ТЕОРЕТИЧЕСКИЙ И ПРИКЛАДНОЙ НАУЧНО-ТЕХНИЧЕСКИЙ ЖУРНАЛ

Издается с сентября 2010 г.

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

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.

УДК 004.457

Д. В. Силаков, канд. физ.-мат. наук, старший архитектор,
ЗАО "РОСА",
e-mail: denis.silakov@rosalab.ru

RPM5: новый формат и инструментарий распространения приложений для ОС Linux

Статья посвящена RPM5 — набору инструментальных средств для установки, обновления и удаления программного обеспечения в ОС Linux, использующего возможности современных системных библиотек и аппаратных средств для повышения удобства управления ПО как для разработчиков, так и для пользователей.

Ключевые слова: *Linux, управление пакетами ПО*

Проблема распространения ПО в ОС Linux

Операционная система GNU/Linux — это не просто ядро Linux, инструментарий и библиотеки GNU. Для получения полнофункциональной системы производители конкретных вариаций Linux предоставляют пользователям согласованный и самодостаточный набор ПО, включающий не только системные компоненты и библиотеки, но и различные приложения, средства конфигурирования системы и пр. Подобный набор ПО образует *дистрибутив* Linux.

Характерной чертой большинства дистрибутивов Linux является использование развитых систем управления пакетами — инструментария для управления процессами установки, удаления, настройки и обновления ПО, входящего в дистрибутив. Использование подобных систем подразумевает распространение приложений и прочих программных компонентов в виде *пакетов* — файлов специального формата, включающих в себя архив с файлами приложения, а также различные метаданные — описание пакета, цифровую подпись, список компонентов, от которых зависит приложение, и многое другое.

В настоящее время наибольшее распространение в корпоративном секторе получили дистрибутивы Linux, использующие менеджер пакетов RPM и одноименный формат пакетов. Изначально RPM расшифровывался как RedHat Package Manager, однако в настоящее время общепринятым является использование рекурсивного акронима RPM Package Manager.

К числу дистрибутивов, использующих RPM, относятся RedHat Enterprise Linux и SUSE Linux Enterprise Server. Менеджер пакетов RPM включен в стандарт Linux Standard Base (LSB) в качестве унифицированного формата для распространения ПО в Linux [1]. В этом формате распространяются все инструментальные средства, разрабатываемые рабочей группой LSB [2] для облегчения использования стандарта. В таких системах, как Ubuntu и Debian, использующих другой популярный формат — Deb, пакеты RPM могут быть установлены с помощью утилиты alien.

История RPM насчитывает уже более 15 лет, причем последние десять лет в большинстве дистрибутивов используется четвертая версия формата и инструментария — RPM4. Безусловно, развитие RPM4 не стоит на месте. Как в сам формат, так и в инструментарий время от времени вносятся различные дополнения и улучшения. Однако все эти изменения носят эволюционный характер. В то же время за последние годы в мире ИТ произошли существенные перемены, нацеленные на повышение эффективности программных продуктов и их надежности, например, получили широкое распространение многопроцессорные машины, позволяющие эффективно выполнять задачи параллельно, получили активное развитие средства работы по сети и т. д. Подобные изменения непозволительно игнорировать в такой ключевой технологии, как RPM.

В 2007 г. была инициирована разработка RPM5 [3] — ответвления оригинального проекта, нацеленного на

активное добавление новых функциональных возможностей и избавление от устаревших (даже если последнее ведет к частичной потере обратной совместимости со старыми версиями инструментария и собранными с их помощью пакетами). На настоящее время RPM5 уже готов к промышленному использованию и применяется в дистрибутивах Unity, Ark Linux, POCA, Mandriva и ряде других проектов, оценивших перспективы обновленного механизма управления пакетами.

Нововведения RPM5 предназначены как на пользователей ОС Linux, так и для разработчиков, создающих пакеты в формате RPM. В данной статье дан обзор новшеств обеих категорий. Рассмотрены как уже реализованные функциональные возможности, так и планируемые изменения.

RPM5 для пользователей

Какие бы изменения не вносились в различные составляющие дистрибутива, все они так или иначе нацелены на получение дополнительных преимуществ с точки зрения пользователей. Не является исключением и RPM5. Часть изменений, предлагаемых новым поколением инструментария, нацелена на упрощение процесса разработки дистрибутива и формирования пакетов. Подобные улучшения видны пользователям косвенно, например, за счет более быстрого внедрения новых технологий. Однако RPM5 содержит и наработки, которые будут видны пользователям непосредственно.

Транзакционное управление пакетами. Одним из главных нововведений RPM5 стало транзакционное управление пакетами, в рамках которого установка, обновление или удаление каждого пакета представляется как атомарная транзакция (по аналогии с транзакциями СУБД), которую, в случае необходимости, можно откатить.

Для реализации такой возможности RPM5 отслеживает все системные вызовы, осуществляемые в процессе обработки пакета. При этом отслеживаются действия, выполняемые не только самим инструментарием (размещение файлов в системе, обновление базы пакетов и пр.), но и скриптами, входящими в пакеты. Для действий, выполняемых инструментарием, возможность отката гарантирована; для скриптов в общем случае это сделать не представляется возможным — единственным универсальным решением было бы создание снимка всей системы, но такой подход не представляется разумным (во всяком случае, в рамках инструментария RPM).

Использование параллелизма. Одним из наиболее ресурсоемких действий при установке многих дистрибутивов, распространяющихся в виде набора прекомпилированных пакетов ПО в формате RPM, является установка и обновление этих пакетов. Развертывание дистрибутива общего назначения, состоящего из нескольких тысяч пакетов, может занять порядка часа даже на современных машинах. Обновление дистрибутива до новой версии посредством обновления всех пакетов занимает схожее время.

Средний размер типичного дистрибутива за последние годы существенно вырос — увеличилось как число пакетов, так и их размер. Поэтому, несмотря на стремительный рост производительности компьютеров, время установки и обновления дистрибутивов уменьшилось не сильно. В то же время современные машины предлагают способ увеличения быстродействия приложений, пока еще никак не задействованный в инструментарии RPM — речь идет о возможности параллельного выполнения подзадач одного процесса на разных физических ядрах процессора.

В случае RPM, распараллеливание задач возможно на высоком уровне — например, инструментарий способен принимать на вход сразу несколько пакетов, но обрабатываются они по очереди. В то же время некоторые действия, осуществляемые при такой обработке, абсолютно независимы и для разных пакетов могут выполняться параллельно. К таким действиям относятся:

- проверка подписей пакетов;
- вычисление и проверка контрольных сумм заголовков пакетов и содержащихся в них файлов;
- проверка наличия в системе необходимых пакету зависимостей (при условии, что уже определено, какие из этих зависимостей удовлетворяются пакетами, переданными на вход инструментарию вместе с обрабатываемым).

В настоящее время в RPM5 уже реализован механизм параллельной проверки подписей и контрольных сумм, ведутся работы над параллельной проверкой зависимостей. Текущая реализация основана на использовании технологии OpenMP, хотя для других действий более удобным может оказаться использование многопоточности (в частности, функций из стандартизированного набора POSIX Threads [4]). Эксперименты на репозиториях дистрибутива POCA показали, что использование разработанных механизмов на машине с двумя ядрами ускоряет перечисленные выше проверки в 1,5 раза. Использование параллельного вычисления контрольных сумм посредством утилиты rpmdigest с опцией -- alldigests (для каждого пакета вычисляющей до сотни различных контрольных сумм) на четырехпроцессорной машине позволило достичь семикратного роста производительности.

Теоретически, возможно одновременное выполнение над несколькими пакетами и ряда других действий, например, непосредственной установки пакетов в систему, включающей разархивирование содержимого пакетов и их размещение в файловой системе. Однако одновременное выполнение подобных действий может потребовать соответствующей доработки инструментов, работающих поверх RPM и специфичных для конкретных дистрибутивов. В частности, во многих системах используются триггеры, срабатывающие при появлении в системе определенных файлов. Например, при появлении нового файла ядра можно автоматически прописывать это ядро в меню

загрузчика системы. Перед реализацией параллельного выполнения некоторых действий в инструментарии RPM необходимо изучить готовность существующих триггеров к работе в таких условиях.

Помимо одновременного выполнения действия над разными пакетами, возможно распараллеливание и на более низком уровне. Так, одним из самых затратных с точки зрения ресурсов процессора действий при установке и обновлении является разархивирование содержимого пакета. Для сжатия содержимого пакета в RPM применяется один из инструментов — gzip, bzip2 или xz (реализующий сжатие по алгоритму LZMA). Для gzip и bzip2 существуют реализации, способные распаковывать архив в несколько параллельных потоков — pgzip и pbzip2 соответственно. Однако подобные возможности в RPM в настоящее время не используются, и распаковка остается сугубо однопоточным процессом.

Исследования возможности использования параллельных алгоритмов распаковки ведутся, однако их внедрение сопряжено с существенными техническими сложностями. В частности, многие программы, использующие API RPM, получают информацию о том, какая часть архива уже распакована посредством соответствующих обратных вызовов. Дизайн существующего механизма обратных вызовов rpmtsCallback() не рассчитан на параллельную распаковку, но при этом достаточно сложен и используется многими программными компонентами. К тому же возможность параллельной распаковки архива отсутствует в случае использования xz, а этот способ становится все более популярным благодаря более высокой степени сжатия и скорости разархивирования в однопоточном режиме.

Наряду с исследованиями алгоритмов распаковки, изучается возможность использования алгоритмов параллельного сжатия, что позволит ускорить процесс формирования пакетов в сборочных средах дистрибутивов и на машинах разработчиков. Такая возможность существует как в случае применения gzip/bzip2, так и xz. Также для разработчиков представляется полезной возможность параллельного создания нескольких подпакетов из разных подмножеств одного большого приложения.

Интеграция с высокоуровневыми менеджерами пакетов. В наши дни пользователи Linux редко обращаются непосредственно к инструментарию RPM, работающему в командной строке на уровне отдельных пакетов и их групп. Традиционной схемой для большинства дистрибутивов является использование дополнительных программ управления пакетами, работающих поверх RPM и вызывающих этот инструментарий для обработки конкретных пакетов.

Высокоуровневые менеджеры пакетов — такие как Yum, Apt, Zypper или Urpmi — работают с *репозиториями* — наборами пакетов, сгруппированных согласно некоторому принципу. Например, создатели дистрибутива могут предусмотреть отдельные репозитории для игр, для офисных программ, либо для всех пакетов, не поддерживаемых официально, но собираемых и обновляемых членами сообщества. С использованием высокоуровневых менеджеров пакетов пользователи могут подключать конкретные репозитории (в том числе находящиеся на удаленных машинах), получать информацию о пакетах в них, проводить поиск среди пакетов по имени, описанию либо предоставляемым файлам и многое другое.

В силу исторических причин, высокоуровневые менеджеры пакетов частично дублируют действия, осуществляемые непосредственно RPM. Например, при установке пакета из удаленного репозитория менеджеры Urpmi, Yum и другие проверяют, от каких пакетов он зависит и какие из этих пакетов отсутствуют в системе. Все недостающие зависимости также извлекаются из репозитория. После этого проводится установка всех загруженных компонентов средствами RPM, который снова осуществляет проверку зависимостей для каждого устанавливаемого пакета.

Для избавления от подобного дублирования необходима более тесная интеграция RPM с вышестоящими программами. Одним из возможных путей решения вопроса является вызов инструментария RPM из высокоуровневых программ с явным отключением проверки зависимостей. Однако при таком подходе необходимо гарантировать, что вызывающая программа проверила *все* возможные зависимости; поскольку код программ обычно развивается параллельно с RPM, то в реальной жизни дать подобные гарантии не всегда возможно. Более продуктивным представляется выделение некоторых возможностей RPM в отдельную библиотеку, которая могла бы использоваться как самим RPM, так и другими утилитами. Создание таких библиотек в настоящее время находится на стадии идей и проектирования.

Управление конфигурационными файлами приложений. При обновлении пакета до более новой версии, существующие в системе файлы перезаписываются новыми. Однако для некоторых файлов это может оказаться нежелательно, например, для файлов конфигурации, которые пользователь мог изменить для своих целей. Использовать старые файлы при этом не всегда возможно, поскольку в новой версии программы их формат мог измениться. В существующих системах управления пакетами (не только в RPM) этот вопрос решается сохранением старых версий файлов, явно помеченных создателями пакета как конфигурационные. При этом сохраняется только одна версия, более старые удаляются. Такой подход позволяет избежать многих проблем, связанных с поддержкой конфигурационных файлов, однако при большом числе таких файлов приводит к замусориванию файловой системы. Кроме того, пользователям приходится вручную (либо с помощью дополнительных программ) изучать разницу между файлами и приводить новую конфигурацию в работоспособное состояние. Наконец, в ряде случаев хранение только предыдущей версии файла может оказаться недостаточным.

Разработчики RPM5 предложили более универсальный способ отслеживания изменений файлов, фактически заключающийся в помещении их в систему контроля версий (СКВ), поддерживаемую непосредственно инструментарием RPM. Благодаря своей гибкости, инструментарий может быть собран с поддержкой одной из библиотек, позволяющих реализовать весь необходимый функционал СКВ, например, libgit2 или libsvn.

Использование СКВ для отслеживания истории файлов снимает проблемы, перечисленные выше — число сохраняемых версий файла практически не ограничено, файловая система при этом не замусоривается, а функционал СКВ позволяет не просто получать разницу между определенными версиями, но и осуществлять их слияние. В настоящее время уже проводятся эксперименты по использованию в RPM5 библиотеки libgit2 для хранения истории изменений конфигурационных файлов.

Работа с сетью. Изначально в RPM не было заложено никаких возможностей работы по сети, например, инструментарий позволял получать детальную информацию по заданному пакету (версию, описание, список файлов и др.) только если этот пакет либо установлен в системе, либо если файл с пакетом находится на той же машине, на которой работает инструментарий. Обрабатывать файлы с удаленных машин инструментарий RPM долгое время был неспособен. Вся подобная работа традиционно возлагалась на более высокоуровневые менеджеры пакетов. Однако менеджеры пакетов работают только с репозиториями дистрибутивов и обычно не предусматривают возможности обращения к произвольным файлам на удаленных машинах.

RPM5 и современный RPM4 содержат встроенные средства обращения к удаленным машинам по протоколам FTP и HTTP и поддерживают установку/обновление пакетов, а также получение информации о них непосредственно с HTTP- или FTP-сервера. В рамках проекта RPM5 ведутся работы над расширением списка действий, в которых можно использовать информацию с удаленных машин. В перспективе планируется полностью убрать различие между локальными и удаленными операциями — если какое-то действие можно осуществить с локальным файлом, то это же действие можно будет провести и с файлом на другой машине сети.

RPM5 для разработчиков

Многие новации в RPM5 направлены на упрощение создания пакетов, что экономит время разработчиков и позволяет собрать большее количество пакетов при тех же ресурсах. Рассмотрим основные улучшения для разработчиков.

Файловые триггеры. При установке или удалении многих пакетов возникает необходимость обновлять системное окружение. Например, при добавлении в систему новых библиотек (равно как и при удалении библиотек) нужно обновить кэш динамического загрузчика; при добавлении или удалении иконок для приложений необходимо обновить кэш иконок и т. д. Традиционно такие действия выполняются в скриптах, входящих в состав пакетов. Поддержкой этих скриптов занимаются разработчики, формирующие пакеты. Однако ряд действий одинаков для многих пакетов, и их дублирование во всех скриптах нецелесообразно.

Для избавления от подобного дублирования разработчики RPM5 предложили концепцию файловых триггеров. Триггеры позволяют перенести на инструментарий RPM выполнение действий, инициируемых появлением или удалением из системы определенных файлов.

Каждый триггер состоит из двух частей:

- регулярное выражение для имен файлов, при появлении/удалении которых триггер должен срабатывать;
- перечень действий, которые необходимо выполнять с целевыми файлами.

Опыт использования файловых триггеров в дистрибутивах РОСА и Mandriva показал, что они не только сокращают размер скриптов, входящих в пакеты, но и позволяют избавиться от человеческого фактора, приводящего к появлению ошибок в таких скриптах.

Локализация. Одной из важнейших для пользователей характеристик пакета в формате RPM является его описание. Обычно в пакете присутствуют краткое однострочное описание (*Summary*) и более длинное и развернутое (*Description*). Эти описания демонстрируются пользователю как при работе в командной строке, так и при использовании инструментария более высокого уровня, с графическим интерфейсом. Для удобства пользователя описания желательно демонстрировать на его родном языке, т. е. необходима возможность *локализации* описаний пакетов.

Как ни удивительно, но за все время развития RPM4 удобного способа предоставлять пользователям описание пакетов на различных языках предложено не было. Сам формат RPM предусматривает хранение подобных описаний непосредственно в пакете. Для этого все описания должны храниться вместе с остальными данными, используемыми при сборке: исходным кодом программы, различными патчами, файлом с инструкциями для инструмента сборки rpmbuild и пр. Все существующие переводы помещаются в метаданные пакета при сборке. Инструментарий RPM не предоставляет возможности изменять описание уже собранного пакета, поэтому при обновлении перевода необходима пересборка. Помимо этого, чтобы измененное описание было доступно пользователю, ему необходимо поставить новую версию всего пакета.

Вследствие подобных недостатков помещение переведенных описаний непосредственно в RPM-пакет во многих дистрибутивах считается плохой практикой (см., например, политики сборки дистрибутивов Mandriva [5] или РОСА [6]).

Альтернативным подходом является помещение переводов описаний пакетов в отдельные файлы. Как правило, используются файлы в формате, поддерживаемом утилитами gettext (стандартным механизмом для локализации приложений в Linux). С этими файлами переводчики работают, как при переводе любой программы, использующей gettext. Для каждого языка создается свой файл, в который помещаются переводы описаний всех пакетов на этом языке. Файлы с переводами собираются в отдельный пакет (традиционно называемый *specspo*), который помещается в систему при ее установке. Инструментарий RPM при работе в системе использует файл с описаниями, соответствующий текущим языковым настройкам.

Как показала практика, такой подход достаточно удобен для переводчиков (которые работают с единственным файлом), но имеет ряд недостатков с точки зрения разработчиков дистрибутива.

Ввиду особенностей specspo в совокупности с gettext, в результирующих файлах нет указаний, какое описание к какому пакету относится. Как следствие, при поиске перевода инструментарий вынужден просматривать весь файл в поисках нужных строк, что не лучшим образом сказывается на скорости работы. А при внесении изменений в описание любого пакета, файл с описаниями и переводами необходимо перегенировать полностью — возможности указать, какие строки изменились, не предусмотрено.

При использовании specspo отпадает необходимость обновления пакета для получения его обновленного описания. Тем не менее остается потребность обновления самого пакета specspo.

Дискуссии по поводу более удобного способа локализации описаний пакетов длятся уже много лет (например, обсуждение в списке рассылки Fedora [7]), однако в рамках RPM4 никаких сдвигов не наблюдается.

Разработчиками RPM5 несколько месяцев назад был предложен альтернативный подход, заключающийся в помещении локализованных описаний пакетов в отдельное хранилище (базу данных), которое по ключу *<имя пакета; язык>* будет выдавать необходимый перевод. В качестве прототипа реализовано хранение описаний в базе данных SQLite3. В настоящее время проводится тестирование этой реализации. В случае успеха (и, возможно, после некоторых доработок) новый метод планируется внедрить в дистрибутивы POCA и Mandriva.

Автоматизация подготовки пакета к сборке. Основной сущностью, с которой имеет дело разработчик, собирающий пакет RPM, является *spec-файл*, содержащий инструкции по сборке пакета. Spec-файлы создаются преимущественно вручную и для программ со сложной схемой сборки могут иметь достаточно большой размер. Кроме того, при составлении таких файлов необходимо учитывать специфику конкретных дистрибутивов — местоположение тех или иных файлов, версии сборочных инструментов и компонентов окружения и т. д.

Одним из основных преимуществ систем управления пакетами в целом и RPM в частности является использование механизма *зависимостей*. В общем случае зависимость — это сущность, предоставляемая одним или несколькими пакетами, которую другие пакеты могут потребовать для своей работы. Каждая зависимость представляется в пакете в виде обычной текстовой строки. При установке пакета в систему, инструментарий RPM проверяет, что в ней есть все необходимые пакету зависимости. При удалении пакета происходит проверка того, что этот пакет не требуется для работы других компонентов системы. Такой подход позволяет поддерживать целостность и самодостаточность набора ПО, установленного в системе.

Изначально перечисление зависимостей каждого пакета практически целиком отдавалось на откуп разработчикам, формирующим этот пакет. Естественно, у разных разработчиков были разные представления о том, как именовать зависимости, фактически означающие одно и то же, что в итоге привело к существенным расхождениям в именовании зависимостей среди различных дистрибутивов. Подобное расхождение стало существенным препятствием для создания пакетов, которые могут быть установлены во всех системах. Даже если содержимое пакетов одинаково для разных дистрибутивов, имена зависимостей могут оказаться различными, и для каждой системы потребуется создавать свой собственный пакет.

Частично решить эту проблему позволяет использование автоматических генераторов зависимостей, применяемых и в RPM4, но наиболее активно развивающихся в RPM5. Помимо унификации имен зависимостей и экономии времени разработчиков, использование генераторов снимает ряд проблем:

- при необходимости переименовать зависимость, это достаточно сделать в одном месте (инструментарии сборки) и пересобрать все пакеты, на которых это переименование может сказаться;
- исключается возможность опечаток и прочих ошибок, возникающих вследствие невнимательности разработчика.

В настоящее время в RPM5 реализованы генераторы зависимостей приложений от имен времени исполнения разделяемых библиотек, а также зависимостей от модулей интерпретаторов Perl, Python и Ruby. Отметим, что многие разработчики дистрибутивов добавляют собственные генераторы зависимостей.

Некоторые генераторы используют в своей работе эвристики, что не позволяет гарантировать полное отсутствие ошибок. Однако практика показывает, что доля ошибок на реальных пакетах дистрибутивов существенно меньше одного процента, и ручная обработка случаев некорректной работы генераторов существенно менее ресурсоемка, чем ручное формирование всех зависимостей.

Встроенные интерпретаторы. Одной из востребованных возможностей RPM является выполнение определенных действий в процессе установки, удаления или обновления пакета. Например, если для коррек-

тной работы приложения необходимо наличие в системе пользователя или группы с определенным именем, то такой пользователь или группа могут быть созданы скриптом, автоматически вызываемым на заключительной стадии установки пакета.

Подобные скрипты помещаются в метаданные пакета и традиционно создаются на языке интерпретатора Shell. Выбор этого языка был обусловлен тем, что при установке пакета интерпретатор Shell в системе гарантированно присутствует, чего нельзя сказать об интерпретаторах других языков. Использование же скомпилированных программ для подобных действий редко оправдано, поскольку сильно усложняет процесс сборки и поддержки пакета.

В то же время язык Shell достаточно специфический, и многие сборщики пакетов с ним знакомы недостаточно хорошо. Для многих из них удобнее было бы использовать более современный и выразительный язык, например, Ruby, Python или Perl. Помимо этого, иногда возникают проблемы, которые сложно решить штатными средствами Shell, но которые можно обойти с использованием других языков. Например, если при обновлении пакета создается символьная ссылка и выясняется, что уже существует директория с таким названием, то RPM обновлять пакет откажется (что обусловлено особенностями реализации функции *cp*, используемой для копирования новых файлов на место предыдущих). В то же время такая проблема может быть эффективно решена посредством небольших скриптов на Perl или Lua, выполняющихся перед копированием файлов.

Создатели RPM5 предоставляют возможность использования для выполнения скриптов интерпретаторов, отличных от Shell, посредством их встраивания непосредственно в RPM. Поскольку многие интерпретаторы предоставляют разделяемые библиотеки, реализующие всю нужную функциональность, то такая интеграция обычно не составляет труда. В настоящее время RPM5 может быть собран со встроенной поддержкой Ruby, Python, Perl, Tcl и Lua.

Также стоит отметить поддержку в RPM5 базовых возможностей по работе с реляционными базами данных. RPM5 может быть собран с поддержкой интерфейса доступа к базам данных ODBC и базовой поддержкой языка запросов SQL, что дает возможность обращаться к различным базам данных при установке/удалении пакетов, не требуя наличия каких-то дополнительных утилит и библиотек.

* * *

Рассмотрены основные нововведения RPM5 — системы управления ПО для ОС Linux. Добавление новых функциональных возможностей, использующих поддержанные программными и аппаратными средствами современные технологии, позволило повысить скорость создания пакетов ПО и их установки на системах пользователей, в то же время повысив надежность соответствующих процессов, а также избавив разработчиков от необходимости выполнения ряда рутинных действий при подготовке пакетов.

При этом RPM5 остается обратно совместимым с RPM4 и способен устанавливать пакеты, собранные посредством инструментария четвертой версии. Разработчики RPM5 отслеживают все изменения, происходящие в RPM4, и при необходимости вносят соответствующие модификации в свой продукт.

Опыт использования RPM5 в дистрибутивах POCA и Mandriva показывает, что проект пригоден к использованию в промышленных масштабах в качестве основного инструментария управления пакетами. При этом инструментарий продолжает активно развиваться, и в будущем следует ожидать еще большего числа улучшений. За развитием RPM5 можно следить на официальном сайте, а также на портале Launchpad [8]. Процесс разработки полностью открыт, и принять в нем участие могут все желающие.

Список литературы

1. **Linux** Standard Base Core Specification 4.1. Package Format and Installation. URL: http://refspecs.linuxfoundation.org/LSB_4.1.0/LSB-Core-generic/LSB-Core-generic/packagefmt.html

2. **Силаков Д. В.** Информационно-аналитическая система для разработки и использования базового стандарта ОС Linux (LSB) // Информационные технологии. 2010. № 5. С. 53—58.

3. **Официальный** сайт RPM5. URL: http://rpm5.org/

4. **The Open Group** Base Specifications. Issue 6. pthread.h — threads. URL: http://pubs.opengroup.org/onlinepubs/009695399/basedefs/pthread.h.html

5. **Mandriva** Packaging Policies — Charsets. URL: http://wiki.mandriva.com/en/Policies/Charset.

6. **ROSA** Packaging Policies — Charset Policy URL: http://wiki.rosalab.ru/ru/index.php/Charset_policy

7. **Критика** specspo в списке рассылки Fedora. URL: http://lists.fedoraproject.org/pipermail/devel/2005-November/076390.html

8. **Раздел** RPM5 на портале Launchpad. URL: http://launchpad.net/rpm

УДК 519.714

Л. Д. Черемисинова, д-р техн. наук, гл. науч. сотр.,
Д. Я. Новиков, канд. техн. наук, науч. сотр., Объединенный институт проблем информатики Национальной академии наук Беларуси,
e-mail: cld@newman.bas-net.by

Программные средства верификации описаний комбинационных устройств в процессе логического проектирования

Рассмотрен комплекс методов и программных средств, которые обеспечивают эффективное решение задачи верификации описаний проектируемых комбинационных устройств на логическом уровне и позволяют обнаруживать ошибки на ранних этапах проектирования. Реализованы два подхода к решению задачи верификации: на основе моделирования комбинационной схемы и на основе сведения к задаче проверки выполнимости конъюнктивной нормальной формы.

Ключевые слова: *автоматизация проектирования, верификация, моделирование, выполнимость КНФ*

Введение

Проектирование современных цифровых СБИС представляет собой многоэтапный процесс оптимизации и преобразований проектных решений, начиная с исходного описания на одном из входных языков проектирования и заканчивая схемой в целевом технологическом базисе. Проектные решения могут быть получены как в автоматическом (с использованием программных средств синтеза и оптимизации), так и в автоматизированном (путем корректировки проектного решения человеком) режимах. В любом случае, во избежание распространения ошибки, допущенной на одном из ранних этапов проектирования, до стадии изготовления схемы необходимо верифицировать решения, полученные в ходе проектирования микроэлектронного устройства. В случае верификации всегда предполагается наличие некоторого исходного, быть может, менее формального, описания проектируемой схемы. Задача верификации заключается в доказательстве поведенческого соответствия двух описаний одного и того же устройства, а именно в проверке, находятся ли они в отношении эквивалентности (если оба описания полностью определены) или реализации (если исходное описание не полностью определено).

Эффективное решение задачи верификации особенно важно при проектировании микросхем СБИС для критических применений, где ошибки в их функционировании могут приводить к катастрофическим последствиям — порче дорогостоящего оборудования, срыву важных исследований, чрезвычайным ситуациям. По мере возрастания сложности проектируемых устройств функциональная верификация становится все более необходимым и дорогим этапом процесса проектирования. Официальные издания проектных компаний утверждают, что коллектив разработчиков цифровой аппаратуры тратит до 70 % всего времени проектирования и ресурсов на функциональную верификацию, и, если сложность проекта аппаратуры удваивается, то усилия, затрачиваемые на верификацию, увеличиваются в 4 раза [1]. В связи с этим обстоятельством верификация, позволяющая выявить ошибки проектирования на достаточно ранних его этапах, становится все более и более ответственной стадией процесса проектирования.

В литературе традиционно рассматривается случай, когда оба сравниваемых описания функциональ-

но полностью определены, решению этой задачи посвящены многочисленные научные публикации, ссылки на которые можно найти в работах [1—3]. В то же время исходные описания проектов электронных управляющих устройств на языках VHDL и VERILOG могут иметь неопределенности, которые, в частности, объясняются тем, что поведение разрабатываемых устройств не регламентировано для некоторых входных воздействий, поскольку появление таких воздействий в процессе функционирования устройства невозможно.

В настоящей статье предлагаются средства для решения задачи верификации для более общего случая, когда исходное описание проектируемого устройства функционально не полностью определено. Данная ситуация обычно возникает на начальных этапах проектирования, когда существуют такие комбинации значений входных переменных проектируемого устройства, которые никогда не появятся при нормальном режиме его работы. В процессе проектирования устройства его выходные реакции на такие входные воздействия могут быть доопределены произвольным образом. В этом случае при решении задачи верификации достаточно рассмотреть только возможные сценарии поведения верифицируемого устройства и проверить, имеют ли его выходные реакции специфицированные значения.

Представляемые в работе средства верификации являются частью программного комплекса [4] для автоматизации проектирования интегральных микросхем с пониженным энергопотреблением, выполняемых по КМОП-технологии. Программы верификации ориентированы на тестирование логических описаний верифицируемых устройств большой размерности. В качестве элементного базиса сравниваемых схем допускаются элементы библиотеки КМОП СБИС, а также элементы, реализующие простые функции (типа И, ИЛИ, И-НЕ и т. д.). Сравниваемые описания задаются на языке функционально-структурных описаний SF [5].

1. Основные определения. Постановка задачи верификации

Будем рассматривать n-мерное булево пространство E^n ($E = \{0, 1\}$), состоящее из n-компонентных булевых векторов, каждый из которых представляет набор значений булевых переменных множества $X = \{x_1, x_2, ..., x_n\}$. Интервал a булева пространства задается множеством наборов значений переменных из X и представляется n-компонентным троичным вектором [6]. Ранг интервала равен числу компонент его векторного представления, которые имеют определенные значения (единица или ноль). Интервал ранга k может быть представлен в виде конъюнкции k литералов переменных $x_i \in X$ (под литералом понимается переменная или ее отрицание) и задает 2^{n-k} наборов булева пространства E^n. Например, интервал ранга $k = 2$, заданный троичным вектором 0 — 1 — пространства E^4 представляется множеством наборов {0010, 0011, 0110, 0111}.

Полностью определенная булева функция (ПБФ) $f(X)$ (где $X = \{x_1, x_2, ..., x_n\}$) задается множествами U_f^0 и U_f^1 интервалов (или M_f^0 и M_f^1 наборов) булева пространства E^n, на которых эта функция принимает соответственно нулевое и единичное значения, при этом $U_f^0 \cup U_f^1 = E^n$.

Дизъюнктивная нормальная форма (ДНФ) представляет ПБФ в виде дизъюнкции одной или более конъюнкций, каждая из которых задает интервал из области и U_f^1. Конъюнктивная нормальная форма (КНФ) представляет ПБФ в виде конъюнкции одного или более дизъюнктов. Каждый дизъюнкт, в свою очередь, является дизъюнкцией одного или более литералов (соответствующих разным переменным). Матричное представление КНФ (или ДНФ) задается троичной матрицей, строки которой задают дизъюнкты (или конъюнкции), а столбцы соответствуют переменным [6].

Задача проверки выполнимости КНФ заключается в нахождении такого присваивания (может быть частичного) значений переменным из X, которое обращает КНФ в 1. Если такое присваивание существует, то говорят, что КНФ выполнима, и полученное присваивание называют выполняющим эту КНФ. Иначе КНФ не выполнима, невыполнимая КНФ представляет функцию, тождественно равную 0.

Частично определенная булева функция (ЧБФ) $f(X)$ задается множествами U_f^0, U_f^1 и U_f^{dc} интервалов, на которых $f(X)$ принимает соответственно нулевое, единичное или неопределенное значения, при этом $U_f^0 \cup U_f^1 \cup U_f^{dc} = E^n$. Степень определенности ЧБФ определяется отношением числа наборов в множестве $U_f^0 \cup U_f^1$ к числу наборов в U_f^{dc}. Будем задавать систему ЧБФ $F(X) = \{f_1(X), f_2(X), ..., f_m(X)\}$ множеством многовыходных интервалов (**u**, **t**), каждый из которых задается парой троичных векторов длины n и m. Троичный вектор **u** представляет собой интервал из n-мерного булева пространства наборов значений переменных множества X, троичный вектор **t** задает значения функций $f_i \in F$ на интервале **u**, при этом **u** и **t** могут представляться также конъюнкциями литералов переменных $x_i \in X$ и функций $f_i \in F$. Для каждой функции $f_j \in F(X)$ справедливо: если j-я компонента t^j вектора **t** равна 1 или 0, то на всех наборах из интервала **u** функция f_j принимает соответствующее значение; если же $t^j =$ "—", то либо f_j принимает разные значения (из {1, 0, —}) по крайней мере на двух наборах из интервала **u**, либо ее значение не определено на всем интервале.

Система ЧБФ $F(X)$, заданная множеством I_F многовыходных интервалов ($\mathbf{u}_i$, $\mathbf{t}_i$), может быть представлена парой троичных матриц **U** и **T**, задающих своими строками многовыходные интервалы. Напри-

мер, если $I_F = \{(x_3x_4x_5, f_1), (\bar{x}_2\bar{x}_3\bar{x}_4, \bar{f}_1f_2), (\bar{x}_2x_4x_5, f_2), (\bar{x}_1x_2x_4\bar{x}_5, \bar{f}_1\bar{f}_2), (\bar{x}_2x_3\bar{x}_4, \bar{f}_2), (x_1x_2, f_1\bar{f}_2)\}$, то система представляется следующей парой троичных матриц:

$$\mathbf{U} = \begin{matrix} x_1x_2x_3x_4x_5 \\ --111 \\ -0000 \\ -0-11 \\ 01-10 \\ -0100 \\ 11--- \end{matrix} \qquad \mathbf{T} = \begin{matrix} f_1f_2 \\ 1-1 \\ 012 \\ -13 \\ 004 \\ -05 \\ 106 \end{matrix}$$

В процессе синтеза система ПБФ, ДНФ или ЧБФ реализуется комбинационной схемой S из элементов некоторой библиотеки, в частности, простых элементов типа И, ИЛИ, НЕ. Предполагается, что n входов схемы S соответствуют n аргументам из X и на m выходах реализуются функции $y_i(X)$, соответствующие m функциям системы $F(X)$.

Если исходное описание полностью определено (например, это ПБФ или схема), то задача верификации заключается в проверке эквивалентности между двумя схемными реализациями. Если же исходное описание содержит функциональную неопределенность (например, это система F ЧБФ), то задача верификации сводится к проверке реализуемости системы ЧБФ F схемой S. Условие реализуемости системы F схемой S заключается в том, что на области определения каждой функции $f_i(X) \in F$ значения функций f_i и y_i должны совпадать, т. е. для всех $f_i(X) \in F$ и $y_i(X)$ должно выполняться

$$M^1_{fi} \subseteq M^1_{yi};\ M^0_{fi} \subseteq M^0_{yi}.$$

При решении задачи проверки реализуемости системы ЧБФ F комбинационной схемой S на основе моделирования можно ограничиться частичным анализом сравниваемых описаний — не на всем булевом пространстве E^n, а только на области определения $U^0_f \cup U^1_f$ системы F.

В последние годы большое внимание уделяется разработке методов формальной верификации, в основе которой лежит решение задачи проверки выполнимости КНФ. Программы, решающие задачу выполнимости, называют SAT-решателями (SAT-solvers, SAT — сокращение от *satisfiability*, выполнимость). Задача выполнимости КНФ является центральной задачей теории сложности в теории дискретных систем. Ведущие фирмы в области проектирования микроэлектронных устройств активно ведут исследования по созданию новых эффективных алгоритмов решения этой задачи, и в последние годы получены впечатляющие результаты. Наиболее известными программами решения задачи выполнимости, используемыми в настоящее время в области автоматизации электронного проектирования, считаются Chaff [7], BerkMin [8] и MiniSat [9].

2. Функциональные возможности комплекса верификации

Комплекс верификации включает в себя алгоритмические и программные средства для решения задачи верификации, когда сравнивают:

1) два полностью определенных описания, т. е. по сути, две структурные реализации — схемы в некоторых, может быть и разных, базисах;

2) два описания, одно из которых, исходное, не полностью определено (область определения не покрывает все булево пространство), а второе, порожденное в процессе проектирования (оптимизации, синтеза), полностью определено, т. е. задает, например, логическую схему.

В первом случае, как уже отмечалось ранее, верификация заключается в проверке эквивалентности между двумя схемными реализациями. Во втором случае верификация заключается в проверке отношения реализуемости между описаниями, т. е. эквивалентности описаний на области определения "наименее определенного" из двух описаний — исходного описания.

Предполагается, что сравниваемые описания задают функциональные зависимости между одноименными входными и выходными переменными. Каждое из сравниваемых описаний, принимаемых комплексом, задается в одном из форматов на языке функционально-структурных описаний SF [5]:

- SDF — ДНФ, задающая систему ПБФ в матричном виде;
- SBF — система ЧБФ в матричном виде;
- LOG — система логических уравнений, задающая двухуровневую или многоуровневую схему (в частном случае и систему ДНФ) из вентилей типа НЕ, И, ИЛИ, исключающее ИЛИ и др.;
- 2-connect — двухуровневое иерархическое описание структурной реализации, блоки которого имеют функциональное описание типа SDF, SBF или LOG, а второй уровень задает связи между ними.

Иерархическое описание может задавать схему из библиотечных элементов КМОП СБИС или многоблочную структуру, каждый из блоков которой реализует систему полностью или частично определенных булевых функций. Исходное описание верифицируемого устройства представляется, как правило, в виде системы ДНФ или ЧБФ (к этому виду может быть приведено любое функционально полностью или частично определенное описание в виде логических уравнений), заданной как на наборах, так и интервалах значений аргументов. В том случае, когда проводится оптимизация на структурном уровне, исходным описанием может быть схема, заданная в форматах LOG и 2-connect.

Второе из сравниваемых описаний задает комбинационную схему в базисе многовходовых логических элементов или многоблочную структуру и представляется в формате LOG или 2-connect. В обоих случаях порожденное описание можно трактовать как структурную реализацию исходного описания — схему (двухуровневую или многоуровневую).

3. Алгоритмическая поддержка комплекса

Авторами были предложены два класса методов верификации [10—13], которые эффективны для разных типов сравниваемых описаний и способов их задания. Методы первого класса [10, 13] основаны на моделировании описания устройства, полученного в ходе проектирования (порожденного описания), на тех наборах значений входных переменных, которые входят в область определения исходного описания. Этот подход заключается в подаче двоичных сигналов на входы моделируемой схемы, продвижении их по схеме, соответствующей активации ее выходов и проверке результатов моделирования с ожидаемыми значениями. Однако при моделировании практически возможен только частичный анализ сравниваемых описаний — на некотором подмножестве входных воздействий (так как их число может достигать 2^n, где n — число входных переменных), и моделирование в общем случае не может обеспечить полноту верификации.

В работе [13] предложено обобщение метода двоичного моделирования, позволяющее моделировать поведение схемы на интервалах значений входных переменных. Такой подход значительно повышает порог применимости метода моделирования в плане сложности моделируемых описаний, однако он не решает проблему полноты верификации. Причина в том, что некоторые интервалы приходится иногда раскрывать до наборов значений переменных.

Методы второго класса основаны на формальном доказательстве функциональной идентичности проектов путем сведения задачи верификации к проверке выполнимости некоторой КНФ [2, 3], отражающей структуру сравниваемых описаний. В отличие от методов моделирования эти методы позволяют обеспечить полноту верификации, их развитию и практическому применению способствовал наметившийся в последнее десятилетие существенный прогресс в решении задачи выполнимости КНФ.

Методы моделирования эффективны для случая задания системы ЧБФ на наборах значений аргументов или "небольших" интервалах. Методы на основе выполнимости КНФ эффективны для случая задания системы ЧБФ на "крупных" интервалах, охватывающих большие подпространства булева пространства. В предлагаемых средствах верификации реализован также комбинированный метод, совмещающий в себе методы верификации обоих классов. Основная идея этого метода состоит в использовании сначала моделирования, пока оно эффективно и полно. Далее в сложных для моделирования случаях применяется метод сведения задачи верификации к проверке выполнимости КНФ.

4. Верификация на основе моделирования

Реализованные в рамках программного комплекса методы верификации посредством моделирования основаны на использовании [10, 13]:

— двоичного или троичного моделирования (на наборах или интервалах значений входных переменных), в зависимости от способа задания системы ЧБФ исходного описания;

— параллельного моделирования сразу на всех наборах или интервалах из области определения системы ЧБФ исходного описания;

— уточнения отношения реализуемости между описаниями с функциональной неопределенностью;

— процедур анализа результатов троичного моделирования;

— быстрых вычислений над булевыми и троичными векторами произвольной размерности.

Моделирование схемы на наборах значений переменных или, в общем случае, на интервалах из области определения системы ЧБФ $F(X)$ исходного описания состоит: в последовательной подаче на входы схемы наборов (интервалов) значений входных переменных; вычислении значений сигналов на выходах элементов схемы; сравнении реакций схемы со значениями функций системы $F(X)$ исходного описания.

Выполняется параллельное моделирование [6] предварительно ранжированной многовыходной логической схемы сразу на всех наборах или, в общем случае, на интервалах из области определения системы ЧБФ $F(X)$. При параллельном моделировании схемы на l интервалах (или наборах) значений переменных состояние каждого ее полюса (включая входные и выходные) представляется троичным (или булевым) вектором размерности l, задающим состояния этого полюса при подаче на входы схемы каждого из l интервалов. При этом для выходного полюса каждого i-го элемента, имеющего k входных полюсов, вычисляется локальная функция $\varphi_i(\mathbf{z}_{1i}, \mathbf{z}_{2i}, ..., \mathbf{z}_{ki})$ путем выполнения покомпонентной операции φ_i [13] над троичными (или булевыми) векторами $\mathbf{z}_{1i}, \mathbf{z}_{2i}, ..., \mathbf{z}_{ki}$ длины l, приписанными входным полюсам элемента.

Особенностью троичного моделирования является то, что j-я компонента вектор-результата, вычисляемого для некоторого полюса схемы, может иметь значение "—" не только в том случае, когда функция, реализуемая полюсом, имеет неопределенное значение на всем j-м интервале исходной системы ЧБФ, но и в том случае, когда функция имеет разные значения на разных наборах этого интервала. Такая неоднозначность может проявляться и на выходах моделируемой схемы, что приводит к тому, что для некоторых исходных интервалов невозможно дать однозначный ответ на вопрос, реализуются ли они схемой. Проблемная ситуация может быть разрешена путем повторно-

го, но уже двоичного моделирования схемы на наборах, входящих в анализируемый интервал, или путем проверки реализуемости оставшейся части исходного описания формальными методами верификации на основе проверки выполнимости КНФ.

Преимущество троичного моделирования над двоичным состоит в том, что нет необходимости расщеплять интервалы на наборы, что позволяет резко сократить длину требуемых для моделирования векторов и повысить быстродействие моделирования. Однако следует отметить, что по результатам троичного моделирования не всегда удается однозначно ответить на вопрос, реализуется ли система функций схемой на всех интервалах. Это означает, что может остаться некоторое число интервалов, для которых необходим дополнительный анализ.

5. Формальная верификация на основе проверки выполнимости КНФ

Реализованные в рамках программного комплекса методы верификации посредством сведения к задаче проверки выполнимости КНФ основаны на построении такой КНФ, выполнимость которой свидетельствует о том, что исходное описание не реализуется описанием, полученным в ходе проектирования (порожденным).

Традиционно задача формальной верификации состоит в проверке функциональной эквивалентности пары структурных реализаций одного и того же устройства. При проверке эквивалентности комбинационных схем обе верифицируемые схемы преобразуют в одну комбинационную схему, называемую схемой сравнения [3]. Эту схему получают путем объединения пар одноименных входов сравниваемых схем и подачи пар их одноименных выходов на двухвходовые элементы "исключающее ИЛИ", выходы которых собираются на элемент ИЛИ. Константа 0 на выходе элемента ИЛИ появляется тогда и только тогда, когда исходные схемы эквивалентны. Для схемы сравнения строят КНФ разрешения C_S, которая задает все возможные допустимые комбинации сигналов на всех полюсах элементов схемы, и которую проверяют на выполнимость — для эквивалентных схем КНФ C_S не выполнима.

Конъюнктивная нормальная форма разрешения C_S схемы представляет собой объединение (операцией конъюнкции) КНФ разрешения всех элементов схемы. При ее построении для каждого логического элемента схемы, реализующего функцию $f(z_1, z_2, ..., z_k)$ от k своих входов, вводится булева переменная y, строится функция разрешения $\varphi(y, f) = y \sim f(z_1, z_2, ..., z_k)$ и соответствующая этой функции КНФ разрешения элемента. Например, КНФ разрешения k-входовых элементов И и ИЛИ имеют вид

$$\varphi^{\wedge}(y, z_1, z_2, ..., z_k) = (z_1 \vee \bar{y}) \wedge (z_2 \vee \bar{y}) \wedge ... \wedge (z_k \vee \bar{y}) \wedge (\bar{z}_1 \vee \bar{z}_2 \vee ... \vee \bar{z}_k \vee y);$$

$$\varphi^{\vee}(y, z_1, z_2, ..., z_k) = (\bar{z}_1 \vee y) \wedge (\bar{z}_2 \vee y) \wedge ... \wedge (\bar{z}_k \vee y) \wedge (z_1 \vee z_2 \vee ... \vee z_k \vee \bar{y}).$$

В том случае, когда исходное описание содержит неопределенность, оно приводится к виду системы ЧБФ, и искомая КНФ, которая проверяется на выполнимость, состоит из:

— КНФ разрешения C_S, описывающей все допустимые (или разрешенные) комбинации сигналов на полюсах элементов схемы (или многоблочной структуры), заданной порожденным описанием;

— КНФ запрета C_P, описывающей комбинации сигналов, которые противоречат функциям исходной системы ЧБФ.

В работе [12] доказано, что система ЧБФ $F(X)$ реализуется комбинационной схемой, если и только если КНФ $C = C_S \wedge C_P$ не выполнима.

Комбинационная схема реализует систему ЧБФ, если она реализует каждый ее многовыходной интервал. Иначе говоря, система ЧБФ не реализуется комбинационной схемой тогда, когда хотя бы один многовыходной интервал не реализуется этой схемой. В терминах КНФ разрешения многовыходной интервал $(\mathbf{u}_i, \mathbf{t}_i)$ реализуется схемой, если присваивание значений переменным, выполняющее конъюнкцию $\mathbf{u}_i \wedge \bar{\mathbf{t}}_i$, не является выполняющим для КНФ разрешения схемы [11]. Конъюнктивная нормальная форма, задающая условие нереализуемости интервала, называется КНФ запрета этого интервала. Если $\mathbf{u}_i = x_1^i x_2^i ... x_{n_i}^i$, $\mathbf{t}_i = f_1^i f_2^i ... f_{m_i}^i$, то КНФ P_i запрета интервала $(\mathbf{u}_i, \mathbf{t}_i)$ состоит из $n_i + 1$ дизъюнктов: $P_i = x_1^i x_2^i ... x_{n_i}^i \times \times \left(\bar{f}_1^i \vee \bar{f}_2^i \vee ... \vee \bar{f}_{m_i}^i\right)$.

Условие нереализуемости системы ЧБФ $F(X)$ задается ее функцией запрета $P = P_1 \vee P_2 \vee ... \vee P_l$, где l — мощность множества многовыходных интервалов области определения системы. Представление функции P в общем случае не является КНФ, и чтобы применить SAT-решатель для проверки выполнимости КНФ C, необходимо функцию запрета P преобразовать к виду КНФ C_P. Авторами предложен метод линейной сложности преобразования функции запрета P к виду КНФ C_P путем кодирования интервалов из области задания исходной системы ЧБФ (кодами единичной [11] или логарифмической [14] длины). После кодирования функция P принимает вид КНФ $C_P = (P_1^k \wedge P_2^k \wedge ... \wedge P_l^k) \wedge Q(W)$, где P_1^k — кодированная форма КНФ запрета i-го интервала, $Q(W)$ — КНФ выбора, обеспечивающая выполнимость КНФ $C = C_S \wedge C_P$ в случае, если выполняется хотя бы одна КНФ $C_S \wedge P_i^k$, и невыполнимость, если все КНФ $C_S \wedge P_i^k$ не выполнимы. Вид этой функции [12] зависит от типа кодирования. Например, при кодировании интервалов кодами w_i

длины 1 $Q(W) = \bar{w}_1 \vee \bar{w}_2 \vee ... \vee \bar{w}_l$, а КНФ запрета интервалов $P_i^k = x_1^i x_2^i ... x_{n_i}^i \left(\bar{f}_1^i \vee \bar{f}_2^i \vee ... \vee \bar{f}_{m_i}^i \vee w_i\right)$.

На основе проведенных исследований, позволивших определить области предпочтительного использования методов верификации каждого класса, была показана целесообразность сочетания методов моделирования и сведения к проверке выполнимости КНФ при верификации сложных описаний. Этот метод основан на верификации описаний по частям, с использованием в зависимости от типов и сложности сравниваемых частей описаний и степени их определенности одного из подходящих типов верификации.

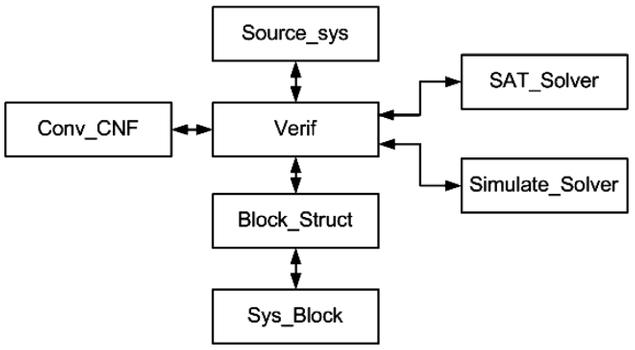
6. Структура программного комплекса верификации

После получения пары сравниваемых логических описаний в одном из упомянутых в разд. 1 форматов программный комплекс работает следующим образом. Исходное описание приводится к виду системы ЧБФ $F(X)$, если оно содержит неопределенность, или к виду логической схемы во внутренней форме. Порожденное описание также приводится к виду логической схемы S, удобной для проведения моделирования. Далее, если оба сравниваемых описания представляют логические схемы, то строится схема сравнения, ее КНФ разрешения и проводится проверка выполнимости этой КНФ. Если исходное описание содержит неопределенность, то осуществляется троичное моделирование логической схемы S на интервалах области определения системы $F(X)$. Если по результатам троичного моделирования не удается однозначно ответить на вопрос, реализуется ли система $F(X)$ схемой S на всех интервалах, а именно — может остаться некоторое множество I интервалов, для которых необходим дополнительный анализ, то для этого множества используется формальный метод верификации путем сведения задачи верификации к проверке выполнимости КНФ $C = C_S \wedge C_P$.

Если в процессе верификации будет установлена неэквивалентность или нереализуемость исходного описания, то проводится диагностика ошибок в порожденном описании. Такая диагностика заключается в нахождении фрагмента исходного описания, не реализуемого порожденным описанием.

Разработанные программные средства верификации состоят из семи основных модулей (см. рисунок) для решения следующих задач:

- обработки и синтаксической проверки исходного описания верифицируемого устройства (модуль Source_sys), заданного в форматах SDF, SBF, LOG или 2-connect языка SF, и далее приведения его к виду системы ЧБФ или логической схемы во внутренней форме;
- обработки и синтаксической проверки порожденного описания многоблочной структуры, заданного в форматах SDF, LOG (модуль Block_Struct) или 2-connect языка SF (модуль Sys_Block), и приведения его к виду логической схемы во внутренней форме;
- построения КНФ, готовой для проверки выполнимости SAT-решателем (модуль Conv_CNF):
  - КНФ запрета исходной системы ЧБФ;
  - КНФ разрешения многоблочной структуры для различных рассмотренных случаев задания ее блоков;
  - КНФ схемы сравнения (для случая, когда сравниваемые описания задают комбинационные схемы);
- проверки выполнимости сформированной КНФ SAT-решателем (SAT_Solver), в качестве которого используется Minisat [9];
- двоичного или троичного параллельного моделирования логической схемы на наборах или на интервалах значений аргументов системы ЧБФ (модуль Simulate_Solver);
- управления процессом верификации (модуль Verif), заключающегося в распознавании типов исходных данных; в выборе подходящего метода верификации, наиболее эффективного в каждом конкретном случае; в диагностике ошибок, если таковые имеются; в организации связи между описанными модулями.



Структура программного комплекса верификации

7. Программная реализация и исследования

Программные средства разрабатывались в среде программирования Visual C++ 6.0. При программной реализации алгоритмов были использованы классы языка C++ для выполнения операций над булевыми и троичными векторами и матрицами произвольной размерности.

В ходе тестовых испытаний было проведено сравнение по быстродействию предложенных программных средств верификации со средствами верификации, реализованными в программных комплексах "Modelsim" [15] фирмы Mentor Graphics и "СиВер" [16] синтеза и верификации, разработанного в ОИПИ НАН Беларуси. Исследования проводились на рабочей станции с процессором Pentium IV 2400 МГц; 1024 МБайт ОЗУ и Windows XP Professional SP2.

Каждый рассматриваемый пример включал систему полностью или частично определенных булевых функций, заданных на наборах или интервалах значений входных переменных, и комбинационную схему из элементов КМОП библиотеки, полученную в результате проектных операций программного комплекса [4]. Примеры verg1, verg2 взяты из практики проектирования промышленных контроллеров [17], примеры R_30_15_300, R_25_10_250 сгенерированы псевдослучайным образом, а остальные примеры (benchmarks) взяты из источников в сети Интернет [18].

Примеры характеризовались следующими параметрами:

— числом n аргументов системы F функций;

— числом m функций системы F;

— числом l наборов или интервалов области определения системы функций;

— числом k элементов схемы S.

При проведении экспериментов измерялись время t_v, t_{msim}, и t_{siver}, затрачиваемые на верификацию примеров с помощью разработанных программных средств и средств "Modelsim" и "СиВер" соответственно.

Результаты сравнения средств верификации приведены в табл. 1 [17] и 2. Следует отметить, что система "СиВер" использует для верификации только формальные методы и рассчитана на полностью определенные описания. По этой причине перед этой проектной операцией неопределенные значения функций исходной системы ЧБФ доопределялись до нулей.

Результаты исследования показали, что предложенный комплекс средств верификации "Verif_PDBF" обладает более высоким быстродействием, выполняет верификацию рассматриваемых примеров на порядки быстрее по сравнению с программой VHDL-моделирования из "Modelsim" и средствами верификации из "СиВер".

Таблица 1

Сравнение комплекса "Verif_PDBF" и системы "Modelsim"

Пример	n	m	l	k	t_{msim}, с	t_v, с
RCKL	32	7	96	745	>10 000	0,05
MAX1024	10	6	1024	2496	0,63	0,08
INTB	15	7	664	1124	2,5	0,15
TIAL	14	8	640	900	5,0	0,13
R_30_15_300	30	15	300	6483	1800,0	0,35
R_25_10_250	25	10	250	3241	85,0	0,08
verg1	17	61	938	1410	10,0	0,08

Таблица 2

Сравнение комплекса "Verif_PDBF" и системы "СиВер"

Пример	n	m	l	k	t_{siver}, с	t_v, с
ADR4	8	5	256	1631	19,25	0,05
ALU1	12	8	19	73	2,75	0,00
CO14	14	1	47	175	2,38	0,01
RCKL	32	7	96	745	5,85	0,05
RYY6	16	1	112	680	5,78	0,00
SYM10	10	1	837	5124	43,75	0,12
Z9SYM	9	1	420	2501	36,25	0,05
MAX1024	10	6	1024	2496	18,0	0,08
B12	15	9	431	2341	39,25	0,05
EX7	16	5	123	811	6,25	0,01
gary	15	11	442	2850	59,5	0,03
INTB	15	7	664	1124	11,25	0,15
m181	15	9	430	2336	39,75	0,05
MP2D	14	14	123	759	6,25	0,01
RD73	7	3	147	1045	9,25	0,01
SHIFT	19	16	100	564	5,0	0,01
TIAL	14	8	640	900	10,25	0,13
verg1	17	61	938	1410	13,75	0,08
verg2	18	63	507	7293	1050,8	1,05

Заключение

Разработанные программные средства решают задачу верификации для разных сочетаний типов сравниваемых проектных решений (полностью или не полностью определенных описаний, двух и многоуровневых схем из вентилей, схем из библиотечных элементов); для разных способов задания (систем функций, определенных на наборах и интервалах значений переменных, систем логических уравнений).

Оболочка программного комплекса обеспечивает: анализ верифицируемых пар проектных решений и определение типа задачи верификации; выбор наиболее эффективного пути решения задачи верификации в зависимости от способа задания, типов проектных решений и их сложности; преобразование форматов задания проектных решений; настройку на базис библиотечных элементов.

Описанные средства верификации ориентированы на тестирование логических описаний верифицируемых устройств большой размерности и включены в состав программного комплекса автоматизации проектирования логических схем в библиотечном базисе, оптимизированных по энергопотреблению [4].

Список литературы

1. **Wiemann A.** Standardized functional Verification. San Carlos, CA USA: Springer, 2008. 289 p.

2. **Ganai M., Gupta A.** SAT-Based Scalable Formal Verification Solutions. New York: Springer-Verlag, 2007. 338 p.

3. **Kuehlmann A., Cornelis A. J. van Eijk.** Combinational and Sequential Equivalence Checking // Logic synthesis and Verification / eds by S. Hassoun, T. Sasao, R. K. Brayton. Kluwer Academic Publishers, 2002. P. 343—372.

4. **Бибило П. Н., Черемисинова Л. Д., Кардаш С. Н.** и др. Синтез логических КМОП-схем с пониженным энергопотреблением // Проблемы разработки перспективных микро- и наноэлектронных систем. — Сб. трудов / под ред. акад. РАН А. Л. Стемпковского. М.: ИППМ РАН, 2012. С. 73—78.

5. **Бибило П. Н.** Кремниевая компиляция заказных СБИС. Минск: Ин-т техн. кибернетики АН Беларуси, 1996. 268 с.

6. **Закревский А. Д., Поттосин Ю. В., Черемисинова Л. Д.** Логические основы проектирования дискретных устройств. М.: Физматлит, 2007.

7. **Ganai M. K., Zhang L., Ashar P., Gupta A., Malik S.** Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver // Proc. of the 39th. ACM/IEEE Design Automation Conference. New Orleans, LA, USA. 10—14 June, 2002. ACM/IEEE, 2002. P. 747—750.

8. **Goldberg E., Novikov Y.** BerkMin: A fast and robust SAT-Solver // Proc. of European Design and Test Conference. Paris, France. 4—8 March 2002. IEEE Computer Society, 2002. P. 142—149.

9. **Een N., Sorensson N.** An Extensible SAT-solver // Theory and Applications of Satisfiability Testing (SAT 2003). Proc. of the International Conference, Santa Margherita Ligure, Italy, 5—8 May, 2003. Microsoft Research. 2003. P. 502—518.

10. **Новиков Д. Я., Черемисинова Л. Д.** Исследование методов верификации описаний с функциональной неопределенностью на основе моделирования // Автоматика и вычислительная техника. 2012. № 5. С. 13—25.

11. **Cheremisinova L., Novikov D.** SAT-Based Approach to Verification of Logical Descriptions with Functional Indeterminacy // 8th Int. Workshop on Boolean problems, Freiberg (Sachsen). 18—19 September, 2008. Freiberg Technische Universität Bergakad, 2008. P. 59—66.

12. **Черемисинова Л. Д., Новиков Д. Я.** Формальная верификация описаний с функциональной неопределенностью на основе проверки выполнимости конъюнктивной нормальной формы // Автоматика и вычислительная техника. 2010. № 1. С. 5—16.

13. **Cheremisinova L., Novikov D.** Simulation-based approach to verification of logical descriptions with functional indeterminacy // Information Theories & Applications (IJ ITA). 2008. Vol. 15. No. 3. P. 218—224.

14. **Cheremisinova L., Novikov D.** SAT-based method of verification using logarithmic encoding // Information Science and Computing. 2009. № 15. P. 107—114.

15. **ModelSim**: HDL Simulation. Mentor Graphics Corporation [Electronic resource]. URL: http://www.mentor.com/products/fv/modelsim/

16. **Бибило П. Н., Кардаш С. Н., Романов В. И.** СиВер — система синтеза и верификации комбинационных логических схем // Информатика. 2006. № 4 (12). С. 79—87.

17. **Бибило П. Н., Новиков Д. Я.** Верификация логических схем, реализующих системы частичных булевых функций // Информатика. 2011. № 3. С. 68—76.

18. **Espresso** examples. University of California Berkeley [Electronic resource]. URL: http://www1.cs.columbia.edu/~cs4861/sis/espresso-examples/ex.

ИНФОРМАЦИЯ

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

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

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

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

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

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

УДК 004.415.2

Ю. Л. Гик, руководитель группы прикладной интеграции, Управление интеграции Департамента управления архитектурой, АКБ "Росевробанк" (ОАО), г. Москва,
e-mail: y.gik@roseurobank.ru

Анализ методологий сервис-ориентированной архитектуры (СОА)

Работа посвящена анализу и систематизации существующих методологий СОА. Проведено исследование методологий компаний IBM, Oracle, Microsoft, Red Hat, Progress Software, Arcitura, OASIS, The Open Group. Проведено сравнение основных методологий СОА компаний Arcitura и The Open Group.

Ключевые слова: *архитектура предприятия, СОА*

Введение

Деятельность большинства современных предприятий достаточно сложна и многообразна. Для ее автоматизации, как правило, не хватает возможностей одной информационной системы. Рано или поздно предприятие "вырастает" и оказывается в ситуации "зоопарка" информационных систем, построенных на разнообразных технологиях, использующих разные, зачастую несовместимые между собой базы данных, протоколы передачи, операционные системы и другие средства вычислительной техники. На этом этапе перед предприятием остро встают задача интеграции данных и приложений, необходимых для эффективной его деятельности, а также задача построения архитектуры средств и систем автоматизации предприятия как общего плана развития своих информационных ресурсов и их взаимосвязей. Последнюю задачу призвана решать теория и методология сервис-ориентированной архитектуры (СОА). Однако при решении этой задачи системные архитекторы обычно сталкиваются с рядом трудностей, обусловленных различными трактовками этой теории.

Несмотря на многолетнюю историю существования СОА и обилие внедренных в соответствии с этой архитектурой масштабных проектов, не существует ее единой общепризнанной теории и методологии. Фактически каждый крупный поставщик программного обеспечения (вендор) имеет собственную методологию СОА или, что еще хуже, набор собственных методологий. Сервис-ориентированная архитектура по своей природе независима от поставщиков технологий и рассчитана на решения, использующие разные технологии, и подобный разнобой в методологиях ведет в конечном итоге к снижению качества внедряемых в архитектуре СОА проектов.

Наличие множества методологий выгодно прежде всего самим поставщикам, которые, пользуясь отсутствием единой методологической базы, подгоняют особенности своих методологий СОА под собственные линейки продуктов. Поставщик не упускает возможности подстроить теорию СОА под свои маркетинговые потребности.

Основными владельцами и разработчиками методологий СОА являются IBM (в блоке с консорциумом The Open Group) и компания Arcitura, бывшая SOA School. Другие крупные поставщики программного обеспечения (Oracle, Microsoft, Red Hat, Progress Software) также имеют свои, не столь проработанные методологии, использующие так или иначе идеи IBM и/или Arcitura. В последнее время налицо центробежные тенденции, когда поставщики склонны более прорабатывать собственные методологии СОА, чем договариваться о единой стандартной методологии.

Целью настоящей работы является анализ существующих методологий СОА и их систематизация. Источниками информации являются открытые ресурсы на интернет-сайтах поставщиков. Сайты многих вендоров имеют отдельные разделы, посвященные СОА. Методология Arcitura изложена по книгам Томаса Эрля. Методология Oracle изложена по соответствующим книгам, рекомендованным компанией Oracle.

Анализ строится на сопоставлении методологии Arcitura (как наиболее проработанной) с методологи-

ями других поставщиков и консорциумов, в том числе IBM, Oracle, Microsoft, The Open Group, OASIS, Red Hat, Progress Software. Выделяются основные идеи конкретной методологии (концепты) и проводится сравнение с аналогичными идеями методологии Arcitura и, по возможности, с другими.

Краткая история SOA

Сервис-ориентированная архитектура не является революционной идеей. Концепция SOA стала результатом эволюционного развития множества теорий и концепций, имеющих отношение к распределенным компьютерным вычислениям и интеграции: от объектно-ориентированной парадигмы до стандартов CORBA и DCOM.

Термин SOA впервые появился в статье аналитиков Гартнер [1] в 1996 г. Статья содержала определение SOA как стиля многозвенных вычислений[1] и описание преимуществ сервис-ориентированных конфигураций.

Практическое развитие SOA получила в начале 2000-х гг. с появлением и развитием веб-сервисов. Веб-сервисы хорошо подходили для реализации теоретических идей SOA и поддерживались всеми основными поставщиками программного обеспечения на уровне международных стандартов. Классические веб-сервисы являются основной технологией реализации SOA — хотя в последнее время они испытывают конкуренцию со стороны RESTful-сервисов[2].

В середине 2000-х гг. появились первые стандарты и методологии. В 2004 г. Али Арсаньяни (IBM) написал статью [2], положившую начало SOMA — сервис-ориентированному моделированию и анализу. В 2006 г. группа OASIS опубликовала стандарт [3], содержащий основные определения SOA.

В те же годы получают развитие следующие концепции: интеграционной шины (ESB), управления бизнес-процессами (BPM), второго поколения веб-сервисов (стандарты WS-*). Эти концепции начинают играть важную роль в SOA и находят свое отражение в методологиях. В 2008 г. Томас Эрль издает книгу "Шаблоны проектирования SOA", взятую на вооружение основными вендорами. В это время SOA находится на пике популярности, в крупных компаниях создаются SOA-подразделения, внедряется большое число проектов. Поставщики программного обеспечения создают специализированные пакеты для внедрения SOA (SOA Suites). Возникает система сертификации SOA-специалистов (компаний IBM, Oracle, Arcitura).

В настоящее время тема SOA утратила ореол популярности и моды и перешла в режим спокойного функционирования и эволюционного развития. Объединение концепции SOA и идей событийного программирования (CEP) получило развитие в SOA 2.0. Идеи SOA активно используются в популярных в наше время облачных технологиях.

Подход Arcitura

Компания Arcitura (бывшая SOA School) работает в области образования и теории SOA, управления бизнес-процессами (BPM) и облачных технологий со второй половины 1990-х гг. Ее лидер Томас Эрль является автором книг по SOA и смежным направлениям, он признанный в IT-отрасли авторитет, консультирующий многие крупнейшие корпорации — от Пентагона до российского Сбербанка. Методология SOA компании Arcitura является одной из наиболее проработанных и востребованных.

Основы SOA. Arcitura определяет сервис как единицу прикладной логики с сервисной ориентацией [4]. Сервисная ориентация — это парадигма проектирования, состоящая из набора принципов дизайна [5]. Таким образом, сервис определен как прикладная сущность, удовлетворяющая следующему набору принципов дизайна [5]: стандартизированный контракт; слабая связь; абстракция; повторное использование; автономия; отсутствие сохранения контекста; понятность (*discoverability*); способность к композиции.

Сервис-ориентированная архитектура определена как архитектурная модель, состоящая из сервисов [4].

Теория SOA, помимо общих принципов, состоит из набора важных для SOA концептов, развитой системы шаблонов проектирования и организационных аспектов методологии построения корпоративной SOA.

Концепты SOA. В данном подходе выделяют следующие концепты.

- Сервисная композиция — согласованная совокупность сервисов [5]. Сервисы в композиции могут занимать роли контроллера, подконтроллера или члена композиции. В теории мало сказано о возможной прикладной реализации сервисных композиций. Однако в прикладных книгах Эрля ("SOA with .NET", "SOA with Java"), служащих дополнением к основной методологии, достаточно большое внимание уделено вопросам интеграционных шин (ESB) и бизнес-процессов (BPM). Поэтому реализацией сервисных композиций являются прежде всего процессы BPM и ESB.
- Хранилище сервисов — коллекция независимых, стандартизированных и управляемых сервисов, которые определяются функциональными потребностями предприятия или его части [5]. Хранилища могут быть разных уровней — уровня предприятия или его домена (части предприятия).
- Arcitura выделяет следующую классификацию сервисов (сервисные модели) [4]:
  - сервисы сущностей;

[1] "A service-oriented architecture is a style of application partitioning and targeting (placement). It assumes multiple software tiers and usually has thin clients and fat servers (i.e., little or no business logic on the client), but it is more than that. It organizes software functions into modules in a way that maximizes sharing application code and data".

[2] Томас Эрль готовит отдельную книгу, посвященную вопросам построения SOA с помощью RESTful-сервисов.

- сервисы задач;
- утилитарные сервисы.

• В теории подчеркивается, что СОА не зависит от конкретной прикладной платформы и технологии. Однако большая часть материала ссылается на веб-сервисы, и эта технология воспринимается как основная, на базе которой можно строить СОА масштаба предприятия.

• Интеграционные шины (ESB) упоминаются в теории в двух ролях: как предтеча и источник идей для СОА, а также как шаблон проектирования. В шаблоне ESB представлены основные компоненты шины и их роли. Шаблон представляет абстрактную интеграционную шину с минимальными функциональными возможностями. В практических руководствах, описывающих построение СОА на Java и .NET, акцентируется возможность построения СОА без шин и приводятся примеры сервисных композиций на базе интеграционных шин.

• Аналогичную ESB роль в теории СОА играет и теория управления бизнес-процессами (BPM), с теми же двумя ролями — предтечи-источника идей и практической реализации сервисных композиций.

Архитектура СОА обладает следующими преимуществами по сравнению с предшествующими типами архитектур [5]: тесные внутренние взаимодействия сервисов; объединение (федерация) сервисов в рамках организации; диверсификация вендоров; лучшее согласование работ подразделений бизнеса и IT; увеличение отдачи от инвестиций (ROI); повышение гибкости организации; сокращение нагрузки на IT-подразделения.

Шаблоны проектирования. Arcitura обладает развитым набором шаблонов проектирования СОА по следующим направлениям: хранилища сервисов; фундаментальные шаблоны; шаблоны реализации сервисов; шаблоны сервисов безопасности; шаблоны проектирования сервисных контрактов; шаблоны связи с унаследованными системами; шаблоны управления СОА; шаблоны композиции сервисов; составные шаблоны.

Шаблоны СОА являются наиболее популярной частью теории СОА компании Arcitura, и ни в одной другой методологии не встречается столь проработанной коллекции шаблонов. Они охватывают весь цикл жизни сервисов — от первичного проектирования (фундаментальные шаблоны и, в частности, принцип функциональной декомпозиции) через проектирование (например, каноническая схема) и разработку (например, обратный вызов сервиса) к сопровождению и организационным вопросам практического построения СОА в организации.

Организационные аспекты. Управление СОА в теории Arcitura делится на две части:

— управление СОА-проектами;

— стратегическое управление СОА.

В первой части определяется цикл жизни сервисов (аналогично моделям зрелости методологий-конкурентов) — от анализа и дизайна через разработку и тестирование к внедрению и эксплуатации. Предлагается методика версионности сервисов и описываются особенности проектных команд и ролей в СОА-проектах.

Во второй части определяются политики, накладываемые на сервисы, а также рекомендуются технологии и инструментальные средства для управления СОА.

Деятельность компании Arcitura не ограничивается только вопросами теории СОА. Компания ведет обширную образовательную программу. Доступна богатая сетка сертификаций по СОА — от профессионала СОА до архитектора и аналитика СОА. Arcitura инициировала создание СОА-симпозиума — площадки для выработки решений, касающихся СОА. В их продвижении участвуют все основные поставщики программного обеспечения, предлагающие решения по СОА.

Подход OASIS

OASIS — это некоммерческий консорциум, созданный для продвижения открытых стандартов. Консорциум основан группой компаний, в число которых входят IBM и Microsoft. Спонсорами OASIS являются Oracle, Red Hat и другие крупные вендоры. Методология СОА изложена по документу [3], имеющему статус стандарта. Методология OASIS оказала большое влияние на современную методологию СОА компании IBM.

Основы СОА. Методология описывает архитектуру, принципиально независимую от конкретных технологий и продуктов, таким образом она обладает высокой степенью абстракции.

В данном подходе СОА — это экосистема, т. е. место, где люди, машины и сервисы взаимодействуют друг с другом. Отметим, что такое определение принципиально отличается от понятия СОА у компании Arcitura, описывающей только сервисы.

В экосистеме отсутствует иерархия, т. е. каждый член экосистемы (человек, машина или сервис) является равным участником.

Ключевыми принципами экосистемы СОА являются следующие:

• СОА — это среда обмена благами между независимо действующими участниками;

• участники (а также заинтересованные стороны) имеют одинаковые права претендовать на собственность ресурсов, доступных с помощью СОА;

• поведение и производительность участников определяются правилами, зафиксированными в соответствующих политиках и контрактах.

Целями СОА являются: эффективность, конфиденциальность, масштабируемость.

В качестве принципов СОА рассматриваются нейтральность по отношению к технологии; экономия (избегание сложностей в дизайне); разделение интересов; применяемость.

Отметим кардинальное отличие этих принципов от принципов дизайна сервисов у Arcitura (стандартизированный контракт, слабая связь и др.). Это объясняется более широким определением СОА как экосистемы.

Модели СОА. В методологии описывается большое число моделей по следующим срезам: сервисы с точки зрения бизнеса; реализация СОА; управление СОА.

Список основных описываемых в рамках методологии OASIS моделей включает следующие модели: заинтересованных сторон и участников; ресурсов; прав собственности; необходимых функций и возможностей; социальной структуры; действий в социальном контексте; ролей в социальных структурах; управления социальными структурами; предложения; описания сервисов; видимости сервиса; взаимодействия с сервисом; политик и контрактов; управления; безопасности; сервисов как управляемых сущностей.

Если сравнивать данную методологию с методологией Arcitura, то можно заметить, что бóльшая часть этих моделей не имеет аналогов у Arcitura вследствие различных объектов анализа. Некоторые модели OASIS имеют механизмы, которые используются в Arcitura в виде принципов (модель исполнения контекстов OASIS и принцип понятности Arcitura) или шаблонов.

Уже на примере сравнения методологий Arcitura и OASIS видно, насколько сильно они отличаются на уровне принципов, сущностей, а также самих объектов анализа.

Подход The Open Group

The Open Group — это глобальный консорциум, конкурент OASIS, занимающийся вопросами продвижения IT-стандартов. Платиновыми членами консорциума являются IBM и Oracle. Членами консорциума являются фактически все значимые IT-компании. Методология СОА компании The Open Group изложена в книге [6]. Теория СОА компании The Open Group является основой современной методологии компании IBM. Методология СОА компании The Open Group — прямой конкурент методологии Arcitura, претендует на роль ведущей методологии СОА.

Методология СОА компании The Open Group включает основные и дополнительные понятия СОА (именуемые строительными блоками СОА), инфраструктуру СОА, модели зрелости сервисов и СОА-онтологии.

Основы СОА. Сервис — это логическое представление повторяемого бизнес-процесса [6]. Сервис самодостаточен. Сервис может состоять из других сервисов. Сервис является черным ящиком для своих потребителей.

Сравнивая это определение с соответствующим определением Arcitura, можно заметить, что определение сервиса The Open Group является упрощением по отношению к Arcitura. Последняя добавляет в определение сервиса принципы стандартизованного контракта, слабой связи, отсутствия сохранения контекста и понятности. Другие принципы Arcitura (автономия, повторное использование, абстракция, способность к композиции) присутствуют в определении The Open Group. Отметим, однако, что принципы СОА у Arcitura проработаны гораздо глубже, чем у The Open Group.

Сервис-ориентированная архитектура определяется как архитектурный стиль, который поддерживает сервисную ориентацию. Сервисная ориентация — это стиль мышления в терминах сервисов, сервисной разработки и сервисных результатов [6].

Архитектурный стиль СОА обладает следующими свойствами:

- стиль строится на дизайне сервисов, отражающих реальную бизнес-активность;
- сервисы подчиняются бизнес-правилам, для сервисных композиций применяется оркестровка сервисов;
- инфраструктуре СОА рекомендуется использовать открытые стандарты для повышения уровня взаимодействия и прозрачности;
- реализации сервисов зависят от технического окружения, они должны определяться контекстом и описываться внутри этих контекстов;
- стиль требует строгого управления описанием и реализацией сервисов;
- стиль требует "лакмусовых тестов", определяющих качество сервисов.

Arcitura определяет сервисную ориентацию как парадигму, состоящую из восьми принципов сервисного дизайна (сервисный контракт, слабая связь и др.). Arcitura не содержит на уровне базовых принципов рекомендаций применять конкретные технологии (открытые стандарты рассматриваются наряду с закрытыми технологиями поставщиков программного обеспечения). Также Arcitura настаивает на построении сервисов, независимых от конкретных технологий и технических окружений. Эти различия в подходах Arcitura и The Open Group принципиальны.

Сервис в методологии The Open Group имеет провайдера, потребителей и выдает результат, необходимый потребителям. Здесь нет концептуальных разногласий с Arcitura.

Основные понятия СОА. *Бизнес-процесс* — совокупность взаимосвязанных мероприятий и задач предприятия, осуществляемая определенным повторяемым способом. Отметим, что в методологии Arcitura нет понятия бизнес-процесса.

Эктор (*Actor*) — сущность (человек или программа), осуществляющая какие-либо действия. Аналогов этому в методологии Arcitura нет.

Событие — то, что может произойти. Оно не обязательно связано с бизнес-процессами. Аналогов этому определению в методологии Arcitura нет.

Отсутствие аналогов этих терминов, присущих теории управления бизнес-процессами, в определениях Arcitura служит хорошей иллюстрацией различ-

ных моделей построения СОА-методологий у Arcitura и The Open Group. Arcitura пытается выделить чистую СОА, вне других теорий и методологий (прежде всего, управления бизнес-процессами BPM). The Open Group включает элементы теории BPM в свою методологию СОА.

Сервисное описание — это описание сервиса словами и диаграммами. Аналогом в Arcitura служит принцип понятности сервиса.

Сервисный контракт — это соглашение между провайдером сервиса и его клиентами. Принцип сервисного контракта в Arcitura является одним из базовых и проработан намного лучше. Понятию сервисного контракта посвящена отдельная книга Томаса Эрля (Web Service Contract Design & Versioning for SOA), где подробно объяснены стандарты веб-сервисов и приводятся практические примеры использования.

Сервисная политика — набор действий, удовлетворяющих требованиям провайдеров и потребителей. В методологии Arcitura политики сервисов определяются как части контрактов.

Сервисная композиция — коллекция совмещенных сервисов, действующих как один сервис. В методологии Arcitura принцип сервисной композиции является базовым и, соответственно, более глубоко проработанным: выделяются контроллеры, подконтроллеры и участники композиций, их свойства, вводятся понятия примитивной и сложной композиции и др.

Программа — набор компьютерных инструкций для решения определенной задачи. В методологии Arcitura нет аналогов.

Информационный элемент — сущность, обладающая знанием о чем-либо. В методологии Arcitura аналогов этому нет.

Элемент данных — представление информационного элемента. В методологии Arcitura нет аналогов.

Хранилище данных — технологичная сущность, хранящая элементы данных. В методологии Arcitura нет аналогов.

Обмен сообщениями — программы могут обмениваться сообщениями с помощью сервисов обмена сообщениями. В методологии Arcitura есть соответствующий шаблон проектирования.

Исследование сервисов — способ получения информации о сервисе. Алгоритм включает исследование хранилища сервисов и выяснение функциональных возможностей имеющихся сервисов, например, в целях принятия решения о разработке нового сервиса или применении существующего в хранилище. В методологии Arcitura есть аналогичный базовый принцип понятности сервисов и описывается похожий алгоритм.

Обертка тела сервиса — принцип взаимодействия с сервисом через фасад — специальный интерфейс с ограниченным, но необходимыми набором параметров, настроенными правами доступа. В методологии Arcitura взаимодействие соответствует базовому принципу сервисного контракта, хорошо проработанному, который включает детальное описание основных стандартов веб-сервисов.

Виртуализация сервиса — функциональные возможности сервиса определяются контрактом, реализация тела сервиса скрывается. Благодаря виртуализации сервиса появляется дополнительная свобода действий у разработчика (в этом случае не важно, на какой технологии реализовывать логику сервиса — главное, чтобы контракт остался неизменным). В методологии Arcitura есть соответствующий базовый принцип абстракции сервиса.

Обработка событий — поток событий обрабатывается специализированным событийным процессором. Аналогов этому в методологии Arcitura нет.

Управление СОА. Управление СОА — это организация и контроль деятельности и персонала на предприятии, которые связаны с разработкой, внедрением и применением СОА.

Управление СОА подчиняется следующим принципам:

- деятельность по управлению СОА должна подчиняться корпоративным, IT и архитектурным стандартам предприятия;
- СОА-сервисы и решения должны удовлетворять архитектуре предприятия;
- существующие сервисы должны учитываться в первую очередь при построении СОА-решений;
- между провайдерами и потребителями сервисов должны существовать контракты;
- процессы управления СОА должны учитывать уровни и риски проектов.

Разработка и внедрение СОА регулируются следующими документами, регламентирующими управление: "Портфолио по управлению решениями"; "Портфолио по управлению сервисами"; "Управление этапами жизненного цикла сервисов"; "Управление этапами жизненного цикла решений".

Управление СОА содержит строгий набор ролей и должностных обязанностей, например: собственник домена; бизнес-аналитик; старший архитектор и других, подобных им.

Отметим, что в подходе Arcitura похожая методика управления СОА (набор соответствующих ролей и др.), но она ориентирована более на специфику разработки и сопровождения сервисов.

Управление СОА содержит собственный метод, включающий четыре фазы: планирование; определение; внедрение; мониторинг. Этот метод является основным при внедрении СОА на предприятии. В подходе Arcitura выделено восемь аналогичных фаз: анализ; дизайн; разработка; тестирование; развертывание; использование; исследование; версионирование.

The Open Group рекомендует использовать методологию TOGAF для построения СОА, где подробно определены необходимые для ее реализации шаги и расписаны соответствующие модели дизайна. В подходе Arcitura не содержится рекомендаций по использованию дополнительных методологий.

Модель зрелости (OSIMM). Модель определяет уровень зрелости СОА на предприятии по следующим атрибутам: бизнес; организация; методы; приложения; архитектура; информация; инфраструктура.

Возможны следующие уровни зрелости: силос; интегрированный; уровень компонент; уровень сервисов; уровень составных сервисов; уровень виртуализируемых сервисов; уровень динамически настраиваемых сервисов.

Аналогичная модель зрелости Arcitura имеет существенные базовые отличия. Она оценивает только зрелость предприятий по отношению к сервисам и выделяет следующие уровни зрелости: предприятие не имеет представления о сервисах; предприятие имеет представление о сервисах; предприятие допускает применение сервисов; сервисы предприятия соответствуют его бизнес-процессам; сервисы предприятия ведутся его бизнес-подразделениями. Эти уровни зрелости описывают не уровень непосредственно сервисов как в OSIMM. Более того, Arcitura определяет сервисы как динамически настраиваемые виртуализируемые и способные быть в составе других сложных сервисов. Это означает, что все уровни зрелости по OSIMM, кроме высшего, в подходе Arcitura бы не рассматривались как СОА, поскольку сервисы в понимании этой методологии отсутствуют. Модель зрелости OSIMM определяет уровень соответствия сервисной архитектуры предприятия протекающим в нем бизнес-процессам.

СОА-онтология. СОА-онтология является технологическим стандартом, описывающим в графическом и текстовом виде все элементы СОА и их взаимосвязи. В онтологии даются основные определения СОА и детально описаны все основные понятия СОА, взаимоотношения с другими элементами, а также их свойства. Для такого описания используется UML-нотация. Основные понятия рассматривают как классы, а взаимосвязи — как связи между классами. Для описания свойств построены xml-представления. Приводятся примеры использования классов.

Отметим базовое отличие от подхода Arcitura. В методологии Arcitura не применяется UML, поскольку СОА рассматривается как надмножество над объектно-ориентированным программированием (ООП). Сервисы в подходе Arcitura не являются классами или объектами ООП, к ним неприменимо наследование, однако есть контракт, отсутствующий в ООП. По этой причине в методологии Arcitura применяется собственная графическая нотация[3].

Итак, мы видим, что методологии СОА компаний Arcitura и The Open Group описывают примерно одну область (в отличие от методологии OASIS). Однако в деталях эти методологии существенно отличаются. Иногда эти отличия касаются степени проработки материала. Однако многие отличия носят принципиальный характер. К их числу относятся вопрос нотации описания сервисов, отношение к управлению бизнес-процессами и др.

Отметим запутанность стандартов по вопросам СОА у консорциумов, занимающихся выработкой открытых стандартов. Так, в 2009 г. консорциумами The Open Group, OASIS и OMG был выпущен специальный документ "Navigating the SOA Open Standards Landscape Around Architecture". Его создание объяснялось необходимостью понимания сферы ответственности стандартов по СОА, выпущенных этими компаниями. Само существование этого документа свидетельствует об отсутствии прозрачности и сложности стандартов упомянутых компаний.

Подход IBM

Подход IBM к разработке и продвижению методологии СОА чрезвычайно важен, поскольку компания стояла у истоков СОА и имеет несомненный авторитет в IT-отрасли. По этой причине рассмотрим эволюцию методологий IBM. Здесь можно выделить три этапа этой эволюции: ранний (COMA), зрелый (Smart SOA) и текущий.

Ранний этап (COMA). Севис-ориентированное моделирование и архитектура (COMA) — одна из первых методологий СОА. COMA — это метод с ролями и последовательностями действий по производству артефактов, относящихся к идентификации, спецификации и реализации сервисных компонентов и процессов. Анализ и моделирование с помощью COMA являются технологически независимыми процессами, однако они устанавливают контекст для применения определенных технологий на поздних этапах жизненного цикла решения. Существует компонент COMA для унифицированного процесса разработки Rational (RUP) [7].

Идеи COMA впервые изложил Али Арсаньяни в работе [2] "Сервис-ориентированное моделирование и архитектура" в 2004 г. В статье обосновывается недостаточность объектно-ориентированного дизайна для проектирования сервисов: "В первую очередь, текущие методы OOAD не обращаются к трем ключевым элементам СОА — сервисам, потокам и компонентам, реализующим сервисы. Возникает потребность иметь также возможность напрямую обращаться к методикам и процессам, необходимым для идентификации, определения и реализации сервисов, их потоков и композиции..."

Сервис определяется как имеющий три стороны: поставщик; потребитель; посредник. Посредник определяется как обладающий спецификой компонент для обслуживания реестра сервиса.

Выделяются такие свойства сервиса, как наличие слабой связи, способность к масштабированию и формированию композиций, возможность вызова через стандартные протоколы и наличие описания. В целом, эти свойства близки более поздним определениям Arcitura и The Open Group.

[3] Вопросы нотации СОА окончательно не решены. Помимо нотации Эрля, наиболее перспективной нотацией можно считать SOA ML консорциума OMG, являющуюся адаптацией UML к специфике СОА.

Для составных сервисов рекомендуется использовать хореографию сервисов и возможности интеграционной шины (ESB). Отметим важную роль интеграционной шины, возникшую еще в первых методологиях IBM.

Архитектура СОА делится на семь уровней, а именно: уровень операционной системы; уровень корпоративных компонентов; уровень сервисов; уровень бизнес-процесс; составной уровень; уровень обеспечения доступа или презентации; уровень интеграции.

Уровни описывают статичные архитектурные срезы СОА, к каждому из них рекомендуется разрабатывать свои пакеты документов. Отметим интеграцию на высшем уровне абстракции СОА, когда архитектура обеспечивает прежде всего решение задачи интеграции. Здесь есть важное идейное расхождение как с видением Arcitura, так и с более поздними методологиями IBM. В общем-то, СОА в понимании Arcitura предназначена для решения архитектурных задач, а не интеграции. Хотя следует заметить, что теоретически можно построить СОА в однородной среде без решения интеграционных задач.

В COMA приводится описание процесса моделирования СОА, состоящее из следующих шагов: идентификация сервисов; спецификация сервисов; реализация сервисов, компонентов и потоков.

Arcitura, очевидно, опирается на COMA в своей методологии, посвященной управлению СОА.

Smart SOA. Методология Smart SOA изложена в книге [7]. Эта методология определяется как стиль для создания архитектуры уровня предприятия на принципах сервисной ориентации с целью добиться более глубокого взаимодействия между бизнесом и поддерживающими его IT-системами. Сервисная ориентация — это принцип представления бизнеса как набора сервисов. Сервис определяется как повторяющаяся задача внутри бизнес-процесса. Сервисы являются самоописываемыми объектами, удовлетворяют требованиям качества сервиса, являются управляемыми структурами и могут объединяться в составные сервисы. Эти базовые определения близки аналогичным определениям Arcitura.

Smart SOA строится на следующих концептуальных составляющих: первые шаги СОА; модель зрелости сервисов интеграции (SIMM); сервис-ориентированное моделирование и архитектура (COMA); моделирование компонентов бизнеса (CBM); метод управления СОА.

Первые шаги СОА определяют абстракции СОА высокого уровня, к числу которых относятся: люди; процессы; информация; повторное использование; доступность. Эта составляющая является развитием положений методологии OASIS Group.

Модель зрелости сервисов интеграции — это модель, принятая консорциумом The Open Group (OSIMM), которая первоначально разработана инженерами IBM и рассмотрена более подробно в предыдущем разделе.

Моделирование компонентов бизнеса — способ моделирования и анализа предприятия. Представляет собой карту областей бизнеса с определенными характеристиками, возможностями и процессами. Выделяются три уровня абстракции предприятия, а именно стратегический; управляющий; исполнимый.

Метод управления СОА включает стратегические вопросы управления СОА. К их числу относятся: выявление лиц, ответственных за функционирование сервисов; определение модели обмена сообщениями (синхронный/асинхронный); вопросы безопасности сервисов; вопросы версионности, тестирования и мониторинга сервисов. Управление СОА в главном созвучно соответствующим идеям компаний Arcitura и The Open Group.

Smart SOA предлагает для реализации СОА набор IBM-продуктов из линеек Rational, WebSphere и Tivoli.

Smart SOA имеет много пересечений с методологией Arcitura. Существенным отличием является принципиальная независимость методологии от конкретного поставщика программного обеспечения в Arcitura и вполне естественная привязка к продуктам IBM в Smart SOA.

Современная ситуация. Современное понимание СОА в IBM строится на развитии методологии The Open Group, в разработке которой IBM принимала активное участие. IBM принимает эту методологию как основную и под нее предлагает методику привязки продуктов своих линеек для построения СОА. IBM имеет подробные руководства по внедрению методологии The Open Group, а также карты соответствий принципов методологии возможностям своих продуктов.

IBM базирует современную методологию СОА на следующих стандартах The Open Group: модель зрелости открытых сервисов интеграции (OSIMM); стандарт СОА-онтологии; стандарт эталонной архитектуры СОА.

Часть этих стандартов рассмотрена и проанализирована более подробно на соответствие методологии Arcitura в предыдущем разд.

Управление СОА определяется собственным стандартом SGMM, содержащим циклы разработки сервисов, их безопасность и мониторинг. Эти вопросы более подробно освещены в методологии Arcitura по управлению сервисами.

Отметим наличие у IBM шаблонов СОА. Этот набор шаблонов не так богат как в Arcitura и частично перекрывается с ним в плане функциональных возможностей.

Принципиальным отличием методологии IBM от методологии Arcitura является наличие привязки к конкретным продуктам.

Подход Oracle

Методология СОА компании Oracle не отличается строгостью и цельностью в сравнении с аналогичной методологией IBM. В методологии Oracle используются идеи как Arcitura, так и IBM.

Основы СОА. Сервис — это термин, который понимают и бизнес-подразделения, и IT-подразделения. Сервис обладает следующими свойствами [8]:

- свойство инкапсуляции — сервис создает разграничение между своими провайдером и потребителем;
- сервис имеет интерфейс, который определяется в терминах входных и выходных параметров;
- сервис характеризуется контрактом или соглашением об уровне сервиса (SLA), который отражает атрибуты, описывающие качество сервиса, производительность, доступность и стоимость.

Сервис — это объединение (инкапсуляция) данных и бизнес-логики. Сервис состоит из интерфейса, имеет реализацию и определенное поведение. Интерфейс определяет набор операций, обеспечивающих функциональные возможности сервиса. Провайдер сервиса обеспечивает выполнение контрактов на основе интерфейса. Потребители сервиса взаимодействуют с сервисом посредством контракта. Функциональные возможности сервисов могут быть выделены из разных источников, в том числе баз данных, унаследованных приложений, программ на разных языках программирования [9]. Отметим сходство определений с соответствующими идеями Arcitura и расхождение с ними в деталях. В подходе Arcitura, например, не выделяются понятия интерфейса и операций, которые являются атрибутами контракта.

СОА не зависит от применяемой технологии. Однако СОА ассоциируется с веб-сервисами и связанными с ним стандартами OASIS GROUP [9].

В построении архитектуры Oracle ссылается на документы The Open Group (например, SOA Reference Architecture). Классификация сервисов на сервисы сущностей и сервисы задач взята из методологии Arcitura [9].

Для выполнения своих задач сервисы должны удовлетворять следующим условиям [9]: слабая связь; понятность (простота восприятия); прозрачность обнаружения как возможность исполнения сервиса из любого места; автономия; управление состоянием (желательно не хранить состояния, чего, однако, иногда нельзя избежать); возможности повторного использования; способность к композиции.

Можно заметить, что здесь повторяются, хотя и улучшенные, принципы СОА компании Arcitura.

В методологии упоминаются также шаблоны проектирования СОА-решений от компании IBM, включая бизнес-шаблоны; шаблоны интеграции; композитные шаблоны; шаблоны приложения; шаблоны периода исполнения.

Методология СОА компании Oracle служит теоретической частью продукта Oracle Fusion Middleware.

Подход Microsoft

Microsoft имеет собственную методологию СОА, которая включает базовые положения для решений на основе продуктов своей линейки (WCF, WPF, .NET).

Основы СОА. Сервис в рамках подхода Microsoft представляет собой компонент, способный выполнить свою задачу. Возможности сервиса описываются с помощью языка WSDL (*Web Service Dafinition Language*).

Определение сервиса — сущность, с помощью которой удовлетворяются потребности клиентов сервиса в соответствии с контрактом.

Выполнение сервиса — реализация функциональных возможностей экземпляра сервиса.

Сервис-ориентированная архитектура — набор исполнимых компонентов с опубликованным описанием интерфейсов.

Сервис должен удовлетворять следующим принципам дизайна [10]: независимость по отношению к технологии; использование стандартных протоколов; понятность; повторное использование; абстракция реализации; опубликованные интерфейсы; наличие формального контракта; релевантность уровня гранулярности. Отметим сходство этих принципов с принципами дизайна СОА компании Arcitura.

К ключевым компонентам СОА относятся [11]: методологии Microsoft; сервис; сообщение; динамичное обнаружение; масштабируемость сервисов; веб-сервис.

Отметим наличие технологической привязки к веб-сервисам на уровне ключевых понятий методологии, что является существенным отличием от методологии Arcitura, настаивающей формально на независимости методологии от технической реализации.

Подход Red Hat

Компания Red Hat имеет линейку продуктов JBoss, в том числе SOA Platform. Базовые положения последнего продукта и определяют СОА-методологию Red Hat.

Основы СОА. Сервис-ориентированная архитектура — это парадигма дизайна программного обеспечения. Сервисы не обязательно должны являться веб-сервисами. Сущности, представленные протоколами FTP или JMS, также могут являться сервисами [12].

Сервис-ориентированная архитектура — это архитектурный стиль, имеющий целью достижение слабой связи между взаимодействующими программными агентами.

Сервис — это единица работы провайдера сервиса для осуществления результата, необходимого клиенту. Провайдер и клиент — это роли, которые берут на себя программные агенты [13]. Отметим наличие концепта "программный агент", отсутствующего в других методологиях.

В СОА существуют три роли: провайдер сервиса; сущность, запрашивающая сервис; сервисный брокер, отвечающий за связь двух предыдущих ролей.

Отметим ориентацию методологии на интеграционную шину, что является важным идейным отличием от других методологий СОА (в определении Arcitura интеграционная шина — это лишь один из архитектурных шаблонов).

Отметим также, что во многих публичных выступлениях, посвященных СОА, Red Hat ссылается на методологию Arcitura [14].

Подход Progress Software

Компания Progress Software имела[4] линейку продуктов СОА (Sonic, Actional). Базовыми положениями проектирования этих продуктов является собственная методология СОА. Отдельные положения этой методологии используют и другие вендоры, например, Microsoft.

Основы СОА. Сервис-ориентированная архитектура в подходе Progress Software — это ориентированный на практическое использование создаваемой системы архитектурный стиль, обеспечивающий интеграцию и повторное использование процессов и сервисов [15].

В рамках этого подхода используют следующие принципы дизайна сервисов [16].

- Сервис инкапсулирует определенные данные и поведение. Вызываемый сервис называется провайдером сервиса, вызывающий — потребителем сервиса.
- Сервис должен поддерживать определенный интерфейс (контракт), который могут использовать потребители сервиса.
- Сервис должен внедряться таким способом, чтобы быть доступным для вызова со стороны другого программного обеспечения с помощью стандартных протоколов.
- Сервис не имеет избыточных зависимостей от программного обеспечения, что позволяет легко их комбинировать.

Эти принципы хорошо соотносятся с принципами дизайна СОА компании Arcitura.

Методология, помимо общих принципов СОА, содержит модель зрелости СОА, которая включает следующие уровни зрелости СОА на предприятии: начальные сервисы; архитектурные сервисы; совместные и бизнес-сервисы; измеряемые бизнес-сервисы; оптимизированные бизнес-сервисы.

Модель зрелости эквивалента аналогичным моделям компаний Arcitura и The Open Group.

СОА и облачные технологии

Облачные технологии подчиняются стандартам СОА. В частности, эталонная архитектура СОА (SOA RA) компании The Open Group применима к облачным технологиям [17]. Основные понятия СОА этой методологии — сервисные компоненты, сервисы, бизнес-процессы и др. — относятся и к облачным технологиям. Для облачных технологий предъявляются требования, отражающие специфику отдельных архитектурных слоев. Эти требования касаются доступности, производительности, масштабируемости и безопасности проектируемых продуктов.

В методологии Arcitura облачные технологии рассматривают наряду с СОА-архитектурой, дают важные определения, при управлении проектами выделяют роли, характерные для облачных технологий.

Заключение

Методологии СОА прошли достаточно долгий эволюционный путь. В настоящее время можно наблюдать соперничество двух существенно отличающихся методологий, а именно — Arcitura и The Open Group/IBM. Другие крупные поставщики ПО пытаются выстраивать собственные методологии СОА прежде всего в интересах своих маркетинговых потребностей, как правило, используя идеи двух основных методологий.

[4] В конце 2012 г. Progress Software продала линейку продуктов СОА компании Trilogy.

Список литературы

1. **Schulte W. R., Natis Y.** "Service Oriented" Architectures, Part 1. URL: http://www.gartner.com/id = 302868
2. **Арсаньяни А.** Советы по программированию Web-сервисов: Сервис-ориентированное моделирование и архитектура. URL: http://www.ibm.com/developerworks/ru/library/ws-soa-design1/
3. **OASIS** Reference Architecture for Service Oriented Architecture. URL: http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/csd03/soa-ra-v1.0-csd03.pdf
4. **Erl T.** SOA Design Patterns. Upper Saddle River, NJ: Prentice Hall, 2009.
5. **Erl T.** SOA Principles of Service Design. Upper Saddle River, NJ: Prentice Hall, 2008.
6. **The Open Group**. SOA Source Book. What is SOA? URL: http://www.opengroup.org//soa/source-book/soa/soa.htm
7. **Woolf B.** Exploring IBM SMART SOA Technology & Practice. Florida: Maximum Press, 2008.
8. **Reynolds A., Wright M.** Oracle SOA Suite 11g R1 Developer's Guide. Birmingham, UK: Packt Publishing, 2010.
9. **Heidi B., Manas D., Jayaram K., Demed L., Prasen P.** Getting Started with Oracle SOA Suite 11g R — a hands-on tutorial. Birmingham, UK: Packt Publishing, 2009.
10. **Understanding** Service Oriented Architecture. URL: http://msdn.microsoft.com/en-us/library/aa480021
11. **Service** Oriented Architecture. URL: http://msdn.microsoft.com/en-au/architecture/aa948857
12. **JBoss** Enterprise SOA Platform 5 ESB Services Guide. URL: http://docs.redhat.com/docs/en-US/JBoss_Enterprise_SOA_Platform/5/pdf/ESB_Services_Guide/JBoss_Enterprise_SOA_Platform-5-ESB_Services_Guide-en-US.pdf
13. **JBoss** ESB. Basics of SOA. URL: http://www.jboss.org/jbossesb/resources/SOABasics.html
14. **Презентация** "Designing SOA with JBoss Tools". URL: http://www.slideshare.net/eschabell/designing-soa-with-jboss-tools
15. **SOA Infrastructure** Progress Software. URL: http://www.progress.com/en/soa-infrastructure.html
16. **Principles** of a service-oriented architecture. URL: http://www.progress.com/en/soa-infrastructure.html
17. **The IBM advantage** for SOA Reference Architecture Standard. URL: http://www.ibm.com/developerworks/webservices/library/ws-soa-ref-arch/

УДК 004.7; 004.056; 004.94

А. А. Зензинов, студент[1], программист[2], e-mail: andrey.zenzinov@gmail.com,
Л. К. Сафин, студент[1], программист[2], e-mail: safin.l91@gmail.com,
К. А. Шапченко, канд. физ.-мат. наук, стар. науч. сотр.[2,3], e-mail: shapchenko@iisi.msu.ru,
[1] МГУ имени М. В. Ломоносова,
[2] НИИ механики МГУ имени М. В. Ломоносова,
[3] Институт проблем информационной безопасности МГУ имени М. В. Ломоносова

К созданию виртуальных полигонов для исследования распределенных компьютерных систем

Представлен подход к автоматизации процесса создания виртуальных макетов для одного достаточно широкого класса распределенных вычислительных систем, предназначенных для проведения научных исследований. Проанализированы существующие способы автоматизации отдельных действий и изложены практические результаты, полученные авторами в рамках создания и апробации прототипов программных средств для создания виртуальных макетов.

Ключевые слова: *распределенные компьютерные системы, грид-вычисления, виртуализация, автоматизация, информационная безопасность*

Введение

Разработка и исследование технологий и программных средств в составе распределенных компьютерных систем, в том числе средств защиты информации, как правило, включают в себя большие объемы работ на этапах проектирования, разработки и проведения тестовых испытаний программного обеспечения (ПО). Ряд таких действий предполагает выполнение экспериментов с рассматриваемой распределенной системой. Для более эффективной их реализации представляется целесообразным проводить испытания не на реальной распределенной системе (в том числе по той причине, что проектируемая система может еще не существовать), а на ее макете. На таком макете, в условиях близких к реальным можно испытывать программные компоненты исследуемой системы. Подобные макеты для проведения испытаний — экспериментальные полигоны — широко используют при разработке и исследованиях различного рода распределенных систем [1, 2].

Использование средств виртуализации для построения макетов позволяет моделировать различные архитектурные решения в распределенных системах. При этом существенно упрощается процесс построения (развертывания) модели распределенной системы и подготовка к проведению экспериментов. Идея состоит в использовании одного или нескольких компьютеров ("хостов" виртуализации) с развернутыми на них наборами виртуальных машин (ВМ), на которых поддерживается ПО узлов моделируемой распределенной системы. Такой подход широко используют при развертывании компонентов распределенных систем, например, в облачных вычислениях. Поддержка аппаратной виртуализации на хостах виртуальных машин обеспечивает функционирование виртуального макета с относительно низкими накладными расходами.

Постановка задачи

В настоящей работе в качестве класса моделируемых распределенных систем рассматриваются системы грид-вычислений, предназначенные для параллельного исполнения вычислительных заданий. В таких системах, в отличие от систем облачных вычислений вида "инфраструктура как сервис", вычислительные узлы не являются ВМ. Для моделирования систем с виртуальными узлами необходимо иметь возможность реализации систем управления ВМ, именуемых гипервизорами, на другой виртуальной системе. Добиться этих целей позволяет технология вложенной виртуализации [3]. Однако в настоящее время эта технология в применении к широко распространенным гипервизорам находится на ранней стадии разработки и пока не имеет полноценной реализации для ее эффективного использования на практике. По этим причинам вложенная виртуализация в данной работе не рассматривается.

Задачи, при решении которых используют виртуальные макеты таких распределенных систем, направлены на исследование средств обеспечения их информационной безопасности (средств защиты информации). Пусть существует распределенная система, работающая на основе Globus Toolkit — набора ПО и служб для выполнения грид-вычислений. Для тестирования средств обеспечения безопасности ресурсов такой системы необходимо учитывать различные варианты действий злоумышленника, в том числе тех, моделирование которых требует проведения большого объема разноплановых экспериментов. К их числу относятся: атаки вида "отказ в обслуживании" DoS, DDoS (*Denial of Service* и *Distributed Denial of Service*); использование уязвимостей в ПО (так называемые "эксплойт"-атаки); атаки на инфраструктуру распределенной системы, включая подмену ее частей и др.

Для обоснования полноты программы экспериментов по моделированию атак необходимо рассматривать различные наборы параметров. Эти параметры включают различные архитектурные решения исследуемой системы; варианты расположения систем злоумышленника относительно исследуемой системы; конфигурации и состав систем обеспечения безопасности; сценарии проведения атак. В итоге формируется серия (набор) экспериментов на каждый вид моделируемой атаки. Так как необходимо проверять различные архитектурные конфигурации, то возникает необходимость перестроения модели исследуемой распределенной системы. Такие действия могут быть эффективно автоматизированы с использованием виртуальных макетов.

Процесс построения системы представляется в виде следующей последовательности действий:

- создание набора узлов;
- установка операционной системы (ОС) на каждом узле;
- настройка системных параметров на каждом узле;
- установка и настройка на каждом узле дополнительного ПО;
- построение сетевой инфраструктуры распределенной системы.

Реализация перечисленных действий вручную трудоемка и требует большого количества времени. При этом оператор, проводящий эксперимент, вынужден выполнять относительно однообразные и во многом повторяющиеся действия по подготовке эксперимента, а именно по развертыванию макета распределенной системы.

Как правило, оператор, производящий развертывание макета системы, имеет достаточно четко формализованное представление о ее конфигурации, которое включает описание узлов распределенной системы, ее сетевой архитектуры, набора программных средств на отдельных узлах, а также ряда других атрибутов и характеристик, которые адекватно представляют систему на этом этапе. Следует заметить, что на основании такого же набора атрибутов и характеристик выполняется построение реальной системы. Вместе с тем при построении макета распределенной системы могут быть смоделированы и дополнительные характеристики, в том числе отражающие особенности каналов связи в распределенной системе. К таким особенностям можно отнести различия в скорости и задержках передачи данных, пакетную фильтрацию, а также различные неполадки в сетях пакетной передачи данных, например, связанные с дублированием и потерей пакетов.

Процесс создания виртуального макета целевой системы требует от оператора выполнения заранее запланированных действий для каждого ее узла. Часть этих действий сопровождается запуском неинтерактивных процессов, таких как копирование дисковых образов, установка программных пакетов. Для того чтобы приступить к следующему действию оператор вынужден дождаться завершения этих процессов, на которые затрачивается значительное время. Кроме этого оператор при построении макета может допустить ошибку, которая в соответствующих настройках может привести к потере работоспособности системы. В целях исключения подобных ошибок оператора, представляется целесообразным уменьшить долю неавтоматизированных действий.

С учетом изложенного выше возникает задача в следующей постановке: необходимо автоматизировать процесс развертывания и настройки виртуального макета экспериментального полигона по определенной конфигурации. Потребуем от системы развертывания такого макета выполнения следующих требований:

- поддержка различных типов узлов распределенной системы, которые описывают заранее определенные наборы узлов, например, вычислительные, сетевые шлюзы и др.;
- готовность макета системы к выполнению функций в рамках планируемых экспериментов, а именно — на узлах должно быть установлено необходимое ПО и настроен режим удаленного доступа к узлам, например, по протоколу SSH;

- используемое в системе ПО должно быть с открытым исходным кодом в целях обеспечения его эффективной модификации;
- должна присутствовать возможность автоматизированного моделирования характеристик, отражающих особенности каналов передачи данных.

Различные типы узлов определяются поставленными перед ними задачами, которые диктуются функциональным назначением распределенной системы в целом. В системах грид-вычислений, например, могут быть востребованы следующие типы узлов: распределяющие задания; распределяющие сертификаты; сетевые шлюзы; реализующие вычислительные функции. Такие узлы обладают различными конфигурациями и составом ПО. При этом типов узлов в рассматриваемом классе распределенных систем, как правило, немного.

Подходы к автоматизации и требования к системе развертывания

Процесс развертывания макета системы — виртуального полигона — разбивается на описанные выше действия. При этом узлы моделируемой системы реализуются в виде ВМ. В силу того что виртуализация позволяет эмулировать сетевые устройства, можно реализовать основанную на них сетевую инфраструктуру моделируемой системы. Для автоматизации действий по созданию набора ВМ можно воспользоваться кросс-платформенной библиотекой libvirt [4]. Она реализует унифицированный способ управления различными средствами виртуализации, а также предоставляет свой интерфейс для написания программ на различных, широко распространенных языках программирования. Используя libvirt, а также известные приемы автоматизации с помощью сценариев командной оболочки и программ на языке Python, можно решить задачу автоматизации процессов создания набора необходимых ВМ.

Установка ОС обычно сопровождается ответами оператора на определенные вопросы программы-инсталлятора о подлежащих установке пакетах, пользователях, часовом поясе, а также других параметрах ОС. Таким образом, этот процесс, как правило, является интерактивным. Однако существуют различные решения в форме сетевой установки и установки по заданному файлу с ответами. Эти решения успешно используют во многих современных системах, например, совместимых с Debian GNU/Linux и Red Hat Enterprise Linux.

Редактирование конфигурационных файлов можно проводить автоматически с помощью инструментальных средств обработки текстовых файлов, в том числе задавая выполняемые действия с использованием регулярных выражений. Существуют специальные системы автоматизации конфигурирования ОС и ПО, такие как Chef и Puppet. Они позволяют централизованно управлять изменением конфигурации.

Резюмируя результаты анализа различных подходов к решению отдельных задач на направлении автоматизации действий по построению виртуальных макетов распределенных систем, выделим общие идеи и требования, которые стоят перед системой их развертывания. К их числу относятся:

- использование универсальных конфигураций [5, 6];
- использование шаблонов ВМ [5, 6];
- возможность создания набора копий шаблонов ВМ;
- автоматизация процедуры инициализации [6];
- возможность использовать как заранее подготовленные дисковые образы, так и автоматическую установку на новые дисковые образы;
- возможность вносить изменения вручную.

Универсальная конфигурация в рассматриваемом контексте предполагает единый подход к описанию моделируемых систем. Такой подход означает, что все такие системы имеют общий набор изменяемых параметров, которые будут определены ниже. Если в моделируемой системе большое число однообразных узлов, то представляется целесообразным использовать специальные шаблоны узлов в виде отдельных ВМ с необходимой конфигурацией и составом ПО. В таком случае узлы одного типа создаются ("клонируются") по сформированному шаблону, возможно, с внесением дополнительных изменений в настройки отдельных узлов-клонов. Существует два способа такого клонирования узлов в виде ВМ:

- полное клонирование — создание полных копий дисковых образов;
- инкрементальное клонирование — базовый дисковый образ используется в режиме "только для чтения", а изменения дисковых образов узлов-клонов записываются в отдельные файлы специального вида.

Второй из отмеченных способов позволяет значительно уменьшить необходимый для его реализации объем дискового пространства и сократить время развертывания, что особенно важно в условиях проведения большого числа экспериментов. Отметим, что некоторым аналогом такого подхода, но в применении к оперативной памяти, является использование технологий KSM (*Kernel SamePage Merging*) и UKSM (*Ultra KSM*) [7].

Что касается предъявляемого требования возможности вносить изменения вручную, то его выполнение позволяет оператору изменять отдельные параметры экспериментального полигона без необходимости его перестраивания. Кроме этого, оператор должен иметь удаленный доступ к узлам для возможности выполнения интерактивных действий. Таким образом, это позволит расширить объем проводимых тестовых испытаний, предоставляя возможность моделировать ряд случаев, когда в ходе эксперимента возникает необходимость вмешательства оператора.

Можно выделить следующие требования для задания общей конфигурации моделируемой системы:

- должны быть перечислены все необходимые типы ВМ с указанием числа создаваемых копий;
- следует указать параметры ВМ для каждого типа (выделяемые ресурсы, адрес дискового образа);
- необходимо указать общие параметры виртуализации (например, тип гипервизора ВМ);
- следует описать сетевые настройки (используемые виртуальные сети, адреса сетевых шлюзов);
- должно быть описано дополнительно устанавливаемое ПО.

Этот набор является достаточным для построения экспериментальных полигонов. Таким образом, меняя параметры универсальной конфигурации можно определять широкий класс систем.

Пример реализации

На настоящее время авторами разработана автоматизированная система развертывания ВМ с помощью библиотеки libvirt. Эта система поддерживает создание макета (виртуального полигона) распределенной системы на основе набора ВМ различных типов. Общая схема создания полигона, изображенного на рис. 1, включает следующие действия:

- задание универсальной конфигурации полигона в формате JSON (рис. 2, *а*);
- задание существующих дисковых образов шаблонов ВМ;
- создание инкрементальных копий заранее подготовленных образов, настройка сетевых конфигураций, ssh-конфигураций;
- создание XML-описания для каждого экземпляра ВМ;
- через libvirt создание и запуск ВМ по этим XML-описаниям.

Подготовка дисковых образов включает создание образа файловой системы, установку ОС и некоторого набора ПО. Как отмечалось выше, установка ОС и ПО может быть автоматизирована. Для каждого определенного в универсальной конфигурации типа узлов создаются отдельные образы. Устанавливаемое на образ ПО определяется задачами, которые призваны выполнять узлы этого типа.

На рис. 2, *а* представлена конфигурация системы, состоящей из узлов трех типов "gw" (одна единица), "node-1" (четыре единицы) и "node-2" (восемь единиц). Для каждого типа указан размер выделяемой оперативной памяти и путь к дисковому образу. В поле "network" перечислены используемые данным типом узлов сети, которые описываются в конфигурационном файле "Network.cfg" (рис. 2, *б*).

В данном примере описываются две сети. Сеть "network1" представляет собой сеть, объединяющую узлы типа "node-1". Узлы типа "node-2" подключаются к сети "network2". При этом узел "gw-1" играет роль сетевого маршрутизатора и подключается к обеим сетям сразу.

Следует отметить, что оператор вручную задает только конфигурации полигона и сети и создает шаблоны ВМ. Остальные действия выполняются автоматически. В создаваемом макете автоматически настраивается удаленный доступ к созданным ВМ по протоколу SSH.

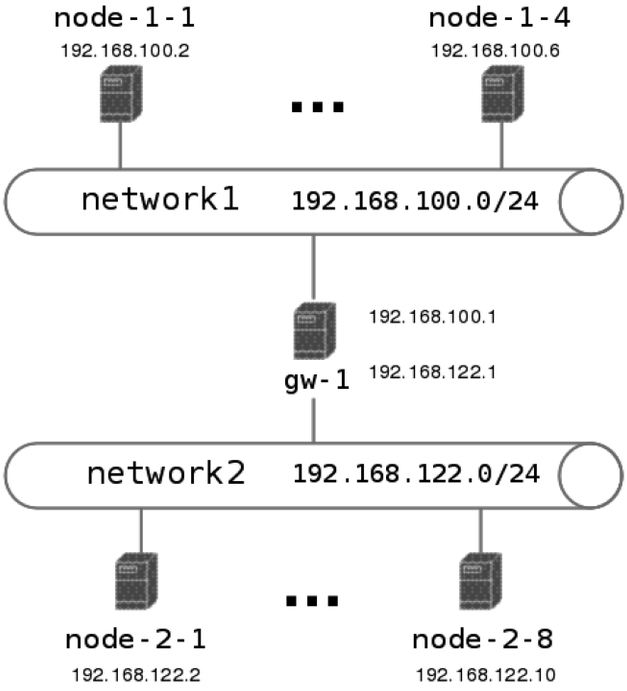


Рис. 1. Схема полигона

В процессе моделирования каналов связи используется следующее представление компьютерной сети. Рассматривается набор ВМ — узлов распределенной системы, каждому из которых присвоен уникальный IP-адрес. Кроме того, заданы изолированные виртуальные подсети (созданные с помощью средств виртуализации), для каждой из которых задан свой диапазон IP-адресов. Каждая ВМ должна принадлежать хотя бы одной из таких подсетей. Задача моделирования каналов связи для сетей в таком представлении заключается в реализации следующих функций:

- маршрутизации между подсетями;
- моделирования характеристик и особенностей каналов связи, включая их пропускную способность, задержку в передаче пакетов данных, нарушения работоспособности и/или плохую работу сети (разрывы соединения, дублирование или большие потери пакетов данных).

Вследствие того, что современные средства виртуализации не предоставляют механизмов моделирования перечисленных особенностей компьютерных сетей в полном объеме, предлагается следующий подход к решению этой задачи. Введем в модель дополнительные виртуальные узлы-маршрутизаторы, которые будут выполнять функции по моделированию каналов связи и по маршрутизации пакетов данных. Такие

```
{
"type" : "kvm",
"machines" : {
  "gw" : {
    "number" : 1,
    "memory" : "524288",
    "disk" : "/images/1.img",
    "network" : [network1, network2]
  },
  "node-1" : {
    "number" : 4,
    "memory" : "262144",
    "disk" : "/images/2.img",
    "network" : [network1]
  },
  "node-2" : {
    "number" : 8,
    "memory" : "262144",
    "disk" : "/images/3.img",
    "network" : [network2]
  }
},
"netconfig" : "Network.cfg"
}
```

а)

```
{
"networks" : {
  "network1" : {
    "range_start" : "192.168.100.2",
    "range_end" : "192.168.100.255",
    "gateway" : "192.168.100.1",
    "netmask" : "255.255.255.0",
    "nat" : "yes"
  },
  "network2" : {
    "range_start" : "192.168.122.2",
    "range_end" : "192.168.122.255",
    "gateway" : "192.168.122.1",
    "netmask" : "255.255.255.0",
    "nat" : "yes"
  }
}
}
```

б)

Рис. 2. Пример конфигураций:
а — полигона; *б* — сети (Network.cfg)

виртуальные маршрутизаторы ставят "в разрыв" того канала связи, который необходимо смоделировать. К особенностям подобной архитектуры можно отнести:

- слабые связи между узлами макета вследствие отсутствия необходимости ручной настройки маршрутизации как на узлах макета, так и на хостовой машине;
- хорошую масштабируемость, поскольку любые изменения в модели достигаются изменением конфигурации набора машин-маршрутизаторов.

Следует отметить, что поскольку виртуальные маршрутизаторы являются дополнительными ВМ, то возрастает потребление вычислительных ресурсов хост-машины.

Идея объединения подсетей продемонстрирована на рис. 1. Легко заметить, что две подсети с TCP/IP-масками 192.168.100.0/24 и 192.168.122.0/24 объединены с помощью узла-маршрутизатора. На виртуальном маршрутизаторе установлены две сетевые карты и для каждой из подсетей узел-маршрутизатор указан как шлюз. Кроме того, на узле-маршрутизаторе необходимо включить перенаправление IP-пакетов.

В общем случае, когда необходимо объединить между собой сразу несколько подсетей, на виртуальный маршрутизатор устанавливается несколько сетевых карт, по одной на каждую из подсетей. Таким образом выполняется объединение подсетей.

Набор характеристик каналов связи, которые целесообразно моделировать (пропускная способность, задержка передачи данных, нарушения работоспособности), повторяет подобные типовые характеристики, которые моделируются во многих средствах имитационного моделирования компьютерных сетей, например, в Opnet Modeler [8].

Программная реализация виртуального маршрутизатора основывается на формировании специализированного дистрибутива ОС, выполняющего функции по маршрутизации пакетов и по моделированию характеристик каналов связи. В рамках настоящей работы успешно опробованы два варианта реализации. Первый вариант основан на использовании механизмов ОС FreeBSD: межсетевого экрана ipfw и системного средства dummynet для управления трафиком, в том числе фильтрация пакетов по нескольким характеристикам, таким как ширина полосы пропускания, задержка передачи данных, процент потерь пакетов и размер очереди [9]. Второй вариант выполнен на базе ОС GNU/Linux с использованием модуля ядра Netem, позволяющего моделировать ограничение полосы пропускания, задержки передачи, потери и порчу пакетов, передачу дубликатов, а также некоторые другие характеристики пакетной передачи данных [10].

С использованием разработанных программных средств была подготовлена и проведена серия экспе-

риментов по параллельному выполнению заданий. Эксперименты продемонстрировали, что созданный с помощью разработанных средств автоматизации процесса развертывания макет распределенной системы отвечает требованиям к функциям, которые должна выполнять рассматриваемая система, в частности, ее узлам могут быть удаленно переданы вычислительные задачи для их последующего выполнения.

Испытания проводились посредством удаленного доступа к узлам системы по протоколу SSH. Были созданы программные сценарии, позволяющие в автоматизированном режиме запустить DDoS-атаку на один из узлов, остальные узлы были атакующими. Атака осуществлялась с использованием четырех узлов, с которых проводилась атака на один и тот же узел. Моделирование атаки прошло успешно. Был заблокирован доступ по сети к узлу, который подвергался атаке. Сценарий проводимого эксперимента за счет использования функциональных возможностей созданной системы развертывания являлся параметризуемым. В числе других параметров можно было задать число узлов определенного вида, диапазон назначаемых им сетевых адресов (для сценариев, имеющих большое число действующих систем). Таким образом, конфигурация моделируемой системы может изменяться, а используемый сценарий при этом остается универсальным.

Эксперименты проводились на системе с процессором Intel i5-3450 и 16 Гбайтами оперативной памяти. На таком хосте был развернут макет распределенной системы, состоящей из 200 узлов. Время на его развертывание составило 24 мин. Получить такой результат позволило использование средств дедупликации памяти, таких как UKSM. На момент создания полигона использование оперативной памяти составляло 14 Гбайт, а в течение одного часа сократилось до 7 Гбайт.

Заключение

Авторами был создан прототип программного комплекса развертывания виртуального макета распределенного сетевого полигона по его конфигурации. На макетах, создаваемых с помощью разработанных программных средств, продемонстрирована возможность моделирования процессов функционирования исследуемых распределенных систем в штатном режиме даже при наличии атак на отказ в обслуживании.

В дальнейшей работе планируется добавить поддержку автоматизированного развертывания специализированного ПО для распределенных систем и параллельных вычислений, такого как программный комплекс Globus Toolkit и реализации MPI. Планируется также добавить поддержку взаимодействия узлов физической инфраструктуры с узлами виртуальными в составе единого распределенного экспериментального полигона. Такой подход позволит расширить возможности построения макетов смешанных полигонов и увеличить возможное число узлов моделируемой системы.

Список литературы

1. **Grossman R., Gu Y., Sabata M. et al.** The open cloud testbed: A wide area testbed for cloud computing utilizing high performance network services. Препринт arXiv:0907.4810. 2009.

2. **Krestis A., Kokkinos P., Varvarigos E. A.** Implementing and evaluating scheduling policies in gLite middleware. Concurrency and Computations: Practice and Experience. Wiley, 2012. URL: http://www.ceid.upatras.gr/faculty/manos/files/papers/cpe_2832_Rev_EV.pdf

3. **Джонс М. Т.** Вложенная виртуализация для облака нового поколения. IBM.COM. 2013. URL: http://www.ibm.com/developerworks/ru/library/cl-nestedvirtualization/

4. **Libivrt**. The virtualization API. URL: http://libvirt.org

5. **Yong Kui Wang, Jie Li.** Automate VM Deployment. IBM.COM. 2009. URL: http://www.ibm.com/developerworks/linux/library/l-auto-deploy-vm/

6. **Использование** VCenter Server. V-GRADE. Виртуализация серверов и станций на базе VMware ESXi и VMware View. URL: http://www.vsphere5.ru/doku.php?id = using-vmware:vcenter-server

7. **Using** KSM (Kernel Samepage Merging) with KVM. KVM — The Linux Kernel-Based Virtual Machine. URL: http://www.linux-kvm.com/content/using-ksm-kernel-samepage-merging-kvm

8. **Тарасов В. Н., Коннов А. Л., Ушаков Ю. А.** Анализ и оптимизация локальных сетей и сетей связи с помощью программной системы Opnet Modeler. Портал магистров Донецкого национального технического университета ДонНТУ, 2003. URL: http://masters.donntu.edu.ua/2010/fkita/gaskova/library/article6.htm

9. **The dummynet** project. Luigi Rizzo. 2010. URL: http://info.iet.unipi.it/~luigi/dummynet/

10. **Netem.** The Linux foundation. 2009. URL: http://www.linuxfoundation.org/collaborate/workgroups/networking/netem

УДК 004.415.2

М. М. Гумеров, канд. техн. наук, зам. нач. отд., ООО "Уфимский научно-технический центр",
e-mail: mgumerov@gmail.com

Некоторые проблемные вопросы программирования в Delphi

Рассматриваются некоторые проблемные вопросы программирования на Embarcadero Delphi и примеры низкоэффективных решений. Объясняется природа их неэффективности и предлагаются альтернативы.

Ключевые слова: *Delphi, интерфейс, ООП, событие, агрегат, ресурсы, VCS, MSBuild*

Введение

Цель работы, описанной в статье, состоит в том, чтобы повысить эффективность создания и сопровождения программных продуктов, создаваемых при помощи среды разработки Embarcadero Delphi, путем уменьшения последствий принимаемых при разработке неэффективных решений, а также полного использования возможностей данной среды.

В настоящей работе для достижения этой цели рассматривается ряд задач, возникающих при разработке ПО, на этапах от проектирования до отладки. Рассматриваются свойственные Delphi особенности, снижающие эффективность некоторых решений таких задач (вплоть до того, что некоторые стандартные функции Delphi не работают должным образом), и предлагаются более эффективные способы их решения.

Использование интерфейсов в качестве средства абстракции

В теории использование интерфейсов вместо абстрактных классов дает дополнительную степень свободы. Это достигается за счет того, что отдельные подмножества методов класса могут быть представлены разными интерфейсами, никак не связанными друг с другом наследованием. Кроме того, классы из не связанных друг с другом иерархий могут реализовывать один и тот же интерфейс. Можно обсуждать, насколько ценны эти новые возможности на практике. Однако в данной статье автор считает нужным отметить некоторые особенности использования интерфейсов, которые могут привести к усложнению разработки и сопровождения программ.

Обычно концепция интерфейсов в Delphi объясняется исходя из нужд компонентной объектной модели COM или, реже, удаленных вызовов по протоколу CORBA. И в этом контексте, когда лишь некоторое подмножество абстракций приложения представлено в виде интерфейсов, последние ведут себя хотя и иногда неожиданно, но не очень осложняют этим весь процесс разработки. Однако ситуация меняется, если пытаться использовать интерфейсы как отдельный уровень абстракции в приложении, а именно вместо абстрактных классов.

Первая особенность состоит в том, что в Delphi восходящее приведение (*upcast*) интерфейсов, т. е. переход от интерфейса-потомка к интерфейсу-предку не обязательно приводит к вызову метода QueryInterface (или оператора as), даже если речь идет об интерфейсах COM. Вместо этого такое приведение обычно выполняется простым "усечением" (с точки зрения компилятора) интерфейса-потомка до предка, так как таблица методов интерфейса-потомка начинается с таблицы предка. Иными словами, если интерфейс *IB* унаследован от интерфейса *IA* и есть переменные *a:IA* и *b:IB*, то $a := b$ оказывается равносильно pointer(*a*) := pointer(*b*); *a._AddRef*().

Возможность приведения интерфейсов в обход вызова метода QueryInterface особенно важна в ситуациях, когда объект предоставляет свою реализацию метода QueryInterface, отличную от общепринятой. Обычно QueryInterface каким-то стандартным для

конкретного языка программирования образом возвращает определенную данным объектом реализацию запрошенного интерфейса; объект может переопределить это поведение, если ему необходимо влиять на то, каким образом он приводится к различным интерфейсам. Можно представить, например, ситуацию, в которой для удаленного (*remote*) COM-объекта может создаваться представитель (*proxy*) для данного интерфейса лишь в тот момент, когда этот интерфейс реально запрашивается каким-то кодом; или же объект может возвращать указатель на реализацию интерфейса, взятую у другого объекта (агрегированного). Приведение интерфейса в обход метода QueryInterface приведет к тому, что эта схема не сработает.

Эта особенность проявляется также при сравнении указателей на интерфейсы в целях определения, например, находится ли уже добавляемый экземпляр в некой коллекции (такой, как TInterfaceList). Для COM-объектов общее правило состоит в том, что два интерфейса относятся к одному объекту, если после приведения к интерфейсу IUnknown вызовом QueryInterface значения полученных указателей совпадают. Документация по IUnknown::QueryInterface гласит, что опрос IUnknown у любого интерфейса заданного объекта должен возвращать один и тот же указатель. Таким образом, либо коллекция должна принимать в расчет это правило (и не допускать простого "усечения" интерфейсов при сравнении, а всегда вызывать QueryInterface), либо использующий ее код должен добавлять в нее не исходные указатели на интерфейсы, а указатели уже после приведения к IUnknown (а потом код, выбирающий интерфейс из коллекции, должен опять приводить этот интерфейс к конкретному интерфейсу-потомку).

Следует также отметить, что в Delphi допускаются интерфейсы, не имеющие глобального идентификатора (GUID). Формально они не являются COM-совместимыми, и к ним неприменимо принятое в COM соглашение о равенстве указателей на IUnknown. Имеет смысл либо вообще воздержаться от использования таких интерфейсов, либо разработать какую-то собственную конвенцию относительно способа их сравнения.

Усечение интерфейсов также отражается на допустимости присваивания значения переменным-интерфейсам. Если интерфейс *IB* наследует *IA*, то этот факт означает, что объект, реализующий *IB*, реализует и методы, вошедшие в *IB* из *IA*. Однако компилятор Delphi не требует и не обеспечивает (в отличие, например, от платформы .Net) того, что этот объект будет реализовывать еще и сам *IA* как интерфейс. При этом, если объект *obj* реализует *IB*, но не *IA*, приведение такого объекта к *IA* вызовом (*obj* as *IA*) приведет к ошибке времени выполнения. Вместе с тем усечение интерфейсов неожиданным образом позволяет обойти это правило: при var *i*: *IA* присваивание *i* : = *obj* успешно выполнится. Налицо противоречие, которое следует иметь в виду.

Другая особенность Delphi касается функций, получающих интерфейс как const-параметр (неизменяемый). Как и в случае с параметрами-строками, при такой передаче не увеличивается счетчик ссылок у интерфейса. Однако в отличие от строк, для интерфейсов их объекты после конструирования имеют в счетчике ссылок 0, а не 1. Вследствие этого факта, если и при получении интерфейса счетчик тоже не увеличивается, то внутри функции вида *f* (const *instance*: IMyObject) объект *instance* будет иметь нулевой счетчик. Однако при этом счетчик вполне может изменяться уже внутри *f*, например, если присвоить *instance* другой переменной, или передавать в качестве параметра другому методу, который принимает этот параметр уже не как const. В таком случае счетчик увеличится на 1, а затем уменьшится на 1, снова достигнув 0, и сработает автоматическое разрушение объекта при достижении нулевого значения счетчика ссылок. В контексте вызова функции *f* (TMyObject.Create()) было бы еще приемлемо, если бы экземпляр разрушался после возврата из *f*(), однако в описанной ситуации он может разрушаться еще в процессе выполнения *f*(), например, в таком случае:

```
procedure f(const instance: IMyObject);
var copy: IMyObject;
begin
  copy : = instance; //вызывает instance._AddRef
  instance.DoSomething(); //работа с instance
  copy : = nil; //вызывает instance._Release

  //Теперь instance указывает на разрушенный
  //экземпляр, хотя f() не предпринимала никаких
  //мер для его разрушения
  //и пытается дальше с ним работать
  instance.DoSomething(); //работа с разрушенным
  //экземпляром приводит к ошибке
end;
```

В принципе, для обхода такого нежелательного поведения достаточно везде в коде при выполнении вызовов вида *f*(TMyObject.Create()) выполнять явный вызов QueryInterface. Более того, практика показывает, что достаточно и оператора as: *f*(TMyObject.Create() as IMyObject). Однако оба эти варианта выглядят лексически и логически избыточно. Можно избегать объявления интерфейсов как const-параметров, что удобнее. Однако если у функции три аргумента const, а четвертый — нет, возникают вопросы, какой же смысл вкладывается в то, что четвертый параметр объявлен не с такими же ограничениями на запись, как остальные три. Кроме того, опытные разработчики нередко вырабатывают привычку делать все параметры неизменяемыми, и этот автоматизм может привести к тому, что какие-то интерфейсы случайно все же будут объявлены const.

Еще одно интересное решение состоит в повсеместном следовании принципам внедрения внешних зави-

симостей (*dependency injection*): вместо создания в методе экземпляров класса передавать в этот метод фабрику, создающую такие элементы и возвращающую их интерфейсы. В этом случае можно конструирование почти любых экземпляров спрятать внутрь фабрик, а создание самих фабрик сосредоточить на верхнем уровне приложения или вовсе автоматизировать. Таким образом, поскольку на объект *instance* к моменту вызова *f*(*instance*) уже имелась интерфейсная ссылка, уничтожение ему не грозит.

Создание оберток и агрегатов с использованием интерфейсов

Выше отмечались трудности, возникающие при попытке обернуть интерфейс без его согласия. Если обертывающий объект пропускает вызовы методов *IA* через методы своей собственной реализации *IA* (т. е. осуществляет делегирование интерфейса), то таким способом он сможет делегировать лишь заранее известный набор интерфейсов. Этот факт означает, что он должен знать, что обернутый экземпляр (обозначим его *actor*) поддерживает именно определенный набор интерфейсов и никакие другие.

Как же с учетом отмеченных выше обстоятельств работают обертки-заместители (*proxy*) в COM [1]? Они генерируются автоматически для каждого интерфейса, а если метод интерфейса возвращает другой интерфейс, то вокруг возвращенной им реализации интерфейса тоже генерируется переходник. Этот механизм можно рассматривать как возможность для обертки поддерживать неограниченное число заранее неизвестных интерфейсов, добавляя поддержку по требованию. Чтобы реализовать такой подход, нужно, во-первых, иметь возможность генерировать исполняемый код в процессе работы (для чего могут оказаться нужны повышенные привилегии в системе безопасности ОС), и, во-вторых, уметь конструировать обертку, зная лишь идентификатор затребованного интерфейса. Запросить полную информацию об интерфейсе можно, если интерфейс описан в библиотеке типов, зарегистрированной в системе. Если же в рассматриваемом приложении не все интерфейсы фигурируют в библиотеке типов (ведь для этого придется отказаться в этом интерфейсе от тех типов данных, которые несовместимы с COM, например, обычного string), то непонятно, как агрегирующий объект должен распознавать структуру "недекларированных" интерфейсов.

Начиная с версий Delphi XE, у программиста есть документированный доступ к внутренней информации о типах (RTTI) выполняемой программы. Это дает большую свободу, чем запрашивание информации из библиотеки типов, поскольку информация RTTI генерируется для многих типов, не совместимых с COM. В принципе, на основе этой информации можно генерировать обертки динамически, как это происходит, например, при использовании CORBA. В то же время некоторые типы данных несовместимы и с RTTI, например, некоторые типы-перечисления.

Возможно еще одно решение. Если у обертки запрашивают интерфейс, который она не реализует, она может перезапросить этот интерфейс у *actor* (обернутого экземпляра) и вернуть его. Однако такой подход означает, что контроль над ситуацией уходит из обертки: если возвращенный интерфейс привести к любому другому, будет возвращен интерфейс из *actor*, уже не обернутый ничем. Кроме того, в этом интерфейсе могли быть какие-то методы, которые обертка, по ее логике, должна была бы обрабатывать каким-то особым образом. Именно для этого и реализуется обертывание. Более того, "ослабевает" правило симметричности QueryInterface: если у интерфейса *IA* (поддерживаемого оберткой и *actor*) опросить *IB* (поддерживаемый только *actor*), а затем у *IB* опросить *IA*, то опрос хотя и пройдет успешно (это требование к QueryInterface), однако возвращенное значение *IA* будет отличаться от исходного и укажет уже на другой экземпляр. Такое поведение QueryInterface не запрещено спецификацией. Вместе с тем оно не исключает ситуации, когда кто-либо из программистов может по ошибке написать код, рассчитывающий на полную симметрию (на то, что исходный и полученный указатели на *IA* совпадают).

Из отмеченной выше ситуации предложен изящный выход в компонентной объектной модели COM, в которой дается особое определение агрегирования. Оно понимается так, что объект, помимо своих интерфейсов, экспортирует также интерфейсы какого-то другого, агрегированного (того, которым он владеет) объекта. При этом если у агрегированного объекта опрашиваются интерфейсы, поддерживаемые агрегирующим, то он возвращает указатели на интерфейсы агрегирующего. Как нетрудно понять, такое решение (при котором сторонний код, приводя один интерфейс к другому и работая с ними, абстрагирован от того факта, что работает при этом с двумя различными объектами) требует кооперации обоих объектов. Иными словами, агрегированный объект должен знать, что принадлежит агрегирующему, и более того — какому именно. Однако если удается выполнить это условие, то такая схема работает, и можно распространить решение COM и на другие ситуации.

Впрочем, даже если обозначенная задача решена, а именно — можно генерировать обертки автоматически, или двухсторонне поддерживается агрегирование, — то для большинства задач, ассоциирующихся с созданием оберток (мостов, декораторов и др.), этого недостаточно. Этого решения достаточно для получения информации о факте вызова неизвестного заранее метода, но недостаточно для дифференцированной обработки этого факта в зависимости от метода. Для такой обработки обертка должна была бы иметь информацию не только о расположении вызванного метода в памяти или о его названии, но и о его назначении. Однако это невозможно, поскольку в обертку в момент написания ее исходного кода нельзя включить знание обо всех заранее неизвестных интерфейсах и

методах. При этом следует отметить, что обертки, за исключением отдельных классов применений, создаются именно в целях дифференцированной обработки различных вызовов, ввиду чего в общем случае автоматическое создание оберток оказывается бесполезным.

Еще одна сложность состоит в том, что объект, реализующий некий интерфейс, может поддерживать и любые другие интерфейсы. Эти другие при этом могут иметь отношение к тем функциям, которые востребованы оберткой, но могут быть этой обертке неизвестны (хотя бы на момент ее написания). Обертка, таким образом, может быть неполноценной. Вместе с тем следует признать, что такого рода ситуации провоцируются поддержкой большого числа интерфейсов одним объектом. Обычно такой подход к проектированию неудачен, поскольку означает избыточную ответственность класса (либо же единый с точки зрения бизнес-логики интерфейс разбит на несколько частей).

Создание генерализованной системы событий и подписки

Пусть существует объект, который характеризуется рядом свойств. Свойства представлены другими объектами, агрегированными данным. Свойства могут изменяться как по запросу их объекта-владельца, так и извне (например, посредством окна редактирования свойств в стиле "Object inspector"). В связи с возможностью их изменения не по инициативе владельца, появляется необходимость снабдить объект способом реагирования на изменение его свойств.

Принцип инверсии зависимостей предлагает уведомлять заинтересованных потребителей не за счет знания о конкретных потребителях во время написания программы, а за счет рассылки уведомлений списку абстрактных получателей, который во время написания программы неизвестен и в котором потребители регистрируются уже в процессе работы программы. Для этого есть классический шаблон проектирования "подписчик" (*publish-subscribe*) [2]. Однако остается вопрос о том, каким должен быть интерфейс подписки.

Прямое и архитектурно экономичное решение этого вопроса может быть следующим. Вводится общий интерфейс IEventProvider, позволяющий подписать слушателя, реализующего интерфейс IEventListener, на событие с обобщенным интерфейсом IEvent. Например, IEventListener может реализовывать метод Notify(event: IEvent). При таком подходе необходимо одним методом получать все сообщения, и в нем писать условную логику для различных видов сообщений. И, как часть этой условной логики, выполнять в этом методе восходящее приведение типа, чтобы из абстрактного сообщения о событии IEvent получить интерфейс, предоставляющий доступ к информации, зависящей от специфики конкретного сообщения (к примеру, некий интерфейс IPropertyChangeEvent для работы с сообщением об изменении какого-либо свойства). Можно оспаривать эффективность именно такого способа представления свойств, он приведен просто в качестве примера подписки на конкретные типы событий при наличии общего интерфейса в шаблоне "подписчик".

И восходящее приведение типов, и условная логика в методах в чрезмерном количестве вредны, в литературе, например, в работе [3], предложены некоторые способы отказа от них. Чтобы избежать восходящего приведения и условной логики, можно под каждый тип событий заводить отдельные интерфейсы источника событий и подписчика. Такой подход приводит к высокой избыточности системы типов. Кроме того, при наличии нескольких свойств одного типа по-прежнему придется применять в методе-обработчике условную логику. В Delphi XE можно использовать обобщенные типы (*generics*) для обобщенного определения интерфейса отправителя, но с получателями это не сработает.

Как представляется автору, хорошим решением может быть то, которое применяется в VCL или в .Net, а именно ссылки на обработчики событий в виде делегатов. Вместо того чтобы определять получателя событий как полноценный интерфейс или класс, он может быть определен как ссылка на метод-обработчик. Такой подход позволяет определить для обработки каждого типа свойств или даже каждого отдельного свойства специфический метод, причем типизированный, а значит, не требующий для своей работы восходящего приведения типов. Тип делегата может быть определен как указатель на метод (*function of object*) или, в версии XE, как ссылка на функцию (*reference to function*).

Использование метода ProcessMessages для устранения эффекта "зависания"

Для устранения эффекта "зависания" приложения — отсутствия реакции на действия пользователя и прекращения прорисовки окон — при выполнении длительных операций в главном потоке документация Delphi предлагает использовать метод TApplication.ProcessMessages.

К сожалению, реализация ProcessMessages такова, что применять ее следует с осторожностью. Этот метод просто выбирает из очереди сообщений все находящиеся в ней сообщения и по очереди обрабатывает их. При этом обрабатываются и такие, как сообщения о навигации в меню программы, нажатиях "горячих клавиш" и подобные им. С учетом этого, если в ходе длительной операции вызвать ProcessMessages, пользователь сможет воспользоваться этим, чтобы войти в меню программы и запустить другую операцию, или повторно запустить ту же самую. Как правило, это нежелательный эффект. Более того, продолжение работы остановленной операции после окончания обработки новой может быть уже невозможно или может привести к порче данных, например, если в ходе со-

хранения открытого в программе документа было проведено его закрытие.

Автор рекомендует воздерживаться от использования ProcessMessages и вместо этого, например, организовывать длительную обработку в отдельных потоках исполнения (хотя в Delphi это нередко бывает затруднительно). В то же время в ходе недавнего обсуждения ProcessMessages с посетителями сообщества RSDN один из участников ознакомил автора с любопытным подходом, который во многих случаях дает достаточно гарантий безопасности, чтобы пытаться все-таки применять ProcessMessages.

Суть предложения в том, чтобы создавать модальное окно, затем запускать длительную обработку. В этом случае вызов ProcessMessages позволит обслуживать сообщения о перерисовке и сообщения от таймера, или, например, от сетевых подключений Windows Sockets, или от горячих клавиш уровня приложения (hotkeys, не путать с shortcuts). Вместе с тем наличие модального окна не позволит пользователю в ходе этого вызова осуществить какие-либо действия в других окнах приложения.

Можно предложить еще один способ защиты от повторного вызова операций, хотя с точки зрения пользователя он не столь элегантен. Если в приложении есть унифицированный механизм запуска операций (например, к его образованию может приводить следование паттерну "команда", включающему как раз обособление операций), такой механизм может отследить начало и окончание выполнения операции, и заблокировать запуск операции, если другая операция еще не завершилась.

В некоторых случаях в приложениях используют ProcessMessages, чтобы поглотить сообщения о нажатии клавиш "мыши", оставшиеся после выполнения запущенной операции. В этих случаях можно воспользоваться функцией PeekMessage в режиме PM_REMOVE для изъятия конкретных групп событий. Так же можно пытаться использовать PeekMessage (с указанием конкретного подмножества сообщений) для организации собственного метода выборки сообщений ProcessMessages, которые следует обрабатывать, пока некая операция еще длится.

Следует также знать о другой возможной проблемной ситуации, и хотя она гораздо более экзотична, но не представляется невероятной. Цикл обработки сообщений в ProcessMessages не имеет ограничений ни по времени работы, ни по числу обработанных сообщений. Если некий источник будет помещать в очередь новые сообщения также часто как цикл обработки будет их изымать, то такой цикл может длиться очень долго. В проекте, в котором принимал участие автор, однажды по ошибке один экспериментальный компонент (не несший полезной нагрузки) включал таймер, работавший с интервалом 1 мс. Разумеется, реальная частота обработки ограничена квантами времени Windows, но суть не в этом. Если обработка этих событий (на слабых компьютерах) едва успевает за поступлением новых сообщений, а в какой-то момент времени пользователем запускается длительная операция, содержащая вызов ProcessMessages, то приложение с точки зрения пользователя не завершает операцию. При этом, однако, приложение реагирует на его действия и даже позволяет запускать новые операции. Получается своеобразное "зависание", при котором приложение не "висит".

Сборка проекта с помощью утилиты MSBuild

В версии Delphi 2007 впервые силами самой компании CodeGear была обеспечена возможность сборки проекта средствами утилиты MSBuild от Microsoft, доступной в составе .Net. В версиях Delphi до XE включительно использование MSBuild было полезной возможностью, поскольку альтернатив для сборки проекта в пакетном режиме было всего две: собственный консольный компилятор DCC32 и запуск интегрированной среды с заданием автоматического выполнения компиляции. Однако запуск среды мог приводить к появлению диалоговых окон с запросами (что неприемлемо, если запуск проводится на сборочном сервере без участия пользователя), а компилятор DCC32 использовал конфигурацию, хранящуюся в .DOF-файле проекта, а не в файле .DPROJ, что приводило к дублированию и не позволяло удобным способом выбирать одну из нескольких конфигураций для сборки, заданных в файле .DPROJ.

В более поздних версиях Delphi (во всяком случае, в XE3) собственный компилятор читает конфигурации из файла .DPROJ, хотя все так же не позволяет при запуске выбрать одну из описанных в нем конфигураций. В то же время MSBuild лишен этого недостатка, и его использование практически напрашивается в качестве решения, с учетом того факта, что файл .DPROJ фактически имеет синтаксис, совместимый с MSBuild.

К сожалению, работа по обеспечению компиляции средствами MSBuild не была доведена до конца.

Вероятно, первая трудность, возникающая при применении MSBuild для сборки проекта на Delphi — работа с файлами ресурсов Windows (.res-файлами). Как известно, Delphi создает такой файл для каждого проекта, размещая там минимум ресурсов для библиотеки VCL, а также (опционально) пользовательские ресурсы, такие как иконка приложения. Интегрированная среда Delphi пересоздает этот .res-файл проекта при сборке проекта, поэтому добавлять такой файл в систему управления версиями неудобно (файл будет постоянно изменяться у всех участников проекта) и излишне. В этом случае можно положиться на автоматическое создание файла с ресурсами VCL (или копировать его из какого-то образца), а пользовательские ресурсы хранить в системе управления версиями в отдельном res-файле, который будет компилироваться не автоматически, а по определенной команде.

Еще более эффективно использовать для пользовательских ресурсов .rc-файл (исходное задание для сборки .res-файла) и по нему генерировать .res-файл

при сборке, при помощи компилятора ресурсов, однако сам постоянно меняющийся .res-файл не добавлять. В .rc-файле при этом следует включить и иконку приложения, и информацию о версии, и что-либо еще, обычно указывающееся через настройки проекта Delphi.

В обоих этих случаях следует иметь в виду, что создавать начальный (только с ресурсами VCL) .res-файл способна лишь среда Delphi, но не компилятор. Ввиду этого, если минимальный .res-файл проекта не будет присутствовать в системе управления версиями исходного кода, то его все равно для сборки с помощью MSBuild следует поместить рядом с файлом проекта (обычно достаточно для этого перед сборкой скопировать его из эталона такого файла .res).

Хранение ресурсных файлов только в виде исходного кода имеет смысл не только для тех .res-файлов, которые изменяются Delphi автоматически. Наличие всего одной версии файла вместо двух (.rc и .res) позволяет избежать возможности несогласованных изменений в этих версиях и просто уменьшить число артефактов в проекте. Необходимый для компиляции проекта .res-файл можно получить из .rc-файла в процессе компиляции.

В Delphi XE правильная обработка .rc-файлов выполняется при помощи директивы компиляции такого вида: {$R 'tt.res' 'C:\MyPath\tt.rc'} в головном файле проекта (.dpr) либо в каком-то из модулей (*units*). Вместе с тем компилятор Delphi все еще несовершенен, и одна лишь эта директива хоть и будет встраивать .res-файл в выходной файл проекта, но не будет вызывать перекомпиляцию .res-файла при изменении .rc-файла. Чтобы обеспечить автоматическую сборку, можно добавить вручную вызов компилятора ресурсов при сборке проекта (это можно сделать через раздел Build Events в настройках проекта).

Существует и гораздо более удобный и стандартный вариант — включить в файл описания проекта, который начиная с Delphi 2007 совместим с MSBuild (еще удобнее бывает передавать MSBuild файл группы проектов: *.groupproj), элемент, вызывающий обработку .rc-файла компилятором ресурсов при компиляции проекта. Этот способ не является каким-то трюком (будь так, элемент мог бы быть автоматически удален при сохранении проекта), он предусмотрен в среде Delphi (IDE) и реализуется простым добавлением .rc-файла в проект (в контекстном меню проекта — пункт "Add..."). При этом в .dpr-файл точно так же добавится директива $R, но дополнительно в файл .DPROJ добавится элемент <RcCompile>, вызывающий компиляцию .rc-файла при сборке проекта.

Следует отметить, что в этой директиве .res-файл указывается без полного пути к нему. Можно указать и полный путь, но в этом случае следует также указать в настройках проекта выходную папку компилятора ресурсов, так как если она не задана, то компиляция из IDE разместит .res-файл "рядом" с .rc-файлом, тогда как MSBuild разместит его в папке файла проектной группы (а искать его затем будет по тому пути, который указан в $R). В этом случае либо IDE не найдет файл, либо MSBuild.

Заключение

Представленные в статье результаты анализа некоторых вопросов, которые возникают при разработке программного обеспечения с использованием Embarcadero Delphi, могут быть полезны. С их помощью можно избежать неэффективных решений в применении интерфейсов, при создании оберток и агрегировании объектов, ликвидации эффекта "зависания" программ и разработке новых систем обмена оповещениями в программе. Предложенные рекомендации по подключению ресурсных файлов Windows позволяют удобным образом осуществлять сборку проектов Delphi с использованием только консольного компилятора dcc32 и средства управления сборкой MSBuild.

Список литературы

1. **Rogerson D.** Inside COM. Microsoft Press, 1997.
2. **Gamma E., Helm R., Johnson R., Vlissides J.** Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.
3. **Kerievsky J.** Refactoring to Patterns. Addison-Wesley, 2004.

ИНФОРМАЦИЯ

25 – 28 сентября 2013 г. Новосибирск, Академгородок
Всероссийская конференция
"Индустриальные информационные системы" –
ИИС-2013

Направления работы конференции

- Индустриальные информационные системы: математическое, программное и техническое обеспечение
- Автоматизированные системы управления технологическими процессами
- Информационные системы мониторинга

Ñàéò êîíôåðåíöèè: http://conf.nsc.ru/iis2013/ru

УДК 004.89:004.93

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

Разработка компонента синтаксического анализа предложений русского языка для интеллектуальной системы обработки естественно-языкового текста

Приведено описание структуры синтаксического компонента системы обработки текста, а также алгоритмов отдельных блоков этого компонента: фрагментации предложения, построения простых синтаксических групп, выделения предикатного ядра и заполнения актантной структуры предиката. Предложенный подход позволяет формировать предикатную модель, которая наилучшим образом отражает смысл предложения, так как в предикатах указывается не только аргументная структура и число актантов, но и их семантическое содержание.

Ключевые слова: *обработка естественного языка, синтаксический анализ, предикативное ядро предложения, простые синтаксические группы, актантная структура предиката*

Введение

Выявление формальных структур естественного языка (ЕЯ), формализация языка в целом, построение конструктивной теории и компьютерной модели языка являются приоритетными направлениями информатики на протяжении последних десятилетий. Системы информационного поиска, диалоговые системы, инструментальные средства для машинного перевода и автореферирования, рубрикаторы и модули проверки правописания так или иначе проводят обработку естественно-языковых текстов (ЕЯ текстов, *Natural Language Processing*, *NLP*). Таким образом, область применения NLP-систем достаточно разнообразна, а в виду большого роста объемов текстовой информации и сложностях, возникающих при попытках формализации естественного языка, анализ ЕЯ текстов представляет собой очень актуальную проблему.

Структура связного текста как языковой конструкции высшего уровня содержит в себе языковые единицы всех уровней — морфологического, лексического, словосочетаний, синтагм и предложений. Поэтому стадии полного лингвистического анализа ЕЯ текста являются стандартными: морфолого-лексический анализ, синтаксический анализ и семантический анализ. В результате формируются модели текста, адекватно отражающие его словообразовательные, грамматические и смысловые конструкции.

В отличие от морфологического анализа, хорошо отработанной лингвистической процедуры, реализованной во множестве разнообразных исследовательских и коммерческих проектов, синтаксический анализ для всех предложений ЕЯ текста — это не решенная в полном объеме задача. Трудности обосновываются неспособностью существующих формальных математических моделей и их программных реализаций охватить всю сложность и многообразие языковой системы, особенно для языков с относительно свободным порядком слов, каким является, например, русский язык. Синтагмы, полученные в результате членения языкового потока речи, анализируются чаще всего пословно, без учета семантико-синтаксических свя-

зей между членами предложения. Общая наука о построении предложения и машинное распознавание синтаксических конструкций в данном случае фактически не пересекаются.

Без компонента синтаксического анализа создание интеллектуальной NLP-системы представляется проблематичным, он необходим для выделения из предложения смысловых элементов: логического субъекта и предиката, прямых и косвенных дополнений, а также различных видов обстоятельств. Это позволяет повысить интеллектуальность процесса обработки текстовой информации путем обеспечения работы с более обобщенными семантическими элементами.

Анализ сложных синтаксических конструкций невозможен без оснащения NLP-систем описаниями и моделями, созданными лингвистами-профессионалами. Одна из таких моделей легла в основу лингвистической базы данных системы автоматического синтаксического анализа предложений русского языка. Она опирается на принципы построения элементарных простых предложений.

Система синтаксического анализа была разработана авторским коллективом в ходе выполнения работ по проекту "Исследование и разработка программного обеспечения понимания неструктурированной текстовой информации на русском и английском языках на базе создания методов компьютерного полного лингвистического анализа", грант Минобрнауки 2012-1.4-07-514-0018.

Предлагаемый подход к формированию синтаксических моделей использует предикативность — одну из важнейших характеристик простого предложения. Ни одна теория или концепция синтаксической организации предложения не обходит стороной свойства предикативности. Глагол является определяющей частью языка, предложений без глагола или без предикативного слова не существует. Предикат — центральная синтаксема в семантическом простом элементарном предложении, формирующая его семантико-синтаксическую структуру. Предикативно связанные грамматические субъект и предикат квалифицируются как главные члены предложения, поскольку они формируют его конструктивный минимум. Более того, предикатная модель наилучшим образом отражает смысл предложения, так как в предикатах указывается не только аргументная структура предложения и количество актантов, но и их семантическое содержание.

В работе приведено описание структуры синтаксического компонента лингвистического процессора, формирующего предикатную модель предложения, а также алгоритмов, используемых блоками, которые входят в состав этого компонента.

Общая структурная схема компонента синтаксического анализа

Анализ синтаксической структуры предложения должен выполняться на основе информации лексического уровня, полученной на этапе графематического и морфологического анализа. При этом каждой словоформе предложения приписывается соответствующий набор (наборы — в случае морфологической омонимии) морфологической информации (МИ). Таким образом, на вход компонента поступает множество наборов морфологических интерпретаций слов (N штук) предложения:

$$S = (s[1], ..., s[i], ..., s[N]),$$

где $s[i] = \{s[i][1], ..., s[i][j], ..., s[i][N]\}$ задает набор морфологических интерпретаций i-ой словоформы в предложении и является массивом пар (лемма, МИ).

Обозначим структуру, описывающую $s[i]$, *Morf_Interpr*.

Функционирование компонента синтаксического анализа осуществляется последовательной работой его четырех блоков (см. рисунок):

- фрагментация предложения;
- построение простых синтаксических групп в пределах каждого сегмента;
- выделение предикатного ядра (подлежащее-сказуемое);
- заполнение актантной структуры предиката.

Алгоритмы, реализованные в этих блоках, используют лингвистическую базу данных, состоящую из следующих словарей:

- словаря шаблонов союзов;
- словаря вводных слов и конструкций;
- словаря моделей управления (МУ) предлогов;
- словаря шаблонов минимальных структурных схем (МСС) предложений;
- словаря валентностей глаголов.

Обозначения, используемые в схеме:

{*Segment*} — массив структур, описывающих состав сегментов предложений, полученных после фрагментации;

A — матрица размерностью $N \times N$, элементы которой $a\_s[i][j]$ для предложения из N слов отражают наличие и тип связи между словами $s[i]$ и $s[j]$ как на синтагматическом, так и на предикативном уровнях ($s[i]$ — главное слово);

$a[Pred][Subj]$ — элемент матрицы A, где *Pred* — номер слова, являющегося сказуемым, *Subj* — номер слова, являющегося подлежащим;

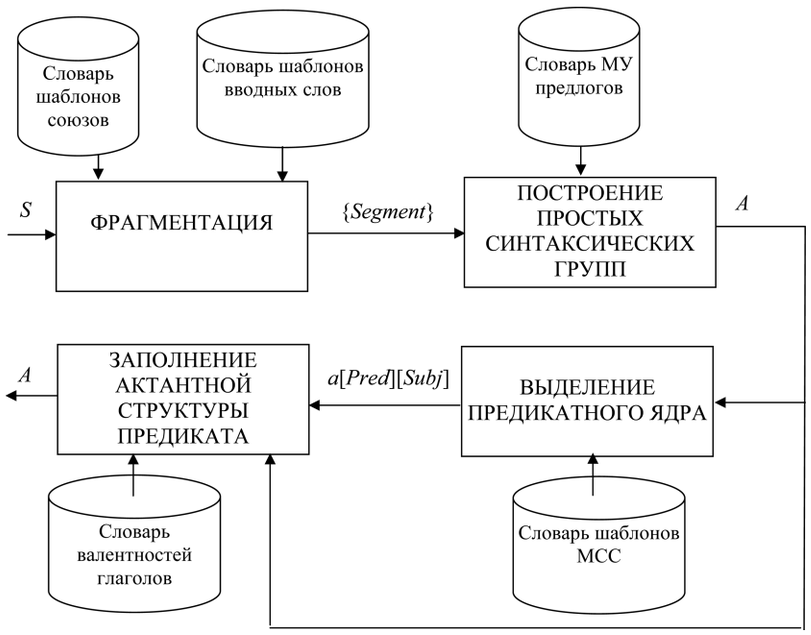
Результирующая матрица A для предложения из N слов задает дерево зависимостей (одно или несколько в силу омонимии), отражающее предикатную структуру предложения.

Подробно компоненты структуры *Segment* и алгоритмы последовательного заполнения матрицы A, лежащие в основе соответствующих блоков, описаны ниже.

Алгоритм фрагментации

Общую схему действий при фрагментации можно представить в виде последовательности шагов [1]:

- членение предложения по знакам пунктуации и союзам на исходные отрезки (будем их также назы-



Функциональная схема взаимодействия блоков компонента синтаксического анализа

вать начальными сегментами), определение вершин и типов начальных сегментов;

- установление иерархии между сегментами с помощью следующих синтаксических правил: вложения контактно расположенных сегментов (причастных, деепричастных оборотов, обособленного определения); определения однородности между контактно расположенными сегментами.

В результате работы алгоритма формируется массив структур типа *Segment* (табл. 1).

Для расстановки границ сегментов введем множество знаков препинания:

$$Prep = \{",", ".", "!", "?", ":", ";", "-", "(", ")", "{", "}", "[", "]", "..."\}$$

Границу сегмента ставим в двух случаях:

- после знака препинания из множества *Prep*/{"-"}, если он не входит в состав определенных графематическим анализом единиц (сокращения, дробные числа, буквенно-числовые комплексы и др.);
- после слова из множества союзов без запятой (следовательно, переменная *S_R* может быть только из множества сочинительных союзов).

В том случае, если несколько знаков препинания идут подряд, по ним проходит одна граница. Также не строятся сегменты, не содержащие ни одного слова.

Тип сегмента *type_sg* равен одному из значений, указанных в табл. 2, и определяется по следующему алгоритму:

- если в сегменте по порядку, указанному в табл. 2, найдено слово соответствующей части речи без

Таблица 1

Описание структуры Segment

Компоненты структуры	Тип данных	Примечание
type_sg	*int*	Тип сегмента (значение от 0 до 9 согласно табл. 2)
L, R	*int*	Номера слов в предложении, которые являются левой и правой границей сегмента
first	*Morf_Interpr*	Морфологическая интерпретация словоформы, являющейся главным словом сегмента (в случае *type_sg*=9 может быть нулевой и зависит от типа сегмента)
S_L	*Conj*	Шаблон союза, являющегося началом сегмента
S_R	*Conj*	Шаблон союза, являющегося концом сегмента
Hoz	*Morf_Interpr*	Морфологическая интерпретация словоформы, являющейся главным словом другого сегмента, подчиняющего текущий сегмент в случае его вложенности
Sosed	*Segment*	Сегмент, связанный с текущим однородной связью
Примечание: *О структуре типа *Conj* речь пойдет ниже.		

Таблица 2

Тип сегмента (*type_sg*) предложения для русского языка

Глагол в личной форме	Краткое причастие	Краткое прилагательное	Предикативное слово	Причастие	Деепричастие	Инфинитив	Вводное слово	Иное
1	2	3	4	5	6	7	8	9

Таблица 3

Морфологическая информация для отнесения сегмента к типу ТИРЕ

*S*1	*S*2	*S*3
Существительное в именительном падеже	Существительное/местоимение в именительном падеже	—
Существительное в именительном падеже	Прилагательное в именительном падеже	—
Местоимение в именительном падеже	Существительное/местоимение в именительном падеже	—
Местоимение в именительном падеже	Прилагательное в именительном падеже	—
*S*1 = "это"	Существительное/местоимение в именительном падеже	—
*S*1 = "у"	Существительное/местоимение в родительном падеже	Существительное/местоимение в именительном падеже
Существительное в именительном падеже	*S*2 из множества {как, словно, будто, что, точно, как будто}	Существительное в именительном падеже

омонимии, то тип определен, *first* — указатель на соответствующее слово (если *type_sg* = 9, то first = 0).

- устанавливается *type_sg* = 0 (тип ТИРЕ), если *type_sg* = 9 и в сегменте есть тире (не первым и не последним символом сегмента). Кроме того, сегмент имеет тип ТИРЕ, если в составе сегмента есть последовательность двух или трех слов (*S*1, *S*2, *S*3), обладающих следующей МИ (табл. 3).

В сегменте типа ТИРЕ *first* — указатель на слово, стоящее до тире.

Для расстановки границ сегментов понадобится база данных союзов и устойчивых словосочетаний, которые могут рассматриваться как союзы, а также шаблоны для их описания (табл. 4).

По выражению синтаксической связи союзы делятся на сочинительные и подчинительные, могут

Таблица 4

Типы союзов и шаблоны для их описания

Типы союзов	Выражаемые отношения	Шаблоны
Copulative (соединительные)	Отношения перечисления	{1,0,"и",0,1,1,Copulative}
Separative (разделительные)	Отношения взаимоисключения, чередования действий, явлений, признаков	{1,0,"или",0,1,1,Separative} // односоставные двусловные {2,0,"не","то",0,1,0,Separative}
Adversative (противительные)	Отношения противопоставления, несоответствия, различия	{1,0,"но",1,1,1,Adversative} {1,1,"не","а",1,0,0,Adversative} {2,0,"все","же",1,0,0,Adversative}
Explanatory (пояснительные)	Отношения пояснения	{2,0,"а","именно",1,0,0,Explanatory}
Bridge (сопоставительные, градационно-сопоставительные)	Отношения сопоставления	{1,1,"как","так",1,0,0,Bridge} {2,2,"не","только","но","и",1,0,1,Bridge} {2,1,"хотя","и","но",1,0,0,Bridge} {3,1,"не","то","что","а",1,0,0,Bridge}
Connecting (присоединительные)	Отношения присоединения, т.е. присоединяют к предложению что-то добавочное, дополнительное	{1,0,"и",1,0,1,Connecting} {2,0,"а","поэтому",1,0,0,Connecting} // односоставные трехсловные {3,0,"да","еще","и",1,0,0,Connecting}

быть односоставными (простые, двусловные, трехсловные) и многосоставными. Для их описания были разработаны шаблоны, у каждого из которых следующие значения полей:

1. число слов первой части;
2. число слов второй части;
3. союз;
4. предполагается ли запятая перед союзом: 1 — да, 0 — нет, 2 — может стоять перед союзом, если он во второй части;
5. возможно ли многократное повторение, разделенное запятыми (открытое употребление союза): 1 — да, 0 — нет;
6. допускается ли разделение двух однородных членов (закрытое употребление союза): 1 — да, 0 — нет;
7. тип союза.

Эти шаблоны образуют структуру *Conj*.

Вводные слова и вставные конструкции находят с помощью соответствующей базы данных (базы вводных слов и вставных конструкций), они не несут никакой синтаксической и семантической нагрузки, а также каждое вводное слово и конструкция должны выделяться или отделяться знаками препинания. Вставные конструкции всегда заключены в различные виды скобок и кавычек.

Выделение границ сегментов, содержащих вводные слова и конструкции, будет проводиться следующим образом:

- поиск в сегменте вставного слова или конструкции по базе данных;
- поиск знака разделителя из множества *Prep* (если слово или конструкция стоит в середине сегмента, то такой сегмент должен быть отделен знаками препинания с двух сторон);
- если в сегменте найдена какая-либо открывающая скобка, то все слова, которые стоят до закрывающей скобки будем относить к вставной конструкции.

В зависимости от типа сегментов и типа подчинительного союза по эвристическим правилам можно их укрупнить, осуществляя над ними операции подчинения и однородности.

Правила для подчинения

- Если *type_sg* = 5 (причастный оборот) и нет согласованного с причастием синтаксического существительного, анализируем предыдущий сегмент на наличие такого. В случае обнаружения согласованного существительного полагаем, что *Hoz* — найденное существительное из предыдущего сегмента, поскольку это слово является главным для причастного оборота сегмента в случае его вложенности.
- Если *type_sg* = 6 (деепричастный оборот), анализируем в контактно расположенных сегментах наличие глагола. Поскольку деепричастие — самостоятельная часть речи, которая обозначает добавочное действие, объединяет в себе свойства глагола и наречия и показывает, каким образом, почему, когда совершается действие, вызванное глаголом-сказуемым, то главным словом для сегмента-деепричастного оборота является глагол. Полагаем, что *Hoz* — найденный ближайший глагол.
- Если *type_sg* = 6 (деепричастие) и сегмент содержит прилагательное и не содержит согласованного с ним синтаксического существительного по роду, числу и падежу, проверяем предыдущий сегмент на согласованное существительное с прилагательным. Если оно найдено, имеем случай обособленного определения. Полагаем, что *Hoz* — найденное существительное.

Правила однородности

Проводим поиск по шаблону: если два контактно расположенных сегмента соединены одним из сочинительных союзов, содержащихся в соответствующей базе (в конце первого стоит союз, следовательно, *S_R* не равно 0), или у сегментов *S_R* (союз в конце сегмента) одинаковый, проверяем эти сегменты на наличие слов с одинаковыми грамматическими характеристиками. При обнаружении таковых полагаем, что *Sosed* — однородный сегмент слева.

Если сегменты соединены разрывным сочинительным союзом, то *type_sg* = 2, *Sosed* — однородный сегмент справа.

Алгоритм построения простых синтаксических групп в пределах каждого сегмента

К простым синтаксическим группам будем относить группы на атрибутивном уровне, группы с предлогом, сравнительные конструкции, составное сказуемое, конструкции с лексемами "каждый" или "один", употребляемые с предлогом "из", генитивное определение в постпозиции.

Построение внутри сегментов простых синтаксических групп начинается с поиска по морфологическим характеристикам групп, соответствующих атрибутивному уровню описания (табл. 5), а именно: признак объекта/субъекта/действия + объект/субъект/действие, мера признака объекта/субъекта/действия + объект/субъект/действие.

При дальнейшем изложении синтаксические группы существительного или глагола на атрибутивном уровне описания будем называть именной (ИГ) или глагольной (ГГ) группой.

Для русского языка приведем правила построения предложных групп (ПГ): предлог + существительное/местоимение-существительное, стоящее в падеже согласно модели управления предлога. Главное слово этой группы — предлог.

Правила для построения сравнительной конструкции для русского языка приведены в табл. 6. Главное слово в этой конструкции — прилагательное.

Две или три контактно стоящих группы могут образовывать составное сказуемое, если главное слово этих групп имеет хотя бы один омоним из множества, описанного в табл. 7. Остальные омонимы удаляются.

Таблица 5

Элементы атрибутивного уровня описания

Компоненты группы	Морфологические признаки	Примеры
Объект/Субъект	Существительное, местоимение-существительное	Вася, он
Действие	Глагол	Гуляет
Признак объекта	Полное прилагательное, согласованное с объектом по роду, числу и падежу	***Признак объекта + объект:*** прекрасная погода
	Одиночное порядковое числительное или количественная группа (последовательность числительных), согласованное с объектом по роду, числу и падежу	Двадцать восемь квартир
	Местоимение-прилагательное, согласованное с объектом по роду, числу и падежу	Свое дело
	Наречное числительное, если объект — существительное во множественном числе, родительном падеже	Мало мест
Признак действия	Наречие	***Признак действия + действие:*** интересно рассказывать
Мера признака	Наречие	***Мера признака объекта + признак:*** очень известный
	Лексемы "такой" или "самый", согласованные с признаком по роду, числу и падежу	Такая старательная

Таблица 6

Синтаксические группы, соответствующие сравнительной конструкции

Компоненты группы	Пример
Наречие + прилагательное в сравнительной степени	Гораздо сильнее, значительно больше
"Более"/"менее" + прилагательное (может быть и в краткой форме)	Более сильный, менее привлекателен
Прилагательное в сравнительной степени + ИГ в родительном падеже	Левее сапога, умнее человека, краше тебя, ниже твоего дома

Глагол-связка — это глагол, принадлежащий множеству лексем: {"быть", "стать", "становиться", "делаться", "являться", "представляться", "казаться", "считаться", "слыть", "бывать", "состоять", "оказаться", "выглядеть", "заключаться"}.

Группу составного сказуемого в дальнейшем рассматриваем как одно слово.

Правило построения конструкции с лексемами "каждый" или "один": "каждый"/"один" + ПГ с предлогом "из". Главное слово — "каждый"/"один". Например, *один из них; каждый из ваших людей.*

Правило построения генитивного определения в постпозиции: ИГ + ИГ в родительном падеже. Главное слово — существительное первой ИГ. Например, *рука человека; стол отца; набор тяжелых грузов.*

В результате работы этого алгоритма для предложения из N слов заполняется матрица A_S размерностью $N \times N$, элементы которой $a\_s[i][j]$ указывают на наличие и тип связи между словами $s[i]$ и $s[j]$ на синтагматическом уровне ($s[i]$ — главное слово) и могут принимать следующие значения:

- -1, если $s[i]$ и $s[j]$ не связаны на синтагматическом уровне;
- 1, если $s[j]$ — признак;
- 2, если $s[j]$ — мера признака;
- 3, $s[i]$ — прилагательное в сравнительной конструкции;
- 4, $s[j]$ — генитивное определение в постпозиции;
- 5, если $s[j]$ принадлежит множеству лексем {"каждый", "один"};
- 6, если $s[i]$ — предлог.

Алгоритм выделения предикатного ядра

Множество простых предложений русского языка задается перечнем МСС предложений, описывающих предикативный минимум предложения [2]. В минимальные схемы предложений входят формы слов, которые перечислены в табл. 8.

В табл. 9 приведены шаблоны МСС и примеры соответствующих предложений [3].

МСС незаменимы при автоматическом выделении составного сказуемого, а также для случаев, когда отсутствует в явном виде глагол-связка и сказуемое выражено предикатом, не являющимся глаголом (тип МСС 2...4 в табл. 9).

Во всех сегментах предложения, не являющихся вложенным и однородным, проводится последовательный поиск шаблона МСС от 1 до 17. Согласно

Таблица 7

Морфологические признаки главных слов групп, образующих составное сказуемое

Морфологические признаки главного слова первой группы	Морфологические признаки главного слова второй группы	Морфологические признаки главного слова третьей группы	Пример
Спрягаемая форма глагола (вспомогательный глагол — фазовый или модальный)	Инфинитив	—	Я продолжаю думать
Краткая форма прилагательного/причастия (предикативное прилагательное)	Инфинитив	—	Он намерен начать
Глагольно-именной оборот, где глагол имеет спрягаемую форму	Инфинитив	—	Она изъявила желание трудиться
Краткая форма прилагательного/причастия (предикативное прилагательное)	Инфинитив	Инфинитив	Я должна решиться уехать
Спрягаемая форма глагола	Инфинитив	Инфинитив	Отец хотел начать работать
Спрягаемая форма глагола-связки	Существительное в творительном падеже	—	Он являлся новатором
Спрягаемая форма глагола-связки	Прилагательное в творительном падеже	—	Сестра стала взрослой
Спрягаемая форма глагола-связки	Прилагательное в сравнительной степени	—	Туман стал гуще
Глагол-связка в среднем роде	Предикатив (наречие), кроме "много", "мало"	—	Утром было прохладно
Спрягаемая форма глагола	Глагол-связка в личной форме	Существительное в творительном падеже	Брат хотел стать ученым
Спрягаемая форма глагола	Глагол-связка в личной форме	Прилагательное в творительном падеже	Он пытался выглядеть умным
Спрягаемая форма глагола	Глагол-связка в личной форме	Прилагательное в сравнительной степени	Парень мечтал стать сильнее

Таблица 8

Формы слов, входящих в МСС предложений

Форма слова	Сокращение
1. Показатели предикативности	
Спрягаемые формы глагола (не инфинитив)	*V*(*f*)
Спрягаемые формы связки служебных слов *быть, стать, являться*	*Cop*(*f*)
Инфинитив глагола или связки	*Inf*
Форма третьего лица единственного числа	*V*(*sn*,3)
Форма третьего лица множественного числа	*V*(*pl*,3)
2. Имена и наречия	
Именительного и творительного падежей существительных/существительных-местоимений	*N*1 и *N*5
Беспредложные и предложные формы любого косвенного падежа, способные сочетаться со связкой	*N*2...*pr*
Именительного и творительного падежа прилагательных и страдательных причастий	*Adj*1 и *Adj*5
Краткие формы и компоративы прилагательных и страдательных причастий	*Adj*(*f*)
Наречия, способные сочетаться со связкой (предикативы)	*Adv_pr_pr*

Таблица 9

Шаблоны ММС простых предложений русского языка

№ шаблона	МСС	Примеры предложений
1	*N*1 *V*(*f*)	Грачи прилетели. Дела делаются людьми
2	*N*1 *Cop*(*f*) *Adj*1 *N*1 *Cop*(*f*) *Adj*5 *N*1 *Cop*(*f*) *Adj*(*f*)	Ночь была тихая (тихой, тиха). Ночь тихая (тиха). Ночь была тише дня
3	*N*1 *Cop*(*f*) *N*1 *N*1 *Cop*(*f*) *N5*	Он (был) студент. Он был студентом
4	*N*1 *Cop*(*f*) *N*2 ...*pr* *N*1 *Cop*(*f*) *N*5...*pr* *N*1 *Cop*(*f*) *Adv_pr*	Дом (будет) без лифта. Чай — с сахаром. Глаза (были) навыкате
5	*Inf V*(*f*)	Курить воспрещалось. Отмалчиваться следует. Не мешало б нам встречаться чаще
6	*Inf Cop*(*f*) *N*1 *Inf Cop*(*f*) *N*5	Любить иных − тяжелый крест. Дозвониться было проблемой
7	*Inf Cop*(*f*) *Adj*1 *Inf Cop*(*f*) *Adj*5 *Inf Cop*(*f*) *Adj*(*f*)	Промолчать — самое разумное. Промолчать было самым разумным. Промолчать — разумно
8	*Inf Cop*(*f*) *N*2...*pr* *Inf Cop*(*f*) *Adv_pr*	Промолчать было в его правилах. Молчать некстати. Идти трудно
9	*Inf Cop*(*f*) *Inf*	Отказаться было обидеть. Быть студентом − это постоянно учиться мыслить
10	*Cop*(*pl*,3) *N*2...*pr* *Cop*(*pl*,3) *Adv_pr*	Дома были в слезах. На Родине от его подвигов были в восторге. С ним были запросто
11	*Cop*(*f*) *N*1	Будет дождь. Была зима. Шепот. Робкое дыхание. Тишина
12	*Cop*(*sn*,3) *Adj*(*f*,*sn*)	Было темно. Ночью будет морозно
13	*Cop*(*pl*,3) *Adj*(*f*,*pl*)	Результатом были довольны. Отказом были обижены
14	*Cop*(*sn*,3) *N*2...*pr* *Cop*(*sn*,3) *Adv_pr*	Будет без осадков. Было поздно
15	*V*(*sn*,3)	Скрипело, свистало и выло в лесу. Ему нездоровится. У него кипело на сердце
16	*V*(*pl*,3)	За столом зашумели. Его обидели
17	*Inf*	Не нагнать тебе бешеной тройки. Быть рекам чистыми. Быть по-вашему

найденному шаблону МСС элементу матрицы *A_S a_s*[*Pred*][*Subj*] присваивается значение 0.

Выделение остальных членов простого предложения (остальных семантически значимых объектов и атрибутов) и семантически значимых связей проводится с помощью последовательного сравнения слов предложения с актантной структурой найденного предиката, с использованием словаря валентностей глаголов русского языка.

Алгоритм заполнения актантной структуры предиката

Для полученного в результате выделения предикатного ядра сказуемого находим правосторонние актанты (т. е. не являющиеся подлежащим) и заполняем валентные гнезда для предиката. В используемой предикатной модели содержится семь слотов, соответствующих валентным гнездам предиката, причем но-

Таблица 10

Тип, семантика и морфологические характеристики валентных гнезд

Номер валентного гнезда	Наличие предлога	Падеж актанта	Семантический падеж
1	–	Именительный	Субъект
2	–	Винительный	Объект
3	–	Дательный	Адресат
4	–	Творительный	Инструмент
5	+	Родительный, предложный	Начальный локатив
6	+	Родительный, предложный	Конечный локатив
7	+	Родительный, предложный	Средний локатив

Таблица 11

Словарная статья из словаря валентности для глагола "являться"

Глагол	Семантический класс	Валентные гнезда	Морфологические признаки актантов	Шаблон МСС
Являться	Предложения, отображающие ситуацию собственно бытия	1 *7	1 — *N*1 *7 — *N*(в, на)6, *Adv*	*N*1 *V*(*f*)
		1^0 *7	1^0 — *N*3; *N*0 *7 — *N*(в, на)6, *Adv*	*Inf*
		1^0 *7	1^0 — *N*0 *7 — *N*(в, на)6, *Adv*	*V*(*pl*,3)

Примечания:

- цифровые индексы в столбце "Валентные гнезда" указывают на необходимое заполнение определенных валентно обусловленных ячеек (1 — левосторонний актант, или субъект действия; 2, 3, 4, 5, 6, 7 — правосторонние актанты и, соответственно: объект, адресат, инструмент, исходный, конечный, промежуточный локативы);
- звездочка при цифровом индексе в столбце "Валентность" указывает на необязательное заполнение данной валентно обусловленной ячейки предиката;
- 1^0 – надстрочный символ "0" при цифре указывает на нулевое заполнение ячейки субъекта;
- (в) (на) — буквенные символы между *N* и цифровым индексом, обозначающим падеж имени существительного, называет предлог, с которым возможно заполнение данной ячейки.

мер валентности определяет ее тип, семантику и морфологическое выражение (табл. 10). Таким образом, актанты выступают в качестве семантических падежей и интерпретируются как роли в отношениях действия и состояния, которые выражаются предикатом.

Для заполнения актантной структуры найденного предиката ранее была разработана семантико-синтаксическая классификация глаголов. Глаголы одного класса, как правило, имеют один шаблон заполнения валентных гнезд, включающий часть речи актанта, предлоги, с которыми он может употребляться, информацию о падеже актанта и МСС, которые соответствуют вариантам заполнения валентных гнезд. Пример словарной статьи словаря валентностей глаголов для глагола "являться" приведен в табл. 11.

В ходе заполнения актантной структуры предиката уточняются значения элементов $a[Pred][j]$, где j — номера слов-актантов предиката (j не равно *Subj*). Элемент $a[Pred][j]$, не равный -1, указывает на наличие предикативной связи и может принимать следующие значения:

- 7, если $s[j]$ — объект;
- 8, если $s[j]$ — адресат;
- 9, если $s[j]$ — инструмент;
- 10, если $s[j]$ — начальный локатив;
- 11, если $s[j]$ — конечный локатив;
- 12, если $s[j]$ — средний локатив.

Таким образом, в результате работы компонента синтаксического анализа получается дерево зависимостей (одно или несколько в силу омонимии) для предложения из N слов, которое задается в матричном виде с помощью матрицы A, имеющей размерность $N \times N$. Элементы матрицы, $a[i][j]$, являются целыми числами и указывают на наличие и тип связи между словами $s[i]$ и $s[j]$ как на синтагматическом, так и на предикативном уровнях.

Пример работы алгоритмов синтаксического анализатора

Рассмотрим работу алгоритма на примере предложения "Механика — часть физики, которая изучает закономерности механического движения и причины, вызывающие или изменяющие это движение" (учебник Т. И. Трофимовой "Курс физики", Москва, "Высшая школа", 2001).

На вход компонента синтаксического анализа поступает последовательность морфологических интерпретаций словоформ $S = (s[1], ..., s[i], ..., s[15])$, представленная в табл. 12.

В результате фрагментации получаем последовательность структур *Segment*, представленных в табл. 13.

После раскрытия анафор фрагменты объединяются в простые предложения, в том числе с помощью подчинительных союзов:

- Механика — часть физики

• Часть изучает закономерности механического движения и причины, вызывающие или изменяющие механическое движение.

Результат выявления предикатной структуры для нашего примера приведен в табл. 14.

Таблица 12

Морфологическая интерпретация словоформ, входящих в анализируемое предложение

№ слова	Леммы	Морфологическая информация словоформы
1	механик	(механик; Род.п. М.р. Ед.ч. Существ. Одушевл.)
		(механик; Вин.п. М.р. Ед.ч. Существ. Одушевл.)
	механика	(механика; Им.п. Ж.р. Ед.ч. Существ. Неодуш.)
2	часть	(часть; Им.п. Ж.р. Ед.ч. Существ. Неодуш.)
		(часть; Вин.п. Ж.р. Ед.ч. Существ. Неодуш.)
3	физик	(физик; Им.п. М.р. Мн.ч. Существ. Одушевл.)
	физика	(физика; Им.п. Ж.р. Мн.ч. Существ. Неодуш.)
		(физика; Род.п. Ж.р. Ед.ч. Существ. Неодуш.)
		(физика; Вин.п. Ж.р. Мн.ч. Существ. Неодуш.)
...	...	...
15	движение	(движение; Им.п. С.р. Ед.ч. Существ. Неодуш.)
		(движение; Вин.п. С.р. Ед.ч. Существ. Неодуш.)

Таблица 13

Результаты фрагментации предложений после объединения однородных рядов

Сегменты предложения	Тип сегмента
Механика — часть физики	ТИРЕ
которая изучает закономерности механического движения и причины	Глагол в личной форме
вызывающие или изменяющие это движение	Причастие

Таблица 14

Предикативный минимум простых предложений, входящих в состав предложений исходного текста

Составляющие простые предложения	Шаблон МСС	Предикативный минимум (субъект-предикат)
Механика — часть физики	Существительное в именительном падеже+ копула + существительное в именительном падеже	Механика — являться частью
Часть изучает закономерности механического движения и причины, вызывающие или изменяющие механическое движение	Существительное в именительном падеже + спрягаемая форма глагола	Часть — изучать

Таблица 15

Синтаксические группы, полученные из исходного текста с помощью синтаксических правил

Фрагменты предложения	Синтаксические группы	Название групп и правил
Механика — часть физики	часть физики	Генитивное определение в постпозиции
которая изучает закономерности механического движения и причины вызывающие или изменяющие это движение	закономерности движения	Генитивное определение в постпозиции
	механического движения	Объект + Признак объекта
	причины, вызывающие или изменяющие это движение	Объект + Признак объекта

Таблица 16

Заполнение валентных гнезд для предикатов текста примера

№ предложения	Предикат	1. Субъект	2. Объект	3. Адресат	4. Инструмент	5—7. Локативы
1	являться	механика	частью	—	—	физики
1	изучать	часть	закономерности, причины	—	—	—
2	являться	движение	изменением	—	—	—

Примечание. Прочерки означают отсутствие актантов в предикатной структуре предложения

Далее осуществляется построение синтаксических групп внутри полученных простых предложений (в которых актанты предикатов — главные слова) с помощью синтаксических правил, выявляющих синтаксические связи между словами. Построенные группы приведены в табл. 15.

Заполненные валентные гнезда для предикатов текста примера приведены в табл. 16.

Заключение

В работе приведено описание структуры компонента синтаксического анализа в составе лингвистического процессора и алгоритмов, используемых при его разработке. Эти алгоритмы опираются на лингвистические знания в виде семантического словаря предикатов, словари шаблонов и набор правил выделения синтаксических связей пар слов, и позволяют получить синтаксическую модель предложения в виде предикатной структуры. Такое представление синтаксических связей описывает не только аргументную структуру и число актантов предиката, но также учитывает их семантическое содержание, используя семантическую классификацию предикатов. Развитием данной работы может стать создание системы понимания текста, которое тесно связано с выявлением предикатных структур, характеризующих смысл предложений, а также цепочек этих предикатных структур, которые опосредуют прагматику текста.

Список литературы

1. **Сокирко А. В.** Семантические словари в автоматической обработке текста (по материалам системы ДИАЛИНГ). дис. ... канд. техн. наук. URL: http://www.aot.ru/docs/sokirko/sokirko-candid-3.html#3-4

2. **Белошапкова В. А., Брызгунова Е. А., Земская Е. А.** и др. Современный русский язык: Учебник для филологических специальностей высших учебных заведений / Под ред. В. А. Белошапковой. — 3-е изд., испр. и доп. М.: Азбуковник, 1997. 928 с.

3. **Дорохина Г. В. Гнитько Д. С.** Автоматическое выделение синтаксически связанных слов простого распространенного неосложненного предложения // "Сучасна інформаційна Україна: інформатика, економіка, філософія": матеріали доповідей конференції, 12—13 травня 2011 року. Донецьк, 2011. Т. 1. С. 34—38.

ИНФОРМАЦИЯ

8—10 ноября 2013 г. в г. Львов пройдет Четырнадцатая Международная конференция в области обеспечения качества ПО
"Software Quality Assurance Days" .

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.

Предыдущая 13-я конференция проходила в Санкт-Петербурге, ее участниками стали более 600 профессионалов. Организатором традиционно выступает компания "Лаборатория тестирования". (http://www.sqalab.ru/)

Обращаем внимание, что 10 ноября пройдет дополнительный день SQA Days English Day, на котором будут представлены доклады на английском языке. Это отдельное событие в рамках конференции. Количество мест на этот день будет ограничено.

Сайт конференции: http://sqadays.com/index-news.sdf/sqadays/sqa_days14

CONTENTS

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

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

Сдано в набор 30.04.2013 г. Подписано в печать 19.06.2013 г. Формат 60×88 1/8. Заказ PI713
Цена свободная.

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