij} — вероятность того, что сбой не появится в компоненте i на уровне j, $i \in \{1, ..., N_j\}$, $j \in \{1, ..., M\}$;

PL_{nm}^{ij} — условная вероятность появления сбоя в компоненте m на уровне n при появлении сбоя в компоненте i на уровне j, $i \in \{1, ..., N_j\}$, $j \in \{1, ..., M\}$, $n \in \{1, ..., N_m\}$, $m \in \{1, ..., M\}$;

TA_{ij} — относительное время доступа к компоненту i на уровне j, $i \in \{1, ..., N_j\}$, $j \in \{1, ..., M\}$, которое определяется как отношение среднего времени доступа к компоненту i на уровне j к числу сбойных компонентов на малых уровнях архитектуры во время доступа к компоненту; параметр TA_{ij} имеет смысл использовать, если компонент работает на удаленных или изолированных системах;

Nta_{ij} — число сбойных компонентов на малых уровнях архитектуры во время доступа к компоненту i на уровне j, $i \in \{1, ..., N_j\}$, $j \in \{1, ..., M\}$;

TC_{ij} — относительное время анализа сбоя в компоненте i на уровне j, которое определяется как отношение среднего времени анализа сбоя в компоненте i на уровне j, $i \in \{1, ..., N_j\}$, $j \in \{1, ..., M\}$ к числу сбойных компонентов на всех уровнях архитектуры, анализируемых в одно и то же время; к анализу относится время на воспроизведение ошибки и ее локализацию;

Ntc_{ij} — число сбойных компонентов на всех уровнях архитектуры, анализируемых в одно и то же время с компонентом i на уровне j, $i \in \{1, ..., N_j\}$, $j \in \{1, ..., M\}$;

TE_{ij} — относительное время устранения сбоя в компоненте i на уровне j, определяемое как отношение среднего времени восстановления в компоненте i на уровне j, $i \in \{1, ..., N_j\}$, $j \in \{1, ..., M\}$ к числу сбойных компонентов на всех уровнях архитектуры, в которых происходит устранение сбоев в одно и то же время;

Nte_{ij} — число сбойных компонентов на всех уровнях архитектуры, в которых происходит устранение сбоев в одно и то же время с устранением сбоев в компоненте i на уровне j, $i \in \{1, ..., N_j\}$, $j \in \{1, ..., M\}$;

TU_{ij} — относительное время использования компонента i на уровне j, которое определяется как отношение среднего времени использования компонента i на уровне j, $i \in \{1, ..., N_j\}$, $j \in \{1, ..., M\}$ к числу компонентов на всех уровнях архитектуры, используемых в одно и то же время;

Ntu_{ij} — число компонентов на всех уровнях архитектуры, используемых в одно и то же время с компонентом i на уровне j, $i \in \{1, ..., N_j\}$, $j \in \{1, ..., M\}$;

K_{ij} — глубина программной избыточности компонента i на уровне j, $i \in \{1, ..., N_j\}$, $j \in \{1, ..., M\}$;

Z_{ij} — множество версий компонента i на уровне j, $i \in \{1, ..., N_j\}$, $j \in \{1, ..., M\}$;

RT_{ij} — среднее время выполнения модуля i на уровне j, $i \in \{1, ..., N_j\}$, $j \in \{1, ..., M\}$;

T_{ij} — трудоемкость разработки компонента i на уровне j, $i \in \{1, ..., N_j\}$, $j \in \{1, ..., M\}$;

T_{ij}^{k} — трудоемкость разработки версии k компонента i на уровне j, $i \in \{1, ..., N_j\}$, $j \in \{1, ..., M\}$, $k \in Z_{ij}$ в чел.-ч;

NVX_{ij} — трудоемкость разработки среды исполнения версий (приемочного теста для RB (*recovery block*, блок восстановления) или алгоритма голосования для NVP (*N-version programming*, N-версионное программирование);

B_{ij} — дихотомическая переменная, принимающая значение 1 (тогда $NVP_{ij} = 0$, $RB_{ij} = 0$), если в программном компоненте не используется программная избыточность, иначе она равна 0;

NVP_{ij} — дихотомическая переменная, принимающая значение 1 (тогда $B_{ij} = 0$, $RB_{ij} = 0$), если в программном компоненте введена программная избыточность методом NVP, иначе равна 0.

RB_{ij} — дихотомическая переменная, принимающая значение 1 (тогда $B_{ij} = 0$, $NVP_{ij} = 0$), если в программном компоненте введена программная избыточность методом RB, иначе равна 0.

TR — среднее время простоя системы, определяемое как время, в течение которого система не может выполнять свои функции;

$MTTF$ — среднее время появления сбоя, определяемое как время, в течение которого сбоев в системе не происходит;

S — коэффициент готовности;

T_s — общая трудоемкость реализации системы.

При построении мультиверсионного компонента из K версий методом N-версионного программирования (NVP) для любого K надежность равна [3]

$$R_{ij} = p_{ij}^{v}\left(1 - \prod_{k \in Z_{ij}}^{K}\left(1 - p_{ij}^{k}\right)\right),$$

где p_{ij}^{v} — вероятность безотказной работы алгоритма голосования, p_{ij}^{k} — вероятность безотказной работы версии $k \in Z_{ij}$.

При построении мультиверсионного компонента из K версий методом блока восстановления (RB) [3]:

$$R_{ij} = \sum_{k \in Z_{ij}}^{K} p_{ij}^{k} p_{ij}^{AT} \prod_{l=1}^{k-1}\left(\left(1 - p_{ij}^{l}\right)p_{ij}^{AT} + p_{ij}^{l}\left(1 - p_{ij}^{AT}\right)\right),$$

где p_{ij}^{AT} — вероятность безотказной работы приемочного теста для компонента i, $i = 1, ..., N$ на уровне j, $j = 1, ..., M$; p_{ij}^{k} — вероятность безотказной работы версии $k \in Zij$.

Вероятность сбоя таких компонентов равна

$$PF_{ij} = 1 - R_{ij}.$$

Программный компонент, участвующий в критически важных циклах управления, должен выполнить вычисления за такое время, чтобы время всего цикла не превышало критическое. Среднее время выполнения программного компонента i на уровне j вычисляется как сумма времени работы компонента без сбоев и времени простоя компонента:

$$RT_{ij} = TU_{ij}Ntu_{ij}\left(1 - \left(PF_{ij} + \sum_{ab \in E_{ab}}\left(PL_{ij}^{ab}\, PF_{ab}PU_{ab}\right)\right)\right) +$$
$$+ (TA_{ij}Nta_{ij} + TC_{ij}Ntc_{ij} + TE_{ij}Nte_{ij}) \times$$
$$\times \left(PF_{ij} + \sum_{ab \in E_{ab}}\left(PL_{ij}^{ab}\, PF_{ab}PU_{ab}\right)\right),$$

где PL_{ij}^{ab} — условная вероятность появления сбоя в компоненте i на уровне j при появлении сбоя в компоненте a на уровне b, $a \in \{1, ..., N_j\}$, $b \in \{1, ..., M\}$, $i \in \{1, ..., N_m\}$, $j \in \{1, ..., M\}$; PU_{ab} — вероятность использования компонента a на уровне b; PF_{ab} — вероятность появления сбоя в компоненте a на уровне b.

При расчете трудоемкости разработки учитываются затраты на реализацию среды исполнения версий модуля (NVX_{ij}) и затраты на реализацию каждой версии $\left(T_{ij}^{k}\right)$ [4]. Трудоемкость разработки системы рассчитывается следующим образом:

$$T_s =$$
$$= \sum_{j=1}^{j=M}\sum_{i=1}^{i=N_j}\left(B_{ij}T_{ij} + \left(NVP_{ij} + RB_{ij}\right)\left(NVX_{ij} + \sum_{k \in Z_{ij}}^{K} T_{ij}^{k}\right)\right).$$

Среднее время сбоя равно [1]

$$MTTF = \sum_{j=1}^{j=M}\sum_{i=1}^{i=N_j}\Bigg\{PU_{ij}(1 - PF_{ij}) \times$$
$$\times \Bigg[TU_{ij} + \sum_{(m=1)\&(m \ne j)}^{m=M}\sum_{n=1}^{n=N_m}\left(\left(1 - PL_{nm}^{ij}\right) \times\right.$$
$$\times\left(TU_{nm} + \sum_{l \in D_{nm}}\left(\left(1 - PL_{lm}^{nm}\, TU_{lm}\right)\right)\right) + \sum_{k \in D_{ij}}\left(\left(1 - PL_{kj}^{ij}\right) \times\right.$$
$$\times\left(TU_{kj} + \sum_{(m=1)\&(m \ne j)}^{m=M}\sum_{n=1}^{n=N_m}\left(\left(1 - PL_{nm}^{kj}\right) \times\right.\right.$$
$$\left.\left.\left.\times\left(TU_{nm} + \sum_{l \in D_{nm}}\left(\left(1 - PL_{lm}^{nm}\right)TU_{lm}\right)\right)\right)\right)\right)\Bigg]\Bigg\}.$$

Среднее время простоя системы равно [1]

$$TR = \sum_{j=1}^{j=M}\sum_{i=1}^{i=N_j}\Bigg\{PU_{ij} \times PF_{ij} \times$$
$$\times \Bigg[(TA_{ij} + TC_{ij} + TE_{ij}) +$$
$$+ \sum_{(m=1)\&(m \ne j)}^{m=M}\sum_{n=1}^{n=N_m}\left(PL_{nm}^{ij}\left((TA_{nm} + TC_{nm} + TE_{nm}) +\right.\right.$$
$$\left.\left.+ \sum_{l \in D_{nm}}\left(PL_{lm}^{nm}(TA_{lm} + TC_{lm} + TE_{lm})\right)\right)\right) \times$$
$$\times \sum_{k \in D_{ij}}\Bigg[PL_{kj}^{ij}\Bigg[(TA_{kj} + TC_{kj} + TE_{kj}) +$$
$$+ \sum_{(m=1)\&(m \ne j)}^{m=M}\sum_{n=1}^{n=N_m}\left(PL_{nm}^{kj}\left((TA_{nm} + TC_{nm} + TE_{nm}) +\right.\right.$$
$$\left.\left.+ \sum_{l \in D_{nm}}\left(PL_{lm}^{nm}(TA_{lm} + TC_{lm} + TE_{lm})\right)\right)\right)\Bigg]\Bigg]\Bigg]\Bigg\}.$$

Коэффициент готовности равен

$$S = \frac{MTTF}{MTTF + TR}.$$

Постановка задачи оптимизации архитектуры

На коэффициент готовности системы, трудозатраты на ее разработку влияют значения PF_{ij} и T_{ij}. Прогнозные значения вероятности сбоя могут быть рассчитаны по моделям надежности ПО, а значения трудозатрат по модели *COCOMO II* [5]. Если компоненту сложно детерминировать определенное значение ве-

роятности сбоя и трудозатрат, то ему можно задать массив альтернативных вариантов "вероятность сбоя/трудозатраты" с помощью модели роста надежности *SGRM* или экспертной оценки [1].

Особую задачу составляет оптимальное введение программной избыточности. Причина в том, что надежность компонента в этом случае зависит не только от вероятности сбоя каждой его версии, но и от надежности среды исполнения версий, числа версий, а также от метода реализации их взаимодействия.

Так как модель архитектуры представлена большим числом параметров, для больших программных архитектур достаточно затратно вычислять каждый параметр с высокой точностью. Для вычисления значения каждого параметра может также не хватать статистических данных. По этой причине параметры могут быть оценены экспертно. При этом теряется адекватность значений целевых функций, но найденные при оптимизации показатели все равно будут отражать качество того или иного варианта построения архитектуры ПО.

Общий вид задачи оптимизации архитектуры следующий:

$$S \to \max,$$

$$T_s \to \min,$$

при ограничениях:

$$S \geqslant S^0,$$

$$T_s \leqslant T_s^0,$$

$$RT_{ij} \leqslant RT_{ij}^0,$$

где S^0, T_s^0, RT_{ij}^0 — предельные значения критериев.

Таким образом, представленная задача оптимизации архитектуры ПО имеет два критерия оптимизации, два ограничения, соответствующих каждому критерию и Q ограничений, соответствующих компонентам, к которым предъявляются требования ко времени выполнения.

Инструментальное средство архитектурного дизайна

Автором была разработана программная система *Genetic Algorithm optimization software Architecture* в среде Embarcadero C++Builder 2010 с применением СУБД Microsoft Access 2010. Система предназначена для поддержки принятия решения о выборе программной архитектуры и может быть использована системными аналитиками, проектировщиками или менеджерами проектов.

Система позволяет выполнить следующие действия:

1) ввод компонентной структуры ПО;

2) ввод характеристик программных компонентов и их зависимостей, а также параметров программной избыточности;

3) ввод альтернативных вариантов "вероятность сбоя/трудозатраты" по определенным компонентам;

4) расчет коэффициента готовности и трудозатрат системы;

5) выбор компонентов, характеристики которых следует детерминировать для оптимальных значений коэффициента готовности системы и общих трудозатрат;

6) поиск оптимальных характеристик компонентов с помощью перебора всех возможных вариантов;

7) поиск оптимальных характеристик компонентов с помощью генетического алгоритма с заданными параметрами;

8) вывод в документ Excel программной архитектуры с оптимальными характеристиками.

Время расчета критериев S и T_s (п. 4) зависит от количества компонентов и связей между ними. На ПК со средней производительностью требуется около 1,5 с для расчета архитектуры из 10 компонентов, для архитектуры из 1000 компонентов требуется около 150 с. С ростом числа связей между компонентами требуется еще большее время. Эти показатели обусловлены сложным алгоритмическим заданием функции S, а также большим количеством вычислений на стороне БД, так как модель программной архитектуры имеет реляционное представление.

Число возможных решений рассчитывается следующим образом:

$$W = \prod_{d=1}^{D} \left(Var_d + 2 \sum_{Ver_d = 2}^{L_d} Var_d^{Ver_d} \right) \prod_{f=1}^{F} Var_f,$$

где W — число комбинаций; D — число программных компонентов, в которых возможно введение избыточности; Var_d — число вариантов "вероятность сбоя/трудозатраты" для компонента с возможностью введения избыточности; Ver_d — число используемых версий при введении избыточности ($2...L_d$); L_d — предельно допустимое число версий компонента; F — число программных компонентов, в которых невозможно введение избыточности; Var_f — число вариантов "вероятность сбоя/трудозатраты" для компонента без избыточности.

Проведенный анализ роста мощности пространства оптимизации позволяет сделать вывод о том, что для решения задачи необходимо применение генетического алгоритма (ГА). Причина в том, что задача определения оптимальной архитектуры ПО является NP-полной и перебор всех вариантов ее решения представляется возможным только для малых архитектур и небольшого числа компонентов с недетерминированными характеристиками [2].

Генетический алгоритм

Генетический алгоритм представляет собой метод оптимизации, основанный на концепциях естественного отбора и генетики. В этом подходе переменные, характеризующие решение, представлены в виде ге-

Генотип и фенотип особи

Группа компонентов с возможностью программной избыточности									Группа компонентов без возможности программной избыточности		
Компонент 1				..	Компонент N				Компонент 1	..	Компонент N
NVP/RB	Var_{v1}	..	Var_{v4}		NVP/RB	Var_{v1}	..	Var_{v5}	Var		Var
1	3		0		0	1		0	2		3

нов в хромосоме. Генетический алгоритм оперирует конечным множеством решений (популяцией). Он генерирует новые решения как различные комбинации частей решений популяции, используя такие операторы, как отбор, рекомбинация (кроссинговер) и мутация. Новые решения позиционируются в популяции в соответствии с их положением на поверхности исследуемой функции. Генетические алгоритмы являются универсальным вычислительным средством для решения сложных математических задач.

В программной системе *Genetic Algorithm optimization software Architecture* задаются компоненты, характеристики которых необходимо детерминировать. При этом возможно выбрать один из двух способов определения переменных характеристик.

Первый способ выбирается для компонентов, в которых возможно введение программной избыточности. Для них могут быть изменены следующие характеристики.

1. Метод реализации программной избыточности: мультиверсионное программирование ($NVP_{ij} = 1$, $RB_{ij} = 0$) или блок восстановления ($NVP_{ij} = 0$, $RB_{ij} = 1$). Если выбран метод NVP, то устанавливается значение гена 0, если RB, то 1.

2. Номер варианта Var_{v1} — "вероятность сбоя/трудозатраты". Возможные варианты задаются аналитиком ($1 \leqslant Var_{v1} \leqslant$ *Число вариантов для данного компонента*).

3. Номер варианта $Var_{v2}...Var_{v10}$ — вероятности сбоя для каждой версии компонента, аналогично п. 2 ($0 \leqslant Var_{v2...10} \leqslant$ *Число вариантов для данного компонента*; 0 — версии нет). Предельное число версий программного компонента, если будет применена избыточность, заранее задается аналитиком и не может быть больше 10 версий, так как бóльшее число вносит в компонент сложность, неоправданную по надежности и стоимости.

Если гены $Var_{v2}...Var_{v10}$ принимают значение 0, то считается, что программная избыточность не вводится в данном компоненте ($B_{ij} = 1$).

Второй способ выбирается для компонентов, в которых недопустимо введение программной избыточности. По этой причине для них изменяется только вариант Var вероятности сбоя компонента и соответствующей трудоемкости для достижения этой вероятности сбоя ($1 \leqslant Var \leqslant$ *Число вариантов для данного компонента*).

Таким образом, фенотип особи (решения) формируется из переменных характеристик программных компонентов. В таблице представлен общий вид фенотипа с примером аллелей и локусов.

Генетический алгоритм основан на методе VEGA (*Vector Evaluated Genetic Algorithm*) с независимой селекцией Шаффера при многокритериальной оптимизации [6].

Входные параметры ГА следующие:

- размер популяции (N);
- вероятность скрещивания (*prob_cross*);
- вид скрещивания (*1*, *2*, *3*-точечное, равномерное);
- вероятность разрыва связанных генов (*prob_cross_inter*);
- вероятность мутации особи (*prob_mutate*);
- вероятность мутации гена (*prob_mutate_gen*);
- критерии останова (максимальное время работы *time_ga*, число популяций без улучшения решения (стагнация) *stagnancy*, число популяций *pop_count*);
- процент популяций на обработку ограничений *percent_bound*;
- число ограничений на время выполнения компонентов Q.

Собственно алгоритм реализуется следующей последовательностью действий.

1. Генерация родительской популяции P размером N случайных особей.

2. Расчет критериев для всех особей популяции P.

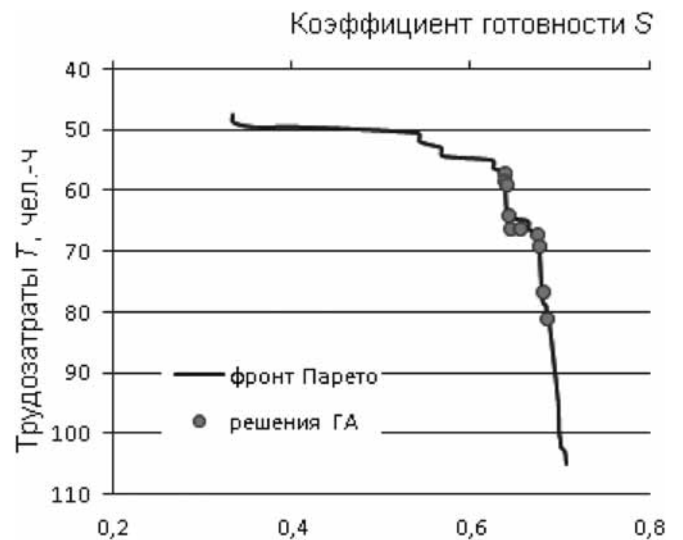


Рис. 1. Недоминируемые решения ГА и фронт Парето тестовой задачи

3. Пропорциональная селекция $N/2$ особей из P по критерию S в промежуточную популяцию P'.

4. Пропорциональная селекция $N/2$ особей по критерию T_s в промежуточную популяцию P'.

5. Скрещивание с вероятностью *prob_cross* $N/2$ случайно выбранных пар особей из промежуточной популяции P'. Добавление N полученных потомков в основную популяцию P.

6. Проведение оператора мутации с вероятностью *prob_mutate* на каждой особи основной популяции P.

7. Расчет критериев для всех особей популяции P.

8. Выбор из популяции P лучших решений. Если в найденных ранее решениях есть лучше, то *stagnancy* = = *stagnancy* + 1.

9. Если не сработал хотя бы один критерий останова, то переход на шаг 10, иначе остановка алгоритма.

10. Если номер популяции меньше, либо равен *percent_bound*×*pop_count*, то переход на шаг 3, иначе переход на шаг 11.

11. Пропорциональная селекция $N/(2 + Q)$ особей из P по критерию S в промежуточную популяцию P'.

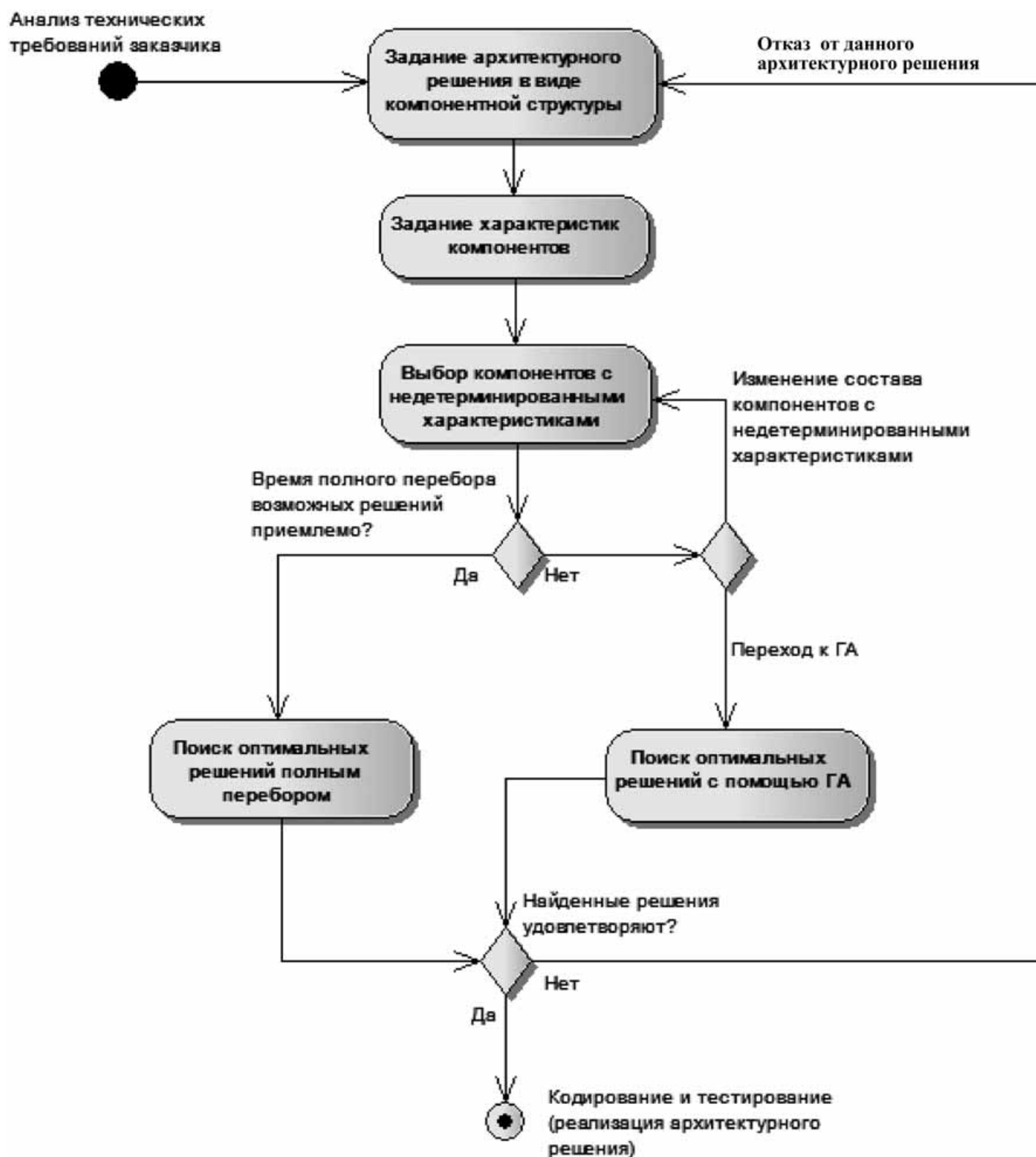


Рис. 2. Алгоритм дизайна программной архитектуры

12. Пропорциональная селекция $N/(2 + Q)$ особей по критерию T_s в промежуточную популяцию P'.

13. Пропорциональная селекция $N/(2 + Q)$ особей по каждому критерию на время выполнения компонента RT_{ij} в промежуточную популяцию P'.

14. Скрещивание с вероятностью *prob_cross* $N/(2 + Q)$ случайно выбранных пар особей из промежуточной популяции P'. Добавление N полученных потомков в основную популяцию P.

15. Проведение оператора мутации с вероятностью *prob_mutate* на каждой особи основной популяции P.

16. Расчет критериев по ограничениям для всех особей популяции P.

17. Выбор из популяции P лучших решений по критериям ограничений. Если в найденных ранее решениях есть лучше, то *stagnancy* = *stagnancy* + 1.

18. Если не сработал хотя бы один критерий останова, то переход на шаг 11, иначе остановка алгоритма.

Недоминируемые решения, полученные во всех популяциях, отбираются в множество Парето. С точки зрения математики решения множества Парето не могут быть предпочтены друг другу, поэтому после его формирования задача может считаться математически решенной [6].

Проведенные испытания показали, что алгоритм работает эффективно. На рис. 1 изображен аппроксимированный график фронта Парето одной из тестовых задач, а точками показаны решения, найденные ГА для той же задачи.

Решения, найденные ГА и удовлетворяющие заданным ограничениям, близки к фронту Парето, а 34 % решений входят в множество Парето.

Применение инструментального средства на этапе архитектурного дизайна

На рис. 2 изображена схема алгоритма дизайна программной архитектуры с применением разработанной системы.

Этап проектирования архитектуры носит итеративный характер, как и весь жизненный цикл программного обеспечения. В случае если значения критериев, найденные полным перебором или с помощью ГА, не удовлетворяют, то системный архитектор может принять решение об изменении компонентной структуры и повторить поиск оптимальных характеристик уже для другого архитектурного решения.

Систему *Genetic Algorithm optimization software Architecture* можно использовать для поиска оптимальных решений при модернизации уже существующего ПО. В этом случае характеристики трудозатрат устанавливаются только для новых архитектурных компонентов.

Таким образом, разработанная автором программная система позволяет, с одной стороны, задавать надежность функционирования будущей системы уже на стадии архитектурного дизайна, с другой стороны, прогнозировать возможность обеспечить заданную надежность при имеющихся трудовых ресурсах.

Список литературы

1. **Русаков М. А.** Многоэтапный анализ архитектурной надежности в сложных информационно-управляющих системах: дис. ... канд. техн. наук. Красноярск, 2005. 168 с.
2. **Жуков В. Г., Шеенок Д. А., Терсков В. А.** Повышение надежности программного обеспечения сложных систем // Вестник СибГАУ. 2012. Вып. 5(45). С. 28—33.
3. **Новой А. В.** Система анализа архитектурной надежности программного обеспечения: дис. ... канд. техн. наук. Красноярск, 2011. 131 с.
4. **Кукарцев В. В., Шеенок Д. А.** Оценка затрат на модернизацию программного обеспечения критических по надежности систем // Вестник СибГАУ. 2012. Вып. 5(45). С. 62—65.
5. **Глазова М. А.** Моделирование стоимости разработки проектов в ИТ-компаниях: дис. ... канд. экон. наук. Москва, 2008. 205 с.
6. **Сергиенко Р. Б.** Автоматизированное формирование нечетких классификаторов самонастраивающимися коэволюционными алгоритмами: дис. ... канд. техн. наук. Красноярск, 2010. 192 с.

ИНФОРМАЦИЯ

Продолжается подписка на журнал "Программная инженерия" на второе полугодие 2013 г.

Оформить подписку можно через подписные агентства или непосредственно в редакции журнала.

Подписные индексы по каталогам: Роспечать — 22765; Пресса России — 39795

Адрес редакции: 107076, Москва, Стромынский пер., д. 4, редакция журнала "Программная инженерия"

Тел.: (499) 269-53-97. Факс: (499) 269-55-10. E-mail: prin@novtex.ru

УДК 512.563.6

Д. В. Баранов, аспирант,
Воронежский государственный университет,
e-mail: Denis.Baranov@dataart.com

Компьютерная реализация устранения избыточных правил в условных системах переписывания

Описана реализация алгоритма исключения избыточности. Он основан на новой алгебраической системе, обладающей семантикой совокупности правил условной эквациональной теории или условной системы переписывания термов.

Ключевые слова: *термы, эквациональная теория, условные правила, логическая редукция, компьютерная реализация*

Введение

Теория систем переписывания предоставляет эффективный аппарат для решения ряда задач искусственного интеллекта и компьютерной алгебры. Ее алгебраические модели создают возможности для автоматизированной верификации и оптимизации соответствующих множеств правил.

Формализм систем переписывания термов (СПТ) применяется в различных разделах информатики и программной инженерии. Он используется для верификации компьютерных программ [1], автоматического доказательства теорем [2], символьного упрощения алгебраических выражений [3] и т. д. При определении системы переписывания отправной точкой служит эквациональная теория, множество правил которой состоит из равенств термов. Правила СПТ получаются путем "ориентации" равенств и пополнения для достижения свойства конфлюэнтности [4]. Аналогичный подход используется в условных СПТ [5].

Важными вопросами, связанными с СПТ, являются эквивалентные преобразования и оптимизация их множеств правил. Критерий эквивалентности систем переписывания предоставляют эквациональные теории. Поэтому исследование условных СПТ может быть основано на эквивалентности условных эквациональных теорий.

В работе [6] представлена методология исследования условных СПТ на основе "решеточных" продукционно-логических структур (*LP*-структур). Полученные результаты, в частности, впервые решают задачу построения логической редукции условной эквациональной теории. Они были обобщены в работе [7], где введена и исследована более сложная *LP*-структура, завершающая исследование задачи минимизации множества правил условной СПТ.

В настоящей работе описана архитектура программной библиотеки, реализующей поиск и исключение избыточных правил в условной эквациональной теории на основе теоретических результатов [7]. Тем самым подтверждена работоспособность созданных ранее алгоритмов и, соответственно, практическая значимость новой *LP*-структуры. Для реализации были выбраны язык *C#* и платформа .Net, как одна из наиболее распространенных на сегодняшний день систем программирования. Код программы, написанной на данной платформе, может быть собран с ключами, отвечающими за оптимизацию по скорости. Таким образом, производительность алгоритмов не будет проигрывать производительности решений, реализованных на других платформах.

Решение представляет собой разделяемую (*shared*) библиотеку. Ее можно в дальнейшем использовать как часть других программных продуктов, в частности, в существующих системах для дополнительного анализа на избыточность имеющихся множеств правил. Для большей гибкости алфавит системы содержит набор символьных констант, которые можно модифицировать в зависимости от поставленных целей.

1. Общее описание алгоритма

Рассматриваемая программная система реализует алгоритм упрощения условной эквациональной теории путем исключения из нее лишних правил. Для этой цели применяется математически обоснованный в работе [7] механизм преобразований. Здесь использованы обозначения и определения, принятые в указанной статье.

Пусть Σ — алфавит, образованный объединением следующих непересекающихся множеств: V — множество переменных; Σ_n, $n = 0,1, \ldots$ — множества n-арных функций (функциональных символов); 0-арные функции называются константами. Множество термов $T(\Sigma)$ определяется рекурсивно:

- $V \subset T(\Sigma)$; $\Sigma_0 \subset T(\Sigma)$;
- если $f \in \Sigma_n$ и $t_1, \ldots, t_n \in T(\Sigma)$, то $f(t_1, \ldots, t_n) \in T(\Sigma)$.

Отображение σ: $V \rightarrow T(\Sigma)$ называется подстановкой; Tr — транзитивный вывод.

LP-структурой (*Lattice Production Structure*) называется эквациональная решетка F с дополнительно заданным на ней бинарным отношением R, которое обладает рядом продукционно-логических свойств [7]. Такая алгебраическая система представляет собой модель условной эквациональной теории. Минимизация системы правил данной теории (исключение избыточности) соответствует построению логической редукции отношения R. Указанный процесс сводится к последовательному выполнению следующих шагов:

1) добавить к R все пары (a, b), для которых $b = \sigma(a)$, $b = f(a)$ либо $b = \text{Tr}(a)$, и обозначить новое отношение R_1;

2) добавить к R_1 всевозможные пары вида $(\sigma(a), \sigma(b))$, для которых $(a, b) \in R_1$, и обозначить новое отношение R_2;

3) добавить к R_2 всевозможные пары вида $(a_1 \cup a_2 \cup \cup a_i \ldots \cup a_m, b_1 \cup b_2 \cup b_i \ldots \cup b_m)$, где $(a_i, b_i) \in R_2$, и обозначить новое отношение R_3;

4) объединить полученное отношение с отношением включения $\supset$ и обозначить новое отношение $\tilde{R}$;

5) построить транзитивную редукцию R^0 отношения $\tilde{R}$;

6) исключить из $\tilde{R}$ содержащиеся в нем пары вида $a \supset b$ и обозначить новое отношение R_{-1};

7) исключить из R_{-1} все пары (a, b) вида $(a_1 \cup a_2 \cup \cup a_i \ldots \cup a_m, b_1 \cup b_2 \cup b_i \ldots \cup b_m)$, где $(a_i, b_i \cup R_{-1}$, причем (a, b) не совпадает ни с одной парой (a_i, b_i), и обозначить новое отношение R_{-2};

8) исключить из R_{-2} всевозможные пары вида $(\sigma(a), \sigma(b))$, для которых $(a, b) \in R_{-2}$, причем (a, b) не совпадает с парой $(\sigma(a), \sigma(b))$, и обозначить новое отношение R_{-3};

9) исключить из R_{-3} все пары (a, b), для которых $b = \sigma(a)$, $b = f(a)$, либо $b = \text{Tr}(a)$.

Из соображений экономии памяти физическое добавление к множеству новых пар следует проводить лишь в случае необходимости. В частности, нет смысла хранить логические тавтологии (шаги 1, 4), если способ кодирования эквациональной решетки и алгоритмы унификаций и обобщений будут допускать их эффективное вычисление.

Непосредственная реализация шагов 2, 3 также требует чрезмерного расхода памяти, соизмеримого с построением булеана на универсуме R. Кроме того, в дальнейших вычислениях из огромного числа добавленных таким образом пар практически использовалась бы лишь незначительная часть. Как следствие, приходим к тезису о целесообразности динамического построения необходимых в дальнейшем пар множества $\tilde{R}$.

Обсудим также вопрос построения транзитивной редукции (шаг 5) отношения $\tilde{R}$. Как показано в работе [8], эта задача вычислительно эквивалентна задаче нахождения транзитивного замыкания, что, например, при применении известного алгоритма Уоршолла составило бы $O(N^3)$ операций. Однако в нашем случае N заменяется выражением $\Omega(2^N)$ и нахождение транзитивной редукции посредством замыкания также не представляется эффективным. В качестве практически реализуемого варианта остается исследование множества пар на транзитивную избыточность. Таким образом, в предлагаемой реализации будем строить транзитивную редукцию отношения $\tilde{R}$ путем исключения пар, связанных транзитивными цепочками.

Работу алгоритма минимизации системы правил можно условно разделить на два уровня: *удаление тавтологий* и *удаление избыточных правил*.

Согласно работе [7], в условной эквациональной логике тавтологии удовлетворяют следующим правилам (не зависящим от множества правил):

1) a: $f(a)$ для любых $a = \{s_1 = t_1, \ldots, s_n = t_n\}$ и $f \in \Sigma_n$;
2) a: $\sigma(a)$ для любых $a \in \mathbb{F}$ и подстановки σ;
3) a: Tr (a) для каждого подходящего $a \in \mathbb{F}$;
4) a: b, a, $b \in \mathbb{F}$ при $a \supseteq b$.

Для построения логических цепочек будем использовать сформулированные в работе [7] правила вывода:

1. $a{:}b \vdash \sigma(a){:}\ \sigma(b)$ $(a, b \in \mathbb{F})$ для любой подстановки σ;
2. $a{:}b,\ a{:}c \vdash a{:}b \cup c(a, b, c) \in \mathbb{F}$;
3. $a{:}b,\ b{:}c \vdash a{:}c\ (a, b, c) \in \mathbb{F}$.

Важно иметь в виду, что в процессе построения логических цепочек необходим учет логических тавтологий всех типов (1—4).

2. Внутренний формат данных

В соответствии с работой [7] реализуем хранение правил переписывания на основе эквациональной решетки. В качестве ее базиса будем использовать уникальные равенства термов. Класс, описывающий атом решетки, выглядит следующим образом:

```
class Equal
{
  public List<int>TermIds {get; set;}
  public int Id{get;set;}
}
```

Поле `TermIds` будет содержать два элемента — левую и правую части равенства. Так как выражения

$a = b$ и $b = a$ согласно эквациональной логике эквивалентны, порядок термов неважен.

Тогда элемент решетки можно описать следующим образом:

```
public class Element
{
  public List<int>EqualIds {get; set;}
  public int Id{get;set;}
}
```

Для элементов решетки (множеств равенств) необходимо реализовать операции пересечения и объединения, а также отношение вложения. Вначале опишем основу обычной решетки, представляемую следующим классом:

```
class LatticeBase
{
  public List<Element>Elements {get; set;}
  public static Element Meet
(Element first, Element Second);
  public static Element Join
(Element first, Element Second);
  public static bool IsEnclose
(Element first, Element Second);
}
```

Далее расширим исходный класс добавлением в него еще трех групп операций, связанных соответственно с функциями, подстановками и транзитивными равенствами термов. Таким образом, получим класс, представляющий эквациональную решетку:

```
class EquationalLattice : LatticeBase
{
  public Element ExecuteFunction
(Element equals, Function function);
  public Element ExecuteSubstitution
(Element equals,
  Substitution substitution);
  public Element GetTr (Element equals);
}
```

Данный класс будет использоваться для обеспечения хранения условных правил, а также их анализа. Правило состоит из предпосылки и заключения, которые являются элементами решетки. Каждое правило нашей системы можно описать следующим образом, используя идентификаторы элементов решетки:

```
class Rule
{
public int PremiseId{get; set;}
public int ConclusionId{get; set;}
}
```

Также для реализации системы переписывания понадобятся функции и подстановки. Информация о каждой из них будет содержаться соответственно в классах `Function` и `Substitution`. Полное описание эквациональной системы будет представлено классом `TermSystem`:

```
class TermSystem
{
  // Свойства класса
  public List<Function>Functions{get;set;};
  public List<Substitution>
Substitutions{get;set};
  public List<Rule>Rules{get; set;}

  public List<Term>Terms{get;set;}
  public List<Equal>Equals{get;set;}
  public EquationalLattice Lattice{get;set;}
  // Методы
...
}
```

Информация о реализации функций, подстановок и термов будет дана в разд. 5, посвященном базовым классам библиотеки.

Таким образом, представлен общий обзор архитектуры системы переписывания, реализующей работу на основе понятийного аппарата эквациональной логики. Данное обстоятельство позволяет сделать более структурированным и читаемым код программы, создаваемой для минимизации систем переписывания.

Для упрощения дальнейшего анализа и минимизации времени работы алгоритмов используются некоторые дополнительные приемы, такие как приведение имен переменных к унифицированному виду, хранение промежуточных данных с пометками для дальнейшего удаления и т. д.

3. Компьютерная реализация алгоритма

Опишем основные положения компьютерной реализации алгоритма минимизации условной эквациональной теории. Алфавит языка, а также общая структура внутренних классов библиотеки будут представлены в следующих разделах.

Основной алгоритм работы системы состоит из следующих пунктов.

1. Анализ исходных данных, трансляция их во внутренние объекты (класс `Parser`).

2. Выявление и удаление логических тавтологий в соответствии с пунктами, описанными ранее (класс `TautologyAnalyzer`).

3. Последовательный перебор правил переписывания, попытка построения логической цепочки для каждого правила. Для построения цепочек используются правила вывода 1—3 условной эквациональной логики (класс `ChainCreator`).

4. Преобразование результата из внутреннего языка системы в язык исходных данных. Предоставление проанализированной и упрощенной системы переписывания конечному пользователю (класс `Parser`).

Пункты 1 и 4 являются служебными и в детальном рассмотрении не нуждаются, в то время как на п. 2 и 3 остановимся немного подробнее.

Начнем с п. 2. После преобразования во внутренний формат находим и исключаем повторения среди правил переписывания и функций. Под повторением подразумеваются правила и функции, отличающиеся лишь именами переменных. Исключение таких сущностей позволит в дальнейшем уменьшить число необходимых операций для анализа системы.

Далее последовательно ищем и исключаем каждый вид тавтологий 1—4, описанных в разд. 1. Исключение явных тавтологий до начала построения транзитивных цепочек позволяет сократить число анализируемых правил, а значит, уменьшить число операций и время работы.

Рассмотрим п. 3. Для экономии ресурсов во время построения цепочки нецелесообразно постоянно хранить все получаемые переходы. Достаточно сохранять лишь последнее состояние. Далее алгоритм единообразен и повторяется для каждого ("анализируемого") правила.

1. Выбрать все правила с левыми частями, тавтологически порождаемыми левой частью анализируемого правила.

2. С помощью правил вывода 2 и 3 перейти к следующему состоянию системы. Для получения недостающих звеньев цепочки допустимо использование неявно присутствующих в системе тавтологий, а также и условных равенств, получаемых с помощью правила вывода 1.

3. Удалить тавтологии из полученной системы.

4. Повторять п. 1—3, пока система не перестанет расти, либо не будет сформировано правило, совпадающее с анализируемым.

5. Если оно найдено, удалить анализируемое правило из системы.

После повторения алгоритма для каждого правила и удаления избыточности анализ системы переписывания будет завершен.

4. Описание алфавита языка системы

Одним из важных факторов, влияющих на реализацию алгоритма, являются исходные данные, т. е. способ представления анализируемой системы переписывания. Было принято решение о том, что система будет передаваться библиотеке в текстовом виде. Такой подход упрощает процесс описания анализируемых систем пользователями.

Чтобы не ограничивать алфавит заранее предопределенным набором символов, он хранится в отдельном файле. Преимуществом данного подхода является повышение гибкости при определении анализируемых систем, а также возможность поддержки одновременно нескольких алфавитов, например, для разных пользователей или систем различных типов.

При инициализации библиотеки данные из указанного файла загружаются в основной модуль библиотеки и используются для преобразования исходных данных во внутренний формат.

В конфигурационном файле для построения алфавита могут содержаться следующие элементы.

- **WordsSeparator.** Базовыми конструкциями описываемого языка являются слова — это могут быть как описания функций, так и правил переписывания. Для разделения входной информации на отдельные слова используется данный символ, например, символ перевода строки, т. е. каждое слово будет начинаться с новой строки.

Пример описания: WordsSeparator = "\r\n".

Пример использования:

$$F(x) = x$$

$$F(y) = y + 3.$$

- **ParametersStart** и **ParametersEnd.** Данные два символа используют для определения в описании функции или подстановки позиций, где начинается и заканчивается описание параметров. Их всегда нужно использовать совместно, т. е. обозначение только начала или только конца области задания параметров недопустимо.

Пример описания:

ParametersStart = "("

ParametersEnd = ")"

Пример использования: $F(x)$

- **ParametersSeparator.** Данный символ используют для отделения описания одного параметра от другого.

Пример описания: ParametersSeparator = ","

Пример использования: $F(x, y)$

- **FunctionBodyStart.** Используется программой для отделения области описания имени функции и параметров от ее тела. Описание функции без описания ее тела недопустимо.

Пример описания: BodySeparator = " = "

Пример использования: $F(x, y) = x + y$

- **EqualsSymbol.** Этот символ обозначает равенства термов.

Пример описания: EqualsSymbol = " = "

Пример использования: $x = y + 1$

- **RuleSeparator.** Данный символ используют для отделения в правиле предпосылки от заключения.

Пример описания: RuleSeparator = ":"

Пример использования: $x = y : y = x$

- **SubRuleSeparator.** Правило переписывания может состоять из нескольких равенств термов. Для разделения таких равенств применяют указанный символ.

Пример описания: SubRuleSeparator = ";"

Пример использования: $x = y; y = 0 : x = 0$

- **ParameterNames.** Это множество имен переменных, которые можно использовать в определении функций, подстановок, термов. В данной реализации это не словарь символов для конструирования имен переменных, а именно полные имена.

Пример описания: ParameterNames = "x", "y"

Пример использования: $F(x, y) = x + y + 1$

- **FunctionNames.** Это словарь имен функций и подстановок, допустимых при описании системы.

Как и в предыдущем случае, это полные имена функций, а не символы, используемые для их конструирования.

Пример описания: FunctionNames = "*f*", "*sum*"

Пример использования: $f(x) = x$ $sum(x, y) = x + y$

При описании системы переписывания допустимо использование любых других символов, но они будут восприниматься как константные выражения.

5. Базовые классы библиотеки

Рассматриваемая библиотека состоит из нескольких основных классов, каждый из которых отвечает за реализацию определенной части алгоритма. Кратко опишем каждый из них, не вдаваясь в излишние технические подробности.

Все классы библиотеки условно можно разделить на две группы: *описательные* и *функциональные*. Под *описательными* подразумеваются классы, содержащие информацию об исследуемой эквациональной системе. Основные из них были представлены выше, остановимся лишь на нескольких ранее не описанных.

- **Function.** Данный класс целиком описывает функцию — ее имя, параметры, тело. Конструктору класса передается текстовое описание функции в рамках заданного алфавита. Во время ее преобразования из текстового формата во внутренний параметры функции будут приведены к стандартному формату.

Данный класс умеет также печатать хранимую функцию в текстовом виде. В этих целях внутри класса инкапсулирован словарь (класс Parameters), предназначенный для преобразования стандартных параметров в исходные и наоборот.

- **Substitution.** Этот класс содержит полную информацию о подстановке. По функциональности он похож на предыдущий класс. Умеет преобразовывать текстовый формат данных во внутренние объекты, а также наоборот.

- **Term.** Данный класс используют для хранения информации о базовой единице системы переписывания — терме. Этот класс умеет сравнивать хранимые термы между собой, а также имеет уникальный идентификатор. Существование идентификатора позволяет при анализе вместо полного сравнения экземпляров сравнивать только их идентификаторы. Это существенно уменьшает стоимость операции сравнения. Возможность находить эквивалентные термы позволяет также исключать повторяющиеся конструкции и не множить сущности, а использовать уже имеющиеся.

Под второй группой — *функциональными* классами — подразумеваются объекты, не участвующие в описании системы в явном виде, но предоставляющие функциональность для ее построения, анализа и вывода полученных результатов. Таких классов три: Parser, TautologyAnalyzer, ChainCreator. Остановимся на каждом из них подробнее.

- **Parser.** Класс, отвечающий за преобразования текстового описания системы во внутренние объекты библиотеки и наоборот. Работает он следующим образом: методу `Initialize` передается путь к файлу, описывающему алфавит. Далее методу `Parse` передается текстовое описание. С использованием загруженного алфавита текст разбивается на части — описания функций, подстановок, правил переписывания. На основе полученных частей конструируются внутренние объекты системы.

- **TautologyAnalyzer.** Как было отмечено ранее (разд. 1), в эквациональной логике существуют четыре типа тавтологий. Данный класс, основываясь на правилах вывода, умеет определять, является ли пара тавтологией. Для этой цели используются заданные на эквациональной решетке операции (в классе `EquationalBase`). Для заданного правила переписывания он умеет также получать любой необходимый тип тавтологий. Такая возможность используется при построении цепочек вывода.

- **ChainCreator.** Это класс, реализующий описанный выше (разд. 1) алгоритм получения цепочки вывода. Принимает на вход правило переписывания и пытается вывести такое же правило, используя остальные правила системы. При выводе использует операции, заданные на эквациональных решетках, и тавтологии, генерируемые классом TautologyAnalyzer.

Рассматриваемая библиотека реализует набор функций и алгоритмов, необходимый для упрощения систем переписывания и обнаружения в них тавтологий. Далее будут изложены особенности и принципы работы с библиотекой, а также приведены примеры ее практического применения.

6. Ключевые особенности

Прежде чем перейти к общим принципам работы с библиотекой, резюмируем сказанное выше и при этом выделим ключевые особенности. Они делают данную библиотеку удобной в использовании, а также минимизируют по времени упрощение систем переписывания.

1. **Использование в качестве основы функционирования математической модели.** Данный подход дает строгое обоснование и предоставляет возможность получения упрощенной системы переписывания за несколько шагов. Использование системы упрощений позволяет сократить до минимума число дополнительных операций, избавляет от построения сложных рекурсий, которые могут оказаться дорогими по объему занимаемой памяти.

2. **Гибкий алфавит, который можно выбирать для описания систем переписывания.** Пользователь может легко изменить входной язык, скорректировав только один файл. Такой описательный механизм позволяет перестраивать библиотеку под собственные нужды, а также при необходимости расширять ее новыми конструкциями.

3. **Использование стандартизованной внутренней формы представления данных.** В процессе преобразования описания из текстового вида в объектный биб-

лиотека приводит описания термов, функций и правил к унифицированному внутреннему формату. Например, присваивает всем переменным, используемым в функциях, стандартные имена. При этом для термов используются уникальные идентификаторы. Такие преобразования позволяют сократить время проверки похожести функций до минимума, сведя его к простому сравнению последовательностей символов или даже целых чисел. Применение данного подхода положительно влияет как на время выполнения отдельных алгоритмов, так и на упрощение работы всей системы в целом.

7. О практическом применении

Важнейший критерий полезности программного продукта — возможность его использования на практике, в реальных условиях. Приведем одну из возможных областей применения рассматриваемой библиотеки.

В финансовой сфере существуют классы задач, для решения которых используют сложные парсинг-системы. Одним из них является разбор электронной почты. Необходимость такого разбора возникает ввиду потребности в извлечении нужной информации (например, стоимости определенных типов ценных бумаг) из огромного потока писем.

Подобные системы разбора достаточно сложны, и для их создания и поддержки могут привлекаться десятки специалистов. В основе автоматизации могут лежать такие теории, как нейронные сети, машинный анализ текста и т. п. Работают они обычно в режиме реального времени. Скорость извлечения информации и безошибочность полученного результата — ключевые параметры оценки эффективности работы. Возможная реализация такой системы может быть основана на шаблонно-ориентированном подходе. Он предполагает формирование наборов шаблонов, которые система в дальнейшем использует для разбора писем. Число таких шаблонов может доходить до десятков тысяч, при этом процент ошибок, тем не менее, остается довольно высоким (около 10...15 %).

Для уменьшения числа ошибок после основного разбора часто проводится ручной. Он более точен, но и более дорог по сравнению с автоматическим разбором. Для снижения затрат на ручную работу создаются правила, которые в дальнейшем могут быть использованы автоматически. Можно сказать, что создается система переписывания термов.

Основным недостатком рассматриваемой системы является ее постоянный рост, соответственно растет и время анализа писем. Добавляются новые правила, которые могут дублировать уже существующие как явно, так и неявно. Соответственно, появляются тавтологии и избыточные правила. Ручной анализ и упрощение такой системы сложны и не дают гарантированного результата. Применение же описанной в настоящей работе библиотеки, после некоторой доработки под существующие условия, может оказаться оптимальным решением.

Анализ системы переписывания можно будет проводить в периоды, когда нагрузка на парсинг-систему минимальна, например, биржа закрыта. Таким образом, проблема избыточного роста системы правил переписывания будет решена, а время работы системы по разбору писем не изменится. Аналогично данная библиотека может быть использована и в других областях, где так или иначе действуют системы переписывания.

Заключение

Нередко при решении поставленной задачи разработчик не учитывает того обстоятельства, что данная задача уже существует в решенном или доказанном виде в математике или логике в виде моделей, теорем или лемм. Предварительный автоматизированный анализ подобных существующих материалов может многократно упростить или даже полностью изменить ход решения. Кроме того, реализация теорий на ЭВМ может подсказать слабые с точки зрения устойчивости или производительности места. Поэтому важно сочетать прикладные теории с их компьютерной реализацией. Подобная реализация описана в настоящей работе.

Список литературы

1. **Воробьев С. Г.** Условные системы подстановок термов и их применение в проблемно-ориентированной верификации программ: автореф. дис. ... канд. физ.-мат. наук. Новосибирск: ВЦ СО АН СССР, 1987.
2. **Hsiang J.** Refutational theorem proving using term-rewriting systems // Artificial Intelligence. 1985. Vol. 25. P. 255—300.
3. **Buchberger B., Loos R.** Algebraic Simplification. Computer Algebra — Symbolic and Algebraic Computation / eds. B. Buchberger, G. E. Collins, R. Loos. Vienna — New York: Springer-Verlag, 1982. P. 11—43.
4. **Klop J. W.** Term rewriting systems // Handbook of Logic in Computer Science. Vol. 2. Background: Computational Structures / eds S. Abramsky, D. M. Gabbay and S. E. Maibaum. 1992. Vol. 2. P. 1—116.
5. **Dershowitz N., Okada M., Sivakumar G.** Canonical Conditional Rewrite Systems // Proceedings of the 9th international Conference on Automated Deduction. May 23—26, 1988 / eds E. L. Lusk, R. A. Overbeek. London: Springer-Verlag. 1988. Vol. 310. P. 538—549.
6. **Махортов С. Д.** Основанный на решетках подход к исследованию и оптимизации множества правил условной системы переписывания термов // Интеллектуальные системы. 2009. Т. 13. Вып 1—4. С. 51—68.
7. **Баранов Д. В.** Алгебраическая интерпретация условных систем переписывания на основе LP-структур // Вестник ВГУ. Серия Системный анализ и информационные технологии. 2010. № 2. С. 131—138.
8. **Aho A. V., Garey M. R., Ulman J. D.** The transitive reduction of a directed graph // SIAM Journal on Computing. 1972. Vol. 1, N 2. P. 131—137.

УДК 800, 801

А. А. Харламов, д-р техн. наук, проф., стар. науч. сотр., Институт высшей нервной деятельности и нейрофизиологии РАН, г. Москва, e-mail: kharlamov@analyst.ru,
Т. В. Ермоленко, канд. техн. наук, доц. нач. отд., Институт проблем искусственного интеллекта, г. Донецк

Семантическая сеть предметной области как основа для формирования сети переходов при автоматическом распознавании слитной речи

Описывается подход к автоматическому формированию сети переходов для заданной предметной области на основе использования однородной семантической сети корпуса текстов этой предметной области.

Ключевые слова: *распознавание слитной речи, сеть переходов между моделями слов, автоматический анализ текстов, семантическая сеть*

Введение

Задача распознавания слитной речи в настоящий момент не считается успешно решенной. Тем не менее известны лицензионные продукты таких фирм, как Nuance [1], Autonomy [2], которые используются в приложениях, в том числе русскоязычных. Хорошо зарекомендовал себя подход к распознаванию слитной речи на основе скрытых марковских моделей, в результате применения которого ошибка распознавания при распознавании спонтанной слитной речи в заданной предметной области в условиях шума машинного зала не превышает в полевых условиях 15 %. Этот подход имеет единственный недостаток, который, тем не менее, существенно ухудшает картину: сеть допустимых переходов между словами в заданной предметной области строится всегда вручную, а это трудоемкий процесс.

Распознавание речи требует обработки и анализа речевых сигналов, их преобразования в элементарные речевые единицы и интерпретации полученной последовательности с учетом лингвистических знаний, чтобы исправить неверно распознанные единицы. Таким образом, разработка любой системы автоматического распознавания и понимания речи невозможна без учета специфики языка, следовательно, проблема распознавания речи находится на стыке таких научных областей как компьютерная лингвистика, фонетика, распознавание образов, цифровая обработка сигналов и др.

Исследованию речевого аппарата и математическому обоснованию частотных характеристик звуков речи были посвящены пионерские работы А. Бела, Г. Фанта и Д. Фланагана, Р. Якобсона и др. Появление ЭВМ привело к необходимости развития методов цифровой обработки речевого голосового сигнала. Важную роль в этой области сыграли работы Б. Голда, Д. Маркела, А. Оппенгейма, Л. Рабинера. Д. Редди, Р. Шафера и др., заложивших научные основы распознавания речи статистическими методами и внесших существенный вклад в развитие методов распознавания речевых голосовых команд. Значительный вклад в развитие технологий распознавания речевых голосовых сигналов внесли известные ученые Х. Сакоэ и С. Чиба в Японии, Ф. Итакура в США, В. М. Величко. Н. Г. Загоруйко. В. М. Сорокин в Советском Союзе. Разработанные методы базировались в основном на статистическом подходе с использованием марковских цепей, критерия максимального правдоподобия и байесовских правил.

Имеющиеся надежные методы распознавания (87...99 %) дикторозависимы, используют небольшой

словарь (до 1000 слов), распознают слова и команды как целые единицы и рассматривают временное представление речи как результат нелинейного сжатия/растяжения. В основу этих систем заложена концепция последовательной обработки речевого сигнала методами динамического программирования, предложенная Т. К. Винцюком в 80-х гг. XX века. Системы, обеспечивающие распознавание небольшого словаря, можно применять для голосового управления оборудованием, речевого запроса к базам данных, встраивания голосовых функций в мобильные электронные устройства.

По мере развития компьютерной техники стало возможным работать с большими массивами данных в режиме реального времени, в результате чего начались исследования по созданию статистических моделей языка, использующих различные единицы речи (словоформа, лемма, морфема и др.) и разработке методов автоматического дикторонезависимого распознавания слитной речи, которые требуют больших словарей. Для распознавания речи с большими словарями используют пофонемный подход, разрабатывают новые решающие правила, основанные на статистических методах или на аппарате искусственных нейронных сетей. Широкое применение получили алгоритмы, в которых для моделирования элементарных единиц уровня фонемы применяют скрытые марковские модели или смеси гауссовских распределений. Однако методы вероятностного моделирования речевых и языковых процессов требуют создания и обработки огромных текстовых и речевых массивов, что связано с использованием больших финансовых и временных ресурсов.

Несмотря на то что распознаванием речи начали заниматься еще в 60-х гг. прошлого века, результаты многочисленных исследовательских групп в этой области остаются довольно скромными. Трудность применения технологий распознавания речи для славянских языков связана со сложным механизмом словообразования. Существенными препятствиями в создании систем автоматического распознавания славянских языков являются их принципиальные отличия от основных европейских: высокая степень флективности, тенденция к фонетической редукции, свободный порядок слов. Это очень затрудняет построение статистических языковых моделей, на которых сегодня базируются все более или менее работающие системы распознавания речи. Фонетическая редукция приводит к "смазыванию" акустических свойств сегментов, что необходимо учитывать при формировании наборов акустических характеристик элементарных языковых единиц. Относительно автоматического распознавания речи это означает, что словоформ, которые так или иначе должны быть учтены при составлении словаря, почти на порядок больше, чем для языков с низкой флективностью. В связи с этим резко возрастает размер словаря и падает точность распознавания. Поэтому во многих современных широко разрекламированных системах, использующих голосовой интерфейс, отсутствует поддержка славянских языков.

На сегодня существует несколько систем, которые используют элементы распознавания речи. Во-первых, это дикторозависимая система распознавания от Microsoft, поставляемая с операционными системами Windows Vista и Windows 7. Для ее настройки и надежной работы пользователь должен озвучить достаточно большой фрагмент текста. Основным недостатком системы для российского пользователя является отсутствие поддержки славянских языков. Во-вторых, это система автоматизированной расстановки субтитров в видеороликах YouTube от Google. Система является дикторонезависимой, однако ориентирована она также на англоязычную аудиторию. Точность распознавания этой системы не очень высока. В-третьих, это дикторонезависимая система голосового поиска от Google, которая поддерживает не только английский язык, но и русский. Имеющиеся отзывы по тестированию этой системы свидетельствуют о не очень высокой точности распознавания слов русского языка. Кроме указанных существует целый ряд менее известных систем распознавания речи, однако перечень проблем, которые в них не решены, остается подобным приведенным выше.

Таким образом, проблема распознавания речи является открытой и поиски новых подходов к ее решению составляют актуальное направление в области искусственного интеллекта. Существующие модели понимания речи значительно уступают речевым способностям человека, а проблема взаимодействия человека с компьютером — комплексная задача, где, во-первых, необходимо учитывать другие, кроме речи, естественные модальности передачи информации (жесты, направление взгляда и т. п.), которые использует человек, а, во-вторых, необходимо использовать информацию верхних уровней (в том числе семантического и прагматического).

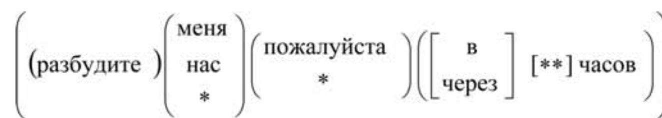
Базовая модель распознавания слитной речи может быть реализована на основе разных подходов. Одним из наиболее известных, хорошо зарекомендовавшим себя, и до сих пор применяемым подходом является подход на основе скрытых марковских моделей [3], имеющий общие корни с алгоритмом динамического программирования, разработанным Т. К. Винцюком. При этом контекст различных уровней учитывается сетью переходов: фонемы соединяются в слова, слова — в предложения, предложения принадлежат текстам, описывающим предметную область. Предполагается, что на вход системы поступает параметризированный речевой сигнал в виде последовательности векторов параметров в пространстве первичного описания. Параметризация сигнала проводится препроцессором,

который моделирует физическое восприятие речевого сигнала с учетом свойств акустического канала и фоновых помех. В ответ на произнесенное речевое сообщение на выходе системы получается результат распознавания в виде одной или нескольких гипотез о произнесенной последовательности слов. Эти последовательности, дополненные оценкой сопутствующих речевых параметров, передаются в блок смысловой интерпретации, где происходит принятие окончательного решения в пользу одной из гипотез. Знания о предметной области, в которой используется система распознавания, позволяют существенно сузить и ускорить поиск.

В одном из подходов к распознаванию слитной речи [4] при преобразовании речи в текст используют словарь, в котором каждое слово задано своим графемным представлением. По этому представлению слова формируется одна или несколько фонемных транскрипций слова. Далее, исходя из фонемного состава слова, формируется акустический прототип слова путем объединения различных прототипов акустических образов фонемного уровня (например, трифонов). При распознавании предъявленная реализация слова сравнивается с преобразованными сформированными акустическими прототипами слова. При преобразованиях прототипов слова сохраняется порядок следования акустических форм фонемного уровня прототипа и варьируется в разрешенных пределах длительность акустических форм. Процесс сравнения и поиск наилучшей меры сходства осуществляется методом динамического программирования. Распознаваемая реализация относится к тому слову, преобразованный прототип которого дал наибольшее интегральное сходство с распознаваемым сигналом.

Так же как прототипы слов из прототипов графем создаются прототипы словосочетаний, а прототипы предложений — из прототипов слов, и рассматриваются их допустимые преобразования. В этом случае процесс перебора допустимых фраз и возможных границ между словами достигается методами динамического программирования. Ответом на слитную речевую последовательность является та допустимая фраза или предложение, преобразованный прототип которой оказался самым похожим на предъявленный для распознавания речевой сигнал. Наиболее просто, но не с наилучшим качеством, решается задача распознавания слитной речи в случае свободного порядка следования слов. Если же учитывать не только лексику, но и синтаксис, и семантику речи, то на порядок слов накладываются дополнительные тематические ограничения (ограничения на сочетаемость слов).

Фонемные эквиваленты графемных представлений фраз и предложений конкретной предметной области, так же как и в случае формирования фонемных



Сеть переходов для просьбы "разбудить в определенное время"

представлений слов, формируют вручную. При этом строят сеть переходов, в которой учитывают все возможные комбинации слов всех предложений всех текстов, описывающих предметную область. Так сеть просьбы "разбудить в определенное время" из предметной области "Гостиница" будет выглядеть следующим образом (рисунок).

На рисунке в круглых скобках указаны подсловари, которые можно менять местами, а в квадратных — подсловари, которые переставлять нельзя. Переставлять подсловари можно только внутри старших скобок. Символ * означает пустое слово, а ** — конкретный час. Не все слова в этой фразе являются ключевыми, только "разбудите" и "часов".

В отличие от трудоемкого метода построения моделей слов с учетом правил комбинаторики элементов, не представляет большой сложности автоматически построить марковскую модель слова. Для этого из речевого корпуса, содержащего определенное число произнесений слов словаря (достаточное для формирования представительной модели) в автоматическом режиме формируются модели слов. Объединение этих моделей в более крупные единицы представляет собой иногда непреодолимые трудности. Модель языка таким образом построить трудно (а для русского языка просто невозможно), так как требуемого для обучения модели количества произнесенных текстов просто не существует физически. Для модели предметной области это ограничение менее критично, но тоже достаточно серьезно. Именно поэтому известные системы распознавания слитной речи работают более или менее прилично на нескольких предметных областях (для которых сформированы модели), и плохо на других предметных областях.

Менее сложна, по сравнению с распознаванием слитной речи, задача распознавания ключевых слов в потоке слитной речи. В этом случае трудоемкий процесс ручного формирования сети допустимых переходов между ключевыми словами в заданной предметной области может быть автоматизирован с использованием технологии автоматического построения однородной семантической (ассоциативной) сети текста на основе корпуса текстов этой предметной области. Семантическая сеть предметной области (текста) является одним из способов представления в сетевом виде семантики предметной области (текста). Особенностью этого автоматического процесса формирования сети переходов является необходимость выявле-

ния ключевых, в заданной предметной области, слов, а также существенно важных для описания предметной области предложений. Предложения не несущие информации о предметной области в этом случае отбрасываются. Чем тщательнее будет проведен отбор, тем более точно будет работать система распознавания речи. После выбора ключевых слов, а также значимых предложений текста, на основе этих предложений строится однородная семантическая (ассоциативная) сеть. Далее эта сеть используется для преимущественного выбора гипотез ключевых слов, которые входят в эту семантическую сеть и находятся на наименьших расстояниях от предыдущего распознанного слова.

Таким образом, в работе ставилась задача заменить трудоемкий процесс ручного формирования сети допустимых переходов между ключевыми словами в заданной предметной области автоматическим процессом с использованием технологии автоматического построения однородной семантической (ассоциативной) сети текста на основе корпуса текстов этой предметной области.

Исследования были проведены в ходе выполнения работ по проекту "Исследование современных способов организации и представления данных и разработка макета адаптируемых сервисов электронных библиотек и других информационных фондов на основе самообучаемой системы квалификации контента", грант Минобрнауки 2011-1.4-514-031-020.

Автоматическое формирование сети переходов

Идея автоматического формирования сети переходов заключается в следующем. В тексте выявляют ключевые понятия в их взаимосвязях. Сеть из ключевых понятий текста, учитывающая их связи (ассоциативная — однородная семантическая — сеть), может быть основой для построения сети переходов. Ключевые понятия выявляют на фоне других, второстепенных слов. Достигается это вычислением частоты встречаемости слов [5], а также попарной встречаемости слов в осмысленных фрагментах текста — предложениях. Далее частотные характеристики слов — вершин сети — перенормируются в их смысловые веса, в результате чего появляется возможность ранжировать и ключевые слова, и предложения текста с точки зрения их важности в этой сети и, следовательно, в тексте (или в корпусе текстов, описывающих предметную область). Другими словами, такой анализ позволяет выявить важные ключевые слова и существенные предложения текста на фоне второстепенных.

Построенная на основе этого множества предложений ассоциативная сеть является сетью переходов между ключевыми словами для этой группы предложений. Сеть, построенная на всех предложениях корпуса текстов, описывающего предметную область ранга выше порогового, является сетью переходов между ключевыми словами для всей предметной области.

Автоматическое формирование ассоциативной сети

Текст представляет собой внутренне структурированную последовательность символов, такую, что различные ее элементы имеют разную сложность и частоту встречаемости. В процессе анализа текста формируются словари $\{B_i\}_j$ этих элементов. Это, например, словари флективных морфем, корневых основ. Выявленные лингвистические единицы в дальнейшем можно использовать для обработки текстовой информации. Словарь флективных морфем можно использовать для морфологического анализа, словарь корневых основ — для выявления ключевых понятий в тексте и формирования однородной (ассоциативной) семантической сети.

Ранее одним из авторов была реализована технология обработки текстовой информации TextAnalyst [6], позволяющая автоматически выявлять ключевые понятия в тексте на основе информации только о структуре самого текста (независимо от предметной области). Для этого формируется частотный портрет текста, содержащий информацию о частоте встречаемости понятий текста, представленных как корневые основы соответствующих слов или их устойчивых сочетаний, встречающихся в тексте, а также об их совместной (попарной) встречаемости в смысловых фрагментах текста (в предложениях). Частотный портрет, таким образом, содержит информацию о частоте встречаемости понятий и их попарной (в терминах их ассоциативной связи) встречаемости в тексте. Использование хопфилдоподобного алгоритма позволяет перейти от частоты встречаемости к смысловому весу (вес связей при этом остается неизменным).

Эта обработка включает несколько этапов. На первом этапе осуществляется первичная обработка: из текста удаляется нетекстовая информация, текст сегментируется на слова и предложения, из текста удаляются стоп-слова, рабочие и общеупотребимые слова, а оставшиеся слова подвергаются морфологической обработке. Для простоты анализа морфологическая обработка проводится с использованием традиционного морфологического словаря — словаря первого уровня $\{B_i\}_1$. Далее формируется словарь второго уровня $\{B_i\}_2$ — словарь корневых основ (и устойчивых словосочетаний). На следующем этапе строится частотный портрет текста, т. е. выявляются частоты p_i встречаемости корневых основ понятий (полученных в результате морфологического анализа) и их устойчивых сочетаний, и частоты p_{ij} их попарной встречаемости в пред-

ложениях текста (т. е. формируется словарь третьего уровня $\{B_i\}_3$). Выявленные таким образом пары слов позволяют построить первичную ассоциативную сеть — частотный портрет текста — объединением второго слова следующей пары со вторым словом предыдущей пары. После перенормировки частотных характеристик понятий сети в их смысловые веса эта сеть становится исходной для формирования сети переходов.

Автоматическое выявление ключевых понятий текста

На третьем этапе обработки, частоты встречаемости перенормируются в смысловые веса с использованием итеративной процедуры, похожей на алгоритм искусственной нейронной сети, предложенной Хопфилдом:

$$w_j(t+1) = \left(\sum_{\substack{i \\ i \neq j}} w_i(t) w_{ij}\right) \sigma\left(\bar{E}\right),$$

где t — шаг итерации; $w_i(0) = \ln p_i$; $w_{ij} = \ln p_{ij} / \ln p_j$ и $\sigma\left(\bar{E}\right) = 1 / \left(1 + \mathbf{e}^{-k\bar{E}}\right)$ функция, нормирующая на среднее суммарное значение всех весов вершин сети $\bar{E}$, которая минимизируется в процессе сходимости сети, а k — коэффициент, задающий скорость сходимости процесса. В результате итеративной процедуры перенормировки наибольшие веса получают понятия, связанные с наибольшим числом других понятий с большим весом, т. е. те понятия, которые "стягивают" на себя смысловую структуру текста.

Полученные таким образом смысловые веса ключевых понятий показывают значимость этих понятий в тексте. В дальнейшем эта информация будет использована для выявления предложений текста, содержащих наиболее важную информацию. Понятия сети ранжируются по их смысловым весам с последующим использованием только тех понятий, которые превышают заранее заданный порог.

В результате получается так называемая ассоциативная (однородная семантическая) сеть N как совокупность пар понятий $\langle c_i c_j \rangle$, где c_i и c_j — понятия, связанные между собой отношением ассоциативности (совместной встречаемости в некотором фрагменте текста):

$$N \cong \{\langle c_i c_j \rangle\}.$$

В данном случае отношение ассоциативности несимметрично: $\langle c_i c_j \rangle \neq \langle c_j c_i \rangle$.

Семантическая сеть, описанная таким образом, может быть переописана как множество так называемых звездочек $z_i = \langle c_i \langle c_j \rangle \rangle$:

$$N \cong \{z_i\} = \{\langle c_i \langle c_j \rangle \rangle\}.$$

Под звездочкой $z_i = \langle c_i \langle c_j \rangle \rangle$ понимается конструкция, включающая главное понятие c_i, связанное с множеством понятий-ассоциантов $\langle c_j \rangle$, которые являются семантическими признаками главного понятия, отстоящими от главного понятия в ассоциативной сети на одну связь. Ассоциативные связи направлены от главного понятия к понятиям-ассоциантам.

Сеть переходов в задаче распознавания ключевых слов в потоке слитной речи

Задача распознавания ключевых слов в потоке слитной речи проще, чем задача точного распознавания слитной речи: в этом случае распознаются не все произносимые слова, а только некоторые — ключевые — имеющие большее по сравнению с другими словами значение (заранее оговоренные) в тексте. Если в случае распознавания слитной речи для успешного распознавания требуется построение полной сети переходов (см. рисунок), которая позволяет существенно сократить оперативный словарь на каждом шаге распознавания от слова к слову, в случае распознавания ключевых слов сеть переходов упрощается. Из нее уходит второстепенная информация, отаются только ключевые слова и переходы между ними.

Ассоциативная сеть, построенная так, как это описано в предыдущих двух разделах, может быть использована для уменьшения числа гипотез при распознавании текущего ключевого слова. В этой сети присутствуют только ключевые слова. Причем после распознавания очередного слова, оперативный словарь ограничивается только ближайшими ассоциантами распознанного ключевого слова, что существенно упрощает саму задачу распознавания, так как текущий словарь системы на данном этапе распознавания при этом существенно редуцируется.

Пример построения ассоциативной сети

Для наглядности представления описанного подхода рассмотрим пример построения сети переходов, на основе текстов разговорников для раздела "Бронирование гостиницы". Сеть формируется на проходе с помощью персонального продукта, реализующего статистический нейросетевой подход к анализу неструктурированных текстов TextAnalyst. Исходный текст примера — это текст из русско-французского разговорника.

Пример. *"Я делал заказ. Места были зарезервированы для меня и моей семьи. Заказ был подтвержден в Париже. У вас есть свободные места? Мне нужна комната. Я хотел бы одноместный номер. Я хотел бы номер с ванной. Я хотел бы номер с двумя кроватями. Нам нужен двухместный номер с дополнительной кроватью. Есть что-нибудь подешевле? Не могли бы Вы*

показать мне комнату получше? Не могли бы Вы показать мне комнату побольше (поменьше)? Комнату с видом на море. Какова плата за обслуживание и налог? Надбавка за обслуживание учтена? Сколько стоит номер, включая завтрак? Завтрак включен? Сколько это стоит в день? Вам нужен залог? Когда я должен освободить номер? Вам нужен мой паспорт? Можете порекомендовать другую гостиницу? Я сниму этот номер на неделю (месяц). Я пробуду два дня. Меня зовут Вася. Где мне расписаться? Можно одолжить Вашу ручку?"

Сформированная ассоциативная сеть этого текста представлена в таблице.

Ассоциативная сеть текста Примера

Родитель	Подчиненный
включая	
	завтрак
включен	
	завтрак
двухместный номер	
	нужен
завтрак	
	включен
заказ	
	делал
	подтвержден
залог	
	нужен
надбавка	
	обслуживание
нужен	
	двухместный номер
	залог
	паспорт
	комната
комната	
	показать
	нужна
обслуживание	
	надбавка
	плата
паспорт	
	нужен
плата	
	обслуживание
подтвержден	
	заказ
показать	
	комната

Конечно, сеть выглядит несколько примитивно, кроме того, в ней есть информационный шум. Сеть построена на маленьком тексте, описывающем предметную область. Для более полного учета переходов необходимо собрать более представительный корпус текстов. Также необходима чистка сети экспертом, и возможно ее дополнение. Однако если учесть, что ручное формирование сетей переходов, описывающих предметную область в системах распознавания речи, — очень трудоемкое занятие, подспорье в построении сети имеет несомненные преимущества.

Отзывы экспертов

Технология построения ассоциативной сети TextAnalyst зарекомендовала себя достаточно хорошо. Если считать критерием качества работы семантического анализатора качество реферата, построенного на основе этого семантического анализа, то можно сказать, что эта технология работает очень устойчиво: "This robustness speaks very well of the core technology and the summarization technique TextAnalyst is using" — такое заключение дала известный специалист в области анализа текстов из Стэнфордского университета Colleen E. Crangle в частном отзыве для немецкой фирмы Arexera.

Данный подход к анализу текстов рассматривался наряду с двумя другими — фирмы IBM и фирмы Oracle — и также признан имеющим несомненные преимущества [7].

Дальнейшие исследования

В рамках данной работы сеть переходов строилась с использованием статистического подхода и для решения задачи распознавания ключевых слов в потоке слитной речи. Наряду со статистическими методами для автоматического анализа текстов используют лингвистические методы. Лингвистический подход к анализу текстов позволяет решить задачу семантического анализа более точно, но только для отдельных предложений текста. Объединение семантических представлений отдельных предложений в единое представление всего текста — до сих пор нерешенная задача. Поэтому совмещение двух подходов — статистического и лингвистического — позволит улучшить качество анализа при возможности построения семантической сети всего текста. В этом случае семантическая сеть всего текста строится статистическим методом, но вместо пар слов в ней учитываются тройки, включающие пару слов и связь между ними. Эти тройки выявляют в процессе синтаксического анализа. Из предложений текста выявляют содержащиеся в них предикатные структуры, а также отношения сочинения, и атрибутивные отношения.

Под предикатной структурой понимается ядерная подлежащно-сказуемостная конструкция, включающая в свой состав n актантов. В общем случае ядро — это глагольная конструкция, а актанты объединяются с ним системой отношений, заполняя его валентные гнезда [8—10]. Таким образом, предикатная структура имеет следующий вид:

$$Subj - R_0 - \mathrm{Pr}ed - R_i - Obj_i, \quad i = 1...n,$$

где $Subj$ — активный субъект (грамматическое подлежащее), который инициирует использование предиката $\mathrm{Pr}ed$ (грамматического сказуемого); R_0 — отношение "быть субъектом"; R_i — предикативные отношения; Obj_i — актанты или узлы предикатной структуры (имя существительное, местоимение, числительное) в их атрибутивной форме, n — число актантов.

Общую схему действий по выявлению обобщенной предикатной структуры можно представить в виде последовательности шагов.

1. Членение предложения по знакам пунктуации и сочинительным союзам на начальные сегменты (фрагментация). Определение вершин и типов начальных сегментов.

2. На декартовом произведении омонимов внутри начальных сегментов построение множества однозначных морфологических интерпретаций каждого сегмента.

3. Построение синтаксических групп для каждой интерпретации сегмента с помощью синтаксических правил, выявляющих синтаксические связи между словами. Оценка синтаксического покрытия каждой интерпретации.

4. Установление иерархии между сегментами с помощью синтаксических правил: вложения контактно расположенных сегментов (причастных, деепричастных оборотов, обособленного определения); определение однородности между контактно расположенными сегментами; определение отношения импликации между сегментами по подчинительным союзам, в них входящим.

Предикатная структура простого предложения выявляется на основе анализа главных членов предложения путем определения шаблона, соответствующего одной из минимальных структурных схем предложений, описывающих предикативный минимум предложения [10]. Использование минимальных структурных схем предложений в качестве формального образца позволяет выявить предикативную основу (структурную схему) простого предложения, и в дальнейшем — его предикатную структуру. Смысловая связь между понятиями предложения (объектом/субъектом) в общем случае может быть описана предикатом, актантами которого выступают данные понятия. Установление таких синтактико-семантических связей позволяет сформировать схему ситуации, описываемой во фразе.

Обусловленный валентностью предиката семантико-синтаксический уровень анализа конструкций, не соответствующий узкому формально-синтаксическому подходу, дает возможность даже из набора неправильных форм (посредством приведения их к начальным формам) определить схему предложения с помощью заполнения валентных гнезд.

Выявление ключевых предикатных структур в тексте возможно с применением того же подхода, который был использован для выявления ключевых понятий и ключевых предложений в тексте в технологии TextAnalyst. Для этого после выявления ключевых предикатных структур текста в ассоциативной сети текста отношения между парами ключевых понятий размечаются их типами из соответствующей предикатной структуры, и, таким образом, звездочка ассоциативной сети заменяется предикатной структурой, причем главное понятие ассоциативной сети совмещается с субъектом, а ближайшие ассоцианты — главным и второстепенными объектами и их атрибутами предикатной структуры. Ассоциативные связи заменяются соответствующими типами отношений предикатной структуры — собственно предикатным отношением между субъектом и главным объектом, а также другими отношениями — валентными связями актантов предиката — субъекта с второстепенными объектами и их атрибутами соответственно.

Заключение

Автоматический смысловой анализ текста с использованием технологии TextAnalyst позволяет выявить ключевые понятия в их взаимосвязях в тексте, а также ранжировать их по степени их смысловой значимости в данном тексте. Ранжирование ключевых понятий, в свою очередь, позволяет ранжировать предложения текста и выбирать наиболее существенные из них для множества текстов, описывающих предметную область. Такая сеть может быть исходной для построения сети переходов между ключевыми словами в задаче распознавания ключевых слов в потоке слитной речи. В дальнейших работах предполагается использовать объединение статистического и лингвистического подходов к автоматическому смысловому анализу текстов, что позволит строить более точную сеть переходов за счет разметки ассоциативных связей между ключевыми словами метками типов их отношений в предикатных структурах соответствующих предложений.

Список литературы

1. http: //www.nuance.com
2. http: //www.autonomy.com
3. **Vintsiuk T., Sazhok M., Vasylieva N., Chollet G.** Acoustic-Phonetic Model Application for Syllable Speech Recognition Output Post-Processing // XII International Conference "Speech and Computer" — Specom'2007. P. 182—187.
4. **Винцюк Т. К.** Анализ, распознавание и смысловая интерпретация речевых сигналов. Киев. Наукова думка, 1987. 267 с.
5. **Харламов А. А.** Нейросетевая технология представления и обработки информации (естественное представление знаний). М.: Радиотехника, 2006. 89 с.
6. http: //www.analyst.ru
7. **Sullivan D.** Document Warehousing and Textmining. NY.: Wiley publishing house, 2001. 542 p.
8. **Загітко А. П.** Теоретична граматика української мови. Морфологія. Донецьк: ДонДУ, 1996. 300 с.
9. **Харламов А. А., Ермоленко Т. В., Дорохина Г. В., Гнитько Д. С.** Метод выделения главных членов предложения в виде предикатных структур, использующий минимальные структурные схемы // Речевые технологии. 2012. № 2. С. 75—85.
10. **Белошапкова В. А., Брызгунова Е. А., Земская Е. А.** и др. Современный русский язык: Учебник для филологических специальностей высших учебных заведений / Под ред. В. А. Белошапковой. 3-е изд, испр. и доп. М.: Азбуковник, 1997. 928 с.

ИНФОРМАЦИЯ

С 1 по 3 октября 2013 в Москве, в ЦВК "Экспоцентр" состоится 2-я международная форум-выставка передовых информационных технологий

"ИТ СТРАТЕГИИ ЛИДЕРСТВА"

Тематическая направленность выставки:

- IT решения по управлению корпоративной информацией
- Инфраструктура и системы хранения данных
- Технологии управления, обучения и подготовки кадров

Контактная информация:

Тел.: (812) 320-8098, 320-0141 *Факс*: (812) 320-8090

E-mail: itcom@restec.ru, ict-dep@restec.ru

УДК 519.256

А. В. Иванова, студент, **А. А. Адуенко**, студент, Московский физико-технический институт,
В. В. Стрижов, канд. физ.-мат. наук, доц., науч. сотр., Вычислительный центр
им. А. А. Дородницына РАН, г. Москва, e-mail: strijov@ccas.ru

Алгоритм построения логических правил при разметке текстов[1]

Предложен метод восстановления структуры библиографических записей BibTeX по их текстовому представлению. Структура восстанавливается с помощью логических правил, определенных на экспертно-заданном множестве регулярных выражений. Для построения набора логических правил предложен алгоритм, использующий тупиковые покрытия. Предложенный алгоритм проиллюстрирован задачей поиска структуры библиографических записей, представленных набором текстовых строк.

Ключевые слова: *библиографическая запись, BibTeX, регулярное выражение, тупиковое покрытие, кластеризация*

Введение

Задан набор текстовых строк, которые составлены в соответствии с некоторыми правилами, например, определен порядок следования информации в строках и способ форматирования. Требуется решить задачу восстановления правил, по которому был построен набор строк. Для различных типов строк и правил, по которым они построены, предложены разные способы решения задачи [1, 2]. В данной работе в качестве текстовых строк рассматриваются библиографические записи, которые строятся по существующим стандартам (ГОСТ 7.82—2001, ISO 690-2:1997, MLA). Однако разные стандарты определяют разный порядок следования полей библиографической записи и разный способ форматирования полей. Более того, библиографическая запись может содержать ошибки.

В данной работе рассматривается задача выделения сегментов библиографических записей и их разметка — определение соответствия между текстовыми сегментами библиографической записи и полями структуры BibTeX. Рассматриваются следующие поля структуры BibTeX: автор или авторы; название; журнал; страницы; том; номер; год; город; издательство; редакторы; ссылка в сети Интернет.

Кроме того, наряду с неразмеченной коллекцией — набором библиографических записей, которые требуют разделения на сегменты и дальнейшей разметки этих сегментов, имеется размеченная коллекция библиографических записей, для которой сегментация и разметка уже проведены. Таким образом, требуется осуществить выделение сегментов и разметку библиографических записей, т. е. классификацию полученных сегментов на классы, соответствующие полям структуры BibTeX, располагая уже размеченной коллекцией. В работе эта задача рассмотрена как задача кластеризации [3]. При этом каждый класс, соответствующий полю структуры BibTeX, описан несколькими кластерами. При наличии размеченной коллекции применимы методы частичного обучения [4], однако в данной работе размеченная коллекция использована лишь для интерпретации получившихся кластеров, т. е. определения, какой класс описывается рассматриваемым кластером.

[1] Работа выполнена при поддержке Министерства образования и науки РФ в рамках Государственного контракта 07.524.11.4002.

Для проведения сегментации и кластеризации сегментов [5] порождались признаки объектов [6]. В качестве объектов были использованы позиции в тексте библиографической записи, в качестве признаков — регулярные выражения [7], задающие шаблоны текстовых строк. Значение признака позиции равно единице, если библиографическая запись, начиная с этой позиции соответствует шаблону. В противном случае это значение равно нулю. Таким образом была порождена бинарная матрица объектов—признаков [6].

В алгоритме кластеризации пошагово строились кластеры сходных объектов. Для определения меры сходства [8] были использованы тупиковые покрытия [9]. Для их построения был применен алгоритм, изложенный в работе [9]. После кластеризации каждому кластеру был поставлен в соответствие класс — поле библиографической записи, которое он описывает. Соответствие было установлено с помощью размеченной коллекции. Кластер относится к тому классу, объектов которого в нем наибольшее число из размеченной коллекции.

Постановка задачи и алгоритм кластеризации

Библиографические записи представлены текстовой строкой T, где $t = |T|$ — число символов в строке. В качестве объектов используются позиции в строке, а для построения признаков используется экспертно заданное множество Θ регулярных выражений мощности $\theta = |\Theta|$. Построим матрицу $\mathbf{D}$ размеров $t \times \theta$ следующим образом:

$$\mathbf{D}(i, j) = \begin{cases} 1, \text{ если на позиции } i \text{ сработало} \\ \text{регулярное выражение } j, \\ 0, \text{ в противном случае.} \end{cases} \quad (1)$$

Обозначим p-й столбец единичной матрицы размеров $r \times r$ как $\mathbf{e}_p^r$.

Для описания алгоритма кластеризации необходимо ввести понятия покрытия матрицы и тупикового покрытия.

Определение. Пусть $\mathbf{L}$ — матрица размера $m \times n$. Множество H, состоящее из r столбцов матрицы $\mathbf{L}$, $r \in \mathbb{N}$, $r \leqslant n$, назовем *покрытием* матрицы $\mathbf{L}$, если матрица, образованная столбцами из H, не содержит нулевую строку. Покрытие H матрицы $\mathbf{L}$ назовем тупиковым, если для каждого $p \in \{1, ..., r\}$ найдется столбец $\mathbf{h} \in H$, который содержит $\mathbf{e}_p^r$, т. е. $h_1 = ... = h_{p-1} = h_{p+1} = ... = h_r = 0$, $h_p = 1$.

Заметим, что если рассмотреть построенную согласно выражению (1) матрицу $\mathbf{D}$, то в силу того, что каждый сегмент библиографической записи содержит значительное число позиций и лишь некоторые из них характеризуют этот сегмент и отличают его от остальных, в матрице $\mathbf{D}$ есть множество нулевых строк, соответствующих таким неинформативным позициям в тексте записи. Отсюда получим, что покрытия у исходной матрицы $\mathbf{D}$ не существует. Поэтому будем рассматривать подматрицу $\hat{\mathbf{D}}$ матрицы $\mathbf{D}$, полученную из нее исключением всех нулевых строк.

Введем следующие обозначения:

$\mathbf{\Omega}$ — некоторая подматрица матрицы $\mathbf{D}$;

$\sigma^T(\mathbf{\Omega})$ — множество тупиковых покрытий матрицы $\mathbf{\Omega}$.

Для двух подматриц $\mathbf{\Omega}_f$ и $\mathbf{\Omega}_e$ матрицы $\mathbf{D}$ с одинаковым числом столбцов определим подматрицу $\sigma^T(\mathbf{\Omega}_f|\mathbf{\Omega}_e)$ — множество тупиковых покрытий матрицы $\mathbf{\Omega}$, которые также являются тупиковыми и для соединенной матрицы $\begin{bmatrix}\mathbf{\Omega}_f \\ \mathbf{\Omega}_e\end{bmatrix}$. Создадим матрицу $\mathbf{\Omega} = \hat{\mathbf{D}}$, полученную из $\mathbf{D}$ исключением всех нулевых строк. Для описания алгоритма кластеризации требуется ввести меру сходства. Предлагаемый алгоритм является пошаговым, причем на i-м шаге рассматривается очередной объект $\mathbf{x}_i$, соответствующий некоторой строке $\mathbf{\Omega}$. Он либо относится к рассматриваемому на i-м шаге кластеру M_j, либо образует новый кластер M_{j+1}. Поэтому требуется определить меру сходства не между объектами, а между объектом и кластером объектов, т. е. набором объектов. Для этого воспользуемся понятием тупикового покрытия и в качестве меры сходства или близости очередного объекта $\mathbf{x}_i$ кластеру M_j будем рассматривать функцию

$$P(\mathbf{\Omega}_f, \mathbf{x}_i) = \frac{\left|\sigma^T(\mathbf{\Omega}_f)\mathbf{x}_i\right|}{\left|\sigma^T(\mathbf{\Omega}_f)\right|}. \quad (2)$$

Здесь $\mathbf{\Omega}_f$ — матрица, составленная из признаковых описаний объектов, входящих в рассматриваемый кластер M_j. Заметим, что $P(\mathbf{\Omega}_f, \mathbf{x}_i) \leqslant 1$ и эта величина определяет сходство объекта $\mathbf{x}_i$ с кластером M_j. Чем $P(\mathbf{\Omega}_f, \mathbf{x}_i)$ больше, тем ближе объект $\mathbf{x}_i$ к кластеру M_j. В числителе формулы (2) находится мощность множества тупиковых покрытий соединенной матрицы $\begin{pmatrix}\mathbf{\Omega}_f \\ \mathbf{x}_i\end{pmatrix}$, а в знаменателе мощность множества тупиковых покрытий матрицы $\mathbf{\Omega}_f$, составленной из признаковых описаний объектов, входящих в кластер M_j. Опишем алгоритм кластеризации подробнее. Для удобства обозначим Ω множество объектов, признаковое описание которых есть строки матрицы $\mathbf{\Omega}$. Зададим некоторый порог $p \in [0, 1]$.

Шаг 1. Положим, что кластер M_1 включает единственный элемент $\mathbf{x}_1$, т. е. $M_1 = \{\mathbf{x}_1\}$, где $\mathbf{x}_1$ — объект, описанный первой строкой $\mathbf{\Omega}$. Если в матрице $\mathbf{\Omega}$

ровно одна строка, то закончить процедуру, иначе перейти к шагу 2.

Шаг 2. Пусть построены классы $M_1, ..., M_{j-1}, j \geqslant 2$. Обозначим множество

$$\tilde{\Omega}_{j-1} = \Omega \setminus \bigcup_{k=1}^{j-1} M_k.$$

Если это множество пусто, то все объекты из множества Ω уже кластеризованы, поэтому заканчиваем работу. Иначе определяем объект $\mathbf{x}' \in \tilde{\Omega}_{j-1}$ такой, что

$$\mathbf{x}' = \operatorname*{argmax}_{\mathbf{x} \in \tilde{\Omega}_{j-1}} P(M_{j-1}, \mathbf{x}). \quad (3)$$

То есть находим элемент, наиболее близкий к текущему рассматриваемому кластеру. Если $P(M_{j-1}, \mathbf{x}') \geqslant p$, то $M_{j-1} = M_{j-1} \cup \{\mathbf{x}'\}$, где $\mathbf{x}'$ определен формулой (3), иначе начинаем добавлять объекты к новому классу $M_j : M_j = \{\mathbf{x}'\}$. Алгоритм заканчивает работу, когда все объекты из множества Ω будут распределены по кластерам.

Интерпретация кластеров. Полученные кластеры требуется проинтерпретировать, т. е. указать для каждого из них класс, который он описывает, иначе указать, к какому полю библиографической записи отнести элементы, в нем находящиеся. Для решения этой задачи используют размеченную коллекцию. Кластер относится к тому классу, объектов которого в кластере из размеченной коллекции больше, чем в прочих кластерах.

Вычислительный эксперимент

Проиллюстрируем предложенный алгоритм на коллекции библиографических записей, представленных в виде текстовых строк. В коллекции 1000 библиографических записей, из которых 100 размечены, т. е. каждому предварительно выделенному сегменту из этих 100 строк поставлена в соответствие метка типа записи. Каждый сегмент принадлежит одному из 11 типов: автор или авторы, название, журнал, страницы, том, номер, год, город, издательство, редакторы, ссылка в сети Интернет. Требуется, пользуясь уже размеченной коллекцией, спрогнозировать разметку оставшейся ее части. Размеченная коллекция использовалась лишь для интерпретации полученных при разметке кластеров сегментов записей.

Для всей коллекции, включая ее размеченную часть, была составлена матрица **D** (1). Приведем примеры регулярных выражений, которые использовались для построения матрицы **D**: наличие четырех цифр подряд, начиная с рассматриваемой позиции, наличие запятой, наличие заглавной буквы, наличие заглавной буквы и точки за ней и т. д. На всей коллекции применялся алгоритм кластеризации с параметром p из отрезка [0, 1]. График зависимости числа

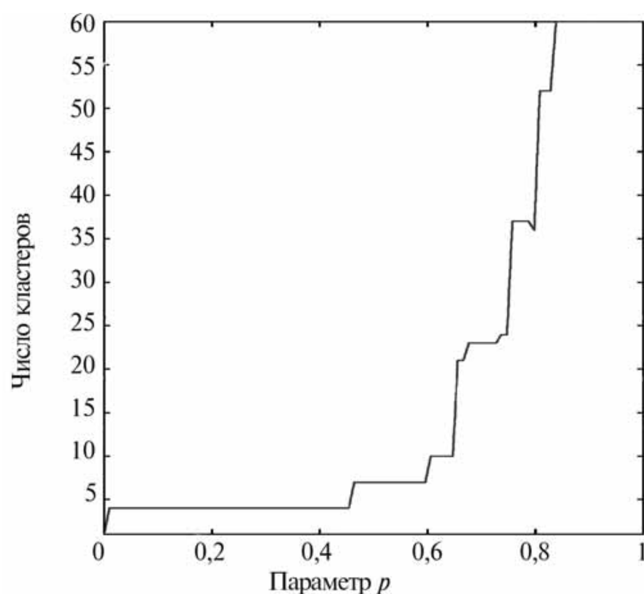


Рис. 1. Зависимость числа кластеров от значения параметра p

выделяемых кластеров от значения этого параметра приведен на рис. 1. При малых $p \ll 1$ число выделяемых кластеров очень мало, напротив — при $p \approx 1$ число кластеров стабилизируется на 60. То, что число кластеров при $p \approx 1$ не равно полному числу сегментов, т. е. не исчисляется тысячами, говорит о том, что набор использовавшихся признаков не является избыточным, т. е. позволяет указать и очень близкие объекты.

При $p = 0{,}83$ число кластеров равно 35. Из них 24 являются шумовыми, т. е. малочисленными и не имеющими элементов из размеченной коллекции. Оставшиеся 11 кластеров поддаются интерпретации с помощью размеченной коллекции. Каждый кластер отнесен к тому типу поля библиографической записи, элементов которого в нем наибольшее число среди записей размеченной коллекции. Кластеры 1, 2 соответствуют полю "авторы", кластеры 3, 4 — инициалам автора, кластер 5 соответствует ссылкам в Интернете, кластеры 6, 7 соответствуют названиям журналов, кластеры 8, 9 — названиям статей, а кластеры 10, 11 — годам выпуска статей или книг. На рис. 2 (см. третью сторону обложки) приведены свойства этих 11 кластеров. Каждому кластеру на диаграмме соответствуют два столбца. Высота синего столбца показывает долю записей (в %) из размеченной коллекции, попавших в кластер и относящихся к тому же типу поля библиографической записи, к которому отнесен и сам кластер. Высота коричневого столбца показывает ту же долю для неразмеченной коллекции.

Опишем некоторые получившиеся кластеры подробнее. Кластер 7, хотя и отнесен к полю названия журналов, содержит почти одинаковое число названий журналов и названий статей. Поскольку отделить

одно от другого, пользуясь только самим написанием, зачастую невозможно, результат вполне закономерен. Возможно, что для улучшения работы алгоритма можно использовать некоторый словарь, содержащий список названий журналов. Кластер 11 отнесен к полю библиографической записи "год", однако туда попало значительное число полей "том", "номер журнала", "страницы". Это также объясняется тем, что различить указанные поля не представляется возможным. Кластер 10, соответствующий тому же полю "год", отличается от кластера 11 тем, что за номером года в сегментах, отнесенных к этому кластеру, есть запятая, а число прочих числовых полей с запятой после номера значительно меньше. Существенное отличие результатов в кластерах 8 и 9 для размеченной и неразмеченной коллекции объясняется тем, что в размеченной коллекции внешние знаки препинания были исключены, а потому в ней не встречались сегменты вида "автор,", "город," или "издательство,", что часто встречается в неразмеченной коллекции и мешает выделению названий статей, содержащих запятые и прочие знаки препинания. В табл. 1 приведены примеры верно и неверно отнесенных к некоторым классам сегментов по кластерам.

Таблица 1

Верно и неверно классифицированные сегменты текста для различных кластеров

Номер и класс кластера	Классифицированные сегменты	
	Верно	Неверно
1: авторы	Hand, D.J. D.N. Potts F.X. Le Dimet	Comput. Syst. U.K. Shalev-Shwartz (Eds.)
2: авторы	Shawe-Taylor Kai-Min Chung MacKay	Prentice-Hall Springer-Verlag TD-Gammon
7: названия журналов	SIAM Journal on computing Neural computation ACM Transactions on Mathematical Software	Sparse matrix test problems Interior-point method for convex programming A trust region algorithm for nonlinearly constrained optimisation
8: названия статей	A QP-free globally convergent, locally superlinearly convergent algorithm for inequality constrained Learning from noisy examples The hardness of approximate optima in lattices, codes and systems of linear equations	Tibshirani, USA MIT Press

Таблица 2

Добавленные регулярные выражения для разных полей структуры BibTeX

Поле структуры BibTeX	Добавленные регулярные выражения
Том	"Vol"; "vol"; C(C); C:C
Номер	"No."; "no."; C(C); C:C
Страницы	"Pp"; "pp."; "pages"; "p."; "page"; C-C
Год	20dd; 19dd
Журнал	"Journal"; "journal'
Издательство	"Publish"; "publish"; "Press"; "press"
Редакторы	"Eds"; "eds"; "ed"; "editor"; "editors"; "edited"; "Editor"; "Editors"

Значительное число ошибок алгоритма объясняется малым размером набора регулярных выражений и тем, что этот набор не учитывает специфику задачи разметки библиографических записей.

Далее расширим набор регулярных выражений с учетом специфики задачи. В набор будут добавлены регулярные выражения, улучшающие выделение отдельных полей библиографических записей (табл. 2). Символом C обозначим некоторое число, а d — цифру.

Результат работы предложенного алгоритма с расширенным множеством регулярных выражений на той же коллекции библиографических строк в текстовом представлении сравнивался с результатом работы онлайн-программы восстановления структуры Brown University (http://freecite.library.brown.edu/).

Каждому полю библиографической записи на диаграмме рис. 3 (см. третью сторону обложки) соответствуют три столбца.

Высота синего столбца определяет долю записей из размеченной коллекции, попавших в кластер и относящихся к тому же типу поля библиографической записи, к которому отнесен и сам кластер для предложенного в работе алгоритма. Высоты зеленого и коричневого столбцов определяют то же для онлайн-программы, с которой происходит сравнение, и для алгоритма, предложенного в работе, при использовании исходного множества регулярных выражений соответственно.

Диаграмма демонстрирует значительное улучшение качества работы алгоритма после добавления новых регулярных выражений. Так строки, соответствующие полям "страницы" и "год", кластеризованы без ошибок. Улучшение качества для поля "город" можно объяснить улучшением общего качества кластеризации строк, соответствующих тем полям, с которыми и происходило смешивание строк, соответст-

вующих полю "город": "издательство", "название журнала". Отсутствие кластера, соответствующего редакторам, объясняется малочисленностью записей, содержащих это поле. Ошибки в полях "том" и "номер" объясняются наличием регулярных выражений, которые срабатывают и для поля "страницы", и для этих двух полей. Более аккуратный выбор регулярных выражений для указанных трех типов полей библиографической записи позволит решить и эту проблему. Полученные результаты сопоставимы с результатами онлайн-программы, что свидетельствует в пользу использованного подхода.

Заключение

В данной работе представлено решение задачи разметки библиографических данных. Использован метод метрической кластеризации, при котором расстояние вводится с помощью тупиковых матриц. В качестве признаков исходных объектов использовалось срабатывание регулярных выражений. В вычислительном эксперименте предложенный алгоритм тестировался на данных библиографических записей, часть которых была размечена. Полученные результаты говорят о применимости метода и указывают на возможные его модификации для улучшения разметки библиографических записей.

Список литературы

1. **Borkar V., Deshmukh K., Saravagi S.** Automatic segmentation of text into structured records // Proceedings of the 2001 ACM SIGMOD international conference on management of data. New York: ACM, 2001. Vol. 30. N 2. P. 175—186.

2. **Christen P., Churches T., Zhu J. X.** Probabilistic name and address cleaning and standardisation // Proc. of the Australasian data mining workshop, 3 December 2002, University House, ANU, Canberra. P. 99—108.

3. **Адуенко А. А., Кузьмин А. А., Стрижов В. В.** Выбор признаков и оптимизация метрики при кластеризации коллекции документов // Известия ТулГу. 2012. № 3. С. 119—131.

4. **Chapelle O., Schölkopf B., Zien A.** Semi-supervised learning. Cambridge: The MIT Press, 2006.

5. **Aliguliyev R. M.** Performance evaluation of density-based clustering methods // Information sciences. 2009. Vol. 179, N 20. P. 3583—3602.

6. **Gabrilovich E., Markovitch S.** Overcoming the brittleness bottleneck using wikipedia: enhancing text categorization with encyclopedic knowledge // Proceedings of Twenty-first national conference on artificial intelligence, Menlo Park CA. 2006. Vol. 21, N 2. P. 1301.

7. **Ахо А., Хопкрофт Дж., Ульман Дж.** Построение и анализ вычислительных алгоритмов. М.: Мир, 1979. С. 535.

8. **Zhao Y., Karypis G.** Criterion function for document clustering: experiments and analysis // Machine Learning. 2001. Vol. 55, N 3. P. 311—331.

9. **Дюкова Е. В., Инякин А. С.** Задача таксономии и тупиковые покрытия целочисленной матрицы. Сообщения по прикладной математике. М.: Вычислительный центр РАН, 2001.

14—17 октября 2013 г. в г. Ярославль состоится

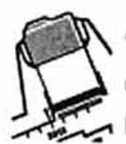
XV Всероссийская научная конференция RCDL'2013 "Электронные библиотеки: перспективные методы и технологии, электронные коллекции"

Серия Всероссийских научных конференций RCDL, труды которых представлены на сайте http://rcdl.ru, нацелена на формирование российского корпуса международного сообщества ученых, развивающих это научное направление, подробно описанное на сайте конференции.

Совместно с конференциями традиционно проводятся сопутствующие диссертационные семинары, на которых авторам работ, отобранных на основе предварительного рецензирования, предоставляется возможность изложить текущие результаты своих исследований, а также обсудить их сильные и слабые стороны с более опытными коллегами.

Труды конференции буду опубликованы в виде сборника текстов принятых полных статей, кратких статей и тезисов стендовых докладов, а также в электронном виде в европейском репозитории трудов конференций CEUR Workshop Proceedings. Лучшие статьи, представленные на конференцию, будут рекомендованы к публикации в изданиях, признанных ВАК, таких, как "Информатика и ее применения", "Программная инженерия", "Системы и средства информатики".

Подробности — на сайте конференции: http://rcdl2013.uniyar.ac.ru



2013 CEE-SEC(R) Разработка ПО

IX Международная научно-практическая конференция «Разработка ПО»

23-25 октября 2013. Москва. Центр Digital October

700+ делегатов из 250+ компаний и 20+ стран

Приз 1000 Евро за лучший исследовательский доклад

Бесплатное участие для докладчиков

Приглашаем докладчиков

Принимаются заявки на выступления по направлениям:

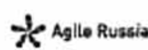
- **Технологии: исследования и практика**, в том числе:
  - Архитектура программных систем
  - Облачные вычисления
  - Технологии и средства программирования
  - Тестирование, верификация и анализ ПО
  - Big data, Smart data
  - Мобильные и кросс-платформенные приложения
  - Приложения для банков и финансовой индустрии
  - Usability и проектирование интерфейсов (UX)
- **Управление: процессы, ресурсы**
  - Управление проектами и продуктами
  - Гибкие методологии (Agile)
  - Управление качеством
  - Управление рисками
  - Человеческий капитал и образование
- **Бизнес и предпринимательство в индустрии разработки ПО**

Срок подачи заявок: 15 июля 2013

Подробнее: www.secr.ru, contact@secr.ru, +7 812 336 93 44

Спонсоры

Партнеры



Уважаемые коллеги!
Приглашаем Вас к авторскому участию в журнале "Программная инженерия"!

МАТЕРИАЛЫ, ПРЕДСТАВЛЯЕМЫЕ В РЕДАКЦИЮ

- Статья, оформленная в соответствии с требованиями
- Иллюстрации и перечень подрисуночных подписей
- ФИО авторов, название статьи, аннотация и ключевые слова на английском языке.
- Сведения об авторах (ФИО, ученая степень, место работы, занимаемая должность, контактные телефоны, e-mail)

ПОРЯДОК ОФОРМЛЕНИЯ ИНФОРМАЦИОННОЙ ЧАСТИ

- Индекс ***УДК*** размещается в левом верхнем углу первой страницы
- ***Сведения об авторах*** на русском языке размещаются *перед названием статьи* и включают инициалы и фамилии авторов с указанием их ученой степени, звания, должности и названия организации и места ее расположения (если это не следует из ее названия). Указывается также e-mail и/или почтовый адрес хотя бы одного автора или организации
- *За сведениями об авторах* следует ***название статьи***
- После названия статьи отдельным абзацем дается ***краткая аннотация***, отражающая содержание статьи (что в ней рассмотрено, приведено, обосновано, предложено и т. д.)
- Затем следуют ***ключевые слова***

ТЕКСТ СТАТЬИ

Статью рекомендуется разбить на разделы с названиями, отражающими их содержание. Рекомендуемый объем статьи 15 страниц текста, набранного на стандартных листах формата А4 с полями не менее 2,5 см шрифтом размером 14 pt с полуторным межстрочным интервалом с использованием компьютерного текстового редактора Word. В указанный объем статьи включаются приложения, список литературы, таблицы и рисунки. Страницы статьи должны быть ***пронумерованы***. В необходимых случаях по решению главного редактора или руководства издательства объем статьи может быть увеличен.

В формулах русские и греческие буквы следует набирать прямо; латинские буквы, обозначающие скаляры, *курсивом*; величины, обозначающие векторы и матрицы, должны быть выделены полужирным шрифтом и набраны прямо (допускается также набор всех величин, обозначенных латинскими буквами, в т.ч. матриц и векторов, светлым курсивом). Стандартные математические обозначения (например, log, sin и т. д.) должны быть набраны прямо.

Номера формул располагаются справа в круглых скобках. Нумерация формул — сквозная, причем нумеруются те формулы, на которые имеются ссылки. Рисунки должны быть выполнены качественно (графическая обработка рисунков в редакции не предполагается). В журнале все рисунки воспроизводятся в черно-белом варианте, за исключением цветных рисунков, размещаемых по усмотрению редакции на обложке.

Статья может быть отправлена по e-mail: prin@novtex.ru

Дополнительные пояснения авторы могут получить в редакции журнала лично, по телефонам: (499) 269-53-97, 269-55-10, либо по e-mail.

CONTENTS

Vyukova N. I., Galatenko V. A., Samborskij S. V. LLVM as an Infrastructure for Compiler Development for Embedded Systems 2

The paper presents an overview of the LLVM infrastructure and its evaluation with respect to the specificity of compiler development for microprocessors used in embedded systems. It discusses LLVM features from the user's point of view, the overall structure of LLVM, and internal representation of a program.

Keywords: compiler, LLVM, embedded systems, intermediate representation

Terekhov A. N., Bryksin T. A., Litvinov Y. V. QReal: Domain-Specific Visual Modeling Platform 11

Approach to a development of software, based on specialized visual languages is described in this article. Existing platforms for creation of specialized visual development environments are examined, QReal platform is described, which is developed on a software engineering chair of SPbSU, overview of its architecture and main functionality is given, examples of its practical use are presented.

Keywords: domain-specific modeling, visual programming, DSM platforms, MetaCASE-systems

Sheenok D. A. Tool Design the Optimal Architecture of Fault-Tolerant Software Systems 20

Considered the problem of planning the cost of achieving the required level of reliability of failover software. Substantiated the necessity of the use of specialized optimization algorithms for solving this problem because of the large number of alternative options for building the software architecture. Described the algorithms of a software system to solve this optimization problem using exhaustive search options, or by using a specialized genetic algorithm. Proposed an algorithm for the use of tools in the software life cycle.

Keywords: software architecture design, optimization, genetic algorithm

Baranov D. V. Computer Implementation of Redundant Rule Elimination in Conditional Rewrite Systems 27

This paper describes the implementation of the redundancy elimination algorithm. It is based on a new algebraic system with the semantics of a collection of rules of the conditional equational theory or conditional term rewriting system.

Keywords: terms, equational theory, conditional rules, logical reduction, computer implementation

Kharlamov A. A., Ermolenko T. V. Semantic Network of Subject Area as a Basis for Transition Network Compilation for Automatic Recognition of Key Words in Speech Flow 33

The task of automatic speech recognition has not decided successfully now yet. But products such companies like Nuance [1], Autonomy [2], and others are known now. The products are used in applications, and in Russian language ones. Hidden Markov Models approach is good for the automatic speech recognition. Mistake of the recognition system in low room in such a case become less than 15 %. There is one, but essential lack: the transition network between word models compiled by hand. But the process is very labor consuming.

Keywords: speech recognition, key words, transition network between word models, text analysis, semantic network

Ivanova A. V., Aduenko A. A., Strijov V. V. Algorithm of Construction Logical Rules for Text Segmentation .41

Consider the method of recovery of BibTeX-structure bibliographic records from their text representation. Structure is recovered using logical rules defined on an expert-given set of regular expressions. Algorithm based on stub covers is proposed for constructing the logic rules. The algorithm is illustrated with the problem of searching the structure in bibliographic records, represented by text strings.

Keywords: bibliographic record, BibTeX, regular expression, stub cover, clustering

ООО "Издательство "Новые технологии". 107076, Москва, Стромынский пер., 4

Дизайнер *Т. Н. Погорелова.* Технический редактор *Е. М. Патрушева.* Корректор *Е. В. Комиссарова*

Сдано в набор 02.04.2013 г. Подписано в печать 21.05.2013 г. Формат 60×88 1/8. Заказ PI613
Цена свободная.

Оригинал-макет ООО "Адвансед солюшнз". Отпечатано в ООО "Адвансед солюшнз".
105120, г. Москва, ул. Нижняя Сыромятническая, д. 5/7, стр. 2, офис 2.