

Программная инженерия

Пр 7
2012
ИН

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.
Назирев Р.Р., д.т.н.
Нечаев В.В., к.т.н.
Новиков Е.С., д.т.н.
Норенков И.П., 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

Галатенко В. А. Категорирование и разделение программ и данных как принцип архитектурной безопасности	2
Разумовский А. Г., Пантелеев М. Г. Валидация объектно-ориентированных программ с использованием онтологий	7
Филаретов В. Ф., Юхимец Д. А., Мурсалимов Э. Ш. Создание универсальной архитектуры распределенного программного обеспечения мехатронного объекта	14
Александров А. Е., Востриков А. А., Шильманов В. П. Принципы организации архитектуры программной системы "Прогноз" на основе библиотеки компонентов	22
Попов С. Е., Замараев Р. Ю. Программный комплекс и язык метаописаний алгоритмов анализа социально-экономических объектов	27
Лихачёв В. Н., Гинзгеймер С. А. Обработка ошибок ограничений для баз данных PostgreSQL	34
Ерофеев Е. В., Пелепелин И. Е. Анализ производительности протокола CMIS на примере его реализации в Alfresco и IBM FileNet	42
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", 2012

Категорирование и разделение программ и данных как принцип архитектурной безопасности

Рассматриваются архитектурные аспекты обеспечения информационной безопасности. В качестве одного из принципов архитектурной безопасности предлагается категорирование и разделение программ и данных. Это особенно важно при разработке и реализации программных средств обеспечения информационной безопасности, способных противостоять деструктивным воздействиям.

Ключевые слова: компьютерная безопасность, архитектурные принципы безопасности, разделение программ и данных

Введение

Для создания эффективного аппаратно-программного обеспечения информационной безопасности (ИБ) любой системы необходимо сочетание мер следующих трех типов (см., например, [1]):

- недопущение определенных, характерных для этой системы видов нарушений ИБ;
- оперативное выявление факторов нарушений ИБ;
- реагирование на нарушения ИБ, включая ликвидацию последствий.

Предпочтительными в этом перечне являются меры первого типа, поэтому системную архитектуру следует выбирать так, чтобы они были реализованы в максимальной степени. Основной архитектурный принцип, направленный на достижение этой цели, — разделение сложной системы на подобъекты с установлением физических границ между ними.

Данный принцип допускает многочисленные реализации, например:

- разделение сетевых потоков данных на пользовательские, административные и связанные с поддержанием ИБ, а также направление этих потоков по физически различным каналам;
- разнесение приложений по физически различным серверам;
- статическое распределение процессорных ядер между программными потоками при отсутствии разделяемой памяти;

- установление отношений недоверия между компонентами систем, а также ряд других.

Ниже будут рассмотрены наиболее перспективные из возможных реализаций.

Категорирование и разделение данных

Под категорированием понимается классификация данных по каким-либо признакам, например, по степени изменчивости (статические и динамические данные), по степени критичности для корректного функционирования систем (пользовательские и служебные данные) и т.п. Данные, отнесенные к различным категориям, целесообразно физически разделить, чтобы там, где это возможно, воспользоваться естественной защитой или предотвратить несанкционированную модификацию критически важных данных.

Статические и динамические данные. Категорирование и разделение данных поддерживается средствами управления доступом. Однако если проводить категорирование и разделение данных на архитектурном уровне, можно добиться более высокого уровня защищенности, сделав некоторые виды нарушений ИБ невозможными.

Прежде всего данные можно классифицировать как статические и динамические. Под статическими понимаются данные, которые после начальной загрузки остаются неизменными в процессе функционирования системы. Изменяемые данные называются динамическими. Статические данные целесообразно

разместить на носителях, допускающих только чтение, но не запись. Это может быть постоянное запоминающее устройство (ПЗУ) или компакт-диск (КД) ПЗУ. Изменить содержимое ПЗУ можно, если располагать физическим доступом к нему и использовать специальную аппаратуру.

В ПЗУ целесообразно размещать статические компоненты операционной системы (программы и данные), а также другие (прикладные) статические компоненты встроенных систем. На КД ПЗУ могут располагаться статические компоненты информационного наполнения web-серверов.

Одним из архитектурных принципов ИБ является невозможность обхода защитных средств. Применительно к разделению данных выполнение этого принципа особенно важно, поскольку современные аппаратно-программные комплексы содержат многочисленные кэши на различных уровнях, что делает возможными атаки типа "отравление кэша". К подобным атакам можно отнести компрометацию разного рода серверов-посредников.

Еще одна возможная разновидность атак — перехват и перенаправление потоков данных. Эта атака может осуществляться на разных уровнях — от системной шины до сервера доменной системы имен (DNS). В первом случае, очевидно, необходим физический доступ к системе; DNS-серверы более уязвимы, в том числе удаленно, в частности, путем атак типа "отравление кэша". Кроме того, дополнительную уязвимость создает само наличие вспомогательной фазы трансляции имен в адреса.

Пользовательские и служебные данные. При выполнении пользовательских приложений используются служебные данные и программы. В среде времени выполнения поддерживаются: стек вызовов; динамически выделяемая память; динамически загружаемые библиотеки и т. п. Это означает, что в такую среду входят не только написанные пользователем функции и фигурирующие в его программе переменные, но и служебные данные (например, такие как адреса возврата, указатели на элементы списка динамической памяти и подобные им), а также функции (например, библиотечные).

Одной из самых эксплуатируемых уязвимостей программ является переполнение буфера (целенаправленный выход за границы массива при отсутствии соответствующего контроля). Чаще всего целью переполнения служит несанкционированное изменение системных данных. Например, при атаке типа "смазывание стека", как правило, изменяют адрес возврата из функции или адрес фрагмента стека. При переполнении буфера в динамически выделяемой памяти несанкционированно изменяют указатели, связывающие между собой элементы списка динамической памяти.

Сделать невозможными успешные атаки типа "переполнение буфера" можно двумя способами:

— контролировать размер значений переменных, не допускать выход за границы массивов;

— отводить под значения-массивы расширяемые сегменты памяти таким образом, что при переполнении буфера исключительные ситуации не возникают, но значения других переменных при этом не затираются.

Если массив является элементом структуры, то в отдельный сегмент нужно выносить всю структуру. При переполнении буфера могут пострадать другие поля этой структуры.

В процессе динамического выделения памяти можно помещать каждый фрагмент в отдельный сегмент (расширяемый или с фиксированными границами) так же, как это предлагалось делать для массивов.

Разумеется, размещать каждый массив или фрагмент динамической памяти в отдельном сегменте слишком накладно. На практике разумным компромиссом является разделение пользовательских и служебных данных так, чтобы последние не пострадали при переполнении пользовательского буфера.

Проиллюстрируем это разделение на примере стека и области для выделения динамической памяти.

В традиционных системах программирования кадр стека вызовов выглядит примерно так, как показано на рис. 1. Предполагается, что стек растет в направлении уменьшения адресов памяти.

Видно, что в стеке традиционных систем программирования пользовательские и служебные данные чередуются. Между ними нет контролируемой (не говоря уже о физической) границы. В силу этих причин переполнение пользовательского буфера способно привести к сколь угодно серьезным последствиям.

Предлагается параллельно поддерживать два стека — пользовательский и служебный, разместив их в разных сегментах памяти (рис. 2). В таком случае переполнение пользовательского буфера не способно при-



Рис. 1. Кадр стека вызовов в традиционных системах программирования



Рис. 2. Разделенные служебный и пользовательский стеки вызовов

вести к несанкционированному изменению служебных данных.

Естественно, разделение пользовательского и служебного стеков вызовов должно поддерживаться соответствующим расширением регистрового файла, появлением двух наборов регистров, обслуживающих стеки.

Аналогичная по сути ситуация имеет место в традиционных подсистемах динамического распределения памяти. Обычно элемент списка динамической памяти выглядит примерно так, как показано на рис. 3.

При такой организации в простейшем случае переполнение буфера может привести к несанкционированному изменению служебных данных в следующих фрагментах динамически выделяемой памяти, имеющих большие адреса, с целью вызвать нужные атакующему эффекты при последующих операциях со ссылками.

Если разнести служебные и пользовательские данные по разным сегментам, картина может получиться такой, как показано на рис. 4.

Отметим попутно, что при подобной организации элемент служебных данных, соответствующий одному выделенному пользователю фрагменту динамической памяти, имеет фиксированный размер, так что в спи-



Рис. 3. Традиционная структура элемента списка динамической памяти

сочной структуре для служебных данных нет необходимости, и поля ссылок становятся ненужными.

Необходимым условием корректного разделения служебных и пользовательских данных является ограничение на направление ссылок: не может быть ссылок из пользовательских данных на служебные.

Возможные обобщения. Очевидно, служебные данные занимают промежуточное положение между пользовательскими данными и структурами ядра операционной системы (ОС). Подобные данные являются частью пользовательского процесса, однако они не должны создаваться и изменяться произвольными (низкоуровневыми) инструкциями.

К ним применим фиксированный набор (относительно высокоуровневых) операций, определяемых системой программирования (точнее, ее частью, обеспечивающей поддержку времени выполнения).

Из истории развития ОС известно понятие кольца защиты, впервые появившееся в революционной ОС Multics (см., например, [2]). Под кольцами защиты понимается набор вложенных областей (уровней привилегий), ограничивающих возможности процессов в диапазоне от пользовательских программ до супервизора. Таких колец может быть несколько, с разными уровнями привилегированности. В каждый момент времени пользовательский процесс функционирует в рамках одного из этих колец, где ему разрешено выполнение определенного набора операций. При переходе из одного кольца в другое происходит переключе-

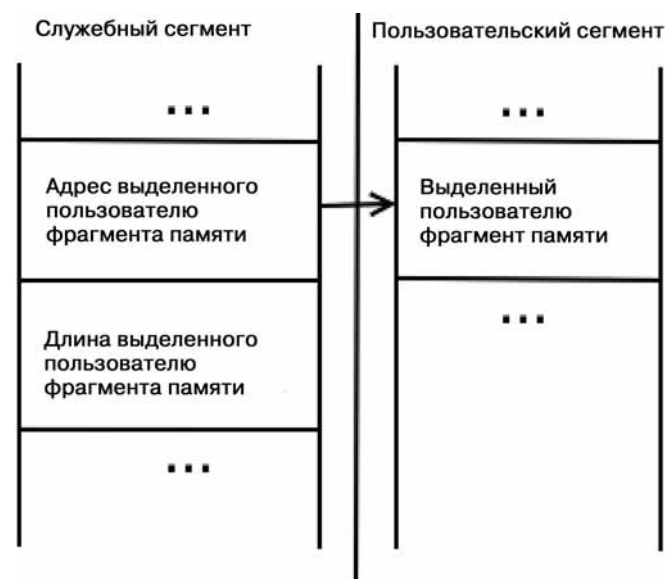


Рис. 4. Структуры элементов в подсистеме динамического распределения памяти при разделении пользовательских и служебных данных

чение контекстов — действие относительно дорогостоящее, так что число подобных действий желательно минимизировать. По этой причине многие встроенные системы, ресурсы которых ограничены, функционируют на одном (самом привилегированном) уровне, что повышает эффективность, но снижает надежность программного обеспечения.

Особенность разделения служебных и пользовательских данных состоит в том, что среди потока команд соответствующего разделения не осуществляется. Для поддержки такого разделения необходимо удвоение некоторых аппаратных компонентов (например, регистров, поддерживающих стек), а также умение определять, с какими данными (служебными или пользовательскими) работает та или иная инструкция. Иногда это очевидно из контекста. Например, инструкции перехода с возвратом и возврата из функции сохраняют счетчик команд в служебном стеке и восстанавливают его из этого же стека. Инструкции сохранения и восстановления регистров, вообще говоря, такой контекстной информации не содержат, они могут использоваться с разными целями, не только в прологе/эпilogue функций, но и по ходу их выполнения (например, для создания временных переменных в силу отсутствия свободных регистров). В таком случае в инструкции требуется явное указание, с данными какой категории эта инструкция работает.

Идеи разделения данных и колец защиты можно скомбинировать, допустив параллельную работу в нескольких контекстах в рамках одного или нескольких потоков команд одного пользовательского процесса. При подобной дисциплине выполнения отпадает необходимость в дорогостоящих переключениях контекстов, однако необходимым условием поддержки такой параллельной работы является размножение некоторых аппаратных компонентов (в первую очередь регистров).

Наличие в одном потоке команд инструкций нескольких категорий, относящихся к разным контекстам, требует поддержки доверенной компиляции и загрузки, а также верификации исполняемого кода (по аналогии с верификацией динамически загружаемого байт-кода).

Если отвлечься от переключения контекстов, то наличие в пользовательском потоке команд привилегированных инструкций можно рассматривать как результат стандартной оптимизации — inline-подстановки тела вызываемых функций, а реализация совокупности системных вызовов становится эквивалентной библиотеке функций.

Допускает обобщение и ограничение на направление ссылок между данными разных категорий: ссылки возможны только из более привилегированных данных в менее привилегированные (из системных в служебные и пользовательские). Если использовать традиционную для информационной безопасности терминологию, должно обеспечиваться "невлияние" менее привилегированных кода и данных на более привилегированные.

Достоинство предлагаемой архитектуры безопасности состоит в том, что она требует только ограниченного размножения аппаратных компонентов. В исходных текстах программ, написанных на языках высокого уровня, никаких изменений не предполагается.

Отметим, что безопасность стандартной современной архитектуры, основанной на переключениях контекстов ядра ОС и пользователя, на проверку оказывается мнимой. Свидетельство тому — наличие и успешное распространение такой разновидности вредоносного программного обеспечения, как руткиты, а также возможность реализации вредоносных свертонких гипервизоров, которые не могут обнаружить даже доверенные платформенные модули (см., например, [3]).

Категорирование и разделение программ

Основное препятствие в обеспечении реальной ИБ — беспрецедентная сложность программного обеспечения (см., например, [4]). На сегодняшний день единственным апробированным средством преодоления этого препятствия является объектно-ориентированный подход, цель следования которому — выстроить многоуровневую архитектуру системы с небольшим числом сущностей на каждом уровне и с регулярной структурой связей между сущностями.

Понижение сложности особенно важно для доверенной вычислительной базы (ДВБ), от корректности архитектуры и реализации которой зависит ИБ всей системы. Категорирование и разделение программ позволяет сузить функциональность программных компонентов, сделав их более простыми и, как следствие, верифицируемыми.

Среди компонентов доверенной вычислительной базы ключевую роль играет монитор обращений. Его необходимо выделять в отдельную сущность, в противном случае корректность всей ДВБ оказывается проблематичной.

Применительно к технологии виртуализации этот вопрос исследуется в работе [5]. В контексте виртуальных машин и гипервизоров не разделяются две роли: монитора обращений и эмулятора ресурсов. Это ведет как к большим накладным расходам и снижению производительности, так и к чрезмерной сложности и потенциальной уязвимости монитора обращений. Монитор обращений должен интерпретировать политику безопасности, проводить ее в жизнь, а не заниматься эмуляцией запрашиваемых ресурсов.

Везде, где возможно, разделенные программные сущности целесообразно статически загружать на различные процессоры (процессорные ядра), а для взаимодействия между ними использовать сетевые механизмы, такие как удаленный вызов процедур. Потенциально это сулит выигрыш не только в безопасности, но и в производительности, так как позволяет распараллелить работу системы.

Категорирование и разделение сетей

Естественным следствием и обобщением разделения пользовательских и служебных данных представляется физическое разделение в компьютерных сетях (по крайней мере в локальных) пользовательских и служебных потоков данных. Физическое разделение подразумевает использование для потоков данных разных категорий различных каналов передачи данных (разных подсетей) и, соответственно, сетевых портов (см., например, [4]).

Развивая идеи, изложенные в работе [6], можно предложить разделение следующих категорий сетевых потоков данных:

- пользовательские (прикладные) данные;
- регистрационная информация;
- потоки данных, возникающие при удаленном администрировании систем и сетей;
- потоки данных, индуцируемые действиями администраторов безопасности.

Важным представляется выделение в отдельную категорию потоков регистрационной информации. Это однонаправленные потоки (от клиентов к серверам протоколирования), формат передаваемых данных известен заранее, поэтому здесь особенно просто выявлять подозрительную активность, не допускать несанкционированной передачи данных.

Заключение

Категорирование и разделение программ и данных — это важный принцип и мощный инструмент архитек-

турной безопасности. Главные цели применения этого принципа (равно как и инструмента) — недопущение определенных видов нарушений ИБ, упрощение архитектуры, понижение сложности систем и, как следствие, повышение уровня их реальной информационной безопасности.

Список литературы

1. **Галатенко В. А.** Основы информационной безопасности / Под ред. В. Б. Бетелина. М.: ИНТУИТ.РУ, 2003. 280 с.
2. **Karger P., Schell R.** Thirty years later: Lessons from the multics security evaluation // Proc. of the 18th Annual Computer Security Application Conference (ACSAC 2002). December 2002. Las Vegas, USA: IEEE, 2002.
3. **Галатенко В. А.** Анализ эффективности архитектур безопасности при использовании технологии виртуализации // Моделирование и визуализация. Многопроцессорные системы. Инструментальные средства разработки ПО / Под ред. В. Б. Бетелина. М.: НИИСИ, 2010. С. 161—175.
4. **Галатенко В. А.** Обзор и анализ архитектурных методов обеспечения информационной безопасности // Информационная безопасность / Под ред. В. Б. Бетелина М.: НИИСИ, 2009. 71 с.
5. **Bratus S., Locasto M. E., Ramaswamy A., Smith S. W.** Traps, Events, Emulation and Enforcement: Managing the Yin and Yang of Virtualization-based Security // Proc. of the International Conference VMSEC'08. Florida, Virginia, USA: ACM Press, 2008. P. 49—58.
6. **Williams P., Anchor K., Bebo J., Gunsch G., Lamont G.** CDIS: Towards a Computer Immune System for Detecting Network Intrusion // Proc. of the 4th International Symposium, Recent Advances in Intrusion Detection, Berlin: Springer-Verlag, 2001. P. 117—133.

ИНФОРМАЦИЯ

Специализированная выставка

СВЯЗЬ. ИНФОРМАЦИЯ. ИТ-ТЕХНОЛОГИИ

4—6 декабря 2012 г.

г. Екатеринбург, МВЦ "Екатеринбург ЭКСПО" (Бульвар-ЭКСПО, 2)

Цель выставки: демонстрация новейших технологий и услуг рынка ИТ и связи, продвижение успешно действующих инновационных программ и проектов информатизации, распространение возможностей их применения во всех сферах бизнеса, государственного управления и общественной жизни.

Выставка охватывает весь спектр новаций в сфере связи и информационных технологий, представляя программное и связанное обеспечение, сервисные компании и консультационные услуги, обеспечивая возможность общения специалистов в области информационных и коммуникационных ноу-хау.

Контакты: тел: (343) 3-100-330, E-mail: postovalova@uv66.ru
http://www.uv66.ru/vystavka/ekb_vystavka/2012/protec_2012_info/

А.Г. Разумовский, аспирант, e-mail: razumovsky.andrey@gmail.com,
М.Г. Пантелеев, канд. техн. наук, доц.,
Санкт-Петербургский государственный электротехнический университет ("ЛЭТИ")

Валидация объектно-ориентированных программ с использованием онтологий

Предлагается новый подход к валидации объектно-ориентированного программного обеспечения, основанный на использовании онтологий предметных областей, представленных на языке OWL, и средств автоматического вывода. Представлена общая архитектура системы семантической валидации. Рассмотрен прототип системы, предназначенной для валидации объектно-ориентированных программ на языке Java с использованием разработанной библиотеки Eriphron.

Ключевые слова: разработка программного обеспечения, объектно-ориентированное программирование, валидация, онтологии

Введение

Постоянный рост сложности программных комплексов приводит к необходимости развивать методы автоматической валидации и верификации программного обеспечения ПО [1, 2]. В современной практике программирования ведущее место занял объектно-ориентированный (ОО) подход, достоинствами которого являются высокая степень абстракции, естественная поддержка декомпозиции приложений и, как следствие, сокращение числа ошибок при создании сложных программных комплексов. В ОО-программировании известны подходы к верификации и валидации, основанные на формальных моделях [3, 4], позволяющие явно контролировать корректность программы. Однако такие модели ориентированы на поддержку конкретных методов верификации и этапов жизненного цикла ПО.

Объектно-ориентированное программирование естественным образом поддерживает моделирование классов и отношений предметной области. При этом многие важные ограничения предметной области могут быть выражены только явно, посредством специального программного кода. Примерами подобных ограничений являются свойства кардинальности, симметричности и транзитивности отношений. Такой

подход к программированию ограничений предметной области имеет очевидные недостатки:

- дублирование — повторное объявление ограничений, уже специфицированных на этапах анализа и проектирования ПО;
- повышение сложности контроля соблюдения ограничений с ростом их сложности, так как это требует добавления к реализации классов кода, контролирующего ограничения этих классов;
- подверженность программного кода ошибкам, как следствие отсутствия гарантий соблюдения ограничений на уровне формальной логики.

Вместе с тем присущие различным предметным областям (доменам) ограничения могут быть естественным образом специфицированы с использованием языков онтологий (в частности, OWL [5]), получивших в последние годы значительное развитие и распространение. Выразительные возможности онтологий могут быть расширены за счет использования сложных правил, представленных на языке SWRL [6]. Это обстоятельство создает принципиальную возможность использования для валидации ОО-программ доменных онтологий в качестве "справочников", содержащих ограничения и правила, присущие предметным областям. Реализация такой возможности предполагает наличие эффективных механизмов доступа к онтологиям из программного кода.

Онтологии в разработке ПО

Использование онтологий в разработке ПО является естественным шагом на пути повышения уровня абстракции и обеспечивает ряд преимуществ, в частности [7]:

- сокращение "концептуального разрыва" (*conceptual gap*), так как разработчики ПО получают средства, более близкие человеческому способу мышления;
- упрощение повторного использования знаний, представленных ранее определенными в различных онтологиях понятиями и отношениями;
- повышение уровня автоматизации за счет возможности автоматического логического вывода над содержащимися в онтологиях формальными знаниями;
- повышение надежности и сокращение стоимости сопровождения ПО за счет использования формальной логики.

Указанные преимущества обусловили в последние годы активное развитие методов разработки ПО с использованием онтологического подхода в рамках направления *Ontology-Driven Software Engineering (ODSE)*. Данное направление предполагает использование онтологий на всех основных этапах разработки ПО — анализа, проектирования и реализации [8]. При этом однажды созданная на этапах анализа и проектирования онтология предметной области может использоваться также и на этапе реализации для дополнения семантики ОО-языков различными ограничениями предметной области, соответствующей конкретному приложению.

Таким образом, онтологические базы знаний можно использовать в комбинации с ОО-средствами программирования в качестве декларативно заданных спецификаций ограничений предметной области. Реализация такой возможности предполагает динамическое отображение объектов приложения (экземпляров программных классов) в экземпляры соответствующих классов онтологии с последующей валидацией и логическими рассуждениями над ними стандартными средствами поддержки логического вывода над онтологиями.

В настоящей статье предложен подход к такому использованию онтологий предметной области и рассмотрен прототип программного средства Epirhgon, реализующего предложенный подход для программ в отдельной предметной области, которые исполняются на языке Java. Выбор языка Java обусловлен его популярностью и развитой поддержкой работы с онтологиями. Вместе с тем предлагаемый подход универсален и может быть реализован применительно к любым ОО-языкам программирования.

Связывание конструкций ОО-языков и онтологий

Основные конструкции онтологий и ОО-языков сходны. В обоих подходах применительно к предметной области выделяют классы, связанные различными отношениями (в частности, отношением наследования). Классы имеют атрибуты, представимые базо-

выми типами данных. При этом базовым типам данных (*int*, *float*, *double*, *boolean*) и близким к ним классам (*String*, *Date* и подобных им) ОО-языков можно сопоставить встроенные типы данных онтологий.

Вместе с тем корректное отображение одних конструкций на другие представляет значительную сложность. В частности, онтологии содержат большое число семантических ограничений, которые нельзя выразить синтаксическими средствами ОО-языков. К ним относятся свойства транзитивности, рефлексивности и кардинальности отношений. Кроме того, в наиболее распространенных современных ОО-языках программирования, таких как *C#* и *Java*, отсутствует понятие множественного наследования, присущего онтологиям. Однако в ряде случаев это ограничение можно обойти, используя, например, интерфейсы [9]. В настоящее время известны три принципиально разных подхода к заданию соответствия ОО-конструкций и онтологий [10].

Прямой (*direct*) подход предполагает прямое соответствие класса онтологии классу или интерфейсу ОО-языка. При этом отношение наследования онтологических классов эквивалентно наследованию интерфейсов или классов в ОО-языке. К достоинствам данного подхода следует отнести наличие соответствующего предметной области API (*Application Programming Interface*, программного интерфейса) и наличие механизмов обеспечения безопасности типов (*type safety*) при работе с API предметной области. Однако, как было отмечено выше, данное соответствие ограничено возможностями ОО-языка, поэтому область его применения ограничена только простыми онтологиями либо достаточно простыми частями более сложных онтологий.

При **непрямом (*indirect*) подходе** для работы с любыми онтологиями используется унифицированный программный интерфейс, не зависящий от конкретной предметной области. Структуры онтологий представляются объектами метаклассов, такими как *OWLClass* и *OWLIndividual*. Наиболее развитыми в данном направлении средствами являются *Jena*, *Sesame*, *OWL API* (все — для языка *Java*). Унифицированный программный интерфейс поддерживает все конструкции языка *OWL* и обеспечивает гибкость выполнения программы. Однако при этом теряется возможность поддержки механизмов обеспечения безопасности типов и эффективного восприятия программного кода.

В настоящее время активно развивается **гибридный (*hybrid*) подход**, комбинирующий достоинства двух рассмотренных выше подходов. Его основная идея — прямое отображение основных и наиболее часто используемых концептов наряду с использованием унифицированного API для остальных [11]. Таким образом обеспечивается как безопасность типов, так и гибкость разработки приложения. Примерами такого подхода являются проекты *Mooor* [12] и *Sapphire*. Недостатком гибридного подхода является снижение производительности, вызванное частыми преобразованиями данных из доменно-ориентированного API в унифицированный.

Ограничения предметной области автомобилей, специфицированные в онтологии

Ограничение	Конструкции OWL
Спортивный автомобиль имеет мощность ≥ 150 л.с.	MinInclusive
Автомобиль не может быть быстрее себя	IrreflexiveProperty
Если автомобиль А быстрее автомобиля В, то В не может быть быстрее А	AsymmetricProperty
Гоночный автомобиль имеет мощность >200 л.с. и объем двигателя >3 л	Min, IntersectionOf
Автомобиль имеет не более пяти дверей	MaxCardinality
Автомобиль имеет не менее двух дверей	MinCardinality
Купе имеет ровно две двери	Cardinality
Хэтчбек имеет три или пять дверей	UnionOf

В связи с изложенным выше следует отметить, что гибридный подход является наиболее эффективным. Однако он также как и непрямой подход применим лишь в случаях, когда приложение непосредственно основано на онтологиях и использует их для реализации основных функций.

В настоящее время большинство бизнес-приложений на этапе выполнения программы используют объекты, а в качестве постоянного хранилища данных — реляционные базы данных. Автоматическая конвертация данных выполняется с использованием одной из реализаций технологии ORM (*Object-Relationship Mapping*), например NHibernate/Hibernate, Cayenne. Отметим, что приложения с традиционной архитектурой также могут выиграть от наличия онтологии предметной области. Задание соответствия объектных конструкций элементам онтологии позволило бы выполнять валидацию объектов на основе онтологии предметной области. При этом также стала бы возможна автоматическая реализация логического вывода над данными, представленными объектами. При мерами применения такого подхода являются:

- проверка семантической корректности части программного кода в Unit-тестах;
- проверка семантической корректности данных, записываемых в БД;
- семантическая валидация пользовательских форм;
- автозаполнение графа объектов с использованием свойств рефлексивности, симметричности, транзитивности.

Наиболее эффективно предлагаемый метод может быть применен в классических бизнес-приложениях, не использующих онтологию в качестве хранилища данных. Вместо этого для адаптации онтологий к существующим подходам предлагается использовать их в качестве "справочника" предметной области, содержащего данные о классах, отношениях и, возможно, некоторых индивидуумах, а также их ограничениях. В таком случае целесообразно использовать прямой подход. Несмотря на ограниченную выразительность, он в меньшей степени влияет на стиль написания программы, что существенно для приложений, в которых онтологии не являются единственной ключевой технологией описания их семантики.

Использование онтологий для валидации и логического вывода над программными объектами

Подход к валидации ограничений с помощью онтологий рассмотрим на примере предметной области, которая связана с автомобилями. Пусть имеется два базовых класса — автомобиль (Car) и дверь (Door), для которых (а также их подклассов) в онтологии определены ограничения, не заданные в конструкциях ОО-языков программирования. Примеры таких ограничений приведены в табл. 1.

При отсутствии специально запрограммированных в приложении ограничений на основе определения классов может быть создан объект *sportCar* класса *SportCar* со значением атрибута *power=100*. Однако с точки зрения семантики предметной области такой объект некорректен, так как нарушает ограничение предметной области, в соответствии с которым мощность спортивного автомобиля не может быть меньше 150 л.с. Объект с данными свойствами может существовать в определенный момент выполнения программы, но при записи в БД такие данные будут некорректны.

В некоторых случаях можно выделять группу объектов, являющихся семантически корректными в отдельности, но нарушающих некоторое ограничение на связывающее их отношение. Пусть, например, имеется два объекта *car1* и *car2*, причем *car2* находится в списке *fasterCars* объекта *car1* (т.е. *car2* быстрее, чем *car1*), а *car1* — в списке *fasterCars* объекта *car2*. При этом, очевидно, нарушена асимметричность свойства *fasterThan*. Аналогично, некорректным будет автомобиль *car*, связанный с одним или семью объектами класса *Door* (имеющий одну или семь дверей).

Контроль рассмотренных ограничений, безусловно, может быть реализован средствами ОО-языка. Однако по мере роста сложности ограничений такой подход становится чрезвычайно трудоемким, поскольку в ОО-языках отсутствует встроенная поддержка логического вывода. По этой причине онтологии целесообразно использовать в качестве базы знаний об ограничениях предметной области совместно с модулем логического вывода (*reasoner*) для рассуждений над данными и проверки соблюдения ограничений. Использование онтологий явно гарантирует семантическую корректность данных, синтезированных программным приложением.

Соответствие между ОО-конструкциями и онтологиями можно также использовать для выполнения автоматических логических выводов над объектами.

Таблица 2

Примеры свойств отношений предметной области автомобилей в онтологии

Описание свойства	Конструкция OWL
Если автомобиль А быстрее автомобиля В, а В быстрее С, то А быстрее С	TransitiveProperty
Если у А та же марка, что и В, то у В та же марка, что и у А	SymmetricProperty

В табл. 2 приведены примеры свойств объектов предметной области автомобилей, которые могут быть специфицированы в онтологии.

Пусть, например, в приложении создается три объекта — *car1*, *car2* и *car3*, причем *car2* и *car3* находятся в списках *fasterCars* объектов *car1* и *car2* соответственно (*car2* быстрее *car1*, а *car3* быстрее *car2*). Основанный на онтологии логический вывод установит, что *car3* быстрее *car1* и добавит *car3* в список *fasterCars* объекта *car1*.

На рисунке представлена общая архитектура системы семантической валидации, иллюстрирующая подход к валидации и логическим выводам над объектами.

Входными данными процесса валидации являются "схема" онтологии предметной области и набор объектов программного приложения. "Схема" онтологии (*TBox*) содержит сведения о классах предметной области и отношениях между ними. Часть онтологии, содержащую сведения об индивидуумах, принято называть *ABox*. Онтология должна быть семантически корректной, а именно — не содержать противоречий.

Объекты данных представляют собой экземпляры классов ОО-языка программирования, соответствующих некоторым классам предметной области. Предполагается, что классы содержат методы *get* и *set* для доступа к атрибутам.

Для выполнения валидации объекты данных преобразуются в индивидуумы онтологии предметной области. При отсутствии возможности выполнить прямое преобразование (например, вследствие различия имен программных классов и их атрибутов от

имен соответствующих им сущностей онтологии), необходимо использовать специальные метаданные, характерные для языка программирования.

На следующем этапе над заполненной онтологией выполняется логический вывод. Для этого можно использовать доступные модули вывода (*reasoner*), например Pellet [13]. Результатом данного этапа является фрагмент онтологии, содержащий выведенные утверждения.

Если модуль вывода не смог реализовать вывод вследствие некорректности онтологии, то работа алгоритма валидации прекращается и фиксируется семантическая некорректность исходных объектов данных для рассматриваемой предметной области.

В случае успешной валидации выполняется обратное преобразование — выведенные в онтологии данные преобразуются в программные объекты. Для этого выведенный фрагмент онтологии анализируется на наличие синтезированных знаний. Если эти знания представляют новые объектные свойства или свойства данных, то они добавляются в исходные программные объекты.

В результате над исходными объектами данных будут выполнены валидация и логический вывод, а синтезированные знания добавлены в объекты программного кода. Онтология при этом играет роль "справочника", содержащего информацию об ограничениях предметной области.

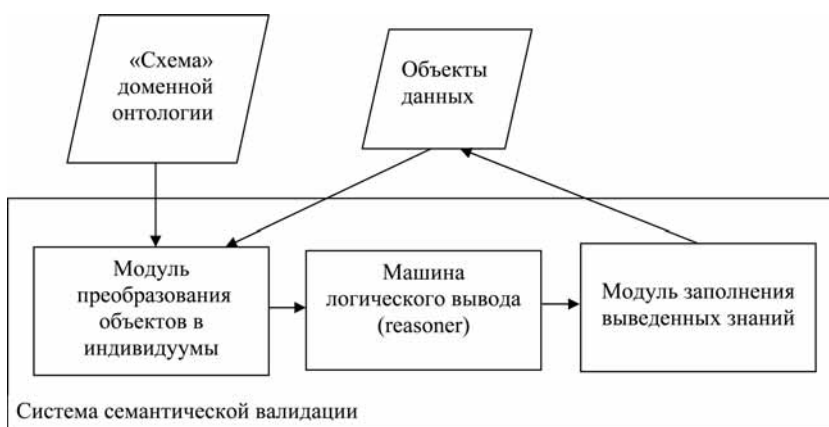
Реализация данного подхода осложняется тем обстоятельством, что онтологии в общем случае (по умолчанию) не поддерживают два важных допущения, относительно содержащихся в них знаний:

- уникальность имен (*Unique Name Assumption* — *UNA*);
- замкнутость мира (*Closed World Assumption* — *CWA*) [12].

Первое из этих допущений (*UNA*) предполагает, что два объекта онтологии, имеющие различные имена, являются различными (если явно не утверждается обратное). Поскольку классы предметной области, как правило, имеют уникальный идентификатор, во многих случаях валидация может быть выполнена с данным допущением.

Допущение о замкнутости мира (*CWA*) является более строгим. Оно предполагает ложность всех фактов, не выводимых из содержащихся в онтологии знаний. Таким образом, описываемый онтологией "мир" является замкнутым. Однако для проведения валидации ограниченного набора объектов данное допущение является уместным.

В отсутствие допущений *UNA* и *CWA* некоторые из ограничений, представленных в табл. 1, такие как ограничения кардинальности, не будут восприняты модулем вывода как ошибки. В зависимости от специфики предметной области и конкретного набора объектов в приложении для валидации и логических выводов мож-



Архитектура системы семантической валидации

Результаты применения подхода с уровнями валидации к различным ограничениям предметной области

Ограничение	Пример нарушения	Plain	UNA	UNA + CWA
Спортивный автомобиль имеет мощность ≥ 150 л.с.	Объект SportCar с power=100	ERROR	ERROR	ERROR
Автомобиль не может быть быстрее себя	Объект Car с самим собой в списке slowerThan	ERROR	ERROR	ERROR
Если автомобиль А быстрее автомобиля В, то В не может быть быстрее А	А в списке slowerCars В, и наоборот	ERROR	ERROR	ERROR
Гоночный автомобиль имеет мощность > 200 л.с. и объем двигателя > 3 л	Объект RaceCar с power=250 и engineSize=2.5	ERROR	ERROR	ERROR
У автомобиля не более пяти дверей	Объект Car с восемью связанными объектами Door	OK	ERROR	ERROR
У купе ровно две двери	Объект Coupe с 1/3 связанными объектами Door	OK	OK/ERROR	ERROR
У хэтчбека три или пять дверей	Объект Hatchback с 4/6 связанными объектами Door	OK	OK/ERROR	ERROR
У автомобиля не менее двух дверей	Объект Car с одним связанным объектом Door	OK	OK	ERROR
<i>Если автомобиль А быстрее автомобиля В, а В быстрее С, то А быстрее С</i>	<i>А в списке slowerCars В, В в списке slowerCars С</i>	<i>OK</i>	<i>OK</i>	<i>ERROR</i>
<i>Если у автомобиля А та же марка, что и В, то у В та же марка, что и у А</i>	<i>А в списке sameBrandCars В, но не наоборот</i>	<i>OK</i>	<i>OK</i>	<i>ERROR</i>

но использовать различные комбинации рассмотренных выше допущений. В связи с этим обстоятельством введем понятие **уровня валидации**, отражающее степень строгости валидации заполненной онтологии, обусловленную принятыми допущениями.

Выделим следующие уровни валидации:

- Plain — стандартные онтологии без каких-либо допущений;

- UNA — принято только допущение UNA;

- UNA+CWA — приняты допущения UNA и CWA.

В табл. 3 представлены результаты валидации на различных уровнях для ограничений, приведенных в табл. 1 и 2. Для результатов валидации использованы следующие сокращения: OK — валидация выполнена успешно; ERROR — ошибка валидации, исходные объекты являются семантически некорректными. В ряде случаев при успешной валидации могут быть выведены новые знания, которые будут добавлены в исходные объекты. Такие примеры в таблице выделены курсивом.

Реализация подхода с использованием библиотеки Eriqron

Предложенный подход к валидации программ на языке Java реализован в библиотеке Eriqron. Для работы с онтологиями Eriqron использует средства OWLAPI [14]. Поддерживаются встраиваемые модули логического вывода. В текущей реализации в качестве модуля вывода по умолчанию используется Pellet [13], частично поддерживается HermiT [15].

Соответствие между структурами Java и онтологий задается с помощью аннотаций, являющихся встроенным механизмом языка Java. При несовпадении имени Java-класса с именем соответствующего ему класса онтологии, с классом ассоциируется аннотация `@RdfClass<имя_класса_онтологии>`. Если имя атрибута Java-класса не совпадает с именем объектного свойства или свойства данных онтологии, атрибут снабжается аннотацией `@RdfProperty<имя_свойства_онтологии>`.

Проиллюстрируем возможности библиотеки Eriqron на примере простого фрагмента онтологии ранее рассматриваемой предметной области, связанной с автомобилями. Ниже приведено ее определение с использованием манчестерского синтаксиса OWL.

```
Ontology (<http://cars.owl>
Declaration (Class (<http://cars.owl#Car>))
Declaration (ObjectProperty
 (<http://cars.owl#fasterThan>))
Declaration (ObjectProperty
 (<http://cars.owl#slowerThan>))
InverseObjectProperties
 (<http://cars.owl#fasterThan>
 <http://cars.owl#slowerThan>)
AsymmetricObjectProperty
 (<http://cars.owl#slowerThan>)
IrreflexiveObjectProperty
 (<http://cars.owl#slowerThan>)
Declaration (DataProperty
 (<http://cars.owl#power>))
```

```
DataPropertyDomain
(<http://cars.owl#power>;Car)
DataPropertyRange (<http://cars.owl#power>
<http://cars.owl#Car>))
```

Данная онтология определяет один класс — Car, со свойством данных power и объектными свойствами fasterThan и slowerThan. Последние два свойства определены как иррефлективные, асимметричные и инверсные друг другу.

Ниже представлен фрагмент реализации Java-класса Car, соответствующего описанному выше онтологическому классу Car.

```
public class Car {

    double power;

    @RdfProperty("slowerThan")
    List<Car> fasterCars = new
    ArrayList<Car>();

    @RdfProperty("fasterThan")
    List<Car> slowerCars = new
    ArrayList<Car>();

    public List<Car> getFasterCars()
    {return fasterCars; }

    public List<Car> getSlowerCars()
    { return slowerCars; }

    public double getPower()
    { return power; }

    public void setPower(double power)
    { this.power=power; }

}
```

Класс реализован в стиле JavaBean (для доступа к свойствам используются методы get и set), являющимся стандартом де-факто для описания классов данных.

Для поддержки возможности валидации и логических выводов в класс добавлены аннотации @RdfProperty, определяющие связь атрибутов fasterCars (более быстрые авто) и slowerCars (более медленные авто) с объектными свойствами fasterThan и slowerThan онтологии. При этом для самого класса Car и атрибута power подобные аннотации не нужны, поскольку их имена совпадают с соответствующими им элементами онтологии. Таким образом, изменения классов, необходимые для поддержки работы библиотеки Eriphron, незначительны.

После задания соответствия для классов данных к объектам этих классов можно применять валидацию и логические выводы. Ниже приведен пример валидации объекта класса Car в предположении, что онтология предметной области хранится в файле cars.owl.

```
// загрузка онтологии с помощью OWLAPI
OWLontology ontology =
OWLManager.createOWLontologyManager().
loadOntologyFromOntologyDocument(new
File("cars.owl"));
```

```
// создание объекта-валидатора
SemanticValidator validator =
new PelletSemanticValidator();

// создание данных для валидации
Car car1 = new Car();
car1.getFasterCars().add(car1);

// собственно валидация
ValidityReport report = validator.validate
(ontology, car1);
System.out.println(report.isValid());
// "false"
```

На первом этапе схема онтологии предметной области с помощью OWLAPI загружается в программную модель. Далее создается объект-валидатор Eriphron, в данном случае использующий модуль логического вывода Pellet. Затем создается объект класса Car. В рассматриваемом примере он является некорректным, поскольку находится в собственном списке fasterCars. Этот факт свидетельствует о том, что он быстрее самого себя.

Отметим, что объект car1 при этом не противоречит никаким ограничениям, содержащимся в Java-коде. Тем не менее, поскольку задана связь списка fasterCars со свойством slowerThan онтологии, при сериализации данного объекта в онтологии будет создан индивидум, связанный сам с собой отношением slowerThan. Поскольку данное отношение является иррефлективным, данный объект является семантически некорректным, что и будет зафиксировано в процессе валидации с помощью библиотеки Eriphron. Таким образом, семантическая валидация Java-объектов данных выполняется без использования каких-либо ограничений, выраженных средствами языка программирования (в данном случае Java).

Следующий пример демонстрирует возможности использования библиотеки Eriphron для выполнения логических рассуждений над объектами.

```
// создание данных для логических суждений
Car car1 = new Car();
Car car2 = new Car();
car1.getFasterCars().add(car2);

// выполнение логического вывода
report = validator.validateAndInfer
(ontology, car1, car2);

System.out.println(report.isValid());
// "true"

System.out.println
(car2.getSlowerCars().get(0));
// car1
```

В данном примере создаются объекты car1, car2 и определяется, что car1 медленнее чем car2. В процессе валидации и логического вывода с помощью Eriphron для объектов car1 и car2 в онтологии будут созданы

индивидуумы, для которых будет выведено, что *car2* быстрее чем *car1*, поскольку свойства *fasterThan* и *slowerThan* определены в онтологии как инверсные друг другу. На завершающем этапе работы Eriphron выполнит добавление выведенных знаний в программный код — в данном примере объект *car1* будет добавлен в список *slowerCars* объекта *car2*. Таким образом, свойства *fasterCars* и *slowerCars* будут заполнены автоматически с помощью онтологии предметной области без определения зависимости между этими свойствами на языке Java.

Eriphron является бесплатной, свободно распространяемой библиотекой с открытым исходным кодом. Она позволяет проводить валидацию и логический вывод над Java-объектами данных, используя лишь "схему" онтологии, описывающую ограничения предметной области и соответствия между структурами Java и онтологий, заданные с помощью аннотаций Java. Таким образом, отпадает необходимость реализовывать эти ограничения в Java-коде, что является трудоемким для сложных предметных областей.

Отметим, что логический вывод над онтологией является достаточно ресурсоемкой процедурой. Так, реализация вывода с помощью Pellet на онтологии из 100 сущностей (классов, свойств и индивидуумов) занимает до 2 с., а на онтологии из 3000 сущностей — до 60 с. [16]. Вследствие этого, средства семантического контроля в настоящее время целесообразно использовать на этапах разработки и отладки ПО, но не в развернутой системе. Вместе с тем активные исследования, направленные на повышение производительности модулей логического вывода, позволяют надеяться на ослабление этого ограничения в будущем.

Заключение

В статье рассмотрены возможность и преимущества использования онтологий на этапе реализации ПО. Показаны сходства основных конструкций онтологий и ОО-языков. Выделены три возможных подхода к отображению конструкций ОО-языков программирования на онтологии — прямой, непрямой и гибридный.

Предлагаемый в статье подход позволяет использовать онтологию предметной области, создаваемую (или заимствованную) на этапах анализа и проектирования ПО, в качестве "справочника" ограничений, присущих классам и отношениям предметной области. Реализация логических рассуждений на онтологии с использованием языков OWL и правил SWRL обеспечивает механизм семантической валидации ОО-программ. При этом нет необходимости реализовывать ограничения и правила на ОО-языке.

В статье представлена реализация данного подхода для языка Java — библиотека Eriphron с открытым исходным кодом. Библиотека Eriphron позволяет выполнять валидацию и логический вывод над Java-объектами. При этом поддерживаются известные в онтологическом инжиниринге допущения об уникальности имен и замкнутости мира.

Существующая реализация позволяет проводить валидацию и логический вывод над объектами "по требованию", при этом разработчик должен вызвать соответствующий метод, передав в качестве параметров онтологию и набор Java-объектов данных. Такой подход можно, например, использовать для проверки семантической корректности объектов перед их записью в БД или в Unit-тестах. Однако для полномасштабного использования данного подхода необходима его полная автоматизация. Дополнительные гарантии качества программного кода могла бы дать валидация части данных перед и после вызова важных методов, их модифицирующих, а также интеграция данного подхода с расширяемыми валидаторами различных ORM-технологий. Наконец, в дальнейшем возможно использование данного подхода при статическом анализе кода, т. е. реализация его в качестве подпрограммы проверки одного из статических анализаторов кода (например, FindBugs). Данные направления являются предметом дальнейших исследований.

Список литературы

1. **Липаев В. В.** Проектирование и производство сложных заказных программных продуктов. М.: СИНТЕГ, 2011. 400 с.
2. **Липаев В. В.** Методология верификации и тестирования крупномасштабных программных средств // Программирование. 2003. № 6. С. 7—24.
3. **Семенов В. А., Морозов С. В., Тарлапан О. А.** Инкрементальная верификация объектно-ориентированных данных на основе спецификации ограничений // Труды Института системного программирования РАН. 2004. Т. 8. Ч. 2. С. 21—52.
4. **Харитонов Д. И.** Раздельная верификация объектно-ориентированных программ с построением протокола C++ класса в терминах сетей Петри // Моделирование и анализ информационных систем. 2009. Т. 16. № 2. С. 92—111.
5. **OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax.** URL: <http://www.w3.org/TR/owl2-syntax/>
6. **SWRL: A Semantic Web Rule Language: Combining OWL and RuleML.** URL: <http://www.w3.org/Submission/SWRL/>
7. **Uschold M.** Ontology-Driven Information Systems: Past, Present and Future // Proc. of the Fifth International Conference on Formal Ontology in Information Systems (FOIS 2008). Amsterdam: IOS Press, 2008. P. 3—18.
8. **Разумовский А. Г., Пантелеев М. Г.** Использование онтологий в разработке программного обеспечения // Известия СПбГЭТУ "ЛЭТИ". 2011. № 8. С. 46—51.
9. **Kalyanpur A., Parsia B., Sirin E., Hendler J. A.** Debugging unsatisfiable classes in OWL ontologies // Web Semantics: Science, Services and Agents on the World Wide Web. 2005. N 3. P. 268—293.
10. **Puleston C., Parsia B., Cunningham J., Rector A. L.** Integrating Object-Oriented and Ontological Representations: A Case Study in Java and OWL // Proc. of ISWC2008. Berlin: Springer, 2008. P. 130—145.
11. **Russell S. J., Norvig P.** Artificial Intelligence: A Modern Approach. Prentice Hall, 2003. 1132 p.
12. **Frenzel C., Parsia B., Sattler U., Bauer B.** Moop — A Hybrid Integration of OWL and Java // Lecture Notes in Business Information Processing. 2011. N 83. P. 437—447.
13. **Pellet: OWL 2 Reasoner for Java.** URL: <http://clarkparsia.com/pellet/>
14. **The OWLAPI.** URL: <http://owlapi.sourceforge.net/>
15. **Hermit OWL Reasoner.** URL: <http://hermit-reasoner.com/>
16. **Sirin E., Parsia B., Grau B. C., Kalyanpur A., Katz Y.** Pellet: A Practical OWL-DL Reasoner // Web Semantics: Science, Services and Agents on the World Wide Web. 2007. N 5. P. 51—53.

В. Ф. Филаретов, д-р техн. наук, проф.,

Д. А. Юхимец, канд. техн. наук, доц.,

Э. Ш. Мурсалимов, аспирант,

Институт автоматике и процессов управления ДВО РАН

Дальневосточный федеральный университет,

e-mail: murs@mail.dvo.ru

Создание универсальной архитектуры распределенного программного обеспечения мехатронного объекта¹

Предложен новый подход к разработке программного обеспечения мехатронных объектов на основе архитектуры JAUS. Такое программное обеспечение представляет собой распределенную информационно-управляющую систему, компоненты которой реализуются таким образом, что их можно размещать на различных вычислительных устройствах без какой-либо модификации их частей, описывающих соответствующие функциональные свойства. Это позволяет существенно расширить область применения указанных систем для произвольных мехатронных объектов, бортовая вычислительная система которых содержит встраиваемые контроллеры.

Ключевые слова: программное обеспечение, мехатронный объект, информационно-управляющая система

Введение

В настоящее время при создании новых типов мехатронных объектов (МО) (роботов, манипуляторов и т. д.) значительная часть трудозатрат приходится на разработку их программного обеспечения (ПО). Это ПО должно выполнять множество функций, связанных с обработкой информации, поступающей от различных бортовых датчиков, и выработкой управляющих воздействий, поступающих на исполнительные устройства МО, т. е. представляет собой информационно-управляющую систему (ИУС). При этом основная сложность создания такой системы состоит в большом разнообразии оборудования (датчиков, исполнительных устройств и т. д.) и интерфейсов обмена данными, входящих в состав МО,

а также функций, которые должна реализовывать указанная ИУС.

ИУС МО может разрабатываться на основе различных подходов. Можно создать централизованную систему в виде одной программы, которая будет последовательно выполнять все необходимые операции [1]: опрашивать датчики, проводить обработку всех поступающих данных и осуществлять выработку управляющих воздействий. Однако при наличии сложных алгоритмов управления и обработки информации, требующих выполнения множества функций в одном цикле, применение этого подхода, использующего последовательную процедуру выполнения операций, помимо усложнения самой программы и ее последующей модификации существенно увеличивает время выработки управляющих сигналов, что значительно ухудшает качество управления МО. При этом ввиду специфичности компонентов централизованной ИУС их использование в процессе создания ИУС для других МО значительно затруднено. Указанные недостатки делают эти ИУС мало применимыми в современных МО.

¹ Работа проводилась при финансовой поддержке Министерства образования и науки Российской Федерации (гос. контракт № 07.514.11.4085 от 17.10.2011 г.) и поддержана РФФИ (гранты 10-07-00395, 11-07-98502).

Другим распространенным подходом к построению ИУС МО является создание распределенных ИУС [1, 2], состоящих из нескольких программ, взаимодействующих между собой и реализующих конкретные функции ИУС (навигация, управление движением, формирование траектории и т. д.). При этом одной из наиболее важных и сложных задач при создании указанных ИУС является разработка и отладка механизмов (протоколов) обмена данными между отдельными управляющими программами. На особенности реализации этих механизмов существенное влияние оказывает состав бортовых вычислительных средств МО, как правило, включающий в себя несколько отдельных вычислительных устройств [3–5], а также стратегия распределения функций ИУС по этим устройствам. Указанное распределение зависит от сложности реализуемых функций (например, функции обработки изображений желательнее размещать на мощных бортовых компьютерах), требуемой частоты их выполнения, особенностей взаимодействия с бортовым оборудованием МО (например, навигационная система должна иметь непосредственный доступ к данным сенсоров МО) и т. д.

Помимо сложностей создания таких ИУС следует учитывать и сложности их последующей модернизации при возможной модернизации МО, связанной как с изменением состава его аппаратной части (интерфейсов передачи данных, а также числа или состава бортовых сенсоров, вычислительных устройств и т. д.), так и с введением дополнительных функций ИУС. Это может потребовать значительной переработки структуры ИУС и механизмов (протоколов) обмена данными между всеми ее функциями.

При разработке ИУС конкретных МО важно предусмотреть и возможность использования ее компонентов без дополнительных модификаций в ИУС других объектов, что позволяет существенно ускорить и упростить их создание. В работе [6] был предложен подход к построению ИУС мобильных роботов, имеющих аппаратно-независимую функциональную часть, что позволяет проводить их достаточно простую адаптацию к различным схемам аппаратной части этих роботов. Однако для реализации этой ИУС используется язык высокого уровня Python, применение которого возможно только на достаточно мощных бортовых компьютерах.

В результате актуальной остается задача создания таких ИУС МО, структура которых допускала бы простую модификацию и использование на большом числе различных МО независимо от состава их бортовых вычислительных систем. Решение этой задачи и описывается в данной работе.

Постановка задачи

В настоящее время существуют универсальные архитектуры ИУС МО, которые достаточно успешно применяются на практике. В качестве базовой целесообразно использовать архитектуру системы *Joint*

Architecture for Unmanned Systems (JAUS) [7], хорошо зарекомендовавшей себя при разработке ИУС для сложных автономных мобильных роботов и манипуляторов. Структурная схема систем, создаваемых на основе указанной архитектуры, представлена на рис. 1.

Указанная ИУС состоит из взаимодействующих подсистем, включающих отдельные устройства (например, пульт оператора, мобильный робот и т. д.). Каждая подсистема содержит набор взаимодействующих узлов, обычно представляющих собой реальные устройства в составе МО или отдельные вычислительные системы, а узлы являются наборами взаимодействующих компонентов, каждый из которых реализует отдельную функцию системы (функцию планирования траектории, закон управления исполнительными устройствами и т. д.). При этом каждому компоненту, входящему в состав ИУС, присваивается адрес, состоящий из номера подсистемы ИУС, номера узла в подсистеме и номера этого компонента в узле.

Взаимодействие между различными компонентами ИУС происходит на основе обмена сообщениями стандарта JAUS, что позволяет обеспечить расширяемость и совместимость указанных ИУС с различными МО, так как компоненты, выполняющие однотипные функции (например, управление движением МО по траектории), в ИУС разных МО будут обмениваться одинаковыми сообщениями. Кроме того, в этом случае становится возможным обеспечить простое сопряжение различных ИУС МО, реализованных на основе указанной архитектуры. В то же время стандарт JAUS не накладывает ограничений на конкретную реализацию ИУС, а требует только того, чтобы компоненты ИУС обменивались стандартными сообщениями JAUS. Однако известные реализации этой архитектуры [8] ориентированы на бортовые компьютеры, работающие под управлением мощных операционных систем, и являются достаточно "тяжеловесными" для использования их в МО, вычислительная система которых обычно строится на типовых контроллерах, выполняющих отдельные функциональные задачи.

В связи с отмеченным выше, в данной работе ставится задача разработки такой архитектуры ИУС МО, которая при минимальных модификациях могла бы применяться на произвольных МО независимо от используемых в них аппаратных средств, а также бес-



Рис. 1. Структурная схема ИУС МО, основанной на архитектуре JAUS

печивала бы удобное наращивание ее функций. При этом указанная ИУС должна позволять простое распределение ее функций на произвольном числе типовых вычислительных устройств.

Требования к реализации ИУС МО

Как видно на рис. 1, базовыми элементами, из которых будет строиться ИУС МО, являются компоненты, реализующие ее отдельные функции. Поэтому необходимо разработать такую программную реализацию этих компонентов, которая без существенных модификаций позволит им работать как на бортовых компьютерах, так и на контроллерах, входящих в состав бортовой вычислительной системы МО.

Сформулируем требования к указанной программной реализации.

1. Необходимо использовать языки С или С++, которые обеспечивают высокую скорость выполнения операций и применимы для всех типов микроконтроллеров, микропроцессоров и операционных систем. При этом необходимо исключить сложные и медленные конструкции — шаблоны.

2. Функции, реализуемые компонентами ИУС, должны быть независимы от аппаратных интерфейсов передачи данных, т. е. изменение этого интерфейса (например, интерфейс UDP на последовательный интерфейс) не должно приводить к изменению реализации этих функций. Удовлетворение указанного требования позволит обеспечить аппаратную независимость ИУС.

3. При реализации архитектуры ИУС необходимо выделить всего два типа компонентов: компонент общего вида, реализующий конкретные функции ИУС, и менеджер, обеспечивающий передачу сообщений между компонентами. Наличие специальных менеджеров обусловлено требованием стандарта JAUS и необходимостью уменьшения числа связей между компонентами. При этом все сообщения между компонентами должны проходить через менеджер соответствующего узла, что позволит каждому компоненту для отсылки сообщений знать только один физический адрес и иметь только один физический интерфейс передачи данных.

4. Компоненты, выполняя свои функции, должны работать параллельно. Это позволит, решая каждую отдельную задачу в нужном темпе, сократить время цикла управления для критически важных задач.

5. Структура компонента должна быть независима от функций, которые он выполняет, и интерфейсов передачи данных. При этом должна обеспечиваться его работоспособность на любых вычислительных устройствах, работающих как под управлением операционных систем, так и на простых встраиваемых контроллерах. Это в случае необходимости позволит перераспределять отдельные функции ИУС между различными вычислительными устройствами при минимальных изменениях в компонентах.

Далее рассмотрим подход, позволяющий обеспечить выполнение указанных выше требований.

Вначале рассмотрим структуру отдельного компонента ИУС. Он должен содержать две независимые части (для выполнения требования 2): интерфейс обмена данными и функциональную часть компонента. При этом для выполнения требования 5 алгоритм его работы должен быть организован как у программы контроллера: последовательное выполнение функций в бесконечном цикле. Для однотипности структуры компонента независимо от реализуемых им функций эту структуру необходимо описывать в терминах объектно-ориентированного программирования.

Диаграмма классов, описывающих компонент ИУС, показана на рис. 2.

При этом используется нотация, предложенная в работе [9].

Как видно из представленной диаграммы, класс COMPONENT, описывающий отдельный компонент ИУС, включает в себя ссылки на два вида объектов: CMP_INTF — объект, описывающий интерфейс обмена данными, и CMP — объект, описывающий интерфейс доступа к функциональным возможностям компонента.

Класс COMPONENT содержит следующие основные параметры:

- MSG — последнее сообщение, полученное интерфейсом обмена данными;
- STATE — структура, содержащая переменные, отражающие текущее состояние компонента;
- CMP* — указатель объекта класса CMP;
- CMP_INTF* — указатель объекта класса CMP_INTF.

Основные методы, которые содержит указанный класс:

- init() — функция инициализации параметров объекта;
- intf_reg() — функция, регистрирующая ссылку на объект класса INTF_CMP, которая связывает этот объект с параметром CMP_INTF*;
- cmp_reg() — функция, регистрирующая ссылку на объект класса CMP, которая связывает этот объект с параметром CMP*;
- run() — функция, описывающая работу компонента.

Классы CMP_INTF и CMP являются абстрактными и обозначены на рис. 2 буквой А в перевернутом треугольнике. Они определяют доступ к интерфейсной и функциональной частям компонента соответственно. Класс CMP_INTF определяет доступ к следующим функциям интерфейса:

- init() — функция инициализации параметров интерфейса передачи данных;
- send() — функция, передающая через заданный физический интерфейс сформированное в процессе работы компонента сообщение;
- receive() — функция, считывающая сообщение с интерфейса обмена данными и сохраняющая результат в переменной MSG, если новых сообщений

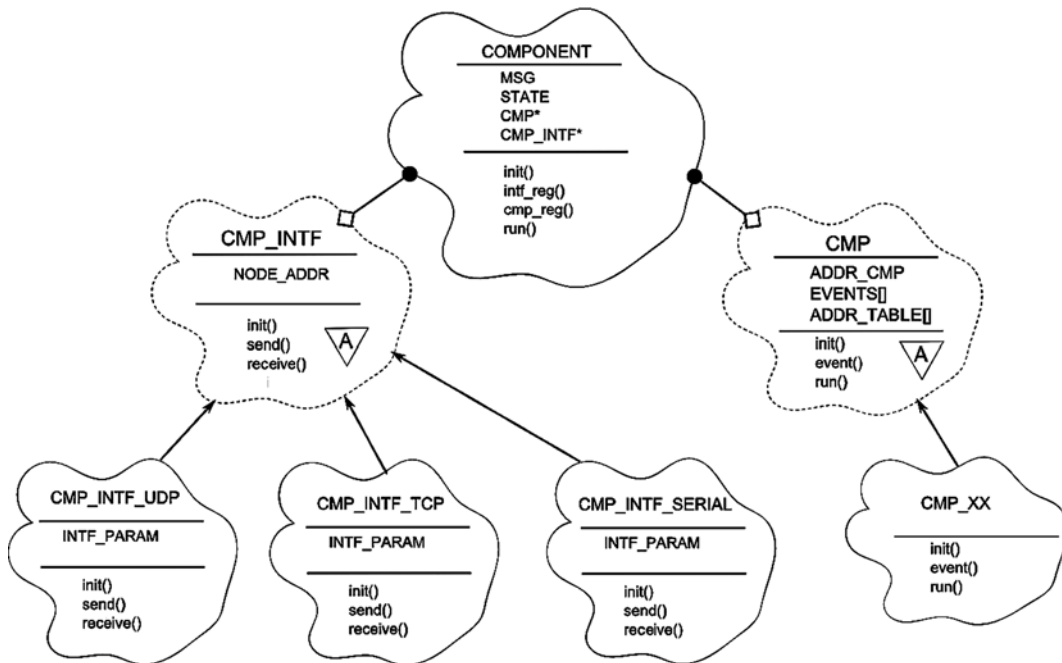


Рис. 2. Диаграмма классов, описывающая компонент ИУС

получено не было, то возвращается значение 0, в противном случае — 1.

При этом классы, описывающие конкретные интерфейсы передачи данных и основанные на базе класса CMP_INTF, должны содержать параметры, задающие физический адрес узла NODE_ADDR и описывающие параметры интерфейса передачи данных INTF_PARAM. В указанных классах особенность реализации функций send(), receive() и init(), а также переменных INTF_PARAM, NODE_ADDR будет зависеть от типа интерфейса передачи данных.

На рис. 2 приведены примеры возможных наследников класса CMP_INTF:

- CMP_INTF_UDP — обеспечивает обмен данными по протоколу UDP/IP;
- CMP_INTF_TCP — обеспечивает обмен данными по протоколу TCP/IP;
- CMP_INTF_SERIAL — обеспечивает обмен данными по последовательному порту.

Как было сказано выше, класс CMP является абстрактным. Этот класс описывает интерфейс доступа к функциональной части компонента с учетом того, что:

- init() — функция выполняет инициализацию параметров компонента;
- event() — функция отслеживает наступление заданных событий и формирует соответствующее сообщение, если никаких событий не наступало, то возвращается 0, в противном случае возвращается 1, а сформированное сообщение сохраняется в переменной MSG класса COMPONENT;

- run() — функция выполняет все специфические для компонента действия.

Классы, созданные на основе класса CMP (на рис. 2 обозначены как CMP_XX), описывают функциональные возможности конкретного компонента ИУС и должны содержать следующие параметры:

- ADDR_CMP — внутренний адрес компонента в ИУС;
- EVENTS[] — массив структур, описывающих какие события должен отслеживать данный компонент (достижение заданного положения, разряд батареи, наличие препятствия и т. д.);
- ADDR_TABLE[] — массив адресов компонентов ИУС, с которыми данный компонент может поддерживать связь.

При этом функции init(), event() и run() для каждого компонента должны быть переопределены при создании нового компонента на основе базового класса CMP. Помимо указанных основных функций в обоих классах можно доопределить необходимые сервисные функции: обработка отдельных сообщений, обработка исключительных ситуаций и т. д.

Таким образом, структура компонента, представленная на рис. 2, позволяет полностью отделить интерфейс передачи данных между компонентами от их функциональной составляющей и комбинировать их в любых вариантах.

Сообщения, которыми обмениваются компоненты, можно представить структурой данных, состоящих из двух основных частей:

- заголовок, содержащий сведения о типе сообщения, адресах отправителя и получателя этого сооб-

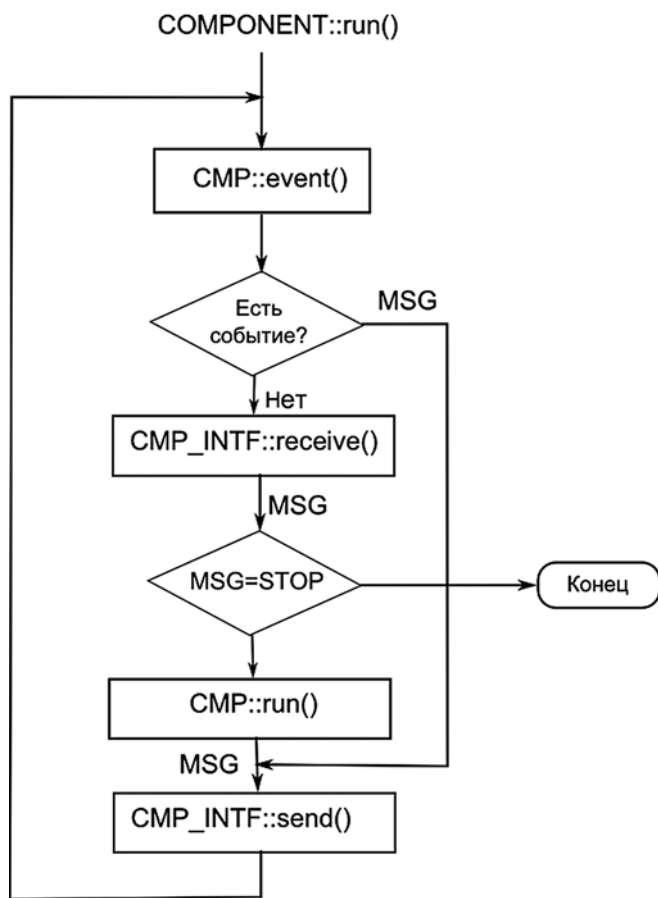


Рис. 3. Блок-схема алгоритма работы отдельного компонента

щения, приоритете и т. д., и имеющий одинаковый вид для всех типов сообщений;

- поле данных, представляющее собой байтовый массив, который должен быть интерпретирован в зависимости от типа сообщения.

Блок-схема алгоритма работы функции run() класса COMPONENT представлена на рис. 3.

Как видно на этой блок-схеме, при работе компонента функция run() класса COMPONENT работает в цикле до тех пор, пока не придет сообщение, прекращающее работу компонента. При этом сам алгоритм содержит несколько последовательно выполняющихся действий:

- 1) проверка на наступление заданных событий посредством вызова функции event() класса CMP, с помощью которой последовательно проверяются условия наступления этих событий и формируются соответствующие сообщения;

- 2) пересылка сформированного в функции event() сообщения в заданный компонент ИУС, если событие наступило, и начало цикла заново;

- 3) считывание сообщения с интерфейса обмена данными и формирование пустого сообщения (сооб-

щения с нулевыми полями), если новых сообщений не приходило;

- 4) обработка поступившего сообщения в соответствии с логикой работы компонента, вызов функции run() класса CMP и формирование ответного сообщения в случае ее выполнения;

- 5) пересылка сформированного сообщения заданному компоненту.

Так как все сообщения проходят через менеджер узла, то для его отправки от одного компонента ИУС другому достаточно знать только физический адрес этого менеджера. При формировании сообщения необходимо знать только внутренний адрес компонента-получателя в ИУС и не требуется знания типа интерфейса передачи данных и физического адреса компонента.

На приведенной блок-схеме алгоритма видно, что его реализация возможна в виде программы, которая может выполняться как на бортовом компьютере под управлением операционной системы, так и на отдельном контроллере. Иногда несколько компонентов можно размещать на одном контроллере. Также возможны случаи, когда одну функцию ИУС могут выполнять несколько стандартных компонентов (например, навигационная система может соответствовать компонентам определения локального и глобального положения МО, скорости его движения и т. д.). Для реализации этих случаев в структуре компонента необходимы следующие изменения:

- параметр CMP* должен быть не просто указателем на объект класса CMP, а массивом указателей;
- опрос компонентов на наличие событий должен проводиться в цикле по всем компонентам, а при первом обнаружении события должно формироваться сообщение, отсылаемое компоненту-приемнику.

Таким образом, предложенная структура компонента ИУС обеспечивает независимость его функциональной части от интерфейса обмена данными и позволяет комбинировать их в любых вариантах без дополнительной модификации.

Структура компонента-менеджера

Особый тип компонента, используемый в ИУС МО с архитектурой JAUS — менеджер, осуществляющий маршрутизацию сообщений. При этом используется два вида менеджеров: менеджер узла, который осуществляет маршрутизацию сообщений между компонентами внутри узла и узлами одной подсистемы, и коммутатор, осуществляющий маршрутизацию сообщений между подсистемами ИУС.

Класс, описывающий компонент-менеджер, в целом соответствует структуре компонента, представленной на рис. 2. Отличие заключается лишь в том, что компонент-менеджер может проводить обмен данными по нескольким физическим интерфейсам. Например, с компонентами, работающими на бортовом компьютере по интерфейсу UDP или TCP, а с компонентами, расположенными на локальных

контроллерах, по последовательному интерфейсу. При этом менеджер должен знать физические адреса компонентов, расположенных в пределах одного узла, а также физические адреса узлов в пределах одной подсистемы.

Используемая структура класса, описывающего компонент-менеджер, показана на рис. 4.

Класс MANAGER содержит следующие параметры:

- MSG — последнее принятое сообщение;
- MODE — переменная, задающая режим работы менеджера или как менеджера узла, или как коммуникатора;
- CMP_INTF*[] — массив указателей на интерфейсы передачи данных;
- ADDR_CMP[] — массив адресов компонентов, относящихся к данному узлу;
- ADDR_NODE[] — массив адресов узлов, входящих в данную подсистему;
- ADDR_SS[] — массив адресов подсистем, входящих в ИУС МО.

Число интерфейсов передачи данных N равно числу всех возможных соединений, которые может установить компонент-менеджер. Это число для менеджера узла равно $N = N_{comp}^{node-i} + N_{node}^{ss-j} + 1$, а для комму-

никатора $N = N_{node}^{ss-j} + N_{ss}$, где N_{comp}^{node-i} — число компонентов в данном узле; N_{node}^{ss-j} — число узлов в текущей системе, N_{ss} — число подсистем в ИУС.

Использование объектов класса CMP_INTF позволяет существенно упростить реализацию менеджера и ускорить его работу, так как каждый менеджер всегда может держать открытыми каналы для передачи сообщений во время его работы.

Основными функциями, обеспечивающими класс MANAGER, являются следующие:

- init() — функция, выполняющая инициализацию компонента-менеджера;
- intf_reg() — функция, регистрирующая интерфейс передачи данных;
- route() — основная функция, определяющая логику работы компонента-менеджера.

Блок-схема алгоритма работы указанной функции представлена на рис. 5.

На рис. 5 видно, что в процессе работы менеджера происходит последовательный опрос всех зарегистрированных интерфейсов и перенаправление полученных сообщений нужному адресату. При получении сообщений они сохраняются в специальном файле для дальнейшего анализа работы ИУС. Указанная

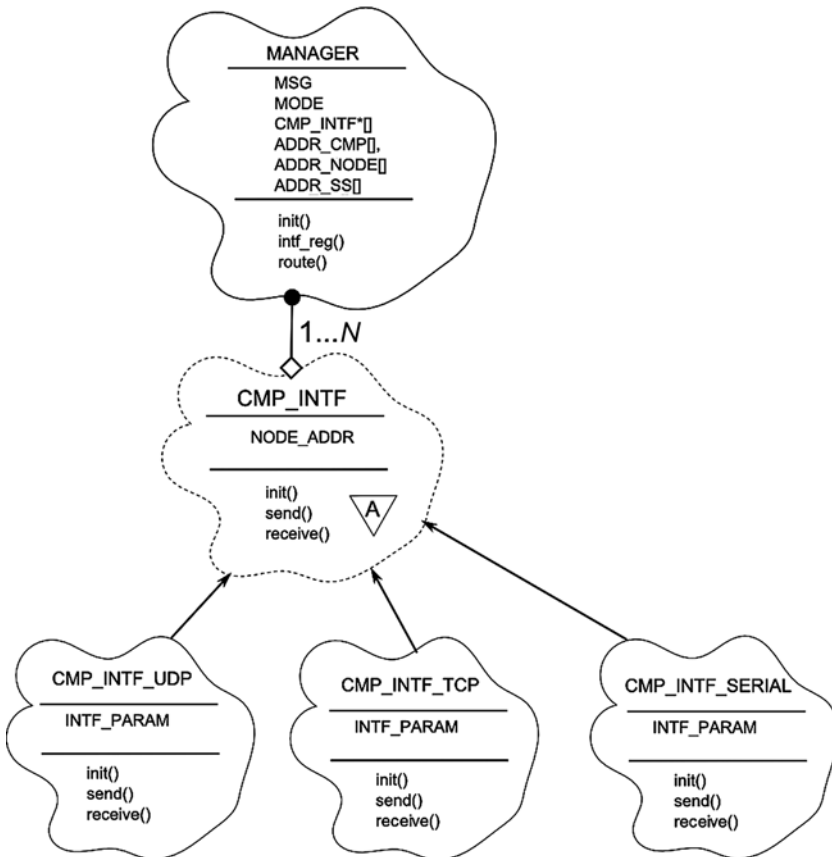


Рис. 4. Структура класса, описывающего компонент-менеджер ИУС

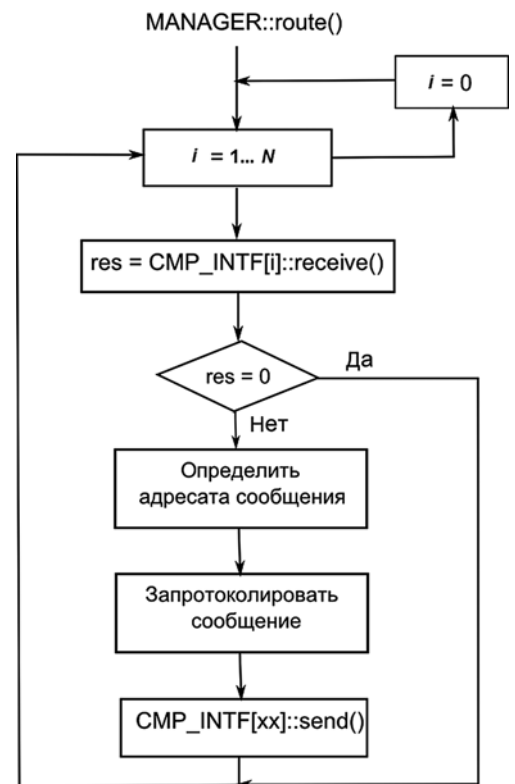


Рис. 5. Блок-схема алгоритма работы функции route() класса MANAGER

функция реализуется на уровне менеджера узла, так как все сообщения между компонентами проходят через него. Из этой же схемы видно, что реализация менеджера без каких-либо модификаций возможна и на любом вычислительном оборудовании.

После описания структуры отдельного компонента рассмотрим работу всей ИУС в целом.

Описание работы ИУС

В начале работы предлагаемой ИУС МО необходимо задать ее конфигурацию, т.е. во всех менеджерах следует сформировать массивы `ADDR_CMP[]`, `ADDR_NODE[]`, `ADDR_SS[]`, задающие соответствие внутренних адресов компонентов, узлов и подсистем их физическим адресам. При этом в каждом компоненте необходимо сконфигурировать интерфейс передачи данных, т.е. задать тип и параметры интерфейса обмена данными, а также физический адрес менеджера узла, к которому относится компонент. Указанные данные могут храниться в специальных конфигурационных файлах и считываться при начальном запуске системы.

Далее каждый компонент ИУС запускается как отдельная программа или на бортовом компьютере, или на контроллере МО. Таким образом, вся ИУС представляет собой совокупность параллельно выполняющихся задач вне зависимости от того, на каком конкретно вычислительном устройстве происходит соответствующая реализация. Это позволяет выделить особо важные задачи, выполнение которых должно проводиться с высокой частотой, и запускать их на конкретных вычислительных устройствах, что повышает надежность работы всей системы в целом.

Таким образом, предложенная архитектура позволяет обеспечить гибкую и простую настройку ИУС на любую конфигурацию бортовых вычислительных устройств МО.

Пример реализации ИУС МО

В качестве примера можно рассмотреть ИУС мобильного робота, представленного на рис. 6.

Указанный робот содержит набор бортовых датчиков, определяющих параметры его движения и положение относительно предметов окружающей среды. Исполнительными элементами его колес являются двигатели постоянного тока. Кроме того, робот содержит телекамеру, способную изменять пространственную ориентацию своей оптической оси с помощью двух сервоприводов. Робот может двигаться по заданному маршруту как автономно, так и в режиме телеуправления, формируемом с помощью задающего устройства, содержащего два аналоговых джойстика.

Структурная схема ИУС указанного мобильного робота показана на рис. 7. Эта ИУС разбита на две подсистемы. Первая реализована на базе ПК, работающего под управлением операционной системы Windows. Она обеспечивает прием и интерпретацию



Рис. 6. Колесный мобильный робот

команд оператора, а также отображение окружающей обстановки и состояния робота. Эта подсистема содержит один узел, в который входят: интерфейс оператора `INTERFACE` и компонент `GAMEPAD`, интерпретирующий команды задающего устройства оператора в задающие сигналы исполнительных устройств робота. В указанную подсистему можно вводить и другие компоненты без внесения изменений в уже созданные.

Вторая подсистема, управляющая перемещением мобильного робота, использует несколько разнотипных вычислительных устройств. Компоненты, непосредственно формирующие управляющие сигналы на исполнительные устройства робота (`PRIMITIVE_DRIVER` и `CAMERA_DRIVER`), установлены на отдельном контроллере `Arduino Nano`. Эти компоненты не содержат сложных алгоритмов и служат интерфейсом между ИУС и исполнительными устройствами робота. На этом же контроллере размещается специальный диагностический компонент `LOW_DIAG`, осуществляющий отслеживание уровней напряжения бортовых источников питания и проверку работоспособности исполнительных устройств робота.

В состав бортовых датчиков входят акселерометры, гироскопы, компас, энкодеры каждого колеса, а также ультразвуковые и инфракрасные датчики расстояния. В навигационной системе используется алгоритм сигма-точечного фильтра Калмана, позволяющего осуществлять совместную обработку всех данных, поступающих с этих датчиков. Этот алгоритм реализован на мощном 32-битовом контроллере `PIC32`. Доступ к данным, формируемым указанным алгоритмом, осуществляется с помощью компонентов, эмулирующих работу более простых датчиков движения робота (`LOCAL_POS`, `LOCAL_VELOCITY` и т. д.). Это позволяет при необходимости полностью заменить алго-

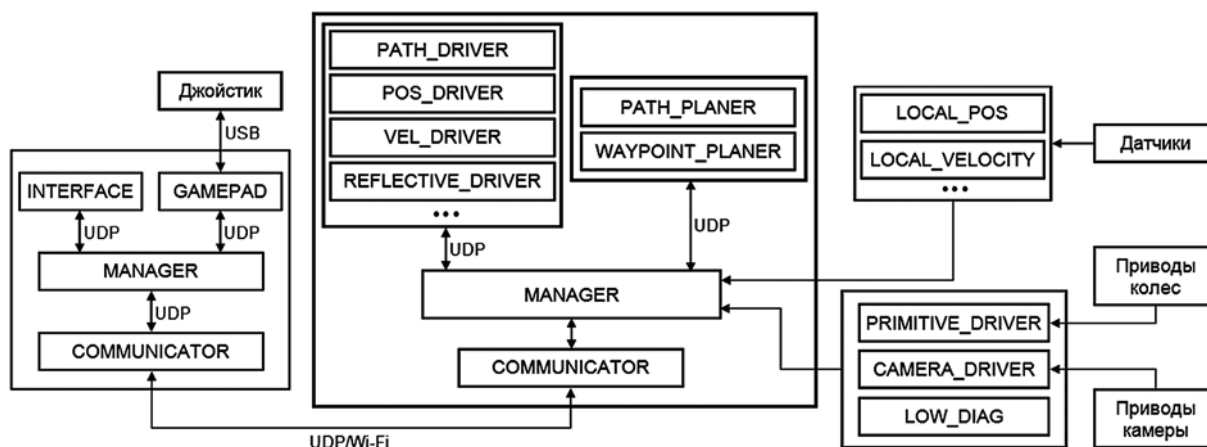


Рис. 7. Структурная схема ИУС мобильного робота, реализованная на основе предлагаемого подхода

ритм работы навигационной системы и набор датчиков без внесения изменений в другие модули ИУС.

Для реализации функций менеджера, коммуникатора, систем управления и систем формирования траекторий используется одноплатный бортовой компьютер, работающий под управлением операционной системы Linux. Связь между менеджером и компонентами, реализованными на бортовых контроллерах, осуществляется по последовательным портам, а между этим менеджером и компонентами, реализованными на бортовом компьютере, — по протоколу UDP. Связь между подсистемами ИУС реализуется по протоколу UDP с использованием интерфейса Wi-Fi.

На бортовом компьютере в виде отдельных программ параллельно работают два набора компонентов. Первый набор формирует программные сигналы движения робота (PATH_PLANNER — компонент, формирующий траекторию движения, WAYPOINT_PLANNER — компонент, формирующий целевую точку), а второй обеспечивает реализацию этих сигналов (PATH_DRIVER — движение по заданной траектории, POS_DRIVER — движение в заданную точку, VEL_DRIVER — движение с заданной скоростью, REFLECTIVE_DRIVER — движение, исключающее столкновения, и т. д.). При этом в текущий момент времени активным может быть только один компонент из набора, который задает режим движения. Это позволяет сократить число одновременно выполняемых задач и экономить ресурсы бортового компьютера.

Следует отметить, что созданная ИУС может быть использована для различных типов мобильных роботов, содержащих различный набор бортового вычислительного оборудования.

Заключение

В данной работе предложен новый подход к построению универсальной ИУС МО, основанной на архитектуре JAUS. Особенность предложенного подхода

заключается в том, что он позволяет так реализовать компоненты ИУС, что их можно размещать на различных вычислительных устройствах без какой-либо модификации их частей, описывающих соответствующие функциональные свойства. Это позволяет существенно расширить область применения указанных ИУС для МО, бортовая вычислительная система которых содержит встраиваемые контроллеры. Кроме того, описанный подход позволяет часть особо важных задач, требующих высокой частоты обновления формируемых сигналов (например, в процессе управления исполнительными устройствами МО), реализовать на отдельных вычислительных устройствах, что существенно повышает эффективность и надежность работы всей ИУС в целом.

Список литературы

1. Siciliano B., Khatib O. Handbooks of robotics. Springer, 2008. 1628 p.
2. Borrelly J.J., Coste-Maniere E., Espiau B. and etc. The ORCCAD architecture // Int. J. Robotics Researches. 1998. Vol. 17. N 4. P. 338—359.
3. Автономные подводные роботы: системы и технологии / Под ред. М. Д. Агеева. М.: Наука, 2005. 308 с.
4. Park I.-W., Kim J.-Y., Cho B.-K., Oh J.-H. Control hardware integration of a biped humanoid robot with an android head // Robotics and Autonomous Systems. 2008. Vol. 56. P. 95—103.
5. Jin J., Jung Ch., Kim D., Chung W. Development of an autonomous outdoor patrol robot in private road environment // Proc. of Int. Conference on Control, Automation and Systems ICCAS 2010. Seoul, Korea, 2010. P. 1918—1921.
6. Кирсанов К. Б., Левинский Б. М., Пряничников В. Е. Интеграционное программное обеспечение интеллектуальных роботов // Информационно-измерительные и управляющие системы. 2009. Т. 7. № 6. С. 35—43.
7. Описание стандартов архитектуры JAUS. URL: <http://www.sae.org>.
8. Открытая реализация архитектуры JAUS — OpenJAUS. URL: <http://www.openjaus.com>.
9. Буч Г. Объектно-ориентированный анализ и проектирование. Второе издание, пер. с англ. М.: Бином, СПб.: Невский диалект, 2000. 560 с.

А. Е. Александров, д-р техн. наук, проф., **А. А. Востриков**, канд. техн. наук, доц.,
В. П. Шильманов, аспирант,
Московский государственный университет приборостроения и информатики,
e-mail: femsystem@yandex.ru

Принципы организации архитектуры программной системы "Прогноз" на основе библиотеки компонентов

Представлена программная система "Прогноз", разработанная на базе технологии, основанной на моделировании семейства программных систем. Представлена библиотека программ, состоящая из элементов, выполняющих основные (прикладные) функции, и вспомогательных элементов, предназначенных для автономного тестирования этих программ, выявления и исправления ошибок. Приведено описание базовых структур данных, которые отражают особенности предметной области и используются для реализации прикладных элементов в составе библиотеки.

Ключевые слова: программная система, программный компонент, библиотека программных компонентов, объектно-ориентированное проектирование

Введение

Для анализа безопасности энергетических систем важную роль приобрело направление, основанное на использовании вероятностных моделей механики разрушения. Это направление широко используется для обоснования надежности и безопасности используемого в составе таких систем оборудования, прогнозирования его ресурса, анализа рисков неблагоприятных событий [1]. Обоснование надежности и безопасности такого оборудования представляет собой сложный, многостадийный процесс, основанный на использовании детерминированных и вероятностных моделей.

Анализ отказов энергетических систем показывает, что подавляющее число случаев, приводящих к наступлению предельных (с позиций разрушения) состояний оборудования, связано с образованием и развитием трещин, достигших критических или опасных размеров [1]. Как правило, причины возникновения таких трещин образуются вследствие несовершенства технологических процессов изготовления объекта и содержатся в нем до начала эксплуатации. Постепенное накопление рассеянных повреждений, вызванных различными условиями эксплуатации оборудования, приводит к образованию макроскопических трещин и

к их развитию. Выросшая до критических размеров трещина может привести к аварийной ситуации. Прогнозирование процессов зарождения и развития трещин может быть сделано на основе разработки и реализации объединенной модели, включающей все стадии разрушения, в том числе зарождение, докритический и критический рост трещин. Достижение предельного состояния, вызванного различными механизмами разрушения, протекающими с разной интенсивностью, зависит от характера эксплуатационных воздействий.

Эксплуатационные воздействия, в свою очередь, определяются режимами эксплуатации, последовательностью и числом повторов этих режимов. Перечень режимов эксплуатации устанавливается главным конструктором объекта эксплуатации. Различные сочетания последовательностей режимов образуют множество сценариев, которые могут приводить к разным последствиям, связанным с проявлением различных механизмов разрушения.

Моделирование процессов зарождения и развития трещин при заданных условиях эксплуатации объекта (оборудования) на основе объединенной модели разрушения является целью разрабатываемой с участием авторов программной системы "Прогноз".

Архитектура построения библиотеки компонентов

При разработке системы, предназначенной для решения рассматриваемой в настоящей работе проблемы, была использована технология, основанная на моделировании семейства программных систем, получившая название порождающее программирование [2]. Эта технология была опробована авторами при разработке ряда программных систем [3].

Основой используемой технологии является порождающая доменная модель. Каждую из составляющих такой модели формируют следующие механизмы идентификации:

- представителя семейства программных систем, соответствующего предметной области;
- отдельных компонентов, из которых осуществляется сборка программной системы.

Сведения о таких семействах и отдельных компонентах аккумулируются в виде определенным образом организованных хранилищ данных. Запросы, на основе которых формируется конечная программная система по заявляемым требованиям, оформляются в виде программы на предметно-ориентированном высокоуровневом языке.

Для автоматизированной сборки программной системы необходимо разработать реализующую ее библиотеку программных компонентов. В качестве таких компонентов были выделены компоненты, реализующие алгоритмы на отдельных этапах. Этапы описываются моделью предметной области и заданными условиями эксплуатации рассматриваемого объекта. Основное требование, предъявляемое к таким компонентам, заключается в том, чтобы они были совместимы по данным при включении их в единый каркас проектируемой системы. С этой целью были использованы объекты-посредники, исключающие прямое взаимодействие между основными объектами классов, реализующих функции отдельных компонентов, а также между этими объектами и файловой системой, которая представляет собой набор файлов входных и выходных данных компонентов, хранимых в дереве каталогов с заданной структурой.

В предлагаемой авторами архитектуре используются три объекта-посредника и объект класса реализации функциональности алгоритма. Объекты-посредники представлены следующими классами: CPathFinder — класс определения путей к данным; CLoader_Saver — класс загрузки/сохранения, CRepository — класс хранилища данных (рис. 1).

Использование предложенной архитектуры построения компонента библиотеки позволяет объектам, обеспечивающим работу с файловой системой, взаимодействовать только с объектом хранилища данных. Со своей стороны объект, реализующий отдельный алгоритм, получает входные данные и возвращает выходные данные в объект класса хранилища данных и полностью освобождается от взаимодействия как с файловой системой, так и с другими объектами. Объект, содержащий таблицу путей к файлам данных,

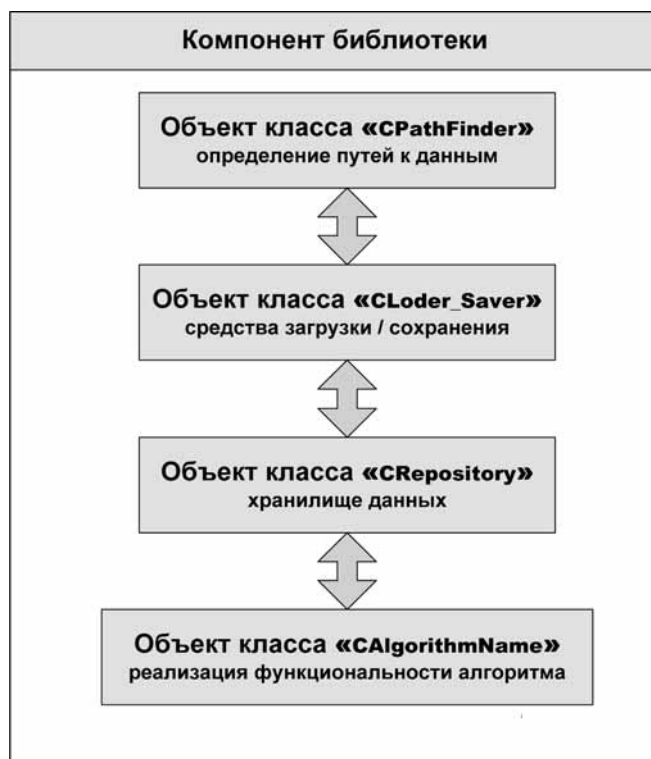


Рис. 1. Архитектура построения компонента библиотеки

позволяет при загрузке и сохранении данных абстрагироваться от особенностей организации данных в файловой системе. Использование этого объекта позволяет избежать изменения кода объекта загрузки/сохранения при изменении структуры каталогов или имен файлов в файловой системе.

Инкапсуляция взаимодействий между объектами библиотеки в объектах-посредниках позволяет исключить прямую связь объектов, реализующих функции алгоритмов, между собой и со средствами загрузки и сохранения данных. Таким образом, при создании новой программной системы на основе библиотеки компонентов, обмен данными между её компонентами или отдельным компонентом и файловой системой не требует внесения изменений в код объектов, реализующих алгоритмы.

Кроме функциональных программных компонентов в состав библиотеки входят программные средства, предназначенные для тестирования их кода. Для проведения тестирования используются средства журналирования (класс CLog). С помощью объекта данного класса происходит сбор информации о ходе выполнения кода и вывод сведений о его состоянии на момент возникновения ошибок. Для обработки ошибок используется класс CSystemErrors, который выводит сообщение о нормальном завершении работы или о следующих типах ошибок:

- по указанному пути не найден файл данных;
- ошибка при доступе к файлу данных;
- неправильный формат файла данных;

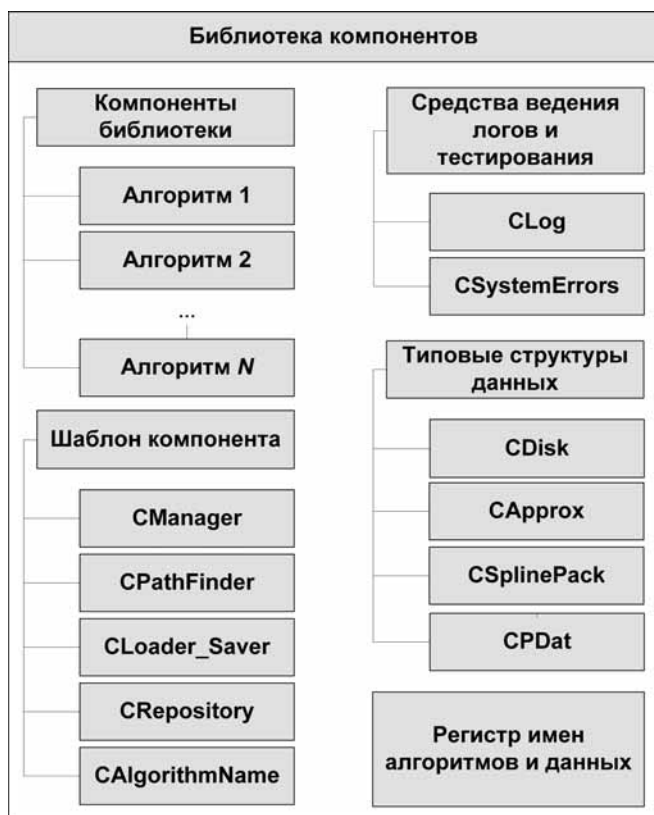


Рис. 2. Состав библиотеки компонентов

- ошибка при создании каталога для файла выходных данных;
- ошибка при передаче данных объекту алгоритма — передаваемый объект данных не содержит данных;
- ошибка при передаче данных объекту алгоритма — передаваемые в объект данные не удовлетворяют заданным условиям.

Совокупность шаблонов классов CManager, CRepository, CPathFinder и CLoader_Saver и класса алгоритма образуют шаблон компонента библиотеки, которая служит для поддержки введенных стандартов разработки программной системы (рис. 2).

В рамках рассматриваемой предметной области были разработаны и включены в состав библиотеки базовые структуры данных, используемые для описания входных и выходных данных функциональных компонентов библиотеки.

В состав этих структур входит полиморфная иерархия наследования базового класса CApprox и его классов-наследников CLineArx и CSpline, предназначенная для хранения функций, заданных таблицей значений. Для расчета промежуточных значений используется кусочно-линейная интерполяция (класс CLineArx) или интерполяции кубическими сплайнами (класс CSpline). Для работы с массивами объектов класса CApprox используется класс CSplinePack.

Отметим, что представление данных, которые используются функциональными компонентами библи-

отеки, в виде таблично заданных функций отражает особенности рассматриваемой предметной области. С помощью объектов класса CSplinePack могут быть представлены: компоненты тензора напряженно-деформированного состояния и тепловые поля, изменяющиеся во времени в точках исследуемого объекта; циклограммы приведенных напряжений; размахи циклов приведенных напряжений; изменения повреждаемости и вероятности повреждаемости в точках объекта по времени; изменение вероятности образования сквозной трещины по времени и т.д. Отмеченная особенность позволяет рассматривать выделенные базовые структуры данных в качестве основы для формирования предметно-ориентированного языка.

В целях сокращения времени разработки шаблон класса CLoader_Saver содержит заготовки методов чтения и записи данных классов CApprox и CSplinePack, которые при реализации нового алгоритма достаточно дополнить именами данных конкретного алгоритма. Для представления в хранилище массивов данных других типов используется класс CPDat. Класс CDisk реализует базовые функции работы с файловой системой. Методы данного класса используются при реализации методов класса CLoader_Saver.

Для регистрации имен алгоритмов и данных используется общий для всей библиотеки заголовочный файл, который предназначен для определения строковых констант, соответствующих уникальным именам алгоритмов и данных. Наличие общего для всей библиотеки перечня уникальных имен позволяет избежать конфликта имен при создании приложений на основе библиотеки.

Порождение объекта класса, описывающего реализацию алгоритма, осуществляется методом Run класса управления CManager. Метод Run используется после исполнения метода Load. Это означает, что к моменту создания объекта класса, реализующего алгоритм, хранилище содержит все полностью сформированные входные и инициализированные выходные данные алгоритма. При этом в таблице путей к файлам данных в объекте класса CPathFinder содержатся пути, позволяющие сохранить результаты работы алгоритма на диск с помощью метода Save. После формирования входного списка указателей на данные объекта, этот список передается его конструктору, а затем вызывается метод запуска на исполнение объекта, реализующего алгоритм.

Формирование приложений на основе библиотеки компонентов

На основе разработанной библиотеки компонентов (алгоритмов) в рамках созданной архитектуры могут быть сформированы различные программные приложения. Ниже представлен пример формирования программного приложения для расчета зарождения трещин в перемычке коллектора парогенератора РУ ВВЭР-1000 [4].

Модель зарождения трещин формируется на основе заданных условий эксплуатации и включает следующую последовательность расчетов.

1. Для всех выбранных режимов, включенных в условия эксплуатации, рассчитываются напряженно-деформируемые и тепловые поля, изменяющиеся во времени в каждой точке исследуемого объекта. Для расчета напряженно-деформируемых и тепловых полей используется автономная программная система конечно-элементного анализа. Результаты расчета конечно-элементной системы передаются в программную систему "Прогноз" в виде файлов.

2. По известной последовательности режимов в системе "Прогноз" формируются циклограммы рассчитанных напряженно-деформируемых и тепловых полей, которые используются для определения главных напряжений и размахов напряжений (например, методом теней [4]).

3. Полученные размахи напряжений рассматриваются совместно с температурными полями и устанавливаются сочетания, наиболее опасные с точки зрения накопления повреждений.

4. Накопленные таким образом повреждения суммируются по всем заданным режимам для анализируемых условий эксплуатации.

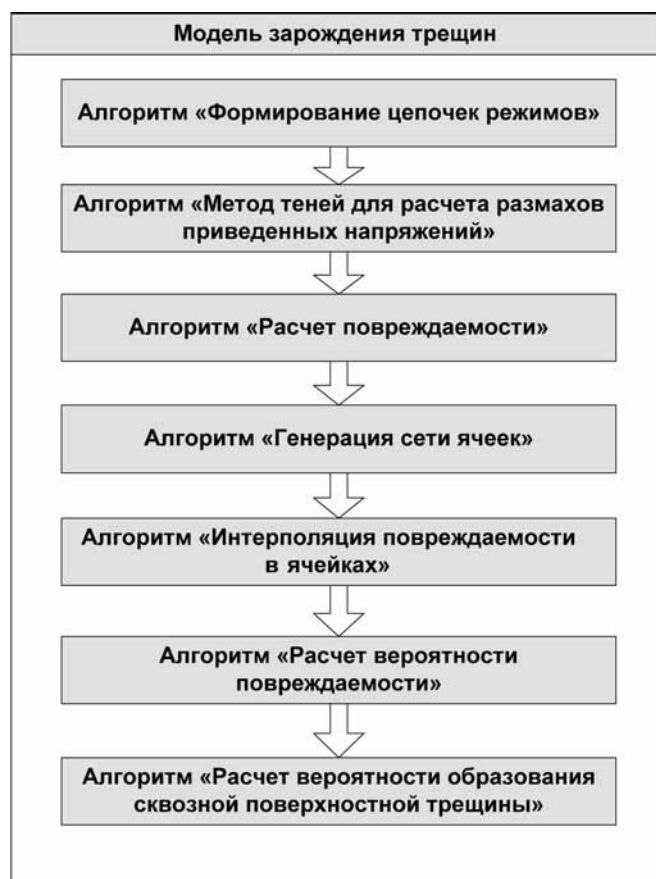


Рис. 3. Модель зарождения трещин на основе разработанной библиотеки алгоритмов

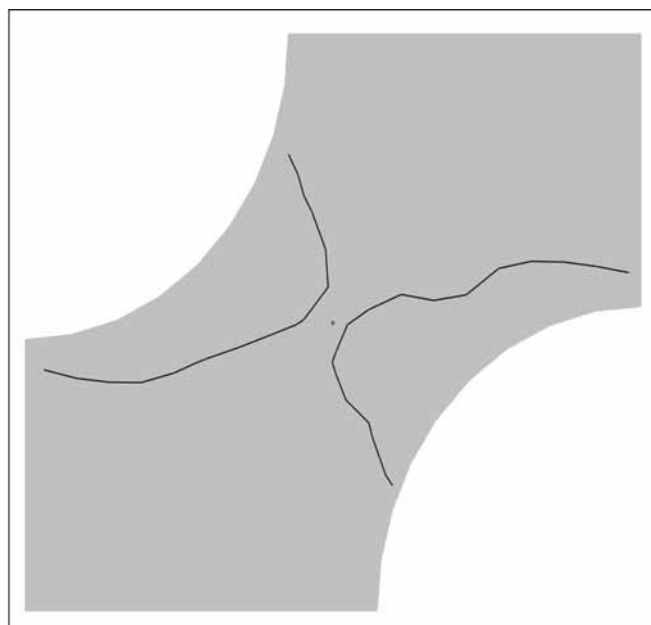


Рис. 4. Распределение повреждаемости по поверхности перемычки коллектора на заданный момент времени

5. На исходной геометрической модели, используемой при конечно-элементных расчетах, генерируется сеть ячеек с размером, характерным для зарождения зародыша трещин.

6. В центре сгенерированных ячеек на основе посчитанных значений повреждаемости в узловых точках конечных элементов с помощью интерполяции рассчитываются значения повреждаемости по времени. В результате определяется суммарная повреждаемость, приводящая к местным локальным разрушениям.

7. На основе известного поля повреждаемости по времени определяется вероятность повреждаемости в ячейках.

8. По известному распределению поврежденных ячеек с учетом вероятности их повреждения рассчитывается вероятность образования сквозной трещины. Сквозная трещина определяется как трещина, достигшая противоположных границ поверхности перемычки.

Последовательность алгоритмов, реализующих модель зарождения трещин и оформленных в виде разработанной библиотеки компонентов, приведена на рис. 3.

Выполненные расчеты, показывающие распределение изолиний повреждаемости в заданный момент времени по поверхности перемычки коллектора парогенератора по реализованной модели зарождения трещин, представлены на рис. 4. Эти изолинии повреждаемости моделируют возможные траектории образовавшихся трещин на поверхности перемычки. Смыкание траекторий приводит к образованию сквозной поверхностной трещины.

Разработанная модель процесса зарождения трещин может быть дополнена моделями докритического роста и критического роста трещин при различных факторах, включая механические, коррозионные и нейтронные воздействия. В результате может быть сформирована модель разрушения с разной степенью детализации в зависимости от заданных условий эксплуатации.

Заключение

Предложенная авторами архитектура построения программной системы "Прогноз" основана на моделировании семейства программных систем. Каждая конкретная реализация программы формируется из отдельных функциональных компонентов (алгоритмов). Для совместимости функциональных компонентов по данным были использованы объекты-посредники, исключающие прямое взаимодействие между основными объектами классов, реализующими функции отдельных компонентов.

В рамках рассматриваемой предметной области были разработаны и включены в состав библиотеки


базовые структуры данных, используемые для описания входных и выходных данных функциональных компонентов библиотеки.

Такие разработанные структуры данных являются основой для построения высокоуровневых предметно-ориентированных языков и формирования на их основе различных моделей, отвечающих заданным целям исследования.


Список литературы

1. **Анализ** риска и проблемы безопасности. В 4-х частях // Часть 1. Основы анализа и регулирования безопасности / Науч. руковод. К. В. Фролов. М.: МГФ "Знание", 2006. 640 с.
2. **Чарнецки К., Айзенкер У.** Порождающее программирование: методы, инструменты, применение. Для профессионалов. СПб.: Питер, 2005. 731 с.
3. **Александров А. Е.** Разработка и проектирование прикладных программных систем на основе порождающего программирования // Информационные технологии. 2008. № 9. С. 2—9.
4. **Александров А. Е., Александров П. А., Востриков А. А.** и др. Отчет по верификации программного комплекса "Прогноз" по расчету вероятности повреждения перемычек коллектора парогенератора РУ ВВЭР-1000. Рег. номер № 01201056086. М.: МГУПИ. 2010. 167 с.

Официальная поддержка:



Правительство Челябинской области



Министерство информационных технологий и связи Челябинской области



Главное управление министерства внутренних дел России по Челябинской области



Администрация г. Челябинска



Южно-Уральская торгово-промышленная палата

1 Первое
Выставочное
Объединение
pvo74.ru

20-22 ноября ЧЕЛЯБИНСК

Генеральный информационный партнер:
БЕЗОПАСНОСТЬ



II СПЕЦИАЛИЗИРОВАННАЯ ВЫСТАВКА IT-ТЕХНОЛОГИИ. СВЯЗЬ. ТЕЛЕКОМУНИКАЦИИ

РАЗДЕЛЫ ВЫСТАВКИ:

- Телекоммуникации, системы телерадиовещания
- Мобильные технологии
- Кабельные и спутниковые технологии
- Автоматизированные системы связи
- Широкополосные сети связи
- Информационная безопасность
- Сетевая безопасность
- Видеоконференции

- Системы интеграции
- Программное обеспечение, биллинговые системы
- Компьютерные и информационные технологии
- Мониторы и телевизоры, оргтехника
- Телефонные аппараты и аксессуары
- Банковские технологии, оборудование и услуги
- Кабели, антенны, матчи
- Услуги операторов сотовой связи

ВЦ "Мегаполис", Свердловский пр., 51А
Тел.: (351) 215-88-77, 231-37-41 www.pvo74.ru

С. Е. Попов, канд. техн. наук, ст. науч. сотр.,

Р. Ю. Замираев, канд. техн. наук, ст. науч. сотр.,

Федеральное государственное бюджетное учреждение науки Институт вычислительных технологий Сибирского отделения Российской академии наук, г. Кемерово,

e-mail: prosluda@gmail.com

Программный комплекс и язык метаописаний алгоритмов анализа социально-экономических объектов

Статья посвящена разработке программного комплекса и языка метаописаний алгоритмов анализа социально-экономических объектов, формализации описательных правил экспертно-расчетных модулей и структурированных данных на примере энтропийного метода анализа. Определяются основные правила составления системно-аналитических профилей, интегрирующих возможность отладки, тестирования и визуализации результатов анализа. На примере исследования состояния регионов Сибирского федерального округа продемонстрированы направления развития и прикладной реализации программного комплекса и языка метаописаний в области анализа социально-экономических систем.

Ключевые слова: язык метаописаний, энтропийный метод анализа, программный комплекс, многокритериальный выбор, функциональные показатели, социально-экономические системы

Введение

Совокупность данных, характеризующих состояние социально-экономических объектов, отличается неоднородностью и паллиативностью. При ранжировании объектов используют экономические показатели самих объектов, внешние социальные факторы, географические особенности расположения объектов в контексте демографической ситуации, экономической безопасности и других процессов, протекающих в обществе. В таком же широком наборе показателей исследуют декомпозиции сложных систем – территориальных и региональных объединений. При этом многие показатели, значимые для принятых в экономике форм отчетности, являются связанными коэффициентами или грубо ранжированными характеристиками.

Дополнительные трудности для анализа социально-экономических объектов создает существенная изменчивость показателей их функционирования, которая определяется характерной для каждого объекта экономической деятельностью, многостадийностью

социальных процессов. Фиксируемые в произвольные моменты времени значения, например, объемы дотационного фонда, количество безработных и малообеспеченных граждан, пенсионеров и т. п. могут сильно отличаться от среднегодовых или среднемесячных значений.

При анализе социально-экономических систем необходимо учитывать такой факт, как статистическая совокупность, под которой понимается планомерный научно-обоснованный сбор сведений о социально-экономических явлениях и анализ полученных данных. Для того чтобы выполнить статистическое исследование, необходима информационная база. Такую базу формирует совокупность социально-экономических объектов или явлений общественной жизни, объединенных качественной основой, общей связью. Однако такие объекты и явления отличаются друг от друга отдельными признаками, например, совокупность населения города, сотрудники предприятий или фирм.

Именно выбор статистической совокупности и ее единиц накладывает существенные ограничения на

многие статистические методы, так как он зависит от конкретных условий и характера изучаемого социально-экономического явления, процесса, например, в задачах кластерного анализа по выявлению "нетипичных" объектов в совокупности.

На практике исследуемые социально-экономические явления чрезвычайно многообразны, поэтому охватить их все сложно и подчас невозможно. Исследователь вынужден изучать лишь часть статистической совокупности, а выводы делать по всей. В таких ситуациях важнейшим требованием является обоснованный отбор той части, по которой изучаются признаки. Например, если изучается активность молодых избирателей, то необходимо определить границы возраста таких избирателей, чтобы исключить людей более старшего поколения. Можно ограничить такую совокупность представителями сельской местности или, например, студенчества.

В условиях существенной неоднородности и разнообразия свойств объектов, интерес и практическую значимость, в конечном счете, приобретают заключения, которые получаются с использованием инвариантных критериев, характеризующих отличительные особенности выбранного объекта на фоне функционально подобных ему. Такой подход к анализу данных был сформулирован как "диагностика состояния уникальных объектов" и реализован в энтропийном методе [1—4]. Энтропийный характер метода в данном случае отражает меру неопределенности информации, полученной в ходе тестового измерения, которое может иметь разные исходы.

Стандартные пакеты прикладных программ предлагают инструменты различного методического уровня для обработки данных социально-экономической направленности. С их помощью удается решать многие важные задачи, но при наличии устоявшихся оценок для типичных объектов и ситуаций в стационарных условиях. Существенным недостатком специализированных программных пакетов для анализа данных является их статичность — один или несколько устойчивых тестов (критериев), адаптированных под конкретный набор данных и реализованных в едином алгоритме от ввода данных до визуализации результатов.

В существующих технологиях без внимания остались более гибкие подходы, обеспечивающие интеграцию компонентов аналитических систем общими синтаксическими правилами, например, метаописанием их поведения. Стандартные технологии, такие как OLAP или Data mining, для формализации решающих правил предлагают использовать инструментальные средства на основе специализированных языков высокого уровня с объектно-ориентированными моделями типа CLIPS и LISP. Такой подход зачастую требует от аналитика глубоких знаний языков программирования.

По причинам, изложенным выше, вполне логичным было бы отображать взаимодействие модулей загрузки, подготовки, обработки и анализа данных в ви-

де разветвленной иерархической схемы действий, приводящей к совокупности (комплексу) аналитических заключений, логически и методически связанных между собой. При этом в качестве средства контроля логики и синтаксиса метаописания расчетных схем целесообразно предоставить аналитику программное средство, в котором были бы реализованы основные и вспомогательные алгоритмы энтропийного метода.

Постановка задачи

1. Разработать единый формат системно-аналитического профиля на базе языка XML, полностью описывающий логику расчета модели в контексте анализа социально-экономических объектов (авторское название языка метаописаний: EAML — *Entropy Analysis Markup Language*).

2. Разработать конструктор, позволяющий создавать, редактировать и проводить верификацию системно-аналитических профилей, обеспечивающий контроль логики и синтаксиса метаописания расчетных схем, их хранение и загрузку в виде файлов XML (авторское название: программный комплекс "ЭнтропияПлюс").

Программный комплекс и его функциональные возможности

Структура системно-аналитического профиля (САП) определяется математической моделью энтропийного метода анализа [2, 3]. Алгоритмы применения его для различных типов данных и математические обоснования приведены в работе [4].

Далее предлагается сконцентрировать свое внимание непосредственно на прикладном аспекте метода и метаописании его математического аппарата в рамках анализа социально-экономических показателей. Будем именовать это *предметной областью*, в которой правомерность всех преобразований вытекает из теоретических основ энтропийного метода анализа (ЭМА), доказываемая и обосновывается в его изложении.

В энтропийном методе анализа выделяют следующие математические модели: энтропийная модель выборочных данных, логарифмическая модель, обобщенные и комбинированные модели, инвертированные модели и обобщенные элементы. Для анализа предметной области в контексте настоящей работы были применены следующие математические модели: логарифмическая и инвертированная модели выборочных данных. Для простоты ссылок, назовем их *математическими преобразованиями*. Именно правила корректности записи последовательности преобразований в рамках энтропийного метода и определяют семантику языка EAML. Синтаксис и вид записи преобразований в "тегированном виде" отражают эвристически структурированный поиск каких-то особенностей данных. В структуре такого поиска предоставляется возможность комбинировать различные

Соответствия элементарных преобразований ЭМА
элементам САП

Таблица 1

Элементарные преобразования		XML-вариант
Название	Выражение	
Логарифмирование	$L_{i,j} = \ln(x_{i,j})$	<code><module id="[№]" name="Logarithm" parentid="[№]([[№],[№]...)]"/></code>
Инвертирование	$I_{i,j} = \frac{1}{x_{i,j}}$	<code><module id="[№]" name="Reverse" parentid="[№]([[№],[№]...)]"/></code>
Стандартизация	$S_{L_{i,j}} = \frac{L_{i,j} - M[L_{i,j}]}{\sigma[L_{i,j}]}$ $S_{I_{i,j}} = \frac{I_{i,j} - M[I_{i,j}]}{\sigma[I_{i,j}]}$	<code><module id="[№]" name="Standardization" parentid="[№]([[№],[№]...)]" direction="matrix column row"/></code>
Извлечение	$I_V = (j_1, j_2, \dots, j_k) = \text{sub}(I_{i,j})$	<code><module id="[№]" name="Extraction" parentid="[№]([[№],[№],[№]])" parameterindex="[№]([[№],[№],[№]...)]" direction="matrix column row"/></code>
<p>Примечание: в таблице представлена часть методов, использованных только для анализа предметной области. Общее число элементарных преобразований соответствует 16 элементам САП.</p>		

математические модели ЭМА, что выражает его результат в виде гистограмм и фазовых портретов.

Семантика записи последовательности математических моделей ЭМА позволяет выделить каждое преобразование в отдельный элемент (тэг) со своим набором атрибутов, задающих правила для интерпретатора. Представляется целесообразным называть каждый элемент согласно его математической модели (табл. 1).

Профиль содержит описательную часть расчета (название профиля, схем и параметров). Расчетная матрица обозначена как элемент `<matrix>`, она включает элементы `<column>` (показатели), путь к источнику данных. Каждый элемент `<module>` формально описывает процедуру преобразования данных, содержит атрибуты `id` и `parentid` соответственно, номер преобразования в последовательности и ссылку на результат предыдущего преобразования (рис. 1).

Атрибут `parentid` может содержать список номеров, необязательно идущих по порядку, что означает, применены комбинации результатов предыдущих процедур. Атрибут `direction` задает направление расчета, может принимать значения: матрица; столбец; строка. Это означает, что в исходном элементарном преобразовании опускаются соответствующие индексы i или j . Соответственно, результатом расчета может быть как матрица, так и столбец или строка. Схема

```
<profile id="Profile 1" name="Профиль 1">
  <matrix startdatabound="" step="" datasource="C:\Temp\regdata.txt">
    <column id="Names" name="Names" isObject="True" />
    <column id="Population" name="Population" isObject="False" />
    <column id="Pop-city" name="Pop-city" isObject="False" />
    <column id="Pop-country" name="Pop-country" isObject="False" />
    <column id="Pop-men" name="Pop-men" isObject="False" />
  </matrix>
  <schema id="schema4" name="Аудит регионов. Красноярский край"
    desc="Регионы СФО" refreshinterval="30000"
    namepatern="column" legend="">
    <streamX axisname="Логарифмическая модель">
      <submatrix startdatabound="" step="">
        <column id="Population" name="Population" isObject="False" />
        <column id="Pop-city" name="Pop-city" isObject="False" />
        <column id="Pop-country" name="Pop-country" isObject="False" />
        <column id="Pop-men" name="Pop-men" isObject="False" />
      </submatrix>
      <submatrix>
        <module id="1" name="Logarithm" parentid="0" />
        <module id="2" name="Standardization" parentid="1" direction="column" />
        <module id="3" name="Standardization" parentid="2" direction="row" />
        <module id="4" name="Sum" parentid="3" direction="column" />
        <module id="5" name="Standardization" parentid="4" direction="row" />
      </submatrix>
    </streamX>
    <streamY axisname="Инвертированная модель">
      <submatrix startdatabound="" step="">
        <column id="Population" name="Population" isObject="False" />
        <column id="Pop-city" name="Pop-city" isObject="False" />
        <column id="Pop-country" name="Pop-country" isObject="False" />
      </submatrix>
      <submatrix>
        <module id="1" name="Reverse" parentid="0" />
        <module id="2" name="Standardization" parentid="1" direction="column" />
        <module id="3" name="Standardization" parentid="2" direction="row" />
        <module id="4" name="Extraction" parentid="3" parameterindex="7" direction="row" />
      </submatrix>
    </streamY>
    <graphexpert description="&lt;graphexpert type=&quot;PhasePlanePortrait&quot;&gt;&lt;groups...
  </schema>
</profile>
```

Рис. 1. Пример САП для анализа социально-экономических объектов

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE profile [
  <ELEMENT profile (matrix, scheme+)>
  <!ATTLIST profile
    id ID #REQUIRED
    name CDATA #REQUIRED
  <ELEMENT matrix (column+)>
  <!ATTLIST matrix
    refreshinterval CDATA #REQUIRED
    connectionstring CDATA #REQUIRED
    remotingtable CDATA #REQUIRED
    startdatabound CDATA #REQUIRED
    step CDATA #REQUIRED
  <ELEMENT submatrix (column+)>
  <!ATTLIST submatrix
    startdatabound CDATA #REQUIRED
    step CDATA #REQUIRED
  <ELEMENT column EMPTY>
  <!ATTLIST column
    id CDATA #REQUIRED
    name CDATA #IMPLIED
    isObject CDATA #IMPLIED
  <ELEMENT scheme (streamX, streamY, graphexpert?)>
  <!ATTLIST scheme
    id ID #REQUIRED
    name CDATA #REQUIRED
    desc CDATA #IMPLIED
    refreshinterval CDATA #REQUIRED
    namepatern CDATA #REQUIRED
    legend CDATA #REQUIRED
  <ELEMENT streamX (submatrix,module+)>
  <!ATTLIST streamX
    axisname CDATA #IMPLIED
  <ELEMENT streamY (submatrix,module+)>
  <!ATTLIST streamY
    axisname CDATA #IMPLIED
  <ELEMENT module EMPTY>
  <!ATTLIST module
    id CDATA #REQUIRED
    mask CDATA #IMPLIED
    direction CDATA #IMPLIED
    parameterindex CDATA #IMPLIED
    parentid CDATA #IMPLIED
    step CDATA #IMPLIED
    name CDATA #REQUIRED
```

Рис. 3. Модель определения типа документа XML САП социально-экономических объектов

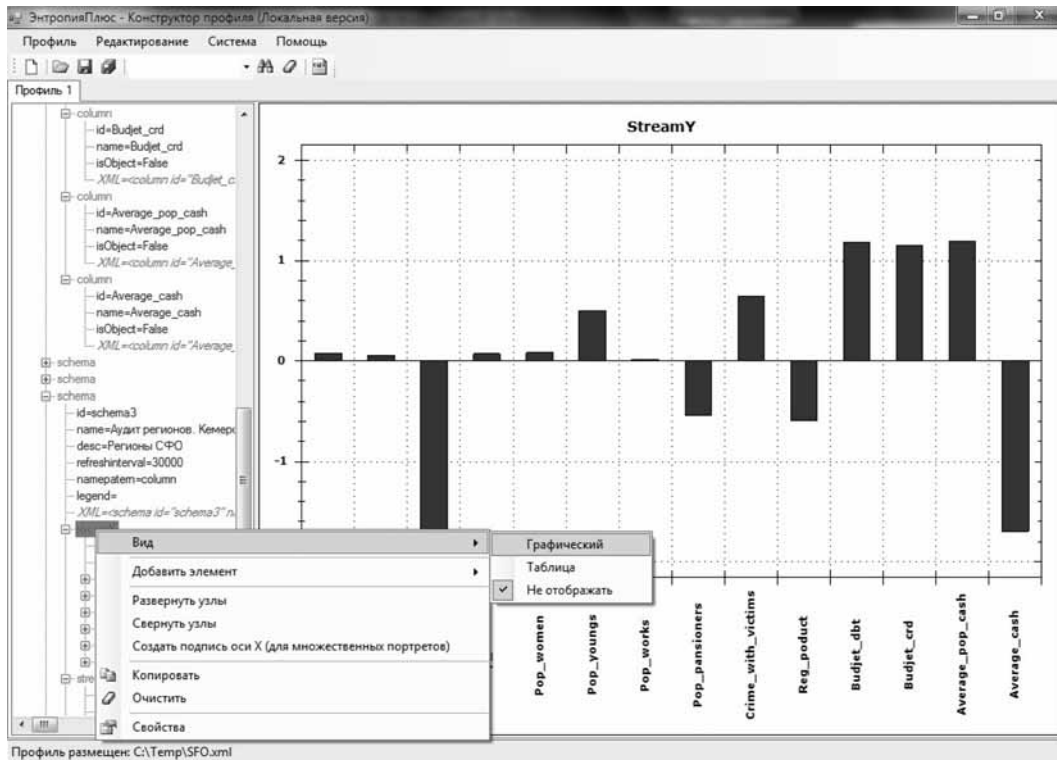


Рис. 4. Промежуточные результаты вычислений расчетной схемы САП

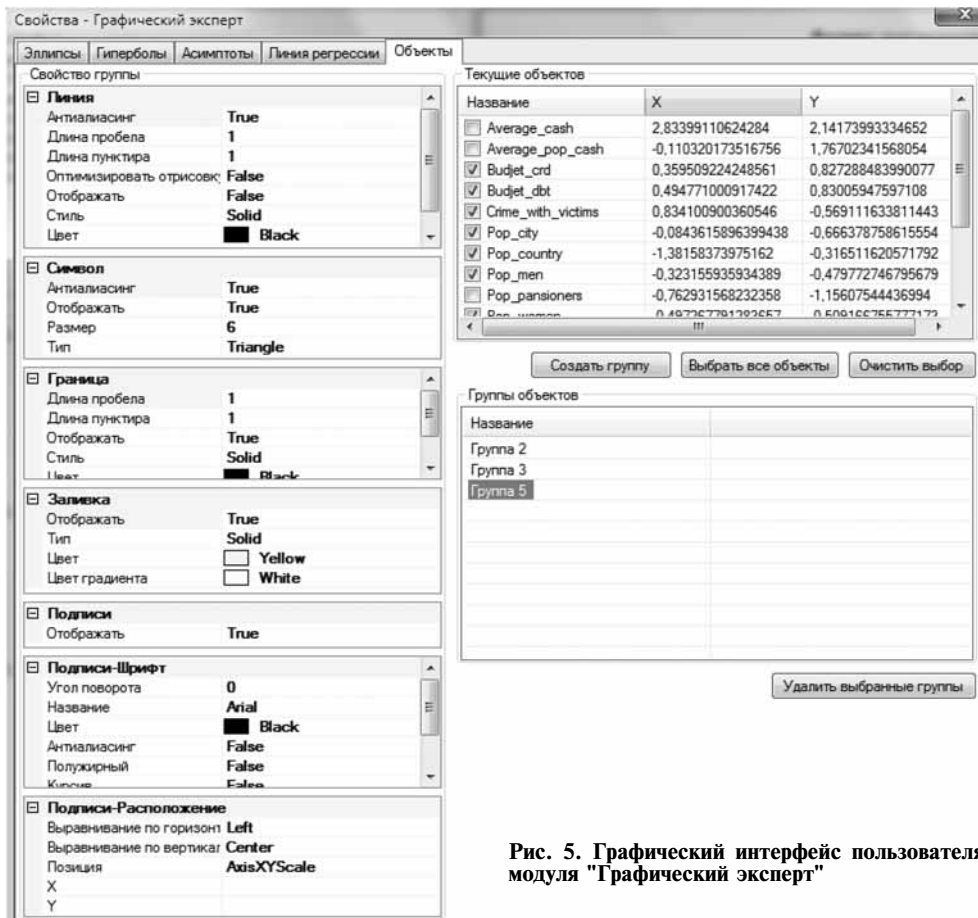


Рис. 5. Графический интерфейс пользователя модуля "Графический эксперт"

разбита на два потока <streamX>, и <streamY>, описывающих элементарные преобразования показателей по осям фазовой плоскости (рис. 1).

Схема содержит системную информацию: интервал обновления расчета; применение процедур подготовки данных; способ именования конечного результата в соответствии с последовательностью преобразований; легенду. В зависимости от потребностей визуализации конечного результата, атрибуты элемента <schema> могут варьироваться. Тэг-элемент <graphexpert> содержит правила для графических элементов (линия, маркер, цвет и т. п.), описывающие комбинации анализируемых объектов на фазовой плоскости (рис. 2, см. третью сторону обложки). Это позволяет аналитику визуально оценить расположение отображающих точек (объектов) относительно границ видов состояния системы, в данном случае социально-экономической. В конструкторе пользовательский профиль формируется в виде дерева (рис. 2, см. третью сторону обложки). Движение по нему задает действия серверного модуля для получения конечного графического представления, что сводит построение профиля к конструированию иерархии объектов-тэгов согласно модели определения типа документа (DTD — *Document Type Definition*) (рис. 3).

Отдельные процедуры конструктора отслеживают изменения структуры дерева и контролируют правила вложенности процедур и целостность схемы.

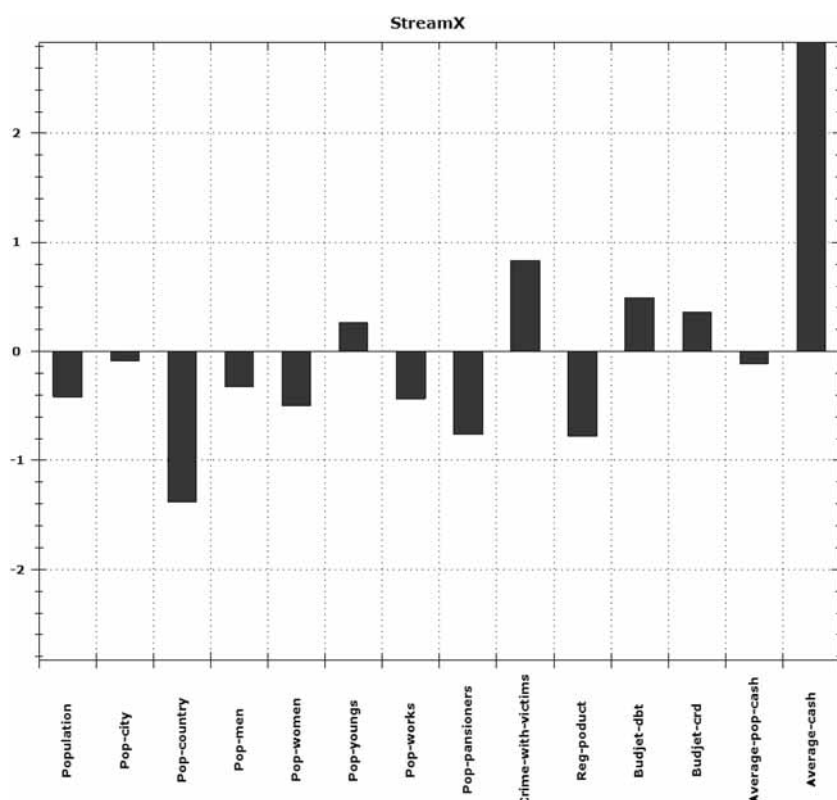


Рис. 6. Гистограмма средних по системе информативностей показателей, представленная в стандартизированном виде

На каждом этапе наращивания дерева процедур существует возможность посмотреть результат того или иного элементарного математического преобразования в виде таблицы данных или графического образа (например, гистограмм) для визуальной оценки логики работы расчетной схемы (рис. 4).

В программном комплексе реализован графический интерфейс для редактирования и сохранения правил графических элементов в XML-шаблоне САП (рис. 5).

Таким образом, разработанная технология расчетных схем и системных профилей обеспечивает единые и прозрачные правила работы серверного и клиентского модулей и снимает вопросы контроля логики анализа и целостности профиля при внесении изменений.

Пример использования программного комплекса "ЭнтропияПлюс" и языка метаописаний EAML в задачах анализа данных

Одним из весьма продуктивных направлений развития и применения EAML и его прикладной реализации является анализ социально-экономических систем. Так в одной из работ было проведено исследование состояния регионов Сибирского федерального округа (Челябинская область включена для фона) по набору показателей (табл. 2), в котором искались качественные отличия регионов в связи бюджетов, доходов и состава населения.

Результатом первой расчетной схемы (рис. 6) является гистограмма средних по системе информативностей показателей, представленная в стандартизированном виде.

Здесь видно, что экстремальное количество информации имеют: заработная плата — как ведущий информативный показатель; численность сельского населения — стратифицированный показателем с явными лидерами и аутсайдерами. На этом фоне подавленными, но различимыми оказываются аналогичные свойства численности пенсионеров и валового регионального продукта. Также существенной независимостью от остальных показателей отличается и уровень преступности.

Частные свойства некоторых регионов определены на фазовых портретах, представленных на рис. 7—9 (см. третью и четвертую стороны обложки), где r — коэффициент корреляции.

Радикально от регионов Сибирского федерального округа, в частности от Красноярского края, отличается высокой негативной связью фазовых координат Алтайский край (рис. 7, см. третью сторону обложки). В системе аналогичных бюджетных показателей в Алтайском

Социально-экономические показатели Сибирского федерального округа [5]

Регионы РФ (Объекты исследования)	Численность населения, тыс. чел (Population)	Численность городского населения, тыс. чел (Pop-city)	Численность сельского населения, тыс. чел (Pop-country)	Численность мужского населения, чел (Pop-men)	Численность женского населения, чел (Pop-women)	Численность молодежи, чел (Pop-youngs)	Численность трудоспособного населения, чел (Pop-works)	Численность нетрудоспособного населения, чел (Pop-pansioners)	Число преступлений против личности (Crime-with-victims)	Валовый региональный продукт, млн руб. (Reg-product)	Местный бюджет – доходы, млн руб. (Budget-dbt)	Местный бюджет – расходы, млн руб. (Budget-crd)	Среднедушевые денежные доходы (в месяц), руб. (Average-pop-cash)	Среднемесячная номинальная начисленная заработная плата, руб. (Average-cash)
Челябинская область	3508	2854	654	1 610 674	1 898 059	564 293	2 193 841	750 599	19294	582 945	52 976	53 822	14161,2	14829,2
Республика Бурятия	961	527	434	453 843	506 899	197 004	611 400	152 338	6649	109 554	16 432	16 662	11298,5	14417
Республика Тыва	314	161	153	148 432	165 508	90 644	193 649	29 647	2611	19 776	7492	7578	7871,2	13614,6
Республика Хакасия	538	382	156	250 280	287 774	96 314	343 128	98 612	2100	64 029	8782	9119	10763,9	14488,4
Алтайский край	2497	1341	1156	1 155 398	1 341 378	396 848	1 572 851	527 077	10 447	223 751	32 502	33 747	9748,6	9731,5
Забайкальский край	1117	711	406	533 777	583 253	228 575	712 211	176 244	5746	113 230	21 981	21 683	10971,5	15142,5
Красноярский край	2890	2187	703	1 346 790	1 542 995	485 996	1 872 172	531 617	13 608	734 414	94 636	98 225	15604,5	18934,7
Иркутская область	2505	1976	529	1 159 775	1 345 802	465 093	1 586 464	454 020	22 783	403 031	42 410	43 767	12881,6	17072,1
Кемеровская область	2822	2397	425	1 300 622	1 521 237	462 913	1 784 521	574 425	17 580	444 352	71 854	73 022	14439,3	15410
Новосибирская область	2640	1992	648	1 214 454	1 425 403	405 042	1 671 520	563 295	13 564	382 186	52 795	55 435	12838,1	15713,6
Омская область	2014	1397	617	928 300	1 085 835	325 597	1 289 921	398 617	8014	301 803	24 951	25 749	13626,5	13524,8
Томская область	1038	717	321	485 179	553 329	164 841	684 738	188 929	5490	216 059	18 953	19 972	13481,7	17675,3

крае наблюдаются заниженные показатели средней заработной платы и среднедушевого дохода. Выделяется регион почти аномально (на границе эллипса) высокой численностью сельского населения.

На фазовом портрете Красноярского края относительно системы выделяются среднедушевой доход и средняя заработная плата — это самый обеспеченный регион (рис. 8, см. четвертую сторону обложки). Высокий уровень линейной связи фазовых координат свидетельствует о подобию структуры показателей в регионе среднему по Сибирскому федеральному округу.

Как "средний" регион выглядит Кемеровская область (рис. 9, см. четвертую сторону обложки). Относительно системы она автономна, так как связь фазовых координат стремится к нулю. Вызвано это низкими показателями средней заработной платы и численности сельского населения относительно других показателей.

На портретах всех регионов была отмечена композиция (синхронное поведение) численности пенсионеров и валового национального продукта (объекты точки в виде оранжевого ромба). Эта особенность подтверждена и в среднем по системе. Трудно представить себе социально-экономические условия и связи, приводящие к такой зависимости. Более вероятной причиной видится некая схема составления отчетности, в которой численность пенсионеров определена как частное пенсионного фонда региона и средней пенсии, а размер пенсионного фонда как доля регионального продукта.

Поскольку фазовое пространство энтропийной модели имеет метрику, то все обладающие специфической соотношения изображающих точек показателей, а также их положения относительно границ легко формализуются с помощью аппарата аналитической геометрии. Как следствие, представляется возможность составления решающих правил аналитической (экспертной) системы, которые, будучи составленными и проверенными на одном наборе данных, легко переносятся на другие аналогичные по составу таблицы.

Заключение

Результатом работы является оригинальная технология разработки систем интеллектуального анализа

данных, основанная на метанаборах описаний модулей решающих и поведенческих правил, и ее прикладная реализация в виде языка EAML.

Заложенный в EAML подход на основе процедурной последовательности позволяет моделировать на уровне понятных терминов (в контексте работы, тэгов) решения не только энтропийного анализа, но и большинства других статистических методов.

Новый подход не только не умаляет достоинств известных языков искусственного интеллекта, но и гарантирует интеграцию различных вычислительных и экспертных модулей на программном уровне с языками C, CLIPS и подобными им. Он позволяет также расширить энтропийный метод в области применения других методов анализа (прикладной статистики, кластерного, корреляционного, регрессионного). Это обстоятельство, по мнению авторов, в совокупности может привести к созданию технологии конкурентной Data mining.

Результаты работы используются для создания междисциплинарного веб-сервиса по обработке и анализу данных в прикладных и научных исследованиях. Примером может служить открытый облачный сервис для диагностики состояния природных сейсмических генераторов (<http://seismatica.appspot.com>) авторов Попова С. Е. и Замараева Р. Ю.

Список литературы

1. Логов А. Б., Замараев Р. Ю., Логов А. А. Анализ состояния систем уникальных объектов // Вычислительные технологии. 2005. Т. 10. № 5. С. 49—53.
2. Логов А. Б., Замараев Р. Ю., Логов А. А. Моделирование тенденций поведения элементов систем уникальных объектов // Вычислительные технологии. 2005. Т. 10. № 5. С. 54—56.
3. Логов А. Б., Замараев Р. Ю., Логов А. А. Алгоритмы энтропийного метода анализа для отображения свойств объекта в фазовом пространстве // Вычислительные технологии. 2005. Т. 10. № 6. С. 75—81.
4. Логов А. Б., Замараев Р. Ю., Логов А. А. Анализ функционального состояния промышленных объектов в фазовом пространстве. Кемерово: Изд-во Института угля и углекислоты СО РАН, 2004. 168 с.
5. Сайт Федеральной службы государственной статистики. [Электронный ресурс]. URL: <http://www.gks.ru/dbscripts/munst/munst.htm>

В. Н. Лихачёв, канд. пед. наук, доц., Калужский университет им. К. Э. Циолковского,
e-mail: lvlad@rambler.ru,

С. А. Гинзгеймер, канд. физ.-мат. наук, доц.,
Калужский филиал Московского государственного технического университета
им. Н. Э. Баумана,
e-mail: ginzgeymer@mail.ru

Обработка ошибок ограничений для баз данных PostgreSQL

Предлагается авторская методика формирования информативных сообщений об ошибках, генерируемых сервером PostgreSQL, на основе анализа структуры базы данных, использования пользовательских названий таблиц и их полей, применения специальных сообщений уровня базы данных и приложения.

Данный подход позволяет сократить объем работ при разработке приложений, улучшить структуру приложений, упростить их сопровождение

Ключевые слова: базы данных, качество программного обеспечения, обработка ошибок, PostgreSQL

При создании программ, работающих с реляционными базами данных, важным является не только обработка ошибок базы данных, но и формирование сообщений об ошибках, понятных для конечного пользователя. К сожалению, данному вопросу практически не уделяется внимания, хотя формирование сообщений о таких ошибках часто может требовать довольно значительного объема работ.

Несмотря на то, что серверы реляционных баз данных при нарушении ограничений базы данных генерируют ошибки, текст этих ошибок часто не содержит достаточной информации для выявления их причины и устранения пользователем системы.

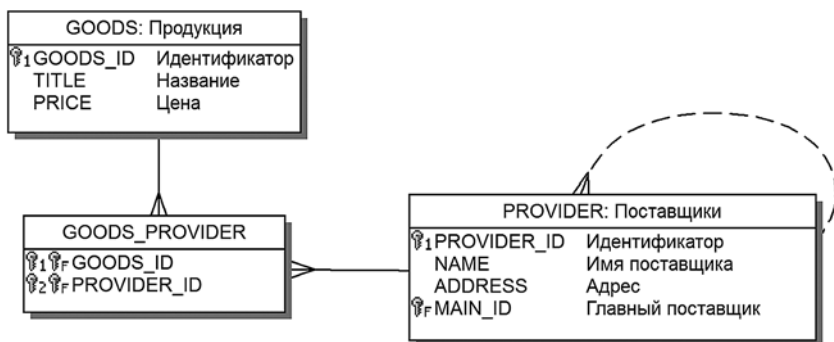
Правильно интерпретировать непосредственные сообщения от сервера баз данных обычно удается только ее разработчику или администратору, которые имеют необходимую квалификацию и потратили достаточное время для изучения структуры базы данных и алгоритмов ее работы. По этой причине разработчики баз данных и клиентских программ обычно программно формируют сообщения о наиболее частых ошибках, которые возникают при разработке и сопровождении базы и будут понятны обычному пользователю.

Затруднения при таком подходе заключаются в том, что число возможных ошибок базы данных возрастает с увеличением ее сложности, а формирование сообщений даже об однотипных ошибках часто ре-

ализуется индивидуально для каждой транзакции. В силу необходимости написания кода практически для каждой потенциально возможной ошибки, часть из них, в том числе те, о которых разработчику известно, оказываются без соответствующих сообщений для пользователя. Как правило, в таких ситуациях для пользователя выводится сообщение об ошибке общего характера или непосредственное сообщение от сервера баз данных, которые в большинстве случаев не могут помочь ему выявить и исправить причину ошибки.

Перечисленные выше причины приводят к тому, что число ошибок, для которых отсутствуют информативные сообщения, также возрастает с увеличением сложности базы данных. Формирование сообщений для каждой отдельной транзакции приводит к тому, что код для формирования сообщений об ошибках оказывается распределенным по всему приложению, что усложняет его сопровождение [1–3].

Решением, которое призвано устранить отмеченную трудность, является использование предлагаемого авторами данной статьи универсального подхода к формированию информативных сообщений об ошибках, генерируемых сервером PostgreSQL при нарушении ограничений базы данных. Такой подход должен позволять автоматически формировать информативные сообщения об ошибках для наиболее часто встречающихся случаев, вызванных ограничениями базы данных. В то же



Взаимосвязь между таблицами "многие ко многим", на рисунке указаны пользовательские названия таблиц и их полей

время он призван предоставлять разработчику возможность самостоятельно формировать информативные сообщения об ошибках при наличии такой необходимости. Разработка такого подхода связана с рядом сложностей, к числу которых относятся следующие.

- Отсутствие у пользователей достаточных знаний о структуре базы данных и особенностях хранения данных в ней, что требует создания довольно информативных сообщений для них.
- Зависимость содержания сообщения об ошибке от назначения программы. Даже для программ, работающих с одной и той же базой данных, может потребоваться формирование различных сообщений об одной и той же ошибке.
- Необходимость учета логической структуры базы данных при формировании сообщений об ошибках.
- Сложность автоматического формирования сообщений для некоторых ошибок, вызванных ограничениями базы данных, имеющих сложную логику, например, для ограничений проверки для таблиц.
- Использование в клиентских программах обозначений таблиц и полей (пользовательских названий), отличных от их имен в базе данных.

Для создания универсального метода формирования информативных сообщений о нарушениях ограничений базы данных, создаваемые сообщения необходимо разделить на две группы:

- универсальные сообщения — сообщения, которые могут быть сформированы на основе анализа структуры базы данных;
- специальные сообщения — сообщения об ошибках, которые определяются разработчиком индивидуально для каждой ошибки.

Формирование универсальных сообщений основано на том, что в информации об ошибке от сервера баз данных обычно указывается тип ошибки (обычно в виде кода ошибки) и название объекта базы данных, который является причиной появления ошибки. Такими объектами обычно являются различные

ограничения базы данных: уникальные и внешние ключи; уникальные индексы; ограничения NOT NULL и др. На основе этой информации об ошибке и информации о структуре базы данных, которая может быть получена из системного каталога базы, могут быть сформированы информативные сообщения об ошибке.

В клиентских программах, работающих с базами данных, обозначения таблиц и полей (пользовательские названия таблиц и полей) обычно отличаются от их имен в базе данных. Например, таблица базы данных может иметь имя "GOODS", а в клиентском приложении данные этой таблицы могут отображаться в справочнике с названием "Товары" или "Продукция" (см. рисунок).

В формируемом сообщении об ошибке для увеличения его информативности необходимы именно пользовательские названия таблиц и полей. Информация о пользовательских названиях таблиц и их полей может задаваться разработчиком базы данных и храниться в базе данных в отдельной таблице или как комментарии для таблиц и их полей. Последний вариант может оказаться более предпочтительным, чем использование для этого отдельной таблицы. При изменении имени поля или его удалении не потребуется дополнительно проводить изменения в отдельной таблице. Для получения пользовательских названий таблиц можно использовать запрос 1.

Запрос 1. Получение информации о пользовательских названиях таблиц

```

SELECT nc.nspname AS table_schema, c.relname AS table_name,
       ds.description AS table_description
FROM pg_namespace nc JOIN pg_class c ON nc.oid = c.relnamespace
LEFT JOIN pg_description ds ON ds.objoid = c.o`id and ds.objsubid = 0
WHERE nc.nspname <> 'information_schema' AND nc.nspname <> 'pg_catalog'
AND nc.nspname not like 'pg_temp%' AND (c.relkind = 'r' or c.relkind = 'v')
--AND ds.description is null or TRIM(ds.description) = "
    
```

В результате запрос возвращает информацию о всех таблицах текущей базы данных: table_schema — название схемы, в которую входит таблица; table_name — название таблицы; table_description — пользовательское название таблицы. Для получения

перечня таблиц без описания в запросе необходимо убрать комментарии с последней строки запроса.

Для получения информации о пользовательских названиях полей таблиц можно использовать запрос 2.

Запрос 2. Получение информации о пользовательских названиях полей таблиц

```
SELECT nc.nspname AS table_schema, c.relname AS table_name,  
       a.attname AS column_name, ds.description AS column_description  
FROM pg_namespace nc JOIN pg_class c ON nc.oid = c.relnamespace  
     JOIN pg_attribute a ON (a.attrelid=c.oid) AND (a.attnum > 0)  
     LEFT JOIN pg_description ds ON ds.objoid = a.attrelid AND a.attnum = ds.objsubid  
WHERE nc.nspname <> 'information_schema' AND nc.nspname <> 'pg_catalog'  
     AND nc.nspname NOT LIKE 'pg_temp%' AND (c.relkind='r' OR c.relkind='v')  
--AND ds.description is null or TRIM(ds.description) = "  
--AND nc.nspname =:table_schema AND c.relname =:table_name
```

В результате запрос возвращает информацию о полях таблиц текущей базы данных: `table_schema` — название схемы, в которую входит таблица; `table_name` — название таблицы; `column_name` — имя поля таблицы; `column_description` — пользовательское название таблицы. Для получения перечня таблиц без описания, в запросе необходимо убрать комментарии с предпоследней строки запроса. Для получения описания полей выбранной таблицы необходимо убрать комментарии с последней строки запроса. В этом случае также необходимо указать в качестве значений параметров `table_schema` и `table_name` имя схемы и таблицы.

Необходимость использования специальных сообщений может появиться в случае, если универсальное сообщение об ошибке по каким-то причинам не отражает реальную причину ошибки или не может быть сформировано. Примером последнего случая являются ограничения проверки для таблиц. В ограничениях проверки для таблиц могут использоваться разнообразные запросы и условия, анализ которых может оказаться довольно сложной задачей. Поэтому для таких ограничений необходимо использовать специальные сообщения, которые определяются на этапе разработки.

Можно выделить следующие две группы специальных сообщений об ошибках, генерируемых сервером баз данных PostgreSQL.

- Специальные сообщения уровня базы данных — сообщения предназначены для использования во всех приложениях, которые работают с общей базой данных.
- Специальные сообщения уровня приложения — сообщения, которые отражают специфику конкретного приложения. Они могут быть необходимы, когда различные приложения должны выдавать пользователю различные сообщения об одной и той же ошибке.

Информация о первой группе сообщений может храниться в базе данных. Вторая группа сообщений может храниться как в приложении (например, в виде ресурсов или в специальном файле), так и в базе данных с указанием приложения, для которого она предназначена.

Универсальные и специальные сообщения об ошибках позволяют реализовать гибкую схему формирования информативных сообщений об ошибках, возникающих при нарушении ограничений базы данных. Как уже было отмечено ранее, для каждой ошибки может использоваться несколько сообщений, а именно:

- специальное сообщение уровня приложения;
- специальное сообщение уровня базы данных;
- универсальное сообщение, формируемое на основе анализа структуры базы данных;

- исходное сообщение сервера баз данных.

Типы сообщений перечислены в порядке приоритета их выбора при появлении ошибки. Если для ошибки определено специальное сообщение, то необходимо вывести именно его. Специальное сообщение уровня приложения при этом имеет в этой схеме больший приоритет, чем специальное сообщение уровня базы данных. Если для ошибки не задано специальное сообщение (уровня базы данных или уровня приложения), то будет выведено универсальное сообщение об ошибке, которое формируется на основе анализа структуры базы данных. Если по каким-то причинам оно не может быть сформировано, то выводится исходное сообщение от сервера баз данных.

Описанная последовательность формирования универсальных и специальных сообщений об ошибках базы данных ориентирована на то, что для большинства ошибок могут быть сформированы универсальные сообщения на основе анализа структуры базы данных. Однако в случае ошибок, для которых эти сообщения недостаточно информативны, или для случаев, когда отдельные приложения требуют особых формулировок сообщений об ошибках, могут использоваться специальные сообщения уровня базы данных и уровня приложения.

Формирование универсальных сообщений об ошибках, генерируемых сервером PostgreSQL при нарушении ограничений базы данных

Универсальные сообщения могут быть сформированы для большинства ограничений базы данных. Далее рассмотрим формирование информативных сообщений для ошибок, которые генерируются сервером PostgreSQL при нарушении наиболее часто используемых ограничений в базах данных.

Не указано значение поля, обязательное для заполнения (ограничение NOT NULL). При нарушении ограничения NOT NULL сервер баз данных PostgreSQL формирует ошибку (табл. 1).

Как можно видеть, в этом сообщении не указывается имя таблицы, для поля которой нарушено ограничение NOT NULL.

Таблица 1

Информация об ошибке NOT NULL сервера PostgreSQL

Код ошибки	23502
Текст ошибки	null value in column "<имя поля>" violates not-null constraint

Возможно несколько вариантов разрешения этого вопроса, которые приводятся ниже.

- Использование уникальных названий полей в рамках всей базы данных. Для этого можно, например, использовать в названии имени поля таблицы короткий префикс из нескольких символов, уникальный для каждой таблицы в базе данных.

- Если формирование информативных сообщений выполняется в клиентском приложении, то для определения таблицы, к которой относится поле, возможен анализ запроса DML, при выполнении которого произошла ошибка, и получение информации об имени изменяемой таблицы из этого запроса.

В результате может быть сформировано сообщение, например следующего содержания: "Необходимо указать значение для свойства "Название" элемента справочника "Продукция!" (см. рисунок).

Нарушение ограничения уникальности. Необходимость ввода уникального значения поля может требоваться в основном в следующих случаях:

- поле входит в главный ключ;
- поле включено в уникальный ключ;
- поле входит в уникальный индекс.

Запрос 3. Получение информации об ограничении уникальности

```
SELECT ns.nspname schema_name, t.relname table_name,
       ds_table.description table_desc, c.relname constrand_name,
       CASE WHEN indisunique AND indisprimary THEN 'primary key'
            WHEN indisunique THEN 'unique index'
            ELSE 'index' END AS "type",
       a.attname field_name, ds_field.description field_desc
FROM pg_index i JOIN pg_class c ON (c.oid = i.indexrelid)
JOIN pg_class t ON (t.oid = i.indrelid)
JOIN pg_namespace ns ON (ns.oid = t.relnamespace)
JOIN pg_attribute a ON (t.oid=a.attrelid) AND (a.attnum = ANY (i.indkey))
LEFT join pg_description ds_field ON (ds_field.objoid = t.oid
AND ds_field.objsubid = a.attnum)
LEFT join pg_description ds_table
ON (ds_table.objoid = t.oid AND ds_table.objsubid = 0)
WHERE ns.nspname =:table_schema AND c.relname =: constrand_name
```

В качестве параметра запроса указывается имя схемы (table_schema) и ограничение уникальности (constrand_name). Запрос возвращает информацию о полях, входящих в ограничение уникальности: schema_name — имя схемы таблицы, table_name — имя таблицы, table_desc — комментарии таблицы, constrand_name — имя ограничения уникальности, type — тип ограничения уникальности, field_name — имя поля, field_desc — комментарий поля. Если ограничение уникальности содержит несколько полей, то запрос вернет несколько записей, каждая из которых будет содержать информацию об одном из полей, входящих в ограничение уникальности.

В результате может быть сформировано сообщение, например, следующего содержания: "Значение свойства "Название" элемента справочника "Продукция" должно быть уникальным!" (см. рисунок).

Таблица 2

Сообщение об ошибке при нарушении ограничения уникальности сервера PostgreSQL

Код ошибки	23505
Текст ошибки	duplicate key value violates unique constraint " <i><имя ограничения></i> "
Дополнительная информация	Key (<i><поле таблицы></i>)= <i><значение></i> already exists

При нарушении ограничения уникальности сервер PostgreSQL генерирует ошибку (табл. 2).

В тексте ошибки указывается ограничение, нарушение которого привело к генерации ошибки сервером PostgreSQL. В дополнительной информации об ошибке указывается поле таблицы, которое входит в ограничение уникальности, и его значение, которое вызвало ошибку. Если ограничение уникальности создано для группы полей, то в дополнительной информации перечисляются все поля ограничения и указываются их значения. Для получения информации об ограничении уникальности можно использовать запрос 3.

Нарушение ограничения внешнего ключа. Можно выделить следующие причины, приводящие к нарушению ограничений внешних ключей.

1. В подчиненную таблицу добавляется запись, в которой для поля внешнего ключа нет соответствующего значения в главной таблице. Аналогичная ситуация происходит при изменении значения поля подчиненной таблицы в случае, если нового значения поля нет в главной таблице.

2. Во внешней таблице выполняется попытка удаления данных, на которые имеется ссылка в подчиненной таблице. При этом в определении связи между таблицами указано ограничение "NO ACTION" для операции удаления данных. При такой связи сервер баз данных не позволяет удалять данные из главной

Таблица 3

Сообщение сервера PostgreSQL об ошибке ограничения внешнего ключа при добавлении или обновлении данных в подчиненной таблице

Код ошибки	23503
Текст ошибки	insert or update on table "<подчинённая таблица>" violates foreign key constraint "<внешний ключ>"
Дополнительная информация	Key (<поле главной таблицы>)=(<значение>) is not present in table "<главная таблица>"

таблицы, если в подчиненной таблице есть записи, связанные с удаляемой записью.

3. Ситуация, аналогичная п. 2, но только для операции изменения данных в главной таблице с ограничением "NO ACTION". Сервер баз данных в этом случае генерирует те же ошибки, что и при удалении данных.

При попытке добавления записи в подчиненную таблицу или изменения записи в ней со значением поля, которого нет в главной таблице, сервер PostgreSQL генерирует ошибку, описание которой приведено в табл. 3.

Таблица 4

Сообщение сервера PostgreSQL об ошибке ограничения внешнего ключа при удалении или обновлении данных в главной таблице

Код ошибки	23503
Текст ошибки	update or delete on table "<главная таблица>"@violates foreign key constraint "<внешний ключ>" on table "<подчиненная таблица>"
Дополнительная информация	Key (<поле главной таблицы>)=(<значение>) Is still referenced from table "<подчиненная таблица>"

При попытке удаления записи из главной таблицы, на которую ссылаются записи из подчиненной таблицы, сервер PostgreSQL генерирует ошибку, описание которой приведено в табл. 4.

Во всех перечисленных сообщениях от сервера баз данных указывается имя внешнего ключа, ограничения которого привели к появлению ошибки. Для получения информации о внешнем ключе можно использовать запрос 4.

Запрос 4. Получение информации о внешнем ключе

```
SELECT nc.nspname AS schema_name,
       c.relname AS table_name, ds.table.description AS table_desc,
       const.conname AS key_name, a.attname AS column_name,
       ds.description AS column_desc, ncf.nspname AS ref_schema,
       cf.relname AS ref_table, ds_tablef.description AS ref_table_desc,
       af.attname AS ref_column_name, dsf.description AS ref_column_desc,
       CASE confupdtype WHEN 'a' THEN 'no action' WHEN 'r' THEN 'restrict'
        WHEN 'c' THEN 'cascade' WHEN 'n' THEN 'set null'
        WHEN 'd' THEN 'set default' END AS update_action,
       CASE confdeltype WHEN 'a' THEN 'no action' WHEN 'r' THEN 'restrict'
        WHEN 'c' THEN 'cascade' WHEN 'n' THEN 'set null'
        WHEN 'd' THEN 'set default' END AS delete_action
FROM pg_constraint const JOIN pg_class c ON c.oid = const.conrelid
JOIN pg_class cf ON cf.oid = const.confrelid
JOIN pg_attribute a ON ((a.attrelid = c.oid)
AND (a.attnum = ANY (const.conkey)))
JOIN pg_attribute af ON ((af.attrelid = cf.oid) AND (af.attnum = ANY (const.confkey))
AND (idx(const.conkey, a.attnum) = idx(const.confkey, af.attnum)))
JOIN pg_namespace nc ON nc.oid = c.relnamespace
JOIN pg_namespace ncf ON ncf.oid = cf.relnamespace
LEFT JOIN pg_description ds ON ds.objoid = a.attrelid
AND a.attnum = ds.objsubid
LEFT JOIN pg_description dsf ON dsf.objoid = af.attrelid
AND af.attnum = dsf.objsubid
LEFT JOIN pg_description ds_table ON ds_table.objoid = c.oid
AND ds_table.objsubid = 0
LEFT JOIN pg_description ds_tablef ON ds_tablef.objoid = af.attrelid
AND ds_tablef.objsubid = 0
WHERE contype = 'f' AND nc.nspname =:schema_name
AND const.conname =:fkey_name AND c.relname =:table_name
```

В качестве параметров для запроса указываются: schema_name — имя схемы; table_name — имя таблицы; fkey_name — имя внешнего ключа таблицы.

Запрос возвращает следующие поля: schema_name — имя схемы таблицы; table_name и table_desc — имя и описание таблицы; key_name — имя внешнего ключа таблицы; column_name и column_desc — имя поля таблицы и его описание, входящие во внешний ключ;

ref_schema — имя схемы главной таблицы; ref_table и ref_table_desc — имя и описание главной таблицы; ref_column_name и ref_column_desc — имя и описание поля главной таблицы, входящего во внешний ключ; update_action, delete_action — правила обновления и удаления внешнего ключа.

Логические связи между таблицами. Кроме информации об ошибке, вызванной ограничением внешне-

го ключа, для формирования информативных сообщений необходимо наличие информации о логической связи между таблицами, для реализации которой используется внешний ключ.

Примером может являться использование внешнего ключа для ссылки на собственную таблицу. Понятно, что в этом случае сообщение об ошибке должно отличаться от случая, когда главная и подчиненная таблица являются различными. Другим примером является формирование сообщений для таблиц с логической связью "многие ко многим". Обычно такая связь между таблицами реализуется с помощью дополнительной таблицы, и внешние ключи непосредственно не связывают таблицы, между которыми реализуется логическая связь. Поэтому если использовать сообщения об ошибке, сформированные на основе только информации об ограничении между таблицами, то они, вероятнее всего, не смогут адекватно отразить причину ошибки.

Сообщения об ошибках для таблиц с логической связью "многие ко многим". Обычно такая связь реализуется с помощью дополнительной таблицы. В дополнительной таблице создаются внешние ключи, которые ссылаются на таблицы, между которыми и реализуется связь "многие ко многим". Чтобы избежать дублирования связей между таблицами, в дополнительной таблице на поля, которые входят в эти внешние ключи, накладывается ограничение уникальности (уникальный индекс, первичный или уникальный ключ). Оба внешних ключа дополнительной таблицы ссылаются на поля таблиц логической связи "многие ко многим", которые входят в их первичные ключи. Для создания новой связи между записями логически связанных таблиц в дополнительную таблицу добавляется новая запись, а для удаления имеющейся связи между записями таблиц с логической связью "многие ко многим" удаляется соответствующая запись из дополнительной таблицы. Это означает, что все изменения выполняются только в дополнительной таблице, и их правильность контролируется ограничениями этой таблицы.

Пример такой взаимосвязи между таблицами "GOODS" и "PROVIDER" представлен на рисунке. В качестве дополнительной таблицы, с помощью ко-

торой реализуется связь "многие ко многим", используется таблица "GOODS_PROVIDER". Для нее не указаны пользовательские названия таблиц и полей, так как эти данные не используются для формирования сообщений об ошибках для таблиц, участвующих в связи "многие ко многим".

Если произошла ошибка, вызванная ограничением внешнего ключа, то для выявления логической связи "многие ко многим" можно использовать ее характерные признаки, к числу которых относятся перечисленные далее.

- Внешний ключ, ограничение которого вызвало ошибку, входит также в ограничение уникальности (уникальный индекс, первичный или уникальный ключ). Это соответствует структуре дополнительной таблицы, используемой для реализации логической связи "многие ко многим".

- В это ограничение уникальности дополнительной таблицы входят поля другого внешнего ключа. Другие поля в это ограничение уникальности дополнительной таблицы не входят. Таблицы, на которые ссылаются эти внешние ключи, связаны логической связью "многие ко многим".

Для таблиц, участвующих в логической связи "многие ко многим", можно выделить несколько отмеченных далее ситуаций, которые могут приводить к появлению ошибок.

1. Делается попытка создания связи для несуществующей записи одной из таблиц, участвующих в логической связи "многие ко многим". Это соответствует ситуации, когда выполняется попытка добавления в промежуточную таблицу записи, значения полей которой нарушают ограничение одного из внешних ключей. Для формирования сообщения об ошибке сначала необходимо определить наличие логической связи "многие ко многим" между таблицами, так как в общем случае о наличии логических связей между таблицами может быть ничего не известно. Для получения информации о внешнем ключе, вызвавшем ошибку, можно использовать приведенный выше запрос 4. Для выявления внешних ключей, которые связаны ограничением уникальности, можно использовать запрос 5.

Запрос 5. Получение информации о внешних ключах, связанных ограничением уникальности

```
SELECT DISTINCT ns.nspname schema_name, t.relname table_name, c.relname constraint_name,
CASE WHEN indisunique AND indisprimary THEN 'primary key'
WHEN indisunique THEN 'unique index' ELSE 'index'
END AS constraint_type, f1.conname AS fkey1, f2.conname AS fkey2
FROM pg_index i, pg_constraint f1, pg_constraint f2, pg_class t, pg_class c, pg_namespace ns
WHERE (i.indisunique or i.indisprimary)
AND ((string_to_array(array_to_string(i.indkey, ','), ',') @> string_to_array(array_to_string(f1.conkey, ','), ','))
AND (f1.contype = ANY (ARRAY['f'])) AND (i.indrelid = f1.conrelid))
AND (string_to_array(array_to_string(i.indkey, ','), ',') @> string_to_array(array_to_string(f2.conkey, ','), ','))
AND (f2.contype = ANY (ARRAY['f']))
AND (i.indrelid = f2.conrelid) AND f1.oid <> f2.oid
AND f1.conkey <> f2.conkey AND (string_to_array(array_to_string(i.indkey, ','), ',') =
(string_to_array(array_to_string(f1.conkey, ','), ',') ||
string_to_array(array_to_string(f2.conkey, ','), ',')))
AND (t.oid = i.indrelid) AND (c.oid = i.indexrelid)
AND ns.oid = c.relnamespace AND ns.nspname =:schema_name
AND c.relname =:constraint_name
```

В качестве параметров запроса необходимо указать ограничение уникальности (`constraint_name`) и имя схемы (`schema_name`), в которой оно создано. Запрос возвращает имена внешних ключей (`fkey1` и `fkey2`) дополнительной таблицы, которые используются для связи таблиц.

В качестве варианта сообщения об ошибке можно использовать, например такой: "Нельзя связать элемент из справочника "Товары" с несуществующим элементом справочника "Поставщики" (см. рисунок).

2. В одной из таблиц логической связи "многие ко многим" выполняется удаление записи, которая связана с записью другой таблицы логической связи. Если для операции удаления внешнего ключа установлено свойство "NO ACTION", то сервер сгенерирует ошибку, вызванную ограничением этого внешнего ключа.

Для определения взаимосвязи между таблицами можно использовать последовательность действий, описанную выше. После получения данных о наличии логической связи между таблицами можно сформировать сообщение об ошибке, например, следующего содержания: "Нельзя удалить элемент справочника "Поставщики", так как он связан с одним или несколькими элементами справочника "Товары" (см. рисунок).

3. Выполняется попытка дублирования связи для записей таблиц, участвующих в логической связи "многие ко многим". В этой ситуации в дополнительную таблицу добавляется запись, которая будет нарушать ограничение уникальности дополнительной таблицы, в которое входят поля внешних ключей. Для выявления этой ситуации при возникновении ошибки нарушения ограничения уникальности необходимо выявить связь между таблицами "многие ко многим" и сформировать соответствующее сообщение для пользователя, например следующего содержания: "Связь между выбранными элементами справочников "Товары" и "Поставщики" уже существует" (см. рисунок).

Сообщения об ошибках для таблиц с логической связью "один ко многим". Можно выделить несколько задач, для решения которых применяется логическая связь "один ко многим":

- реализуется внешний ключ, ссылающийся на ту же самую таблицу (рекурсивно).
- главная таблица используется для ограничения значений, вводимых в поле или несколько полей подчиненной таблицы.
- связь реализуется между таблицами для повышения степени нормализации базы данных.

Отражение в сообщениях об ошибках специфики использования логической связи "один ко многим" позволяет в ряде случаев повысить их информативность. К числу таких случаев относятся следующие.

1. Внешний ключ, ссылающийся на собственную таблицу. В этой ситуации в таблице создается внешний ключ со ссылкой на другое поле той же самой таблицы. По этому признаку данную ситуацию можно отличить от других случаев использования внешнего ключа. Для такого типа внешних ключей могут воз-

никать все варианты ошибок, которые характерны для внешних ключей.

2. Внешний ключ используется для ограничения значений подчиненной таблицы. Можно выделить два варианта реализации, которые используются для этого:

- внешний ключ осуществляет ссылку на поля главной таблицы, которые не входят в ее первичный ключ;
- внешний ключ осуществляет ссылку на поля главной таблицы, входящие в ее первичный ключ.

Выбор варианта реализации, конечно же, будет определяться разработчиком базы данных. Если первый вариант позволяет выявить цель использования внешнего ключа, то при использовании второго варианта выявить назначение внешнего ключа без дополнительной информации практически невозможно, так как ссылка на поля первичного ключа главной таблицы используется во многих других случаях применения внешнего ключа.

Выходом из создавшейся неопределенности может быть использование специальных сообщений об ошибках или хранение в базе данных дополнительной информации о назначении внешнего ключа.

Независимо от того, как реализована связь между таблицами, при возникновении ошибки для пользователя, вероятнее всего, будет важна информация о том, чем ограничиваются значения полей в подчиненной таблице. По этой причине в отличие от случая формирования сообщения для таблиц с логической связью "многие ко многим", в сообщении для пользователя будет необходима информация о полях главной таблицы, которые используются для ограничения значений подчиненной таблицы.

3. Внешний ключ используется для увеличения степени нормализации базы данных. Характерной особенностью применения внешних ключей для повышения нормализации базы данных является использование во внешнем ключе полей главной таблицы, составляющих ее первичный ключ. Для пользователя в такой ситуации обычно важна информация о взаимосвязи записей этих таблиц, но не о способе ее реализации. Поэтому в сообщении об ошибке, как и в сообщениях для таблиц с логической связью "многие ко многим", обычно достаточно указать только таблицы, между которыми реализуется такая связь.

Для получения информации о внешнем ключе можно использовать запрос 5. Поле `constraint_type` этого запроса содержит информацию о типе ограничения, которое используется в главной таблице внешнего ключа для его реализации.

Сообщения об ошибках для таблиц с логической связью "один к одному". Обычно такая логическая связь используется в ситуации, когда данные, которые могут быть, в общем-то, сохранены в одной записи таблицы, в силу требований проектирования базы данных необходимо хранить в разных таблицах. Это может быть вызвано необходимостью повышения производительности базы данных. Подчиненная таблица в этом случае обычно содержит дополнительные данные и ссылается на поля первичного ключа главной таблицы. Дополни-

тельным условием для подчиненной таблицы, которое определяет отношение "один к одному", является ограничение уникальности для полей такой таблицы, которые входят во внешний ключ.

Одна из основных сложностей автоматического формирования информативных сообщений для ошибок ограничений внешних ключей связана с выявлением логических связей между таблицами, для создания которых используются внешние ключи. Ситуация осложняется так же тем обстоятельством, что в ряде случаев для создания логических связей используются ограничения уникальности. В результате одной из основных задач при автоматическом формировании информативного сообщения об ошибке, генерируемой сервером PostgreSQL, является выявление логических связей между таблицами, для реализации которых могут использоваться внешние ключи и ограничения уникальности.

Возможности формирования информативных сообщений на стороне сервера и на стороне клиента базы данных

Необходимо обратить внимание, что формирование информативных сообщений об ошибках может выполняться как на стороне клиента, так и на стороне сервера базы данных. Для этого необходима информация о коде ошибки и текстовое сообщение об ошибке от сервера баз данных. На стороне сервера эта информация может быть получена через переменные SQLSTATE и SQLERRM. Переменная SQLSTATE сохраняет информацию о коде ошибки, переменная SQLERRM — текст сообщения об ошибке.

На стороне клиента для большинства ошибок также доступна детальная информация об ошибке, в качестве которой во многих случаях передается инфор-

мация о значении полей, которые стали причиной генерации ошибки сервером PostgreSQL. К сожалению, данная информация в переменной SQLERRM недоступна (для версии PostgreSQL 9.1 и ниже) и получение этой информации возможно только на стороне клиента PostgreSQL.

В базе данных PostgreSQL имена внешних ключей и ограничений проверок для таблиц не являются уникальными для базы данных, поэтому при получении информации о них необходимо указывать таблицу к которой они относятся.

В сообщениях сервера PostgreSQL об ошибках не указывается имя схемы к которой относятся таблицы и ограничения, при этом в различных схемах имена таблиц и других объектов базы данных могут совпадать. Для того чтобы можно было различать таблицы при анализе сообщений от базы данных необходимо чтобы имена таблиц в пределах всей базы данных были уникальными.

Сервер PostgreSQL позволяет создавать комментарии не только для таблиц и их полей, но и для других объектов базы данных, таких как ограничения проверки для таблиц, внешние ключи, главные и уникальные ключи, индексы. Их комментарии можно использовать для хранения специальных сообщений уровня базы данных и уровня приложений.

Список литературы

1. Лихачев В. Н. Сообщения об ошибках ограничений внешних ключей на примере БД Firebird // RSDN Magazine. 2009. № 2 (URL: <http://www.rsdn.ru/article/db/FKeyErrors.xml>)
2. Лихачев В. Н. Обработка ошибок и формирование сообщений об ошибках для баз данных Microsoft SQL Server 2005/2008 (URL: <http://itband.ru/2010/09/sql-error/>)
3. Лихачев В. Н. Особенности обработки ошибок сервера базы данных Oracle // Oracle Magazine Rus: электронный журнал. 2009. № 6 (URL: http://citforum.ru/database/oracle/error_handling/)

ИНФОРМАЦИЯ

*Продолжается подписка на журнал
"Программная инженерия"
на второе полугодие 2012 г. и первое 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

Е. В. Ерофеев, руководитель группы разработки, Группа компаний "АйТи", Москва,
e-mail: Eerofeev@it.ru,

И. Е. Пелепелин, канд. физ.-мат. наук, вед. науч. сотр., Национальный исследовательский университет "Высшая школа экономики",
e-mail: IPelepelin@blogic20.ru

Анализ производительности протокола CMIS на примере его реализации в Alfresco и IBM FileNet

Обсуждаются результаты тестирования скорости выполнения базовых операций работы с документами, реализованных на основе протокола CMIS версии 1.0, в сравнении с использованием Native API хранилищ ECM-систем Alfresco и IBM FileNet. Результаты показывают значительное снижение производительности при использовании CMIS. Даны практические рекомендации по определению границ, в рамках которых целесообразно использование протокола CMIS.

Ключевые слова: CMIS, производительность, ECM-система, Alfresco, IBM FileNet

Введение

Открытый стандарт CMIS [1] призван обеспечить интероперабельность веб-сервисов по отношению к существующим системам класса *Enterprise content management* (ECM) и хранилищам неструктурированной информации, в первую очередь для систем-лидеров, которые определяются согласно исследованиям Гартнера [2]. Разработанный стандарт позволяет поддерживать одновременную работу с большим числом хранилищ, делая доступным контент, который уже имеется на предприятии. Таким образом, он призван снизить ограничения, связанные с невозможностью работы одновременно с несколькими хранилищами неструктурированных данных в мультивендорных и мультирепозитарных средах крупных организаций, для которых типично наличие сразу нескольких ECM-систем. По данным опросов [3], если бы ECM-решения могли интегрироваться со сторонними информационными системами, они играли бы значительно более важную роль на производстве.

В течение последних двух лет CMIS был реализован на ряде промышленных ECM-платформ, таких как Alfresco, EMC Documentum, IBM FileNet, MS SharePoint. Однако для того чтобы начать использовать CMIS в промышленных разработках, необходимо убедиться, что конкретная его реализация на конкрет-

ной ECM-платформе обеспечивает необходимые параметры производительности.

В данной статье приведены новые результаты исследований, на основании которых каждый желающий сможет сделать для себя определенные выводы о применимости конкретных реализаций протокола CMIS для работы с документами в хранилищах неструктурированной информации.

В настоящее время в сообществе разработчиков сложилось общее понимание того, что реализация протокола CMIS на базе основных ECM-платформ не позволяет добиться нужной эффективности по быстродействию. Однако количественных результатов сравнительного анализа, подтверждающих это мнение, пока никто не опубликовал. В настоящей статье представлены именно количественные результаты, позволяющие устранить этот недостаток.

Особенностью проведенных исследований является то обстоятельство, что сравнение проводилось, прежде всего, между реализацией протокола CMIS на промышленной ECM-платформе и эталонной реализацией тех же самых функций с использованием "родного" программного интерфейса данной ECM-платформы, так называемого *"Native API"*. Кроме этого, в статье представлены замеры быстродействия работы исследуемых функций в зависимости от типа опера-

ционной системы (сравнивались MS Windows и Linux).

Необходимо особо подчеркнуть, что в рамках данного исследования не проводилось сравнительного тестирования производительности CMIS между платформами Alfresco и IBM FileNet. Полученные в ходе тестирования результаты для этих двух платформ также не подлежат сравнению, поскольку тестовые узлы, на которых было развернуто программное обеспечение Alfresco и IBM FileNet, имеют разную конфигурацию аппаратной части с разной производительностью. В связи с этим любое сравнение между ними является некорректным.

Постановка задачи

Для того чтобы ответить на вопрос о границах применимости протокола CMIS, потребовалось провести ряд тестов и получить фактические данные, которые можно анализировать и делать соответствующие выводы. Работа по проведению серии тестов включала в себя, прежде всего, развертывание инфраструктуры двух стендов (узлов): Alfresco и IBM FileNet. Следует еще раз отметить, что аппаратное обеспечение этих стендов было разным. По этой причине сравнительных тестов между Alfresco и IBM FileNet не проводилось.

Для того чтобы обеспечить сравнимость результатов тестов в рамках одной ЕСМ-системы, было разработано специальное программное обеспечение, позволяющее легко переключаться между режимами тестирования, реализуя тем самым единый программный код для всех вариантов, а также корректность сравнения полученных результатов. Тесты проводили на двух массивах текстовых документов. Первый массив содержал 10 документов, второй массив — 1000 документов. Время, которое было затрачено на реализацию каждой отдельной операции при работе с документами хранилища, измеряли в миллисекундах. Для получения общей картины, характеризующей производительность реализации протокола CMIS в каждом из вариантов, были протестированы следующие основные операции по работе с документами хранилища.

- Создание документа с атрибутами. Под документом хранилища здесь подразумевается предметно-ориентированная сущность, которая содержит характеризующие ее атрибуты (поля).

- Создание документов с контентом. В качестве контента здесь может быть любой бинарный файл, например, изображение в каком-либо формате, файл в формате MS Word и т.п.

- Создание связей между документами. Связи могут быть как атрибутивные¹, так и ассоциативные².

¹ Атрибутивная связь — связь, при которой один документ имеет атрибут, ссылающийся на другой документ. Например, документ "Поручение" имеет атрибут "Исполнитель", который ссылается на документ "Сотрудник".

² Ассоциативная связь — связь, при которой документ имеет ссылки на документы без привязки к конкретному атрибуту.

Тип связи зависит от конкретной реализации хранилища. Протокол CMIS поддерживает оба вида связи, но жестко не регламентирует каким из них пользоваться. Под созданием связи подразумевались следующие действия: получение документа; присвоение (атрибуту или помещению в коллекцию связанных) идентификатора другого документа; сохранение.

- Получение документов с атрибутами — извлечение из хранилища карточки документа со всеми его полями.

- Получение документов с контентом — извлечение из хранилища документа с вложенным бинарным файлом.

- Получение связей между документами — извлечение из хранилища информации о связях между документами.

Производительность реализованного протокола CMIS сравнивалась с производительностью Native API, т.е. "родных" интерфейсов, встроенных в ЕСМ-системы Alfresco и IBM FileNet. Для Alfresco, кроме того, было проведено сравнительное тестирование одних и тех же функций работы с документами для случаев, когда сервер приложений Alfresco работает под управлением операционных системы Windows и Linux.

Основная задача, которая решалась данным исследованием, заключалась в необходимости определения возможных потерь и рисков неблагоприятных событий, связанных со снижением производительности при использовании протокола CMIS в собственных разработках. Речь идет о потерях, которые могут возникнуть:

- на стороне протокола SOAP при интенсивной передаче данных, например, при экспорте/импорте больших массивов документов;

- на стороне ЕСМ-системы, при неэффективном преобразовании понятий протокола CMIS во внутренние понятия ЕСМ-системы и обратно.

Для решения поставленной задачи необходимо было провести замеры быстродействия выполнения следующих основных операций при работе с документами с точки зрения программного интерфейса: создание документа, извлечение документа из хранилища, создание связей между документами и извлечение этих связей из хранилища.

Конфигурация стенда и программная реализация

Для проведения экспериментов по замеру скорости выполнения функций работы с хранилищем документов, перечисленных выше, был собран программно-аппаратный комплекс, состоящий из двух узлов.

Узел тестирования CMIS в реализации Alfresco состоял из двух независимых серверов, параметры которых представлены в табл. 1.

Узел тестирования CMIS в реализации IBM FileNet состоял из трех взаимосвязанных серверов (табл. 2).

Состав узла тестирования CMIS в реализации Alfresco

Параметр	Сервер Alfresco (Windows)	Сервер Alfresco (Linux)
Операционная система	Windows XP 32бит	CentOS ver. 5.5
Оперативная память	4 Гбайта	4 Гбайта
Дисковое пространство	40 Гбайт	20 Гбайт
Процессор	2-ядерный Intel Xeon Core I7	2-ядерный Intel Xeon Core I7
Прикладное ПО	Alfresco v3.4.0 (d 3370), Tomcat 6.0.26, MySQL	Alfresco v3.4.0 (d 3370), Tomcat 6.0.26, MySQL

Состав узла тестирования CMIS в реализации IBM FileNet

Параметр	Сервер FileNet	Сервер — контроллер домена	Сервер СУБД
Операционная система	Windows Server 2008 R2 64бит	Windows Server 2008 R2 64бит	Windows 7 64бит
Оперативная память	8 Гбайт	4 Гбайт	4 Гбайта
Дисковое пространство	40 Гбайт	40 Гбайт	40 Гбайт
Процессор	2-ядерный Intel Xeon Core I7	2-ядерный Intel Xeon Core I7	2-ядерный Intel Xeon Core I7
Прикладное ПО	IBM FileNet P8	—	Microsoft SQLServer 2008 R2

Для проведения тестов на языке Java было разработано универсальное клиентское приложение, которое в зависимости от заданных параметров обращается либо к хранилищу документов Alfresco, либо к IBM FileNet. Также в зависимости от заданных параметров используется либо интерфейс протокола CMIS, либо Native API соответствующего хранилища.

Каждый из рассмотренных вендоров реализовал протокол CMIS с отклонениями от спецификации стандарта [1]. Так, например, в случае работы с реализациями CMIS для IBM FileNet и Alfresco пришлось учитывать специфику именования типов и работы со связями для каждой из платформ.

Что касается Native API, то различий, с точки зрения реализации, оказалось гораздо больше, поскольку вендоры не связаны никакими стандартами. Тем не менее с точки зрения выполняемых функций работы с хранилищем документов, Native API реализуются сходным образом.

Следует отметить, что при выполнении операции извлечения документов из хранилища через CMIS возвращаются только указатели на документы, но нет ни атрибутов, ни вложений (контента). Для того чтобы получить атрибуты, необходимо обратиться к каждому документу в цикле, что увеличивает общие затраты по времени. В противоположность к этому подходу, при получении документов через Native API возвращается коллекция документов сразу с атрибутами. Ниже будет дано разъяснение положительных и отрицательных аспектов этих двух подходов.

Полученные результаты и их анализ

В ходе экспериментов, помимо двух разных ECM-платформ, исследовалось также влияние на результат используемой на сервере операционной системы. В частности, система Alfresco была развернута на серверах с операционной системой как Windows, так и Linux. Несмотря на тот факт, что тесты проводились на массивах, состоящих из 10 и 1000 документов, различия в результатах, пересчитанных на один документ, оказались незначительными. По этой причине в настоящей статье приведены результаты только для массива из 10 документов.

В табл. 3 представлены результаты измерений выполнения функций по работе с документами при обращении к хранилищу Alfresco, развернутому на базе операционной системы Windows XP. Результаты получены на коллекции из 10 тестовых документов и представлены в трех столбцах. В столбце CMIS показано время (мс) при обращении через протокол CMIS. В столбце Native API приведены результаты, полученные при обращении через API системы Alfresco. В столбце Сравнительный анализ представлено отношение (частное от деления) времени из столбца CMIS ко времени из столбца Native API. Для сравнения в табл. 4 представлены аналогичные результаты тестов по производительности, но уже для хранилища Alfresco, которое работает под управлением Linux (параметры сервера представлены выше в табл. 1). Данные результаты также получены на коллекции из 10 тестовых документов.

Таблица 3

Результаты тестов для Alfresco (Windows)

Операция над документом	Параметр измерения	CMIS, мс	Native API, мс	Сравнительный коэффициент
Создание документов с атрибутами	Среднее время выполнения теста для одного документа	1279,2	133,1	9,61
Создание документов с контентом		1322,9	352,8	3,75
Создание связей между документами	Среднее время выполнения теста для одной связи	1637,4	313,4	5,22
Получение документов с атрибутами	Среднее время выполнения теста для одного документа	322,7	40,3	8,01
Получение документов с контентом		170,2	149,1	1,14
Получение связей между документами	Среднее время выполнения теста для одной связи	236,3	80,6	2,93

Таблица 4

Результаты тестов для Alfresco (Linux)

Операция над документом	Параметр измерения	CMIS, мс	Native API, мс	Сравнительный коэффициент
Создание документов с атрибутами	Среднее время выполнения теста для одного документа	812	79,3	10,24
Создание документов с контентом		138,8	134,2	1,03
Создание связей между документами	Среднее время выполнения теста для одной связи	1574,8	148,8	10,58
Получение документов с атрибутами	Среднее время выполнения теста для одного документа	326	22,9	14,24
Получение документов с контентом		280,8	97,1	2,89
Получение связей между документами	Среднее время выполнения теста для одной связи	415,4	47,9	8,67

В табл. 5 представлены результаты сравнительного тестирования скорости выполнения базовых операций с документами из хранилища FileNet, которое развернуто под операционной системой Windows. Параметры данного стенда были приведены в табл. 2.

На рис. 1 (см. вторую сторону обложки) представлены данные по производительности, взятые из табл. 3 и 5. Обе платформы развернуты под операционной системой Windows. Для каждой из шести функций работы с документами показано по четыре результата:

- первая колонка — CMIS в реализации FileNet;
- вторая колонка — Native API FileNet;
- третья колонка — CMIS в реализации Alfresco;
- четвертая колонка — Native API Alfresco.

На рис. 1 хорошо видно, что производительность CMIS одинаково сильно (для Alfresco в среднем в 8 раз, для FileNet в 3 раза) проигрывает по сравнению с производительностью Native API. Наиболее затратными являются функции, связанные с созданием документов, где снижение производительности получено примерно в 10 и 4 раза соответственно.

На рис. 2 (см. вторую сторону обложки) представлены данные по производительности, взятые из

табл. 3 и 4. Одна и та же ECM-платформа Alfresco была развернута под операционными системами Linux и Windows, и для каждой из шести функций работы с документами также показано по четыре результата:

- первая колонка — CMIS в реализации Alfresco под Linux;
- вторая колонка — Native API Alfresco под Linux;
- третья колонка — CMIS в реализации Alfresco под Windows;
- четвертая колонка — Native API Alfresco под Windows.

По результатам тестирования хорошо видно, что функции работы с документами, реализованные через CMIS, по производительности значительно проигрывают аналогичной реализации, полученной через Native API ("родной" для ECM-платформы программный интерфейс). В случае Alfresco отличие почти на порядок, в случае FileNet разница меньше, всего в 3—4 раза, однако тоже заметная. Наибольшая разница наблюдается в операциях по созданию документа, созданию связи между документами и извлечению документа из хранилища вместе с его атрибутами.

Существенным может являться тот факт, что при использовании CMIS, в случае, когда необходимо по-

Результаты тестов для IBM FileNet (Windows)

Операция над документом	Параметр измерения	CMIS, мс	Native API, мс	Сравнительный коэффициент
Создание документов с атрибутами	Среднее время выполнения теста для одного документа	264,7	60,2	4,40
Создание документов с контентом		530,2	191,2	2,77
Создание связей между документами	Среднее время выполнения теста для одной связи	591,9	138	4,29
Получение документов с атрибутами	Среднее время выполнения теста для одного документа	74,2	23,9	3,10
Получение документов с контентом		75,1	52,9	1,42
Получение связей между документами	Среднее время выполнения теста для одной связи	119,5	53,1	2,25

лучить список документов, сначала нужно получить перечень указателей на искомые документы, а затем в процессе перечисления поэлементно "подтянуть" необходимые данные по атрибутам. В противовес этому, в случае использования Native API вместе с запуском запроса на получение списка документов сразу же приходят данные по атрибутам.

Пакетная (постраничная) загрузка данных оказывается эффективнее при большом числе результатов запроса. Причина в том, что здесь важно не только быстро выполнить запрос, но и быстро отобразить его результаты, т.е. вывести список документов, для чего требуется подгрузить их атрибуты. В этом случае Native API оказывается в 3–10 раз быстрее CMIS.

Неожиданные результаты показали тесты для одного и того же хранилища, но под разными операционными системами. Так, в частности, для платформы Alfresco, развернутой под Linux, функция создания контента документа через CMIS работает также быстро, как и в случае обращения через Native API. При этом по остальным тестируемым функциям особых различий между операционными системами не наблюдалось.

Выводы

Представленные результаты тестовых экспериментов наглядно показывают, что производительность выполнения основных базовых операций по работе с документами из хранилища, реализованных на основе версии протокола CMIS 1.0, значительно хуже реализации через "родной" программный интерфейс (Native API).

В связи с этим использование CMIS целесообразно в кросс-платформенных приложениях, где необходимо обеспечить максимальную независимость от вендора, где не возникает потребность в работе с большими списками документов и/или не нужно обрабатывать большие массивы документов. В приложениях, где наоборот нужно обеспечить максимальную

эффективность и быстродействие, рекомендуется использовать Native API, или самостоятельно разрабатывать адаптеры, скрывающие от сервисов более высокого уровня особенности обращений к хранилищу ESM-платформы различных вендоров.

Заключение

В статье описана методика тестирования и представлен анализ результатов тестирования основных базовых функций работы с документами и хранилищем документов через протокол CMIS. Результаты позволяют определить границы разумного использования протокола CMIS. Вместе с этим показано, что для ряда задач, где быстродействие становится наиболее критичным параметром, могут быть использованы функции, реализованные через Native API.

Принимая во внимание общий вектор развития систем управления контентом, использование CMIS приведет к тому, что ESM-хранилища станут в большей степени отчуждаемы от конкретной ESM-платформы. Вероятно, крупные вендоры хорошо понимают, что этот факт ослабляет их связи с корпоративными заказчиками. В силу этого они пока не стремятся вкладывать средства в развитие реализации CMIS на своих платформах. В связи с этим не следует ожидать быстрых изменений в производительности реализаций протокола CMIS [4]. С точки зрения функциональности, CMIS, безусловно, не покрывает все возможности, которые дает использование Native API. Однако стандарт развивается и готовится к выходу версия 1.1, а также идет работа над версией 2.0. При этом остается в силе тот факт, что качество и эффективность реализации CMIS целиком и полностью зависят от вендора, и это в первую очередь влияет на производительность приложений, работающих на протоколе CMIS.

Вместе с тем CMIS не обойден вниманием также и Open Source сообщества. Реализация проекта Apache Chemistry [5] позволяет всем вендорам JCR-совместимых приложений поддерживать CMIS.

Низкая производительность CMIS и, соответственно, слабый интерес к разработкам на основе CMIS косвенно подтверждают локальные опросы пользователей ЕСМ-систем, проведенные корпорацией Generis в 2011 и 2012 гг. Их результаты свидетельствуют о том, что знание пользователей о протоколе CMIS в течение года не изменилось и, скорее, пошло на убыль [6]. Вместе с тем этот факт может трактоваться как традиционный для вывода на рынок новой технологии этап спада "информационного шума", который следует за "пиком завышенных ожиданий" и предшествует обычно наступлению зрелости технологии.

Данное исследование осуществлялось при финансовой поддержке Правительства Российской Федерации (Минобрнауки России) в рамках договора № 13.G25.31.0096 о "Создании высокотехнологичного производства кросс-платформенных систем обработки неструктурированной информации на основе свободного программного обеспечения для повышения эффективности управления

инновационной деятельностью предприятия в современной России".

Список литературы

1. **OASIS** Standard Specification. Content Management Interoperability Services (CMIS) Version 1.0. 2010. URL: <http://docs.oasis-open.org/cmisis/CMIS/v1.0/os/cmisis-spec-v1.0.pdf>
2. **Ваулин В.** Оправдает ли CMIS надежды рынка СЭД? URL: <http://www.cnews.ru/reviews/free/dms2010/articles/articles11.shtml>.
3. **Waldhauser S.** 5 myths about the CMIS standard. URL: <http://www.digitallandfill.org/2011/08/5-myths-about-the-cmisis-standard.html>.
4. **Apache** Software Foundation. Apache Chemistry. URL: <http://incubator.apache.org/projects/chemistry.html>.
5. **Bell T., Shegda K. M., Gilbert M. R., Chin K.** Magic Quadrant for Enterprise Content Management. Gartner, 2010. URL: <http://www.ecmforhighereducation.com/wp-content/uploads/2011/09/Gartner-MQ-report-2010.pdf>.
6. **Content** Management Plans for 2012. A Short Survey on CMS and CMIS. Generis. February 2012. URL: [http://www.generis-corp.com/docs/CMIS and CMS 2012 Plans — Survey Responses.pdf](http://www.generis-corp.com/docs/CMIS%20and%20CMS%202012%20Plans%20-%20Survey%20Responses.pdf)

Software Quality Assurance Days

30 ноября - 1 декабря 2012, Минск, Беларусь

SQA Days является конференцией №1 на пространстве СНГ и одним из главных событий в Восточной Европе, посвященных тематике тестирования и обеспечения качества программного обеспечения.

Конференция охватит широкий спектр профессиональных вопросов в области обеспечения качества, ключевыми из которых являются:

- Методики и инструменты тестирования ПО
- Автоматизация тестирования ПО
- Подготовка, обучение и управление командами тестировщиков
- Процессы обеспечения качества в компании
- Управление тестированием и аутсорсинг
- Совершенствование процессов тестирования и инновации

Организаторы конференции приглашают к сотрудничеству докладчиков, партнёров и других заинтересованных лиц.

Twitter лента конференции: [#sqadays12](https://twitter.com/sqadays12)

WWW.SQADAYS.COM

проект компании "Лаборатория тестирования" (www.sqalab.ru)

CONTENTS

Galatenko V. A. Categorization and Separation of Programs and Data as a Architecture Security Principle . . . 2

The article is devoted to the review of architecture security principles. As a one of such principles author propose categorization and separation of programs and data. This is particularly important when designing and implementing software tools to ensure information security, able to withstand the destructive influences.

Keywords: Information Security, architecture security principles, separation of programs and data

Razumovsky A. G., Pantelev M. G. Validation and Reasoning over Objects Using Ontologies. 7

Syntax of object-oriented (OO) language such as Java is not so expressive and doesn't contain similiar semantic constructs as ontology web language does. Therefore, many rules and constrains of the domain cannot be modeled using those languages. Establishing a mapping between similar constructs of OO-languages and ontologies would help expanding validation and reasoning that are usually used over ontologies, to the objects.

Keywords: ontologies, object-oriented programming, software development

Filaretov V. F., Yukhimets D. A., Mursalimov E. Sh. Universal Architecture Development of Distributed Software of Mechatronic Object 14

In this paper the new approach for software development of mechatronic object based on architecture JAUS is offered. This software is a distributed computer control system. Its components are realized so that its one may place on the different computational devices without modification its parts which describe corresponding functional properties. It allows to essentially expand application area of this computer control system for mechatronic object which onboard computational system consist of embedded controllers.

Keywords: software, mechatronic object, computer control system, Joint Architecture for Unmanned Systems (JAUS)

Aleksandrov A. E., Vostrikov A. A., Shilmanov V. P. Principles of the Organization of Architecture of Software System "Prognoz" on the Basis of Library of Algorithms 22

The program system "Prognoz" developed on the basis of the technology based on simulation of family of program systems is provided. The program library consisting of elements, executing the basic (application-oriented) functions, and the auxiliary elements intended for independent testing of these programs, detection and correction of errors is provided. The description of basic data structures which reflect features of data domain is provided and are

used for implementation of application-oriented elements as a part of library.

Keywords: program system, program component, library of program components, object-oriented design

Popov S. E., Zamaraev R. Y. The Software and Meta-Language of Algorithms for Analysis of Social and Economic Entities 27

The article is devoted to the development of the software and algorithms, meta-language of the analysis of social and economic entities, the formalization of the descriptive rules of the expert computational modules and structured data are exemplified by entropy analysis method. Defines the basic rules of making a systematic and analytical profiles that integrate the ability to debug, test and visualize the results of the analysis. On the example of studying the state of the Siberian Federal District demonstrated the direction of the development and implementation of application software system and the language of meta-description in the analysis of social and economic systems.

Keywords: Meta-language, the entropy method of analysis, software, multi-criteria choice, functional performance, social and economic systems

Likhachev V. N., Ginzgemyer S. A. Error Handling Relational Databases of PostgreSQL 34

The article offers the author's method of informative error message formation for the PostgreSQL database based on the analysis of the database structure, usage of table and field user names, and usage of special database-level and application messages.

This approach allows to: considerably decrease the scope of work involved in developing applications working with databases; improve the structure of such applications; simplify the support of such applications.

Keywords: database, software quality, error handling, PostgreSQL

Erofeev E. V., Pelepin I. E. CMIS Performance Results Analysis in Alfresco and IBM FileNet Implementation 42

This article discusses comparative testing performance results of basic operations on documents in a Repository (IBM FileNet, Alfresco) for CMIS realization in comparison with Native API one. Some test results produce a great performance reduction in the case of CMIS using. The practical approaches of bounds definition in use of CMIS are determined.

Keywords: CMIS, performance, ECM-system, Alfresco, IBM FileNet

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

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

Сдано в набор 06.08.2012 г. Подписано в печать 25.09.2012 г. Формат 60×88 1/8. Заказ PI712
Цена свободная.

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