ТЕОРЕТИЧЕСКИЙ И ПРИКЛАДНОЙ НАУЧНО-ТЕХНИЧЕСКИЙ ЖУРНАЛ

Издается с сентября 2010 г.

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

УДК 004.94; 330.4; 519.683

В.Л. Макаров[1], д-р физ.-мат. наук, академик РАН, директор, e-mail: makarov@cemi.rssi.ru
А.Р. Бахтизин[1], д-р экон. наук, вед. науч. сотр., e-mail: albert@cemi.rssi.ru
В.А. Васенин[2], д-р физ.-мат. наук, проф., e-mail: vasenin@msu.ru
В.А. Роганов[2], стар. науч. сотр., e-mail: raduga@butovo.com
И.А. Трифонов[3], студент, e-mail: spaeciallyforyou@gmail.com

[1]Учреждение Российской академии наук Центральный экономико-математический институт РАН (ЦЭМИ РАН)
[2]НИИ механики МГУ имени М.В. Ломоносова
[3]МГУ имени М.В. Ломоносова

Средства суперкомпьютерных систем для работы с агент-ориентированными моделями

На примере разработанной в ЦЭМИ РАН агент-ориентированной модели рассматриваются этапы и методы эффективного отображения счетного ядра мультиагентной системы на архитектуру современного суперкомпьютера. За счет применения суперкомпьютерных технологий и оптимизации программного кода удалось добиться очень высокой производительности.

Ключевые слова: *агент-ориентированные модели, параллельные вычисления, суперкомпьютеры*

Введение

Компьютерное моделирование – интенсивно развивающаяся область научных исследований, востребованная сегодня практически во всех сферах человеческой деятельности. Агент-ориентированный подход к моделированию очень универсален и удобен для прикладников в силу своей наглядности, однако его, как правило, отличает и высокая требовательность к вычислительным ресурсам. Прямое моделирование, например, достаточно длительных социальных процессов в масштабах страны и планеты в целом требует вычислительных средств высокой производительности и больших объемов памяти.

Суперкомпьютеры позволяют на несколько порядков увеличить число агентов и других количественных характеристик (узлов сети, территории охвата) в моделях, первоначально разработанных для использования на традиционных ЭВМ с последовательной архитектурой. По этой причине суперкомпьютерное моделирование является логичным и крайне желательным шагом для тех упрощенных моделей, которые уже прошли успешную практическую апробацию на последовательных компьютерах. Вместе с тем, архитектурно-технологические особенности современных компьютеров вовсе не гарантируют, что программное обеспечение уже апробированной модели немедленно заработает и на суперкомпьютере. Для такого перехода как минимум потребуется распараллеливание счетного ядра, а зачастую и его глубокая оптимизация, поскольку, в ином случае, применение дорогостоящего суперкомпьютерного счета не будет оправдано.

В данной статье, на примере разработанной в ЦЭМИ РАН агент-ориентированной модели (АОМ) рассматриваются этапы и методы эффективного отображения счетного ядра мультиагентной системы на архитектуру современного суперкомпьютера.

1. Теоретические основы агент-ориентированных моделей. Сложившиеся определения АОМ

Определений АОМ достаточно много (см., например, работы [6, 7, 11]). Далее предлагается свое, которое с одной стороны является симбиозом определений, данных наиболее авторитетными экспертами в этой области, а с другой – отражает понимание авто-

рами моделей этого класса. Итак, АОМ – это модель, обладающая следующими основными свойствами.

• *Автономия*. Агенты действуют независимо друг от друга и при этом предполагается, что в моделях нет единой регулирующей структуры, которая контролировала бы поведение каждого агента в отдельности. Однако при этом взаимодействие микро- и макроуровней в моделях осуществляется, как правило, следующим образом: на макроуровне задается общий для всех агентов набор правил, и, в свою очередь, совокупность действий агентов микроуровня может оказывать влияние на параметры макроуровня.

• *Неоднородность.* Агенты чем-то различаются друг от друга, что принципиально отличает АОМ от широко распространенных моделей с агентом-представителем, причем различия между агентами могут проявляться по многим параметрам. В случае агентов, отображающих людей, это могут быть параметры уровня здоровья, дохода, культурного уровня, а также правил принятия решений и аналогичные им.

• *Ограниченная интеллектуальность агентов* (или ограниченная рациональность). Это свойство означает, что агенты модели не могут познать нечто большее, выходящее за рамки макросреды модели.

• *Расположение в пространстве*. Имеется в виду некоторая "среда обитания", которая может быть представлена как в виде решетки (как в игре "Жизнь" [13]), так и в виде гораздо более сложной структуры, например, трехмерного пространства с заданными в нем объектами.

Кроме перечисленных, общей особенностью всех АОМ и вместе с тем их главным отличием от моделей других классов является наличие в них большого числа взаимодействующих друг с другом агентов. Существуют АОМ, число агентов в которых достигает нескольких миллионов, например, модель, разработанная под руководством Дж. Эпштейна [18]. Обычно в моделях социально-экономических систем присутствуют агрегированные агенты, представляющие собой либо отрасль, либо регион, либо совокупное домохозяйство. Спецификация агента при этом проводится путем оптимизации соответствующей функции полезности или в модель включаются рассчитанные ранее экзогенные параметры, отражающие результаты решений агента. В литературе эти два подхода часто подвергаются обоснованной критике, поскольку в большинстве случаев они не всегда позволяют получить в рамках таких моделей соответствующие реалиям оценки взаимодействия агрегированных агентов. В то же время за счет более детальной спецификации в АОМ агентов микроуровня можно более адекватным действительности образом добиться изменений параметров макроуровня.

Подводя итог изложенным выше соображениям, отметим, что согласно перечисленным свойствам агент в АОМ является автономной сущностью, как правило, имеющей графическое представление, с определенной для него целью функционирования и возможностью обучения в процессе существования до заранее принятого уровня, который определяется разработчиками соответствующей модели. Примерами агентов могут быть: люди (равно как и другие живые организмы), роботы, автомобили и другие подвижные объекты; недвижимые объекты; совокупности однотипных объектов. Вообще говоря, агентами в АОМ могут быть любые наблюдаемые в реальной жизни объекты, однако основной задачей описания таких агентов в рамках модели является их корректная, в большей степени отражающая реалии спецификация.

Что может быть "внутри" агента?

Как правило, для описания агента используются параметры, переменные, функции, поведенческие диаграммы, представляющие собой, например, схемы UML (*Unified Modeling Language*), отражающие состояния агентов в определенный момент времени.

Агенты, имеющие графическое представление, могут перемещаться в рамках:

• евклидова пространства (2D или 3D);

• ГИС (геоинформационной системы или системы, позволяющей создавать базы данных, сочетающие в себе графическое и атрибутивное представление разнородной информации);

• решетки (в этом случае перемещение агентов происходит строго из одной ячейки в другую);

• сетевой структуры.

Следует заметить, что необходимости в графическом представлении состояния агентов АОМ в пространстве не возникает.

Из истории возникновения АОМ

Концептуальный прототип первой АОМ был разработан в конце 1940-х гг. Однако широкое распространение эти модели получили в начале 1990-х гг. благодаря появлению на IT-рынке микро- и миникомпьютеров относительно высокой производительности и возможности проводить с их помощью крупномасштабные имитационные эксперименты. Принято считать, что АОМ берут свое начало с вычислительных машин Джона фон Неймана, являющихся теоретическими машинами, способными к самовоспроизводству [14]. Джон фон Нейман предложил использовать машины, которые следуют детальным инструкциям для создания точных копий самих себя. Впоследствии данный подход был усовершенствован его другом Станиславом Уламом, который предложил изображать машину на бумаге в качестве набора клеток на решетке [21]. Данный подход стал началом развития клеточных автоматов. Наиболее известной реализацией взаимодействия конечных автоматов стала игра "Жизнь", предложенная Джоном Хортоном Конвеем (*John Horton Conway*), отличающаяся от машины фон Неймана достаточно простыми правилами поведения агентов [13]. Одновременно с перечисленными выше возникла новая методология научного исследования – компьютерное имитационное моделирование, которое в настоящее время включает следующие ос-

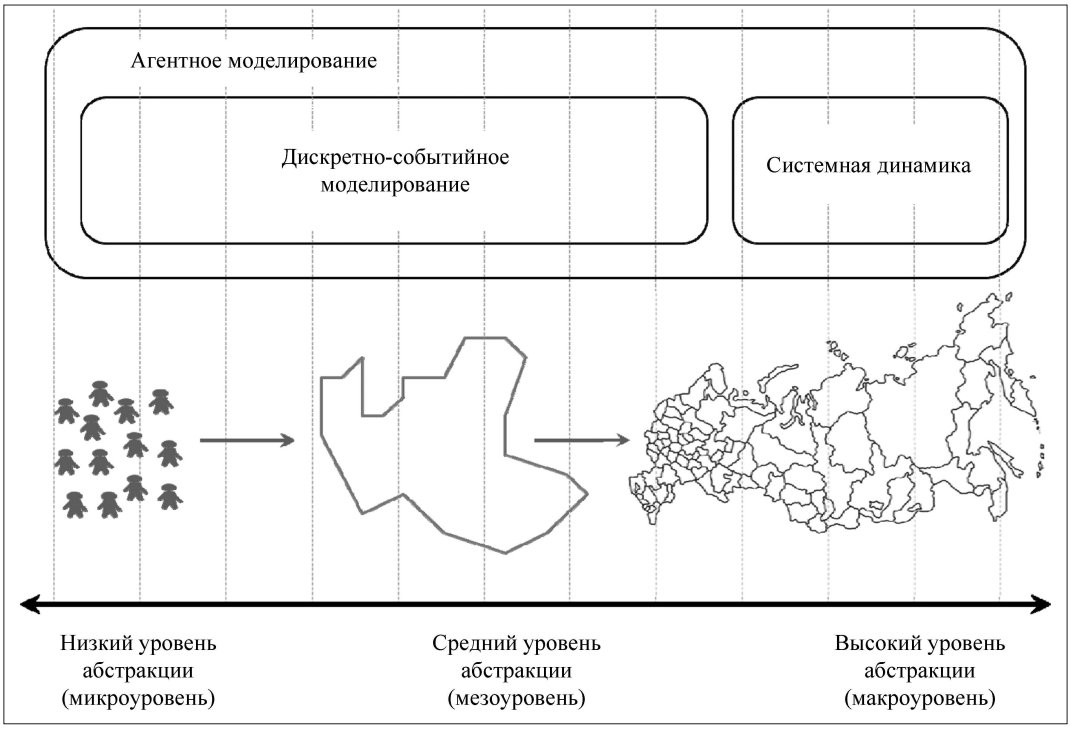


Рис. 1. Соотношение основных подходов к имитационному моделированию различных объектов и трех уровней абстракции

новные направления: системная динамика (СД); дискретно-событийное моделирование (ДС); агентное моделирование. Все эти виды моделирования применяются, в том числе, для решения социальных и экономических задач на разных уровнях абстракции. Агентное моделирование, развитие которого напрямую определяется увеличивающимися вычислительными возможностями современных компьютеров, позволяет представить (смоделировать) систему практически любой сложности из большого числа взаимодействующих объектов, не прибегая к их агрегированию. Появились программные средства (см. раздел 2), позволяющие сочетать все перечисленные выше направления имитационного моделирования.

Следует, однако, отметить, что наибольшие трудности возникают при совмещении объектов разного уровня абстракции в рамках одной модели. В этой связи разработчики математических моделей социально-экономических систем все чаще ставят вопрос об актуальности построения иерархических динамических моделей, включающих субъекты макроуровня и агентов микроуровня, поведение которых должно быть описано более адекватно реально наблюдаемому, нежели применяемые на практике методы их представления. Подходы на основе АОМ позволяют совмещать в себе агентов различного уровня абстракции и более того практически любая модель, основанная на двух других парадигмах имитационного моделирования (СД и ДС) может быть легко конвертирована в АОМ и уже в таком качестве использовать преимущества агентного подхода.

На рис. 1 в виде схемы отображено соотношение различных подходов к моделированию объектов различной степени детализации. Как видно, СД в основном используется для решения задач на высоком уровне абстракции. В свою очередь, модели ДС используются на нижнем и среднем уровнях и, как уже отмечалось, АМ применимо на всех уровнях [1]. Заметим, что к высокому уровню абстракции относятся, например, задачи прогнозирования динамики населения страны. На нижнем уровне реализуются микроуровневые модели (например, модели движения пешеходов). К среднему уровню относятся задачи, связанные, например, с оптимальным планированием перевозок внутри региона и подобные им.

С середины 1990-х гг. АОМ стали использовать для решения большого числа востребованных бизнесом и технологических задач, например:

- оптимизация сети поставщиков и планирование перевозок;
- планирование развития производства;
- прогнозирование спроса на продукцию и объема продаж;
- оптимизация численности персонала;
- прогнозирование развития социально-экономических систем (городов, регионов);
- моделирование миграционных процессов;
- имитация и оптимизация пешеходного движения;

• моделирование транспортных систем;

• прогнозирование экологического состояния окружающей среды.

Этот список можно продолжить.

Преимущества АОМ

Преимущества АОМ перед другими средствами имитационного моделирования заключаются в следующем.

• Методы на основе АОМ позволяют моделировать систему, максимально приближенную к реальности. Как уже отмечалось ранее, степень детализации АОМ, по сути, ограничивается только возможностями компьютеров. Более того, в ряде АОМ передвижение агентов задается без использования сложных формул, однако с помощью заранее определенных маршрутов и простых правил. Такие правила, с одной стороны, имитируют адаптивное мышление в процессе принятия решений, а с другой – позволяют получить неочевидные результаты на уровне агрегированных параметров. Примерами таких АОМ могут быть модели, имитирующие передвижение пешеходов, покупателей в крупных торговых центрах, спецтехники на складах и другие.

• Агентные модели обладают свойством эмерджентности. Например, в одной из разработанных в ЦЭМИ РАН АОМ имитируется работа транспортной системы г. Москвы, в ходе моделирования которой определялось поведение только отдельных агентов, в то время как более общие явления – автомобильные пробки или параметр, отражающий уровень загруженности дорог города, определялись уже в процессе расчетов с использованием другой модели [17].

• Как следует из предыдущего пункта, важным преимуществом агентного моделирования является возможность построения моделей с учетом отсутствия знаний о глобальных зависимостях в рамках моделирования соответствующей предметной области. Важно представлять логику поведения отдельных агентов, что, в свою очередь, может помочь в получении более общих знаний об изучаемом процессе.

• Агент-ориентированное моделирование является гибким инструментом, позволяющим легко добавлять и удалять агентов в модели, а также менять параметры и правила их поведения [7].

Считается, что агентное моделирование дополняет традиционные аналитические методы, а также включает в себя упомянутые выше подходы имитационного моделирования, поскольку последние могут применяться "внутри" агентной модели при формализации ее отдельных активных объектов или агентов [4]. Появление АОМ можно рассматривать как результат эволюции методологии моделирования, а именно – переход от мономоделей (одна модель – один алгоритм) к мультимоделям (одна модель – множество независимых алгоритмов).

АОМ могут объяснить причину возникновения таких явлений, как войны, обрушение рынка акций и других. В идеале АОМ могут помочь идентифицировать критические моменты времени, после наступления которых чрезвычайные последствия будут иметь необратимый характер.

* * *

Большая часть АОМ представлена в свободном доступе в Интернет (на страницах ученых-разработчиков или на популярных порталах, содержащих информацию по данной проблематике). Так, существуют специализированные издания, тематика которых напрямую связана с разработкой АОМ, например, онлайновый журнал JASSS (*Journal of Artificial Societies and Social Simulation*, http://jasss.soc.surrey.ac.uk/JASSS.html), уже ставший достаточно известным. Отметим также российский ежеквартальный интернет-журнал "Искусственные общества" (www.artsoc.ru).

Следует также упомянуть про новое направление в прикладной экономике – агент-ориентированную экономику (*Agent-based Computational Economics*, ACE), основой которой является моделирование виртуального мира, "населенного" автономными агентами (экономическими, биологическими и другой природы). В проект по созданию подобных миров вовлечено много исследователей, разработки которых представлены на сайте www.econ.iastate.edu/tesfatsi. Управление созданным виртуальным миром в соответствии с методологией ACE осуществляется без вмешательства извне, т.е. только посредством взаимодействия агентов [19]. При этом агенты должны обладать способностью к обучению.

2. Программное обеспечение для разработки агент-ориентированных моделей

В настоящем разделе перечислим наиболее известное программное обеспечение для построения АОМ. Вообще говоря, разработка АОМ не требует использования специализированных программ. Можно применять средства разработки широкого профиля (C#, Delphi и другие), однако специализированные программы содержат набор готовых библиотек для эффективного представления агентов и их среды. Реализация таких операций, как визуализация двумерной решетки, перемещение агентов по ней и ряд других с использованием подобных программ значительно упрощена, и в этой связи предпочтительнее использовать имеющийся арсенал описываемых далее средств.

Пакет SWARM является набором библиотек, написанных на языке Objective-C, служащих основой для разработок сложных мультиагентных систем. Этот пакет в свободном доступе представлен в Интернет по адресу http://www.swarm.org.

REPAST (*REecursive Porous Agent Simulation Toolkit*) Symphony представляет собой инструментальный комплекс IDE (*Integrated Development Environment*) для разработки АОМ, по функциональным возможностям сравнимый с описанным далее пакетом AnyLogic. Из достоинств REPAST следует отметить:

✓ удобство разработки моделей;

✓ открытые исходные коды и возможность использования дистрибутива бесплатно;

✓ удобные методы презентации;

✓ встроенную многопоточность.

Простой и доступной программой для разработки АОМ является бесплатно распространяемое приложение NetLogo. Изначально NetLogo был разработан как учебный инструмент, однако сейчас им пользуются не только студенты, но и тысячи исследователей. Он представляет собой программу, которая часто применяется в вузах для обучения студентов основам АОМ. Схожей функциональностью обладает программа StarLogo.

Следует также упомянуть программные средства MASON, EcoLab, SOARS, Cormas, однако больших отличий от предыдущих средств разработки АОМ у них нет.

Отдельного внимания заслуживает пакет AnyLogic [2]. Данный пакет является продуктом нового поколения для разработки и исследования имитационных моделей. Отметим, что он является единственным российским профессиональным инструментом имитационного моделирования, успешно конкурирующим с зарубежными аналогами.

В отличие от перечисленных выше пакетов для построения АОМ, AnyLogic является коммерческим продуктом, и в этой связи он лишен многих недостатков, присущих *open source* продуктам, ответственность за которые их разработчики снимают с себя, главным образом, в силу бесплатного характера распространения.

AnyLogic поддерживает различные подходы к моделированию, в том числе и агентный, для которого содержит специальную библиотеку классов AnyLogic *agent-based library*, предоставляющую возможность задания требуемой функциональности у агентов модели. Следует также отметить, что AnyLogic поддерживает все возможные способы задания поведения агентов – диаграммы состояний (стейтчарты), синхронное и асинхронное планирование событий.

Пакет AnyLogic обладает хорошими средствами визуализации и позволяет имитировать различные процессы производства, бизнеса и др.

Более подробно про саму среду разработки AnyLogic и некоторые теоретические аспекты имитационного моделирования можно прочитать в книге [2].

Основным недостатком перечисленных выше пакетов является отсутствие невозможности разработки проектов, выполняющихся на вычислительном кластере (не предусмотрен механизм распараллеливания процесса выполнения программного кода). Далее рассмотрен опыт зарубежных ученых и практиков по запуску АОМ на суперкомпьютерах.

3. Опыт зарубежных ученых и практиков

В сентябре 2006 г. стартовал проект по разработке крупномасштабной АОМ европейской экономики (EURACE, *Europe ACE – Agent-based Computational Economics*) с очень большим числом автономных агентов, взаимодействующих в рамках социально-экономической системы [8]. В выполнение этого проекта вовлечены экономисты и программисты из восьми научно-исследовательских центров Италии, Франции, Германии, Великобритании и Турции, а также консультант из Колумбийского университета США – нобелевский лауреат Джозеф Стиглиц.

Исследование базируется на методологии ACE в целях преодоления ограничений, лежащих в основе широко распространенных моделей, рассматривающих агрегированных агентов, а также предполагающих их рациональное поведение и состояние равновесия. Для модели используется ГИС с широким перечнем рассматриваемых объектов – предприятия, магазины, школы, транспортные сети и другие.

Как отмечают разработчики, практически все существующие АОМ рассматривают либо отдельные отрасли, либо относительно небольшой географический район и, соответственно, небольшую популяцию агентов. В свою очередь в рамках EURACE рассматривается весь Европейский Союз. И по своим масштабам и сложности разрабатываемая модель является уникальной, а ее численное разрешение требует использования суперкомпьютеров и специально разработанного программного обеспечения.

Для наполнения модели статистической информацией использовались данные (в виде ГИС-карт) статистической службы Европейского союза уровня NUTS-2[1], представляющие информацию о 268 регионах 27 стран Европейского союза.

В модели три типа агентов: домашние хозяйства (около 10^7), предприятия (около 10^5) и банки (около 10^2). Все они имеют географическую "привязку", а также связаны друг с другом посредством социальных сетей, деловых и других отношений.

EURACE реализован с помощью гибкой, масштабируемой среды для моделирования агентных моделей FLAME (*Flexible Large-scale Agent Modeling Environment*), разработанной С. Коакли и М. Холкомбом (более подробно см. www.flame.ac.uk), изначально созданной для моделирования роста биологических клеток, выращиваемых в различных условиях. Используемый во FLAME подход основан на так называемых *X*-машинах, напоминающих конечные автоматы, однако отличающихся от них. Суть в том, что у каждой *X*-машины есть набор данных (своего рода память), а переходы между состояниями являются функциями не только состояния, но и набора данных памяти.

[1] NUTS (фр. *nomenclature des unités territoriales statistiques*) – номенклатура территориальных единиц для целей статистики – стандарт территориального деления стран для статистических целей, разработанный Европейским Союзом и детально охватывающий только входящие в него страны. Существуют NUTS-единицы трех уровней, при этом второй уровень (NUTS-2) соответствует административным округам Германии, графствам в Великобритании и т.д.

Таким образом, каждый агент в системе FLAME представлен *X*-машиной, причем предусмотрено общение между агентами посредством передачи сообщений. При работе в параллельном режиме на суперкомпьютере обмен сообщениями между агентами требует больших вычислительных затрат, в связи с чем агенты изначально были распределены по процессорам в соответствии со своим географическим положением. Тем самым разработчики программы минимизировали вычислительную нагрузку, исходя из первоначального предположения о том, что в основном общение между агентами происходит в рамках небольшой социальной группы, проживающей приблизительно в одной местности. Таким образом, весь модельный ландшафт был поделен на небольшие территории и распределен между узлами суперкомпьютера.

С помощью разработанной модели был проведен ряд экспериментов в целях исследования рынка труда. Не рассматривая детально получившиеся числовые результаты, отметим, что, по мнению авторов, основной вывод исследования заключается в том, что даже у двух регионов со схожими условиями (ресурсы, развитие экономики и др.) в течение продолжительного периода (10 лет и более) макропоказатели могут значительно измениться за счет первоначальной неоднородности агентов.

Подробнее информацию организационного и научного содержания по этому проекту можно найти на веб-странице проекта: www.eurace.org.

В рамках АОМ EpiSims, разработанной исследователями из Института биоинформатики Вирджинии (*Virginia Bioinformatics Institute*), рассматриваются как перемещения агентов, так и их контакты в рамках среды, максимально приближенной к реальности и включающей в себя дороги, здания и прочие инфраструктурные объекты [9]. Для построения модели потребовался большой массив данных, включающий информацию о здоровье отдельных людей, их возрасте, доходе, этнической принадлежности и ряде других характеристик.

Изначально цель исследования заключалась в построении АОМ большой размерности для ее использования на суперкомпьютере, с помощью которой можно будет изучать вопросы распространения болезней в обществе. Однако впоследствии в процессе работы решалась также задача по созданию специализированного программного обеспечения ABM++, которое позволяет осуществлять разработку АОМ на языке C++, а также содержит функции, облегчающие распределение исполняемого программного кода на отдельные узлы суперкомпьютера. Кроме перечисленных функций, ABM++ предоставляет возможность для динамического перераспределения потоков вычислений, а также методы по синхронизации происходящих событий.

ABM++ является модернизированной версией разработанного в 1990–2005 гг. в Лос-Аламосской национальной лаборатории (США) инструментального комплекса, появившегося в процессе построения крупномасштабных АОМ (EpiSims, TRANSIMS, MobiCom). Модернизация этого инструментария и появление первой версии ABM++ произошли в 2009 г.

Межпроцессорные связи между вычислительными узлами в АОМ часто требуют синхронизации происходящих в модели событий. Средства ABM++ позволяют разрабатывать модели, отвечающие этому требованию. Например, в социальных моделях агенты часто перемещаются между различными точками пространства (работа, дом и другие), а на программном уровне этому соответствует смена узла кластера и здесь важно, чтобы модельное время принимающего узла было синхронизировано со временем узла, который агент только что покинул.

Также в ABM++ реализована библиотека MPIToolbox, которая соединяет интерфейс C++ API (*Application Programming Interface*) и *Message Passing Interface* (MPI) суперкомпьютера и обеспечивает более быструю реализацию задач по передаче данных между узлами кластеров.

Разработка ABM++ осуществлялась в Ubuntu Linux – операционной системе с компиляторами gcc/g++. В качестве интегрированной среды разработки рекомендуется Eclipse с модулем расширения (плагином) для поддержки C и C++, а также с плагином PTP (*Parallel Tools Platform*), обеспечивающим разработку и интеграцию приложений для параллельных компьютерных архитектур. Среда Eclipse поддерживает интеграцию с TAU (*Tuning and Analysis Utilities*) Performance System – инструментальным комплексом для анализа и отладки параллельных программ, что также упрощает разработку агентных моделей.

Специалисты другой исследовательской группы также из Института биоинформатики Вирджинии разработали инструментарий для изучения особенностей распространения инфекционных заболеваний внутри различных групп населения – EpiFast. К его достоинствам можно отнести масштабируемость и высокую скорость исполнения. Например, имитация социальной активности жителей Большого Лос-Анджелеса (агломерации с населением свыше 17 млн человек) с 900 млн связей между людьми на кластере с 96 двухъядерными процессорами POWER5 заняла менее 5 минут. Такая достаточно высокая производительность связана с механизмом распараллеливания, предложенным в работе [16].

Как правило, для реализации модели социума на группе процессоров используют несколько естественных способов. Например, один из них базируется на представлении социума в виде набора неструктурированных связей между индивидуумами. При распараллеливании эти связи равномерно делятся на группы, число которых соответствует числу процессоров. Отрицательная сторона такого подхода заключается в сложности отслеживания статуса отдельного человека. Если, например, *i*-й индивидуум инфицирован, то информация об этом

должна быть синхронизирована во всех группах, так как нет сведений о том, в каких из них содержатся связи данного человека. В свою очередь, это влечет за собой большую вычислительную нагрузку. Другой подход основан на разделении людей – агентов модели в соответствии с их географическим месторасположением. Однако в этом случае, учитывая, что население обычно распределено неравномерно, распределение вычислительной нагрузки требует применения достаточно сложных алгоритмов для ее выравнивания, что также создает дополнительную вычислительную нагрузку.

Предлагаемый в настоящей работе метод заключается в равномерном распределении агентов, с соответствующими им односторонними исходящими связями по группам, число которых равно числу процессоров. Соответствующий этому методу вычислительный алгоритм основан на взаимодействии ведущих (*master*) и ведомых (*slave*) процессоров, организованном следующим образом: ведущий процессор "знает", какой из процессоров обслуживает конкретного агента, а каждый из ведомых процессоров "знает" только обслуживаемых (локальных) агентов. В процессе вычислений ведомые процессоры посылают запросы ведущим процессорам относительно связей с нелокальными агентами. В свою очередь ведущие процессоры не осуществляют никаких расчетов, кроме поиска нелокального агента для каждой внешней связи и пересылают запросы соответствующим процессорам. Таким образом, по мнению разработчиков, предлагаемый алгоритм позволяет проводить эффективное и высокоскоростное имитационное моделирование систем с очень большим числом агентов.

Классические модели распространения эпидемий преимущественно основывались на использовании дифференциальных уравнений, однако такой аппарат затрудняет учет связей между отдельными агентами и многие другие их индивидуальные особенности. В свою очередь, АОМ позволяют преодолеть указанные недостатки. Так, в 1996 г. Дж. Эпштейн и Р. Акстелл опубликовали описание одной из первых АОМ, имитирующей процесс распространения эпидемий [10]. Агенты модели, отличающиеся друг от друга восприимчивостью к заболеванию, зависящей от состояния иммунной системы, географически распределены в рамках определенной территории. Вместе с тем, по мнению авторов модели, агенты, число которых всего несколько тысяч, реализуют достаточно примитивное поведение.

В дальнейшем под руководством Дж. Эпштейна и Дж. Паркера в Центре социальной и экономической динамики в Брукингском институте (*Center Social and Economic Dynamics at Brookings*) была построена одна из самых больших АОМ, включающая в себя все население США, т.е. порядка 300 млн агентов [18].

Построенная модель удовлетворяет нескольким требованиям. Во-первых, она позволяет предсказать последствия распространения заболеваний различного типа. Во-вторых, модель ориентирована на поддержку двух сред для вычислений. Одна из них состоит из кластеров с установленной 64-битной версией операционной системы (ОС) Linux, а другая – из серверов с четырехъядерными процессорами и установленной на них ОС Windows. В этой связи в качестве языка программирования был выбран Java , хотя разработчики и не уточняют, какую именно реализацию Java они тогда использовали. Последнее требование заключается в возможной поддержке численности агентов от нескольких сотен миллионов до 6 миллиардов.

Способ распределения агентов между аппаратными ресурсами состоит из двух фаз. Первая фаза заключается в распределении агентов по компьютерам, задействованным в работе, а вторая – в распределении агентов модели по потокам на каждом из компьютеров. В процессе моделирования каждый поток может остановиться (в заранее обусловленное время) для передачи сообщений другим потокам. Все подготовленные к отправке сообщения до определенного момента времени хранятся в пуле сообщений, а затем одновременно отправляются на обработку.

В реализации модели есть также две вспомогательные утилиты. Первая из них управляет потоками на отдельном компьютере, а вторая следит за тем, чтобы все сообщения между потоками были выполнены до момента возобновления вычислительного процесса.

При распределении агентов по аппаратным ресурсам следует учитывать следующие два обстоятельства: производительность узла напрямую зависит от числа инфицированных агентов; контакты, предполагающие передачу сообщений между потоками, требуют гораздо больше вычислительных затрат, нежели контакты, ограничивающиеся локальной информацией. Исходя из этого, возможно различное распределение агентов. С одной стороны, можно поделить все рассматриваемое географическое пространство на равные части, число которых должно соответствовать числу узлов, а затем определить для каждого узла какой-либо географический регион. За счет этого возможна балансировка вычислительной нагрузки между узлами. С другой стороны, можно закрепить определенную территорию, представляющую собой единую административную единицу, за конкретным узлом. Это может способствовать снижению вычислительной нагрузки за счет уменьшения числа контактов, требующих передачи сообщений между потоками. Если первый способ влечет за собой увеличение вычислительной нагрузки за счет ресурсоемких контактов, то второй способ в ряде случаев несет в себе потенциальную опасность значительного дисбаланса между аппаратными ресурсами. Вспышка, например, какого-либо заболевания в одном из регионов загрузит один из вычислительных узлов, в то время как некоторые другие будут простаивать.

Разработанная модель (*US National Model*) включает в себя 300 млн агентов, перемещающихся по карте страны в соответствии с матрицей корреспонденций

размерностью 4 000×4 000, специфицированной с помощью гравитационной модели. С ее помощью был проведен вычислительный эксперимент, имитирующий 300-дневный процесс распространения болезни, особенности которой заключаются в 96-часовом периоде инкубации и 48-часовом периоде заражения. Один из результатов исследования заключался в том, что распространение заболевания идет на спад после того, как 65 % людей выздоровели и приобрели иммунитет. Эта модель была неоднократно использована в Университете Джонса Хопкинса (*Johns Hopkins University*), а также в Департаменте национальной безопасности США для исследований по быстрому реагированию на различного рода эпидемии [12].

В 2009 г. на основе описанной выше модели была разработана ее вторая версия, включающая 6,5 млрд агентов, спецификация действий которых проводилась с учетом имеющихся статистических данных. С ее помощью имитировались последствия от распространения вируса гриппа A(H1N1/09) в масштабах всей планеты.

Отметим, что подобная модель была разработана ранее в Лос-Аламосской национальной лаборатории. Результаты работы с этой моделью были представлены в широкой печати 10 апреля 2006 г. [5]. Для имитационного моделирования на основе этой модели был использован один из мощнейших по тем временам суперкомпьютеров Pink, состоящий из 1024 узлов, с двумя процессорами с частотой 2,4 ГГц и 2 ГБ памяти на каждом из них. С помощью этой крупномасштабной модели, включающей 281 млн агентов, были рассмотрены сценарии распространения различных вирусов (в том числе и H5N1) с учетом различного рода оперативных вмешательств (вакцинация, закрытие школ и введение карантина на некоторых территориях).

Естественно, что для подобных задач можно использовать не только суперкомпьютер, а более дешевые решения. Показателен пример ученых Университета Ньюкасла из Австралии (*The University of Newcastle, Australia*), построивших кластер, включающий в себя 11 узлов – отдельных персональных компьютеров, объединенных в сеть с пропускной способностью до 100 Мбит/с [20].

Программное обеспечение узлов кластера было одинаковое – ОС Debian GNU/Linux и JavaParty – кроссплатформенное программное расширение языка Java (ПО JavaParty включает в себя средства для распределения потоков выполнения кода по узлам кластеров).

Для оценки производительности кластера были использованы три версии одной АОМ, в рамках которой тестировались различные стратегии агентов, принимающих участие в аукционе. Версии различались сложностью стратегий агентов.

Основной вывод по результатам тестирования заключается в том, что использование кластеров оправдано лишь в том случае, если вычислительная нагрузка отдельных узлов достаточно интенсивная. В противном случае распределение потоков наоборот уменьшает скорость работы модели. Отметим, что такое поведение кластерных приложений не является традиционным.

Представляет большой интерес опыт ученых из Оак-Риджской национальной лаборатории (*Oak Ridge National Laboratory*) по распараллеливанию агентных моделей на суперкомпьютерах, построенных с использованием графических процессоров (*graphics processing unit,* GPU) вместо обычных центральных процессоров (*central processing unit,* CPU). Напомним, что для графических процессоров компания NVIDIA разработала специальную архитектуру параллельных вычислений – CUDA (*Compute Unified Device Architecture*), которая позволяет эффективно управлять памятью графического ускорителя, организовывать доступ к его набору инструкций и эффективно организовывать параллельные вычисления, используя упрощенную версию языка C.

Основное отличие между CPU и GPU (помимо механизмов доступа к памяти) заключается в том, что первые созданы для выполнения одного потока последовательных инструкций с максимальной производительностью, а GPU – для выполнения большого числа параллельных потоков.

В настоящее время решения на базе GPU приобретают большую популярность. Так ПК на базе процессора NVIDIA Tesla C2070 с 448 ядрами CUDA может обеспечить пиковую производительность до 1 терафлопа, т.е. он в сотни раз быстрее обычного ПК. Кроме того, подобный суперкомпьютер имеет компактные размеры (чуть больше обычного системного блока), не требует дополнительных систем охлаждения и электропитания (требуется лишь блок питания от 700 Вт до 1 кВт в зависимости от конфигурации). В свою очередь стоимость такого персонального суперкомпьютера с такой потрясающей для своих размеров производительностью составляет 6 000–7 000 долл. США, что несопоставимо меньше стоимости суперкомпьютеров на основе CPU.

Ученые Оак-Риджской национальной лаборатории К. Перумалла и Б. Ааби в многочисленных статьях, размещенных на сайте http://kalper.net/kp, описывают алгоритмы запуска АОМ на суперкомпьютерах двух видов (на основе графических и центральных процессоров). В общих чертах эти алгоритмы схожи с описанными выше. Они основаны на распределении агентов по процессорам, исходя из их географического расположения, однако больший интерес в этих статьях представляет сравнение производительности CPU и GPU при реализации некоторых агентных моделей, предусматривающих графическое отображение.

Например, при тестировании производительности выполнения игры "Жизнь" размерностью 3 750×3 750 (с максимальным числом агентов приблизительно равным 14 млн), по словам исследователей, скорость GPU была почти в 14 раз выше, чем у CPU [15]. Авторы при этом отмечают, что такое преимущество

имеет место только для отдельных задач и, конечно же, GPU не являются полноценной заменой CPU (в их нынешнем виде они просто для этого и не предназначены).

Стоит отметить, что как GPU приближаются к CPU, становясь более универсальными за счет увеличивающихся возможностей по расчетам чисел с двойной и одинарной точностью, так и в CPU увеличивается число ядер, а соответственно, и способность к параллельным расчетам.

4. Адаптация агент-ориентированных моделей для суперкомпьютера: предлагаемый подход

Проблема масштабирования

Важно понимать, что проблема масштабирования программ для суперкомпьютеров довольно фундаментальна. Несмотря на то, что и традиционная последовательная, и параллельная суперкомпьютерная программы реализуют одни и те же функции, целевые установки на их разработку, как правило, разные.

При начальной разработке сложного прикладного ПО, в первую очередь, стараются сократить издержки на программирование, обучение персонала, повысить уровень переносимости с одной платформы на другую и добиться ряда других целей, а оптимизацию откладывают "на потом". Такой подход вполне разумен, так как на ранних стадиях приоритетом разработки является исключительно выполнение функций. Однако после того, как разработанное ПО уже начало внедряться, часто выясняется, что на больших реальных данных для его реализации не хватает производительности. Принимая во внимание тот факт, что современные суперкомпьютеры – это вовсе не "разогнанные" в тысячи раз ПК, для запуска на суперкомпьютере программу приходится существенно видоизменять. Причем эффективно сделать это без специальных знаний и навыков удается далеко не всегда.

При грамотно проведенной работе значительное повышение эффективности обычно достигается на трех уровнях:

- распараллеливание счета;
- специализация вычислительных библиотек, учитывающая особенности задачи;
- низкоуровневая оптимизация.

Специализация и низкоуровневая оптимизация

Перед тем, как начать вычислительные эксперименты на суперкомпьютере, программу следует максимально оптимизировать и адаптировать к целевой аппаратной платформе. Если этого не сделать, то параллельная версия будет лишь хорошим тестом для суперкомпьютера, а собственно сам счет будет весьма неэффективным. По аналогии – нецелесообразно посылать на боевое задание полк необученных новобранцев. Их нужно сначала хорошо обучить выполнять их задачу (специализация, оптимизация программного обеспечения), а также научить эффективно владеть оружием (низкоуровневая оптимизация программного обеспечения). Только в этом случае использование суперкомпьютера будет по-настоящему эффективным.

В универсальных системах моделирования типа AnyLogic предоставляемые процедуры универсальны. А универсальный код часто можно прооптимизировать для конкретного семейства задач.

Выбор системы поддержки моделирования

Прежде всего следует заметить, что АОМ можно программировать и без специальной среды, непосредственно на любом объектно-ориентированном языке (об этом уже упоминалось ранее).

Однако, несомненно, более разумным подходом будет использование одной из хорошо зарекомендовавшей себя системы для АОМ, по причине унифицированной реализации типичных способов взаимодействия агентов. Наиболее известные из них уже были отмечены выше, здесь же рассмотрим систему ADEVS.

Программная система ADEVS представляет собой набор низкоуровневых библиотек для дискретного моделирования, выполненных на языке C++. Из ее достоинств следует отметить:

– простоту реализации;

– высокую производительность моделей;

– поддержку основных численных методов при построении моделей;

– встроенное распараллеливание процесса имитационного моделирования (симуляции) при помощи OpenMP;

– возможность применения стандартных средств распараллеливания;

– достаточно быстрое на настоящее время развитие библиотек;

– кроссплатформенность;

– низкоуровневость (текущие функциональные требования не накладывают никаких ограничений на модель);

– независимость скомпилированного кода от нестандартных библиотек;

– открытый исходный код.

Однако существенными недостатками этого продукта являются полное отсутствие средств презентации и достаточно сложная, по сравнению, например, с AnyLogic, разработка моделей. Как следствие, этот продукт не может использоваться для построения моделей на уровне заказчика. Однако он представляет собой эффективную платформу для реализации параллельных программ имитационного моделирования (симуляторов).

Основными элементами программы с использованием библиотеки ADEVS при построении АОМ обычно являются:

✓ симулятор adevs::Simulator< X >;

✓ примитив агентов adevs::Atomic< X >;

✓ модель (контейнер агентов) adevs::Digraph < VALUE, PORT >.

При этом агент представляет собой следующий класс (рис. 2).

Соответствующие виртуальные функции определяют необходимые для симуляции параметры:

void delta_int() – реакция агента на микроправило;

void delta_ext(...) – микрореакция агента на макроправило;

void delta_conf() – макрореакция агента на макроправило;

```
template <class X> class Atomic: public Devs<X>
        public:
                virtual void delta_int() = 0;
                virtual void delta_ext(double e, const Bag<X>& xb) = 0;
                virtual void delta_conf(const Bag<X>& xb) = 0;
                virtual void output_func(Bag<X>& yb) = 0;
                virtual double ta() = 0;
                virtual void gc_output(Bag<X>& g) = 0;
```

Рис. 2. Представление агента в ADEVS

void output_func() – вычисление выходных воздействий агента;

double ta() – время следующего события для агента;

void gc_output(...) – данные для процедуры сборки мусора.

В силу перечисленных выше достоинств, суперкомпьютерную версию описываемой ниже программы было решено реализовывать на базе ADEVS . В рамках этой работы были разработаны MPI-версия симулятора ADEVS, а также система визуализации процесса счета на базе библиотеки Qt.

Далее даны краткие описания разработанной модели и последующего ее запуска на суперкомпьютере.

Исходная агент-ориентированная модель

Первый этап разработки описываемой ниже АОМ заключался в построении инструментария, успешно решающего задачу исследования на обычных, последовательных компьютерах, а также в настройке параметров модели. После ее успешной апробации с небольшим числом агентов (учитывая их сложность, персональный компьютер с хорошей производительностью способен проводить вычисления с удовлетворительной скоростью над числом агентов около 20 тыс.) было решено конвертировать ее для суперкомпьютера, что являлось вторым этапом разработки. Отметим, что на первом этапе был использован пакет AnyLogic, технические возможности которого позволили достаточно быстро отладить модель и настроить ее параметры.

Вернемся к описанию разработанной модели. Итак, в 2009 г. в ЦЭМИ РАН была разработана АОМ воспроизводства научного потенциала России на базе ГИС.

По своей сути ГИС – это системы, позволяющие создавать базы данных, сочетающие в себе графическое и атрибутивное представление разнородной информации, а также обеспечивающие возможность пространственного анализа данных и представление его результатов в наиболее привычной для пользователей форме (в виде графиков, диаграмм, таблиц, карт и так далее).

Общая схема построения АОМ на базе ГИС состоит из следующих трех основных этапов.

I. Прорисовка среды для функционирования агентов (например, карты страны).

II. Для каждого элемента карты задаются свойства и методы, инициализируемые перед запуском модели с помощью соответствующих запросов к базе данных ГИС.

III. Для каждого элемента карты создается определенное число экземпляров объектов типа "агент".

По указанной схеме была разработана АОМ распространения знаний. Ниже приведено ее концептуальное описание (в виде набора тезисов).

1. Жизненный цикл агента состоит из двух основных стадий (рождение и смерть) и промежуточных состояний, отслеживаемых на каждом шаге работы модели.

2. От момента рождения и до определенного возраста (по умолчанию 18 лет) агент не участвует в процессе производства ВВП.

3. В течение жизни агент может стать либо обычным работником, либо ученым, либо "прикладником". Прослойка ученых создает базис для формирования прослойки "прикладников".

4. Становление ученого. По достижении работоспособного возраста, агент с некоторой вероятностью может стать ученым. Если до 25 лет агент не становится ученым, то он не будет им никогда.

Ученые не участвуют в создании ВВП, но в то же время:

- производят знания, потребляемые "прикладниками", которые участвуют в процессе производства ВВП;
- формируют среду, которая оказывает влияние на число "прикладников".

5. Агент перестает быть ученым (или "прикладником") из-за низкой зарплаты (если заработная плата ученого (или "прикладника") заметно ниже, чем в социуме, то он уходит на работу в другие отрасли). Агент – бывший ученый (или "прикладник") может снова вернуться в науку (или на работу в инновационно-активные предприятия), если заработная плата в науке (или прикладной науке) станет выше, чем в среднем по социуму, и если время отрыва от научной деятельности не превышает некоторого порога (по умолчанию 5 лет).

6. Продолжительности жизни агента-ученого (и "прикладника") выше, чем у обычного человека (по умолчанию на 10 лет). Однако в модели фактор продолжительности жизни не принимается в расчет при выборе профессии.

7. С задаваемой вероятностью (рассчитанной на основе российской статистики) агенты могут иметь ребенка. При этом ребенок агента-ученого (или "прикладника") становится ученым (или "прикладником") с большей вероятностью.

8. В модели предусмотрен экзогенный параметр – средняя зарплата высокоразвитых стран мира. Если в моделируемом социуме средняя зарплата (как у ученых, так и у представителей других профессий) становится намного ниже, чем в других странах, то ученый (или "прикладник") выбывает из социума навсегда (переезд в другую страну).

С помощью разработанной модели можно рассчитать последствия:

– от увеличения заработной платы (всем типам работников);

– от организации инновационных центров;

– от дополнительных инвестиций в науку.

Спецификация агентов модели осуществлялась с учетом следующих параметров (рис. 3, см. вторую сторону обложки):

- возраст;
- продолжительность жизни;
- специализация родителей;
- место работы;
- регион проживания;
- доход.

Спецификация регионов (элементов ГИС) проводилась на основе следующих параметров:

- географические границы;
- число жителей;
- число работников (по типам);
- валовый региональный продукт (ВРП);
- ВРП на душу;
- объем инвестиций;
- объем инвестиций на душу;
- средняя заработная плата;
- средняя продолжительность жизни;
- показатель прироста населения и ряда других.

Для наполнения модели данными использовались статистические сборники: Регионы России; Наука России в цифрах; Индикаторы науки; Индикаторы образования. Кроме того, были использованы социологические базы данных RLMS, а также результаты, полученные с помощью вычислимой модели экономики знаний [3].

На рис. 4 (см. вторую сторону обложки) изображено рабочее окно разработанной АОМ (точки – агенты). Благодаря возможностям ГИС в процессе работы системы можно получать оперативную информацию о социально-экономическом положении всех регионов России, в том числе с использованием картографической информации, меняющейся в режиме реального времени в зависимости от значений эндогенных параметров.

Конвертация модели в суперкомпьютерную программу

Выше уже отмечалось, что существует ряд проблемных вопросов, связанных с использованием средств разработки АОМ для реализации программных средств имитационного моделирования на высокопроизводительных вычислительных кластерных установках. Так и у AnyLogic в связи со сложностью отделения вычислительной части комплекса от той, которая обеспечивает функции визуализации, а также за счет реализации его кода на языке высокого уровня Java производительность выполнения приложений существенно ниже, чем у ADEVS. Кроме того, очень проблематично (либо очень трудоемко) переработать генерируемый код в параллельно выполняемую программу.

Далее представлен алгоритм конвертации модели AnyLogic в суперкомпьютерную программу.

Трансляция модели

На рис. 5 (см. вторую сторону обложки) приведен фрагмент XML-кода (дерево, отображающее структуру модели), сгенерированного программой AnyLogic.

В процессе работы конвертера это дерево транслируется в код C++ программы, вычисляющей эту модель. Проход дерева совершается "в глубину", выделяя при этом перечисленные далее ключевые стадии и совмещая их с выполнением задачи трансляции.

1. Генерация основных параметров, включая поиск корня дерева и считывание параметров дочерних вершин, таких как имя модели, адрес сборки, тип модели, тип презентации.

2. Генерация классов, в том числе:

- построение списка классов;
- считывание основных параметров класса;
- считывание переменных;
- считывание параметров;
- считывание функций;
- генерация списка функций;
- считывание кода функций;
- преобразование кода функций: Java $\rightarrow$ C++;
- считывание используемых фигур и элементов управления;
- генерация кода инициализации фигур и элементов управления;
- генерация кода конструктора, деструктора, визуализатора;
- генерация структуры класса;
- генерация кода заголовочного и *source*-файлов.

3. Генерация симулятора, а именно поиск вершины, хранящей информацию о процессе симуляции (управляющие элементы, значения важных констант, элементы презентации и так далее).

4. Генерация общих файлов проекта, в том числе main.cpp, mainwindow.h, mainwindow.cpp и подобных им.

Импортирование входных данных

В качестве входных данных в модель загружаются данные геоинформационной составляющей исходной модели (карты России), содержащей всю необходимую информацию.

Генерация классов и преобразование кода функций

В ходе трансляции неоднократно возникает задача о преобразовании исходного кода функций из языка

Java в язык C++. Его можно представить в виде последовательных замен конструкций.

В Java нет столь явного, как в C++, различия между объектом и указателем на объект, поэтому структуры работы с ними не отличаются. В силу этого обстоятельства вводится список классов, в которых важно использовать операции с указателем на объект, а не с самим объектом. При этом отслеживаются все переменные этих классов, с последующей заменой в пределах данной функции при обращении к ним на соответствующие обращения к указателям.

В Java и конкретно в библиотеке AnyLogic есть некоторое число функций и классов, аналогов которым нет ни в C++, ни в библиотеке ADEVS. В этой связи были реализованы дополнительные библиотеки shapes.h, mdb-work.h, в которых и реализованы недостающие функции.

На этапе генерации основных параметров списка классов получается название основного класса и названия моделируемых классов-агентов. В код основного класса встраивается процедура добавления агента в зону видимости симулятора.

Статистика и визуализация временных срезов

В силу неинтерактивного режима запуска программы на суперкомпьютерных установках, сбор выходных данных и визуализация были разделены. Такой подход связан с неравномерностью нагрузки на такие установки в разное время суток, а также тем фактором, что монопольный доступ к ним невозможен. После пересчета модели, получившаяся на выходе информация может быть снова визуализирована, например, следующим образом (рис. 6, см. третью сторону обложки).

Доступность для расчетов

На момент проведения тестовых испытаний рассматриваемой модели были доступны три суперкомпьютера (см. таблицу), входящих в первую пятерку суперкомпьютерного рейтинга Топ50 среди стран СНГ (http://top50.supercomputergs.ru//?page=rating в редакции от 21.09.2010).

Из них для проведения имитационного моделирования использовались два суперкомпьютера – "Чебышев" и МВС-100К.

Сравнение производительности

За счет применения суперкомпьютерных технологий и оптимизации программного кода удалось добиться очень высокой производительности.

Как уже отмечалось ранее, обычный ПК с хорошей производительностью способен проводить вычисления с удовлетворительной скоростью над числом агентов около 20 тыс. (поведение каждого из них задается приблизительно 20 функциями), и при этом среднее время пересчета одной единицы модельного времени (один год) составляет около минуты. При большем числе агентов (например, 100 тыс.) компьютер попросту "зависает".

В свою очередь, использование всего 200 процессоров суперкомпьютера и выполнение оптимизированного кода позволило увеличить число агентов до 100 млн, а число модельных лет до 50. При этом такой гигантский массив вычислений был выполнен за период времени, приблизительно равный 1 мин 30 с (в зависимости от типа используемых процессоров).

Отметим также еще одну интересную особенность. По результатам экспериментов с разработанной АОМ было выяснено, что масштабирование модели само по себе имеет определенное значение. Так, при запуске одной и той же версии модели на 50 модельных лет с одинаковыми параметрами (за исключением числа агентов – в первом случае 100 млн агентов, а во втором – 100 тыс. агентов) были получены результаты (а именно масштабированное число агентов), отличающиеся приблизительно на 4,5 %.

В этой связи можно предположить, что, по всей видимости, в сложных динамических системах одни и те же параметры (рождаемость, продолжительность жизни и подобные им) могут приводить к различным результатам в зависимости от размера социума.

Дальнейшее развитие

В дальнейшем планируется обеспечить поддержку более широкого подмножества операторов/конструкций языка Java при определении исходной модели (Subset of Java → C++), а также автоматизировать запуск суперкомпьютерной программы из среды Eclipse (полностью автоматизировать процесс трансляции и распараллеливании программного кода).

Заключение

Несмотря на то что подобная работа может быть проделана практически для любой начальной реализации, в перспективе нужны системы, которые по возможности ликвидировали бы этот непростой этап, который сегодня затрудняет широкое внедрение суперкомпьютерного моделирования.

Доступные для исследовательской группы суперкомпьютеры

Позиция в Топ50	Суперкомпьютеры	Узлы	CPU	Ядра	RAM/узел, ГБайт	TFlops
1	"Ломоносов" (МГУ)	4 420	8 840	35 360	12	414
3	МВС-100К (МСЦ РАН)	1 278	2 556	10 224	8	123
4	"Чебышев" (МГУ)	633	1 250	5 000	8	60

Тот факт, что в исходном представлении модели значительную роль играет код на языке программирования, отражает фундаментальное свойство агент-ориентированного подхода "модель – это программа".

Декларативный подход (набор правил вместо обычной программы) может сильно упростить задачу определения поведения системы.

Проиллюстрируем изложенные выше соображения (рис. 7, см. третью сторону обложки). При возникновении новой задачи мы получаем для нее некоторые исходные данные, а в свою очередь аналитики определяют, к какому классу задач она относится в целях подбора соответствующей спецификации для построения модели. Для этого используется наиболее адекватный DSL (*Domain Specific Language*), который позволяет описывать модели (в качестве IDE, к примеру, может быть Eclipse). Для этого языка, как правило, уже имеется ряд наработок на уже прошедших апробацию задачах этого класса, которые содержат в себе уже адаптированные, учитывающие специфику этих задач библиотеки. В своей совокупности все это образует метаязык, который затем реализуется в некоторой среде разработки и именно в ней, с точки зрения прикладников, в итоге и решается задача. После тестовых прогонов готовой программы на локальной машине следует финальная стадия – запуск на суперкомпьютере.

Общая последовательность действий в процессе разработки будет при этом следующая (рис. 8, см. третью сторону обложки).

1. Выбор языка описания (ЯО) модели (метамодели и ЯО моделей). Первый шаг контролируется специалистами формализации моделей и собственно программистами, которые выбирают язык, наиболее подходящий для описания данной задачи.

2. Описание модели на выбранном языке (этот шаг контролируется заказчиком).

3. Поиск и выбор необходимых шаблонов программирования, по сути, – выбор удобной среды разработки с наиболее подходящим инструментарием.

4. Подготовка алгоритмов распараллеливания и выбор аппаратной части.

5. Выбор библиотек, других программных составляющих и непосредственно программирование.

На четвертом и пятом шагах происходит перевод модели в низкоуровневые языки, подходящие для распараллеливания и, собственно, проводится запуск программы, настройка средств отображения результатов и представление их заказчику (после чего возможны дополнительные итерации).

СПИСОК ЛИТЕРАТУРЫ

1. **Борщев А.В.** Практическое агентное моделирование и его место в арсенале аналитика // Exponenta PRO. 2004. № 3–4 (7–8). С. 38–47.

2. **Карпов Ю.** Имитационное моделирование систем. Введение в моделирование с AnyLogic 5. СПб.: БХВ-Петербург, 2006.

3. **Макаров В.Л., Бахтизин А.Р., Бахтизина Н.В.** Вычислимая модель экономики знаний // Экономика и математические методы. 2009. № 1.

4. **Паринов С.И.** Новые возможности имитационного моделирования социально-экономических систем // Искусственные сообщества. 2007. № 3–4. С. 26–62.

5. **Ambrosiano N.** Avian flu modeled on supercomputer. Los Alamos National Laboratory NewsLetter. 2006. Vol. 7. No. 8.

6. **Axelrod R.** The Complexity of Cooperation: Agent-Based Models of Competition and Collaboration. Princeton: Princeton University Press, 1997.

7. **Bonabeau E.** Agent-based modeling: methods and techniques for simulating human systems. // Proc. National Academy of Sciences. 2002. 99(3): 7280–7287.

8. **Deissenberg Christophe, Sander van der Hoog, Dawid Herbert.** EURACE: A Massively Parallel Agent-Based Model of the European Economy / Document de Travail. 2008. 39. 24 Juin.

9. **Roberts D.J., Simoni D.A., Eubank S.** A National Scale Microsimulation of Disease Outbreaks. RTI International. Research Triangle Park, NC. Virginia Bioinformatics Institute. Blacksburg, VA, 2007.

10. **Epstein J.M., Axtell R.L.** Growing Artificial Societies: Social Science from the Bottom Up Ch. V., MIT Press, 1996.

11. **Epstein J.M.** Remarks on the foundations of agent-based generative social science. Handbook on Computational Economics, Vol. II/ K. Judd and L. Tesfatsion, eds. North Holland Press, 2005.

12. **Epstein J.M.** Modelling to contain pandemics // Nature 460, 687, 2009. 6 August.

13. **Gardner M.** Mathematical Games // Scientific American. 1970. October. P. 120–123.

14. **John von Neumann** Theory of Self-Reproducing Automata. University of Illinois Press. Urbana, 1966.

15. **Perumalla K. and Aaby B.** Data Parallel Execution Challenges and Runtime Performance of Agent Simulations on GPUs. // Proc. of Spring Computer Simulation Conf. Ottawa. Canada, 2008. April.

16. **Bisset K.R., Jiangzhuo Chen, Xizhou Feng, Anil Kumar V.S., Marathe M.V.** EpiFast: A Fast Algorithm for Large Scale Realistic Epidemic Simulations on Distributed Memory Systems ICS'09. Yorktown Heights. New York. USA. 2009. June 8–12.

17. **Makarov V.L., Zhitkov V.A., Bakhtizin A.R.** Moscow Traffic Jam Is Under Attack of an Intelligent Agent-based Model // Proc. of Conf. (edited by Bruce Edmonds and Nigel Gilbert). The 6th Conf. of the European Social Simulation Association. University of Surrey. Guildford. United Kingdom. 2009. 14–18 September.

18. **Parker J.** A Flexible, Large-Scale, Distributed Agent Based Epidemic Model. Center on Social and Economic Dynamics, Working Paper. 2007. No. 52.

19. **Tesfatsion L.** Agent-Based Computational Economics: Modelling Economies as Complex Adaptive Systems. URL: http://www.econ.iastate.edu/tesfatsi.

20. **Lynar T.M., Herbert R.D., Chivers W.J.** Implementing an agent based auction model on a cluster of reused workstations // International Journal of Computer Applications in Technology. 2009. Vol. 34. Issue 4.

21. **Ulam S.** Sets, Numbers and Universes. Cambridge. Massachusetts, 1974.

УДК 004.414+004.415+004.416

Бульонков М.А., канд. физ.-мат. наук, зав. лаб., e-mail: mike@iis.nsk.su
Емельянов П.Г., канд. физ.-мат. наук., стар. науч. сотр.
Институт систем информатики СО РАН

Об опыте реинженерии программных систем

Реинженерия программных систем – важная область информатики, использующая широкий спектр научных методов и технологий, привлекающая в последнее время большое внимание исследователей, инженеров и бизнес-сообщества. В частности, реинжиниринговый проект для программной системы, предназначенной для решения некоторой задачи, может выявить новое знание о характеристиках задачи. В статье приводится общее описание реинженерии, областей ее применения и обсуждаются проблемы, возникающие при автоматизации этой деятельности.

Ключевые слова: *программная реинженерия, долгоживущие приложения, автоматизация реинженерии, извлечение бизнес-логики, извлечение знаний*

Введение

Рассмотрим существующие понятия. В работе [20] представлены следующие определения *программной инженерии*.

• Формулирование и использование четких инженерных принципов (методов) для получения экономически эффективным способом программного обеспечения, которое является надежным и работает на реальных машинах [6].

• Такая форма инженерии, которая применяет принципы компьютерных наук (*computer science*) и математики для достижения экономически эффективных решений задач, имеющих отношение к программному обеспечению [9].

• Применение системного, четко регламентированного и количественно измеряемого подхода к разработке, внедрению, использованию и поддержке программного обеспечения [12].

В этих определениях четко прослеживается мысль (с акцентом на разные детали) о движении от идей и представлений человека (заказчика) о необходимых свойствах программной системы, соответствующей некоторой его деятельности, к их воплощению в программном обеспечении (грубо говоря, в текстах программ). Таким образом, обратный процесс – это получение из текстов программ оригинальных (изначальных) представлений человека о некоторой его деятельности, выраженных формальным образом. Эта мысль и была зафиксирована в широко распространенном определении [8]: *обратная инженерия* – процесс анализа системы для идентификации ее компонентов и их взаимодействий, а также создание представлений системы в другой форме на более высоких уровнях абстракции. Отметим, что эта задача, вообще говоря, не является тривиальной, как может показаться с первого взгляда.

Используемый иногда в России в качестве перевода термина "*reengineering*" термин "перепроектирование" является неполным: лингвистически он соответствует этому английскому слову, но деятельность "в обратном направлении" оказывается значительно шире. Можно указать, по крайней мере, следующее. Перепроектирование предполагает, что известны первоначальные проектные решения. Это предположение, зачастую, неверно для реинжиниринговых проектов, в которых первоначальные решения (точнее, те, что оформились в результате разработки и зафиксированы в программных текстах; в частности, разные части изучаемой программной системы вообще могли не быть объединены изначально каким-либо общим проектом) сначала нужно определить. Отмеченное обстоятельство свидетельствует о том, что результатом реинжинирингового проекта потенциально может быть выявление нового знания (или хотя бы экспликация "несущественных мелких деталей" и "очевидных профессионалам условий применимости") о задаче, для решения которой создавалась про-

граммная система. Еще одной областью демонстрации рассматриваемых различий является реинженерия бинарных и обфускированных программ. Однако обсуждение вопросов этой обширной области деятельности лежит вне рамок данной статьи.

Потребность в реинженерии программных систем возникает в следующих случаях.

• Желание формализовать *бизнес-процессы* (извлечение и описание их логики) в организации, выраженные в программном коде системы. Иногда это может быть единственный документ, содержащий данную информацию.

• *Инспектирование/аудит* программной системы для принятия разного рода решений (дефекты системы, комплексная оценка надежности, стоимость перепроектирования и другие).

• *Перепроектирование* и *перенос* системы на новые платформы, в новые операционные среды, на новые языки программирования. Как пример – смена парадигмы программирования, в которой реализована система.

Следует отметить, что указанные случаи похожи друг на друга и могут быть объединены, так как все они, рассматриваемые в целом, состоят из двух частей: (полу) автоматический сбор семантической информации о программной системе и использование для разных целей извлеченной в процессе ее анализа информации.

Несмотря на то что некоторые современные приложения требуют реинженерии сразу после их написания, все-таки более распространенным объектом для этого являются долгоживущие программные системы. На одной из встреч, например, в качестве перевода на русский язык словосочетания "*legacy application*" было предложено "*ветхозаветное приложение*". Далее будем использовать это выражение именно в таком его понимании. Отметим, что такая ветхозаветность привносит некоторые особенности в работу эксперта. В частности, для программистов-"неофитов", воспитанных в новых математических и технологических парадигмах, методы организации программных систем, данных и управления могут оказаться непривычными, что в случае участия в реинжиниринговых проектах потребует либо серьезной профессиональной переподготовки, либо автоматизации процесса реинженерии, способной в значительной мере экранировать ветхозаветную специфику.

Следует отметить, что значительная часть программных систем, используемых в России, создана с нуля или куплена за рубежом. В последнем случае это также довольно современное программное обеспечение. В этом смысле интересно рассмотреть оценки и предсказания Gartner Inc, сделанные в 2000 [11]. Они особенно интересны, так как не все сбылись спустя 10 лет, см. таблицу.

Можно ли рассматривать этот факт как преимущество России в ИТ-сфере? Предполагая, что используемое в настоящее время российское программное обеспечение разрабатывалось качественно, с использованием основных положений программной инженерии – да. Однако практика показывает, что это далеко не так. В этой связи изучать опыт и применять методы программной реинженерии можно и нужно. И не только поэтому.

• Во-первых, самые современные программные системы требуют внимания и, в связи с ускорением темпов развития и преобразования жизни общества, оперативной адаптации к изменяющимся условиям. Несмотря на то что этап сопровождения и развития эксплуатируемого программного обеспечения обычно рассматривается в рамках прямого процесса, т.е. в соответствии с положениями программной инженерии, это не всегда оказывается возможным.

• Во-вторых, в целом в мире потребность в модификации унаследованных программных средств, в том числе больших по объему кода, велика. Российские исследователи и инженеры-программисты имеют успешный опыт участия в выполнении такого рода проектов (см., например, результаты, представленные в сборниках [1, 2]).

Уровни зрелости языков программирования

Характеристики	**Уровень зрелости (примеры)**					
	Детский (XML, EJB)	**Подростковый** (DHTML, Java)	**Взрослый** (COBOL, C++)	**Зрелый** (Smalltalk, RPG, C)	**Пожилой** (FORTRAN. Ada)	**Престарелый** (PL/1)
Краткосрочная перспектива	Исследовать	Поощрять использование	Оценивать убытки от архитектурной стабильности	Оценивать убытки от архитектурной стабильности	Активно предотвращать использование	Запретить расширение
Долгосрочная перспектива	Активно предотвращать использование	Должны приносить прибыль и архитектурную стабильность	Поощрять использование модельно управляемых процессов	Мигрировать к более общим формам, увеличивать использование инструментальных средств	Избегать нового использования	Заменить обязательно
Ожидаемое время перехода на следующий уровень	1 год	2 года	10 лет	5 лет	5 лет	Конец жизни

• В-третьих, реинженерии требуют не только "ветхозаветные" приложения и не только в целях их сопровождения. У человечества уже заканчиваются приставки для описания пиковой производительности компьютеров, а коэффициент эффективно используемых вычислительных мощностей неуклонно снижается. В этих условиях актуальной является реинженерия приложений в целях извлечения скрытых в них внутренних резервов параллелизма.

• В-четвертых, реинженерию можно рассматривать как общий подход к извлечению, структурированию и обработке информации и свойств разного рода сложных систем (см. ниже общее описание процесса реинженерии). В частности, в этой области лежит деятельность, которая в англоязычной литературе носит название Data/Knowledge Mining. Можно упомянуть медиаобработку текстовых источников и анализ биологических систем. Расшифровку генома, например, можно трактовать как реинженерию – извлечение структурной и семантической (поведенческой) информации из системы с полностью утраченной документацией.

"Все полезные программы уже написаны"

Эта широко известная в кругах сообщества программной реинженерии шутка на самом деле важна для понимания места реинженерии. Приведем несколько конкретных цифр, характеризующих состояние дел. Отметим, что в качестве иллюстративного материала используется язык COBOL. Основная причина – это самый представительный пример в данной области. В меньшей степени то же самое относится и к другим «великовозрастным» в отмеченном ранее смысле языкам программирования, таким как PL/1 или Natural, языкам управления заданиями, например, JCL или ECL, языкам описаний пользовательских интерфейсов, таким как AS/400 Screens или CISC BMS.

На начало нового тысячелетия [21] ситуация выглядит следующим образом:

• в мире имеется 200 млрд строк на COBOL;

• к концу первого десятилетия будет написано еще не менее 5 млрд строк;

• 5 трлн долл. инвестировано в бизнес-системы, написанные на COBOL;

• их объем составляет более 60 % компьютерного кода в мире;

• 2,4 млн программистов на COBOL поддерживают и разрабатывают около 10 млн приложений на этом языке. Отметим, что в 1997 г. в США соотношение приложений 12 млн. на COBOL против 375 тыс. на C/C++ [13, 14].

Что это за код в прикладном плане? Это успешные бизнес-приложения, участвующие, в частности, в управлении огромными денежными потоками. Один пример: объем CICS/COBOL-транзакций, таких как ATM-транзакции, возрос с 20 млрд в день в 1998 г. до 30 млрд в день в 2002 г. (предсказание было дано в работе [4]).

Другой пример, характеризующий заинтересованность бизнес-сообщества в качественном программном обеспечении из этой области: от 300 до 600 млрд долл. было инвестировано для решения **Проблемы 2000 года** – Y2K [5]. Нужно сказать, что Проблема Y2K оказала огромное влияние на развитие реинженерии программного обеспечения. Хотя вопрос реинженерии привлек к себе пристальное внимание в 1980-е гг., именно накануне нового тысячелетия в связи с Проблемой Y2K был проведен широкий и глубокий анализ, в том числе статистический, состояния "ветхозаветных" приложений по всему миру. Ни до этого, ни после столь массовых исследований в данной области не проводилось.

Оказавшись перед лицом возможных существенных финансовых и репутационных потерь, частные и государственные организации на Западе были вынуждены в массовом порядке правильно расставлять знаки препинания: *переписывать не нужно сопровождать*.

Предположим, мы захотели переписать этот код. Вот типичные параметры систем, возникающих в задачах реинженерии:

• десятки различных типов исходных файлов;

• сотни и тысячи исходных файлов (до нескольких десятков тысяч);

• отдельно взятый файл может достигать 50 000 строк;

• до 50 % кода могут составлять определения данных (это свойство "ветхозаветных" приложений действительно представляет определенные трудности для "молодых" программистов, работающих с современными языками и технологиями, в которых данные локализуются и/или структурируются по-другому).

Специалисты оценивают среднестатистическое приложение на COBOL в 1 млн строк кода.

Инженер-программист, имеющий специальную подготовку в анализе такого словообильного языка как COBOL, может проанализировать ("вручную", с соблюдением соответствующих регламентов [15, 24]):

• 250...500 строк кода в час для написания нового кода;

• 1 000...1 500 строк кода в час только для "понимания".

Перечисленные факты в общем случае не указывают на то, что в день производительность инженера-программиста составит тысячи строк. Рекомендуемое время непрерывного инспектирования – 2 ч, после чего качество начинает неуклонно падать. Если оценивать это в терминах Function Points (см., например, [15]; широко используемая методология для оценки сложности программных систем), то один человек может поддерживать и развивать в небольших пределах приложения объемом приблизительно 500 Functional Points. Один Functional Point соответствует приблизительно 100 LOC (Lines Of Code) на COBOL, 20 LOC для объектно-ориентированных языков программирования или автоматических генераторов и более 300 LOC – на ассемблере.

При некоторых предположениях для того, чтобы переписать весь существующий код на COBOL, потребуется более 50 лет. Переписывание "ветхозаветных"

приложений, включая деятельность по отладке и тестированию, оценивается в 25 долл. за строку кода [22]. Если работает действительно хороший специалист, по словам тех же авторов, этот показатель можно довести до 18–20 долл. Проблема усугубляется еще и тем обстоятельством, что в 2010 г. более половины COBOL-программистов – это люди предпенсионного и пенсионного возраста [7, 17].

Таким образом, дилемма – переписывать или с пониманием сопровождать – не имеет однозначного решения. К сожалению, авторы не могут привести точных статистических данных, но с определенной долей уверенности могут утверждать, что подавляющая часть COBOL-приложений и аналогичных, которые можно было без значительных экономических затрат и бизнес-рисков перенести на новые языки и платформы, уже перенесены. Оставшиеся приложения (их объем оценивается в 150...180 млрд строк), представляют собой сложные приложения с большой долей бизнес-знаний. В обозримом будущем они не исчезнут и будут требовать сопровождения. Косвенным подтверждением дальнейшего существования многочисленных COBOL-приложений является тот факт, что последний ISO стандарт COBOL утвержден в 2002 г. (предыдущие ANSI стандарты были утверждены в 1968, 1974 и 1985 гг.), а в 2005 г. началась работа над новым. Исходя из изложенного выше, следует, что без серьезной инструментальной поддержки реинжиниринговая деятельность не может быть эффективной.

Общая характеристика процесса программной реинженерии

Ниже приведена, по сути, развернутая версия определения реинженерии программных систем [8], уточняющая некоторые детали.

- Загрузка анализируемой программной системы (она может быть частично недоступной по причине либо ветхозаветности и, как следствие, наличия только двоичных исполняемых файлов, либо соображений секретности). Соответственно, средства "домысливания" поведения этой части системы должны быть предоставлены и работать корректно.
- Синтаксический разбор компонентов системы для представления их в формализованном и унифицированном виде. Можно сказать, что программный текст – это уже формализованный вид, однако, с одной стороны, он не очень удобен для массовой обработки по причине неэффективности, а с другой стороны, содержит детали, иррелевантные для понимания системы. Модельная унификация (насколько это возможно) внутренних представлений программных компонентов разных типов важна для минимизации числа различных инструментальных средств анализа.
- Идентификация интересующих объектов. Действительно, в программах есть много интересного. Таким образом, появляется необходимость выбора: насколько подробно должна быть представлена информация о системе, всегда ли нужна подробная информация, всегда ли интересны всевозможные аспекты поведения системы и ряд других.
- Первичное атрибутирование выбранных объектов и выявление отношений между ними. Для этого пункта также актуальна проблема выбора, указанная в предыдущем.
- Сбор и анализ семантической информации, изучение различных аспектов поведения системы. Отличие предыдущего пункта от настоящего: в предыдущем речь идет о статических характеристиках системы, достаточно просто собираемых на этапе трансляции или не очень сложной постобработкой, в настоящем – о динамических характеристиках, которые, однако, могут извлекаться и статическими средствами. Сбор и анализ могут проводиться как автоматическими, так и полуавтоматическими средствами.
- Формализация выявленной семантической информации, например, в форме стандартизованных описаний бизнес-процессов.
- Перепроектирование и перенос системы на новые платформы, в новые операционные среды, на новые языки программирования.

Далее рассмотрены некоторые аспекты построения систем реинженерии.

Репозиторий

Весьма важными этапами реинженерии программ (разных видов) являются разработка архитектуры, наполнение, поддержание целостности и предоставление доступа к собственно хранилищу накопленной информации – *репозиторию* системы. Репозиторий реализует концепцию *сущности–отношения*, широко известную в теории баз данных. При этом со стороны прикладного интерфейса – это объектно-ориентированная модель данных, а со стороны реализации – реляционная СУБД.

Важность понятия репозитория объясняется следующим. Тщательность проработки его архитектуры определяет как полноту представления информации о предмете исследования, так и "тонкость" структурных связей между элементами архитектуры, что в конечном итоге определяет общую информативность анализа. Однако в этой ситуации важно найти осмысленный компромисс между информативностью и объемом репозитория, так как излишняя детализация может привести к росту объемов хранимой информации, сложности ее извлечения и манипулирования ею. Целостность данных обеспечивает их актуальность и структурную корректность. Репозиторий должен поддерживать "фантомные" объекты, т.е. объекты, возникающие в процессе построения структурных отношений, когда один конец отношения уже определен, а другой еще нет. И наконец, программная система, предоставляющая доступ к репозиторию, должна обеспечивать разграничение прав доступа к данным, простой и эффективный удаленный доступ к системе, контроль целостности данных, в том числе и при параллельном доступе к ним, а также скорость работы системы.

Несомненно, все этапы, в том числе и предварительные (например, разработка архитектуры репозитория), должны быть поддержаны соответствующим технологическим инструментарием.

Семантический анализ

Ключевую роль в процессе реинженерии играет *анализ программ*. Многие задачи могут быть решены на основе анализа текста программы или ее синтаксической структуры. Типичным примером таких задач является поиск по образцу, который может быть задан как в виде регулярного выражения, так и в виде шаблона синтаксической структуры. Более того, образец может быть "нагружен" дополнительными условиями. При задании такого образца могут использоваться выявленные на предыдущих стадиях анализа отношения между объектами, например, отношения между определением и использованием переменной или функции.

Однако более глубокий анализ требует изучения динамических свойств программ и, в первую очередь, потоков управления и данных. В зависимости от решаемой задачи необходимая точность анализа может варьироваться: для одних задач (например, для определения глоссария программы) достаточно контекстно-нечувствительного анализа, а для других (например, для извлечения бизнес-логики) необходим глобальный контекстно-чувствительный анализ. Масштабируемость алгоритмов анализа чрезвычайно важна для их использования в реинжиниринговых инструментах.

Опыт показывает, что создание полностью автоматизированных, т.е. не требующих вмешательства человека, инструментальных средств анализа нецелесообразно (по крайней мере, на текущем этапе развития вычислительной техники и алгоритмики, в целях применения этих инструментов к реальным программным системам). Учитывая интерактивный, направляемый пользователем характер реинженерии, необходимым условием является время реакции системы. Обеспечить его выполнение можно было бы, заранее выполнив анализ и собрав всю необходимую информацию. Однако сложность процесса анализа и объем информации не позволяют воспользоваться столь прямолинейной стратегией. Действительно, на основе каких-либо других соображений пользователь может выделить в системе лишь относительно небольшое число мест. При этом окажется, что вся остальная информация была собрана впустую. По этой причине здесь также необходимо искать компромиссные подходы. Например, заранее анализ потоков данных выполнять контекстно-нечувствительно, а собранную информацию использовать лишь для "поверхностного" взгляда на систему и как приближение к более детальному анализу, который проводится "точечно".

Средства визуализации свойств программ

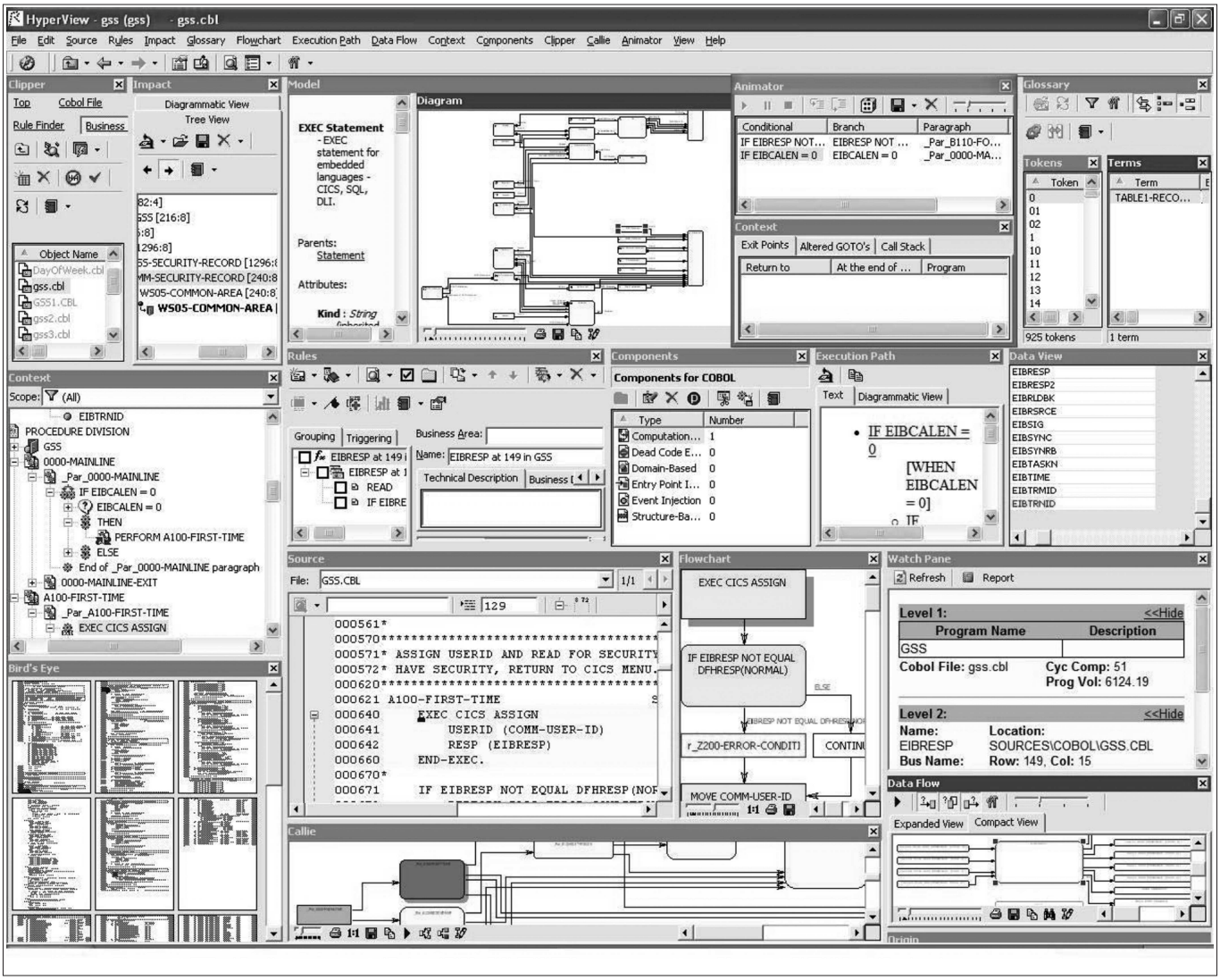
Система реинженерии должна представлять эксперту анализируемую систему в наиболее подходящем виде. Не принимаем при этом во внимание тот факт, что многие элементы системы могут не иметь текстового представления, поскольку доступны только в виде откомпилированных файлов, либо вообще появились в результате работы администратора и хранятся где-то в настройках системы. Ограничимся пока рассмотрением случая, когда доступен исходный код: либо программы, либо командного файла, либо описания структуры файлов и базы данных и т.п.

Наиболее распространенным и часто используемым является именно текстовое представление – возможно, раскрашенное в соответствии с лексикой языка. Однако оно не является наиболее удачным для большого количества другой информации, собранной для реинженерии. Так, для отображения синтаксической структуры программы наиболее уместным будет древовидное ее представление с соответствующим набором манипуляций: схлопывание/распахивание вершин дерева и подобные им.

Сложнее дело обстоит с отображением информации об управляющей и информационной структуре программы. На первый взгляд, наиболее адекватным может показаться графовое представление. Однако здесь опять возникают вопросы эффективности: размеры реальных графов достигают нескольких тысяч вершин и даже самые удачные алгоритмы размещения приводят к тому, что изображение напоминает абстрактную живопись – большое число пересекающихся линий, местами сливающихся в пятна. В лучшем случае на основе такого изображения эксперт сможет судить об общей структуре программы.

Традиционно упомянутая выше задача решается путем использования иерархических графов и их инкрементального построения. Иерархические графы хорошо применимы для отображения управления в структурированных языках программирования, к которым, к сожалению, такие языки, как COBOL, не относятся. Что касается потоков данных, то выявить в них иерархическую структуру можно очень редко, и она неочевидна для эксперта. Инкрементальное построение графа, на первый взгляд, решает эту задачу: эксперт всегда видит только ту окрестность начальной точки исследования, которую он сам обозначил. Однако при таком подходе скрывается степень полноты отображения, а именно неясно, какая часть уже исследована, а какую еще предстоит исследовать. Более того, по мере расширения графа его структура может существенно меняться, и он опять же может стать слишком большим для восприятия.

Таким образом, отображение информации в виде графов хотя и выглядит наиболее "презентабельным", далеко не всегда оказывается полезным для решаемой экспертом задачи. Исходя из изложенного выше, следует большее внимание уделить не проблеме размеще-



Представление программной системы

ния графов, а пониманию тех задач, которые решает эксперт на основе графовой информации. Примерами таких задач могут быть разбиение графа на кластеры, нахождение недостижимых компонентов, нахождение возможных путей из одной точки в другую и подобные им. Для каждой из таких задач может быть создан специальный компонент, который предоставляет адекватные средства управления/навигации, (необязательно графовые) средства отображения информации и средства взаимодействия с другими частями реинжиниринговой системы.

Отдельные компоненты могут визуализировать не только исходную информацию, но и различного рода аннотации и связи, выявленные экспертом в ходе работы, а также выходные артефакты, порожденные автоматически. Естественно, что вся информация такого рода должна быть привязана к исходным текстам программ.

Резюмируя представленные выше соображения, приходим к архитектуре системы визуализации, поддерживающей совокупность "точек зрения" на анализируемую систему. Каждая такая точка зрения отображает некоторый аспект системы или вид деятельности эксперта. Система должна обеспечивать взаимодействие и синхронизацию этих точек зрения. В зависимости от решаемой в конкретный момент времени задачи, эксперт подбирает себе нужный набор инструментальных средств. На рисунке показан момент работы некоторой системы реинженерии, когда открыто максимальное число разных способов визуализации информации – "взглядов" – на анализируемую систему.

Разумеется, в реальной ситуации эксперт откроет лишь несколько существенных для данного анализа представлений программной системы. Этот пример, в частности, демонстрирует важность синхронизированной реакции разных компонентов. Время реакции при этом должно обеспечивать комфортную и производительную работу эксперта.

Извлечение бизнес-логики

Понятие *бизнес-правила* возникло в 1980-х гг. прошлого века. С тех пор было предложено несколько версий определения данного понятия, а также предложено несколько способов задания и хранения бизнес-правил, разработаны стандарты. Эти работы велись несколькими различными группами и сообществами (такими, как Business Rules Group и Business Rules Community). В процессе изучения данного понятия бизнес-правилом называли [3]:

– утверждение, которое определяет или ограничивает некий аспект бизнеса;

– директиву, предназначенную для управления поведением бизнеса;

– атомарную часть переиспользуемой бизнес-логики;

– набор условий, которые управляют деловым событием, чтобы оно происходило так, как нужно для предприятия.

В процессе развития подхода к разработке систем, основанного на бизнес-правилах, создавались различные модели бизнес-правил. Итогом многолетней работы стал утвержденный в начале 2008 г. организацией Object Management Group стандарт "Semantics of Business Vocabulary and Business Rules" [19]. В стандарте особо отмечено, что он предназначен в первую очередь для бизнес-аналитиков, а уже потом для реализации в программных системах.

Прямой процесс, включающий построение бизнес-правил и их дальнейшее использование в разработке приложения, привлек широкое внимание исследователей более 20 лет назад, был хорошо методологически проработан [10, 18] и реализован во многих коммерческих проектах. Однако технологии, реализующие обратный процесс, т.е. извлечение бизнес-правил из существующего программного кода, развиты значительно меньше. Процесс восстановления бизнес-логики программных систем, с учетом размера этих систем, непосилен для человека и нуждается в автоматизации. Опыт показывает, что полная автоматизация невозможна как ввиду возникающих алгоритмических проблем, так и потому, что требует непосредственного участия бизнес-аналитика, владеющего знанием предметной области. Тем не менее, инструментальная поддержка процесса анализа чрезвычайно важна.

Как отмечалось ранее, понимание бизнес-логики программы необходимо для ее эффективного использования и сопровождения. Однако для множества старых приложений одним из основных источников для получения бизнес-логики, а иногда и единственным, становится исходный код. Знания предметных специалистов также необходимы, однако они полезнее в ответе на вопрос о том, как система должна работать, а не о том, как она действительно работает. Вполне возможно, что не все изменения, произошедшие в бизнесе, были корректно отражены в приложении. Таким образом, возникает задача восстановления бизнес-логики, для решения которой необходимо проанализировать код и отделить требования предметной области от особенностей реализации. Первые должны проверяться и изменяться бизнес-специалистами, вторые редактируются программистами.

Генерация по шаблону позволяет на основе набора программных конструкций, выбранных каким-то образом (вручную или некоторыми автоматическими средствами), создавать бизнес-правила с заданным шаблоном имени и набором атрибутов. Данный способ можно отнести к *слабой автоматизации*, поскольку анализ кода – поиск необходимых конструкций – проводится пользователем с помощью других компонентов системы, автоматизация касается лишь непосредственного создания бизнес-правил из каждой заданной конструкции.

Два других способа предоставляют *сильную автоматизацию*: проводится автоматический анализ специального информационного графа, который строится системой на основе анализа потоков данных, а также содержит дополнительную информацию об условных зависимостях, полученную на основе анализа потоков управления. Автоматическое восстановление вычислений строит набор бизнес-правил, последовательное выполнение которых (с учетом условий их выполнения) вычисляет заданную переменную.

Методы сильной автоматизации для построения бизнес-правил близки к *нарезке слайсов* [16, 23]. Однако слайс представляет собой исполняемую программу и содержит много вспомогательной информации, неинтересной аналитику. Это, что естественно, машинно-ориентированная запись последовательностей мелких инструкций, демонстрирующих частный взгляд – один из многих – на вполне конкретный и довольно однозначный бизнес-процесс. В случае межпрограммных вызовов могут быть получены маленькие слайсы, показывающие лишь переход в другую программу. Представление в виде бизнес-правил более удобно для демонстрации целостной картины вычислений, соответствующих интересующему бизнес-аналитика процессу.

И, наконец, важным показателем качества реинжинирингового проекта является характеристика покрытия слайсами и/или бизнес-правилами кода программной системы. Удобные средства проверки этой характеристики – неотъемлемая часть хорошей системы реинженерии.

Заключение

Исследование методологических принципов и инструментария, обучение реинженерии являются чрезвычайно важными задачами. Даже, если предположить, что проблема "ветхозаветного программного

обеспечения" не очень актуальна для России (у нас нет своего, а доступ на международный рынок реинженерии не очень прост), то применение реинжиниринговых подходов к распараллеливанию программ и анализу сложных (непрограммных) систем с лихвой эту важность подтверждают. Авторы полагают, что внедрение методов реинженерии в практику Data/Knowledge Mining окажется очень плодотворным и взаимно полезным для этих сфер научно-технической деятельности.

СПИСОК ЛИТЕРАТУРЫ

1. **Автоматизированный** реинжиниринг программ / Сб. статей под ред. А.Н. Терехова и А.А. Терехова. СПб.: Изд-во СПбГУ, 2000.

2. **Системное** программирование / Сб. статей под ред. А.Н. Терехова и Д.Ю. Булычева. Вып. 2004 г., Вып. 1 (2005), Вып. 2 (2006). СПб.: Изд-во СПбГУ, 2004–2006.

3. **A Brief History** of the Business Rule Approach, 2nd ed. // Business Rules Journal. 2006. Vol. 7. No. 11.

4. **Ulrich W.M.** Remember Cobol? If You Dont, Get Reacquainted // Computerworld. 2001. MAY. URL:http://www.computerworld.com/s/article/print/60683/Remember_Cobol_If_You_Don_t_Get_Reacquainted.

5. **Arranga E., Price W.** Fresh from Y2K: What's Next for COBOL? // IEEE Software. 2000. Vol. 17. No. 2. P. 16–20.

6. **Bauer F.L.** Software Engineering // Information Processing. 1972. Vol. 71. P. 530–538.

7. **Carr D., Kizior R.** Continued Relevance of COBOL in Business and Academia: Current Situation and Comparison to the Year 2000 Study // Proc. of 20th Annual Conf. on Information Systems Education, ISECON'2003. 2003.

8. **Chikofsky E.J., Cross J.H.II.** Reverse Engineering and Design Recovery: A Taxonomy // IEEE Software. 1990. Vol. 7. No. 1. P. 13–17.

9. **CMU/SEI-99-TR-032** Guidelines for Software Engineering Education, Version 1.0. By Bagert D. et al. Software Engineering Institute, Carnegie Mellon University, 1999.

10. **Date C.J.** What Not How: The Business Rules Approach to Application Development. Boston, MA: Addison-Wesley, 2000.

11. **Quiet Storm**: Demand for COBOL Skills Remains High. Gartner's Application Development Research Note TG-09-9946. 2000. February 25.

12. **IEEE STD** 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology. IEEE Computer Society, 1990.

13. **Jones C.** The Global Economic Impact of the Year 2000 Software Problem. Version 5.2. Report of Software Productivity Research, Burlington, MA. 1997. January 23.

14. **Jones C.** The Year 2000 Software Problem – Quantifying the Costs and Assessing the Consequences. Reading, MA: Addison Wesley, 1998.

15. **Jones C.** Estimating Software Costs. New York: McGraw Hill, 2007.

16. **Krinke J.** Program Slicing / In Handbook of Software Engineering and Knowledge Engineering, Vol. 3: Recent Advances. World Scientific Publishing, 2005.

17. **Mitchell R.L.** Cobol: Not Dead Yet // Computerworld, 2006. October 4.

18. **Ross R.G.** Principles of the Business Rule Approach. Boston, MA: Addison-Wesley, 2003.

19. **Semantics** of Business Vocabulary and Business Rules. Object Management Group Recommendations, 2008.

20. **Software** Engineering 2004. Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering. IEEE Computer Society and ACM. 2004.

21. **Stern N., Stern R.** Structured COBOL Programming, 9th ed. New York: John Wiley & Sons, 2002.

22. **Ulrich W.M., Newcomb Ph.H.** Information Systems Transformation. Architecture-Driven Modernization Case Studies. Amsterdam-Tokyo: Morgan Kaufmann OMG Press, 2010.

23. **Weiser M.** Program slicing // IEEE Transactions on Software Engineering, 1984. Vol. 10. Iss. 4. P. 352–357.

24. **Wiegers K.E.** Peer Reviews in Software: A Practical Guide. Reading, MA: Addison-Wesley, 2002.

УДК 004.45

Н.О. Вегерина, инженер-программист, ООО "Инфострой", **А.В. Липанов,** канд. техн. наук, доц., директор ООО "Инфострой", Харьковский национальный университет радиоэлектроники, e-mail: alex@infostroy.com.ua

Экспертная система анализа качества исходного кода программного обеспечения

Рассматривается визуальная система оценки качества исходного кода с помощью подсчета метрик и выдачи текстовых рекомендаций. Современные средства предоставляют возможности для подсчета метрик по исходному коду, однако выдаваемые ими значения являются ненормированными и без указания пределов измерения. Средств, интерпретирующих числовые значения метрик в словесные рекомендации, на сегодняшний день не существует. В рассматриваемой системе предусматриваются возможности загрузки пользователем модуля для анализа, проводимого с помощью функциональности программы Reflector, подсчета метрик, выдачи на основе их значения текстовых рекомендаций с помощью экспертной системы и генерации отчета. Систему можно использовать для анализа исходного кода и получения текстовых рекомендаций по его модификации.

Ключевые слова: *объектно-ориентированные метрики, связность, сцепление, нестабильность, исходный код, программное обеспечение*

1. Современные средства проектирования и анализа исходного кода программного обеспечения

Проблема разработки надежного, масштабируемого, безопасного и работающего с высокой скоростью программного продукта является актуальной, несмотря на огромный прогресс, происходящий в сфере разработки программного обеспечения.

Новые методологии разработки, такие как экстремальное программирование или *Scrum*, позволяют сократить время разработки, а наличие новых платформ и абстрагирование от нижних уровней позволяют избегать многих ошибок. Тем не менее, контроль качества должен осуществляться на самых различных уровнях – начиная с методологического и заканчивая технологическим уровнем, когда процессы контроля качества протекают в автоматическом режиме, например при автоматических сборках проекта.

Если говорить о начальном этапе создания программного продукта – проектировании и разработке его архитектуры, то это наиболее важный момент всего процесса разработки приложения, так как изначально неправильно спланированная архитектура может привести к большому числу ошибок, обнаруживаемых на стадии тестирования, а также ненадежной и небезопасной работе всего приложения. Для решения такой проблемы были созданы средства проектирования, такие как *Rational Software Architect, Borland Together, ArgoUML, PowerDesigner* и ряд других. Все эти продукты предоставляют широкий спектр функций, необходимых для проектирования, среди которых использование языка UML и автоматическая генерация исходного кода для представления архитектуры в наглядном виде, преобразования ее в готовый шаблонный код без учета ошибок, которые могут привести ко многим часам доработки и исправления на стадии тестирования, если в процессе создания приложения не использовать вышеперечисленные средства [3].

Системы проектирования программного обеспечения (ПО) эффективны лишь на первом шаге написания приложения. Если учесть, насколько может измениться структура приложения в процессе разработки (не в по-

следнюю очередь благодаря изменению требований) либо в процессе поддержки, когда отладкой ошибок занимается команда разработчиков, не имевших отношение к написанию данного продукта, которым, однако, необходимо уменьшить число уязвимостей в программном продукте, то средства проектирования становятся абсолютно бесполезными, и возникает необходимость в оценке состояния качества готового исходного кода.

Том ДеМарко, разработавший идею метрик исходного кода, сказал: "Мы не можем управлять тем, что мы не можем измерить". Благодаря ему, оценка кода свелась к получению набора значений некоторых метрик, описывающих данный код [1]. На данный момент это единственный метод оценки исходного кода, который подразумевает анализ внутренней объектной структуры исходного кода, отражающей сложность каждой отдельно взятой сущности, такой, как метод или класс, и внешней структуры, отражающей сложность взаимодействия сущностей между собой (например, связность или наследование) [2]. Наибольший эффект дает применение таких метрик при анализе больших программных систем, когда ручной анализ и просмотр исходного кода могут занимать значительное время.

Метрики программного кода являются важным инструментом и уже сегодня используются многими производителями программного обеспечения. Так, при сертификации на более высокие уровни по моделям ISO/IEC или CMM/CMMI использование метрик кода является обязательным, что позволяет в определенной степени достичь контролируемости процесса разработки [2].

Следует отметить средства для анализа исходного кода, которых на сегодняшний день существует сравнительно небольшое число. Они позволяют подсчитывать объектно-ориентированные метрики по исходному коду. Самыми известными программными продуктами, предоставляющими возможность подсчета наиболее оптимальных метрик, являются *NDepend*, *OxyProject Metrics*, *Resource Standard Metrics*, *Visual Studio 2010*, *CodeMaid*, *CodeSnippets*, *GMetrics*, *SourceAnalyzer*, *Software Index*, *Resource Standard Metrics*. В этих продуктах реализованы функции для подсчета следующих количественных характеристик кода:

- число строк кода (общее число строк, количества физических, логических, закомментированных и т.д.);
- число различных элементов (сборок, классов, интерфейсов, методов и т.д.);
- число метрик элементов (цикломатическая сложность для методов, связность и сцепление для классов и т.д.).

Эти системы позволяют проводить анализ программ, разработанных на C, C++, C#, JAVA, VB. Также существуют системы, которые позволяют вычислять метрики по UML-диаграмме классов, и они не зависят от языка программирования. Примером такой системы является система *SDMetrics*, которая предоставляет возможность посчитать более 100 различных метрик в таких категориях, как размер, наследование, связность, сцепление (реляционная сплоченность), сложность. *SDMetrics* реализует в несколько раз больше метрик, чем другие программы и, таким образом, позволяет наиболее полно оценить архитектуру приложения и выдать необходимые рекомендации.

Приложения, которые вычисляют метрики, дают их ненормированные числовые значения без указания пределов измерения. Проще говоря, если пользователю неизвестен смысл метрики, то ее значение не представляет для него никакого интереса. Более того, числовые значения метрик сами по себе не несут никакой полезной информации. Это приводит к проблеме интерпретации значений метрик и перевода их в осмысленные утверждения о состоянии качества кода и рекомендации по его улучшению.

Среди существующих программных продуктов, нацеленных на проведение в той или иной степени оценки качества исходного кода, не существует приложений, предоставляющих возможности интерпретации метрик в целях предоставления словесных рекомендаций по улучшению исходного кода. В связи с этим, проведение исследований в направлении разработки инструмента для оценки качества исходного кода с возможностью предоставления словесных рекомендаций является актуальной задачей.

В качестве основных задач разработки продукта были определены следующие:

- нормирование выбранных метрик таким образом, чтобы их значения попадали в промежуток [0, 1] для четкого определения уровня качества исходного кода;
- для каждой метрики разбиение промежутка [0, 1] на части и постановка в соответствие каждой части некоторой рекомендации, отражающей состояние проверяемого кода и содержащей указания по его улучшению в случае необходимости;
- реализация алгоритма подсчета каждой отдельной метрики;
- проведение анализа большого числа примеров исходного кода экспертами в целях получения оценки каждого примера и рекомендаций по его улучшению, а также группировка полученных оценок по подобности примеров;
- реализация экспертной системы, выдающей рекомендации на основе каждой метрики в зависимости от ее значения;
- разработка приложения, выполняющего разбор исходного кода, подсчитывающего метрики и выдающего текстовый эквивалент их значений, а также имеющего интуитивный дружественный интерфейс.

2. Метрики, реализованные в автоматизированной системе анализа исходного кода ПО

Существует множество различных классификаций метрик ПО, трактующих метрики с различных позиций и ранжирующих одни и те же характеристики по различным критериям. Одной из таких классифика-

ций может служить разделение метрик на группы по субъектам оценки:

• размер – сравнительная оценка размеров ПО;

• сложность – оценка архитектуры и алгоритмов программной системы (отрицательные показатели этой группы метрик говорят о проблемах, с которыми можно столкнуться при развитии, поддержке и отладке программного кода);

• поддерживаемость – оценка потенциала программной системы для последующей модификации.

Для оценки и контроля качества кода могут непосредственно использоваться метрики сложности: цикломатическая сложность, связность кода, глубина наследования и др.

Разработанная система предоставляет возможность оценки состояния качества исходного кода. Данная функциональность была реализована с использованием метрического метода, для которого были отобраны три метрики – **сцепление, связность и нестабильность**.

Метрики, описанные далее в пунктах 2.1–2.3, были выбраны по следующим причинам.

1. Эти метрики могут быть достаточно легко вычислены, что позволит сделать систему достаточно производительной.

2. Эти метрики наиболее точно позволяют отразить связи между частями исходного кода.

3. Эти метрики дают наиболее полную картину о состоянии исходного кода.

Подсчет вышеуказанных метрик возвращает числовые значения, которые несут сравнительно небольшое количество информации для пользователя. Для решения данной проблемы была использована экспертная система. Таким образом, пользователь программы получает текстовую информацию о состоянии кода, а также рекомендации по его улучшению, если таковые необходимы.

2.1. Сцепление

В компьютерных науках сцепление (*coupling*) или зависимость – это мера того, насколько программный модуль зависит от каждого из остальных модулей.

Сцепление может быть низким (свободным, слабым) или высоким (плотным, сильным). Типы сцепления в порядке от самого высокого до самого низкого следующие (рис. 1) [1]:

• сцепление по содержимому (высокое) – один модуль зависит от внутренней работы другого модуля (имеет доступ к локальным данным другого модуля) или изменяет ее; изменения в вычислении данных во втором модуле повлекут за собой изменения в зависимом модуле;

• совместное сцепление – два модуля разделяют одни и те же глобальные данные; изменения в глобальных данных приведут к изменениям во всех модулях, использующих эти данные;

• внешнее сцепление – два модуля разделяют внешний установленный формат данных, протокол сообщений или интерфейс устройства;

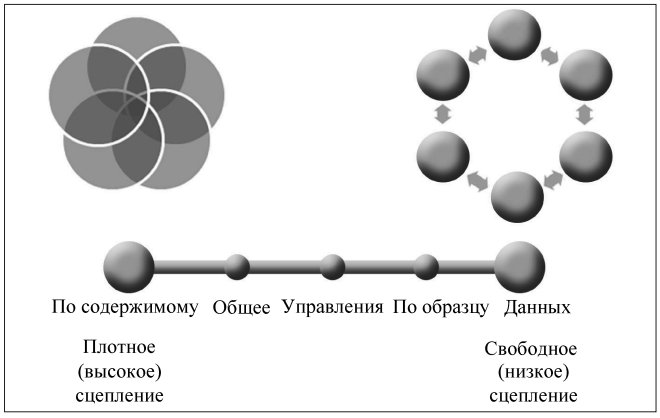


Рис. 1. Концептуальная модель сцепления

• сцепление управления – один модуль контролирует выполняющий поток другого, передавая ему информацию о том, что необходимо выполнить;

• сцепление по образцу – один модуль посылает другому составную структуру данных в то время, как тому необходима только часть этой структуры, это может привести к изменению чтения модулем структуры, так как поле, в котором модуль не нуждается, было изменено;

• сцепление данных – модули разделяют данные, например, по параметрам;

• сцепление по сообщениям – модули не зависят друг от друга, вместо этого они используют интерфейс для обмена сообщениями без параметров (или событиями);

• отсутствие сцепления – модули не зависят друг от друга.

Сцепление между классами A и B возрастает, если [4]:

– A содержит атрибут типа B;

– A вызывает метод (событие и др.) объекта B;

– A содержит метод, который ссылается на B (через возвращаемый тип или параметр).

Низкое сцепление представляет собой связь, при которой один модуль взаимодействует с другим с помощью простого и устойчивого интерфейса и не имеет необходимости знания о внутренней реализации другого модуля.

Формула для вычисления сцепления (1) имеет вид:

$$Coulping = \frac{r_t}{\sum_{i=0}^{n} r_i}, \qquad (1)$$

где r_i – число ссылок на класс (ссылка – это поле, локальная переменная, возвращаемый тип или параметр метода);

r_t – число ссылок на класс, участвующий в подсчете метрики;

n – число классов.

Чем больше модуль сцеплен, тем больше значение сцепления – в пределах приблизительно от 0,67 (низкое сцепление) до 1 (высокое сцепление).

2.2. Связность

В компьютерной науке связность (*cohesion*) – это мера того, насколько сильно связаны между собой методы модуля приложения. Связность является обычным типом измерения и чаще всего выражается как "высокая связность" или "низкая связность". Модули с высокой связностью являются более предпочтительными, так как высокая связность сопряжена с некоторыми желательными характерными чертами программного обеспечения, такими как устойчивость, надежность, повторное использование и понятность, в то время как низкая связность ассоциируется с нежелательными особенностями, такими как сложность поддержки, тестирования, повторного использования и даже сложность понимания.

Связность часто является противоположностью сцеплению – высокая связность соответствует низкому сцеплению и наоборот. Метрики контроля качества исходного кода сцепления и связности были разработаны Ларри Костантином и основаны на характеристиках "хорошей" программной практики, которая уменьшает расходы на поддержку и модификации.

Таким образом, связность уменьшается, если [4]:

– методы класса мало связаны между собой;

– внутри методов выполняется много различных действий, использующих не связанные наборы данных.

Недостатки низкой (слабой) связности [4]:

- возрастание сложности понимания модулей;
- возрастание сложности поддержки системы, так как логические изменения в одном модуле влияют на многие другие модули и, таким образом, изменения в одном модуле влекут за собой изменения в связанных модулях;
- возрастание сложности повторного использования модуля, так как большинству приложений не нужны произвольные наборы операций, предоставляемые модулем.

Типы связности в порядке возрастания от низкой к высокой следующие:

- случайная связность (низкая) – части модуля связаны произвольно либо не имеют важной связи (модули часто используемых функций);
- логическая связность – части модуля сгруппированы, так как они логически выполняют одно и то же, даже если они разные (группы методов ввода/вывода);
- временная связность – части модуля группируются во время выполнения: части модуля обрабатываются в определенное время выполнения программы (метод, вызываемый после обработки исключения, который закрывает открытый файл, создает лог ошибок и оповещает пользователя);
- процедурная связность – части модуля сгруппированы, потому что они всегда следуют определенной последовательности выполнения (метод, проверяющий наличие разрешений и затем открывающий файл);
- коммуникационная связность – части модуля сгруппированы, так как они работают с одними и теми же данными (модуль, который обрабатывает одну и ту же информационную запись);
- последовательная связность – части модуля сгруппированы, так как результат выполнения одной части является входным параметром для другой части (метод, который читает данные из файла, и метод, обрабатывающий эти данные);
- функциональная связность (высокая) – части модуля сгруппированы, так как они сотрудничают при выполнении одной хорошо определенной задачи модуля (например, разбор XML в случае *Expat*(XML)). Существует несколько методов вычисления связности *LCOM* [4]. При вычислении *LCOM* по методу, предложенному Л. Константайном, значения принадлежат промежутку [0,1]. При вычислении *LCOM* по методу Хендерсона – Селлерса (*Henderson–Sellers*) значения принадлежат промежутку [0,2]. Эту метрику обозначают *LCOM HS*. Значение *LCOM HS* большее, чем 1, свидетельствует о плохой связности. Формула (2) используется для подсчета метрики (согласно [4]):

$$LCOM = 1 - \frac{1}{M \cdot F}\sum_{i=0}^{n} MF,\ LCOM\ \ HS = \\ = \frac{1}{M-1}\left(M - \frac{1}{F}\sum_{i=0}^{n} MF\right), \qquad (2)$$

где M – число методов в классе (статических и экземплярных конструкторов, геттеров и сеттеров свойств, методов добавления и удаления событий); F – число экземплярных полей класса; MF – число методов класса, имеющих доступ к определенному экземплярному полю; $\sum_{i=0}^{n} MF$ – сумма MF по всем экземплярным полям класса.

Основная идея метрик заключается в том, что класс абсолютно связан, если все его методы используют все его поля, что означает, что $\text{sum}(MF) = M \cdot F$ и приводит к тому, что $LCOM = 0$ и $LCOM\ HS = 0$;

Классы, в которых $LCOM > 0$, число полей больше 8 и число методов больше 8, являются проблематичными. В то же время сложно избежать таких несвязных классов. Классы, где $LCOM\ HS > 1$, число полей больше 10 и число методов больше 10, должны избегаться. Вышеупомянутое ограничение является более строгим, чем первое [4].

2.3. Нестабильность

Нестабильность (*instability*) – это мера того, насколько зависят классы в пределах модуля. Однако, в отличие от сцепления, где подсчитывалась зависимость между двумя классами, нестабильность подсчитывает, насколько класс зависит от всех классов модуля в целом и насколько все классы модуля в целом зависят от данного класса. Для подсчета данной метрики проводится подсчет связанных метрик – центростре-

мительного и центробежного сцепления [5]. Центростремительное сцепление подсчитывает число классов, которые зависят от данного класса. Центробежное сцепление, наоборот, считает число классов, от которых зависит данный.

Формула для подсчета нестабильности выглядит следующим образом [5]:

$$I = \frac{C_e}{C_a + C_e}, \quad (3)$$

где C_e – центростремительное сцепление, C_a – центробежное сцепление.

$I = 0$ указывает максимально стабильный класс, $I = 1$ указывает максимально нестабильный класс. Большое значение метрики означает, что небольшие изменения в классах, от которых зависит данный, приведут к большим изменениям в данном, а также изменения в данном классе приведут к изменениям в классах, которые зависят от данного.

3. Реализация системы анализа исходного кода ПО

Система NVMetrik была разработана как дополнение и расширение к программе *Reflector* от компании *Red Gate*, являющейся бесплатной утилитой для *Microsoft .NET*, комбинирующей браузер классов, статический анализатор и декомпилятор.

Текущая версия системы поддерживает только *.NET* модули в силу того, что программа *Reflector* позволяет получить исходный код только *.NET* модулей. После выбора файла проводится анализ модуля путем подсчета метрик. Алгоритм подсчета каждой метрики реализован в отдельном классе, являющемся наследником класса *Metric*. Отображение результатов каждой метрики происходит в отдельном элементе управления, который является наследником *MetricControl*. Такая архитектура позволяет быстро и легко добавлять реализацию подсчета новых метрик в программу, и, кроме того, добавление метрик никаким образом не затрагивает существующий код и не приводит к появлению ошибок.

В элементе управления *MetricControl* осуществляется вывод текстовой информации, полученной в результате анализа исходного кода ПО. Данная функциональность реализована с помощью экспертной системы.

Экспертная система – это программа, которая использует знания экспертов о некоторой конкретной узкоспециализированной предметной области и в пределах этой области способна принимать решения на уровне эксперта-профессионала. Работа экспертной системы как части программы представлена на рис. 2.

Каждая метрика имеет некоторый диапазон значений от минимального до максимального. Расчет метрик проводится согласно формулам (1–3). Этот диапазон разбивается на несколько промежутков, для каждого из которых, в зависимости от метрики, была составлена рекомендация на основе оценок экспертов. Эти рекомендации и промежутки, к которым они относятся, хранятся в отдельном файле формата *.nv, основанном на формате xml. Такой файл и является базой знаний для данной экспертной системы. В процессе анализа программа подсчитывает значение метрики, загружает все промежутки, на которые был разбит диапазон значений метрики, из файла и в зависимости от того, в какой промежуток попадает значение метрики, принимает решение о выборе необходимой рекомендации, которая выводится пользователю. Такое решение позволяет убрать зависимость между подсчетом метрики и текстом рекомендации, так как рекомендации могут быть добавлены отдельно, равно как и промежутки диапазона значений. На данный момент файл с рекомендациями является нередактируемым, однако такая функциональность может быть добавлена в программу при необходимости.

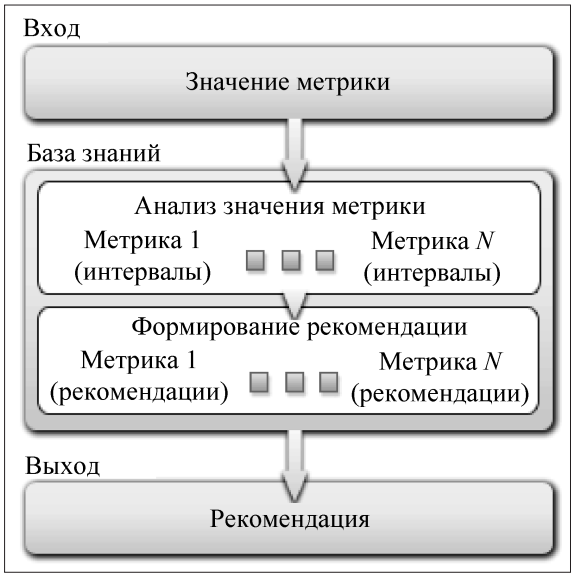


Рис. 2. Схема экспертной системы

Для работы программы, описанной в данной статье, необходимо запустить программу *Reflector*. Так как программа реализована как дополнение, ее необходимо добавить в *Reflector*.

После добавления *NVMetrics* в *Reflector* в меню *Tools* появляется новый пункт *NVMetrics*, выбор которого приводит к появлению элемента управления для анализа dll-модулей. Элемент управления *NVMetrics* состоит из четырех основных частей: панели управления, элемента управления с информацией о загруженном модуле, вкладок с элементами управления, подсчитывающими метрики и отображающими результаты, и элемента управления, отображающего рекомендации по выбранному значению метрики для класса (рис. 3).

Панель управления состоит из трех кнопок: *Open*, *Analyze* и *Get report*. *Open* позволяет выбрать dll-модуль для анализа, после чего в верхней части основного элемента управления появляется информация о загруженном модуле: название и путь к файлу. Нажатие на кнопку *Analyze* приводит к анализу выбранной сборки.

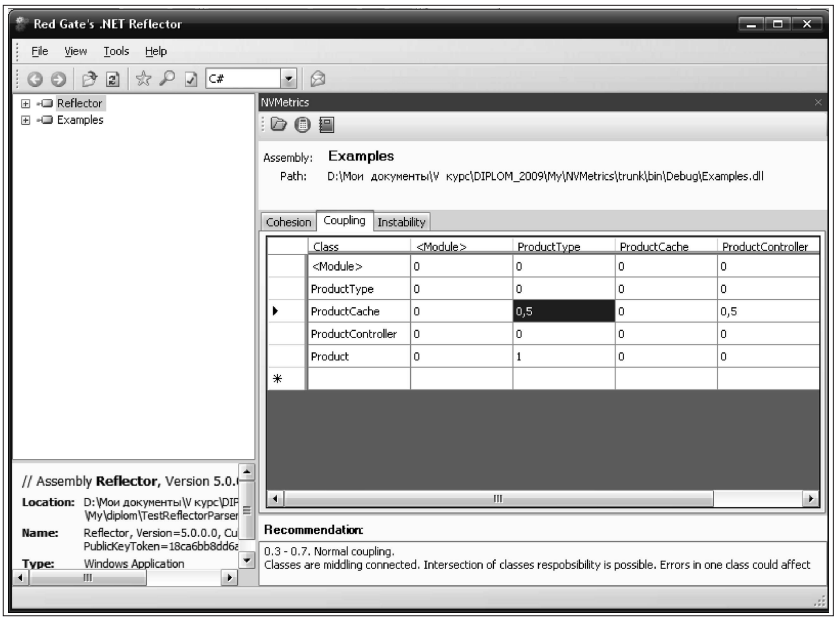


Рис. 3. Отображение результатов подсчета метрик

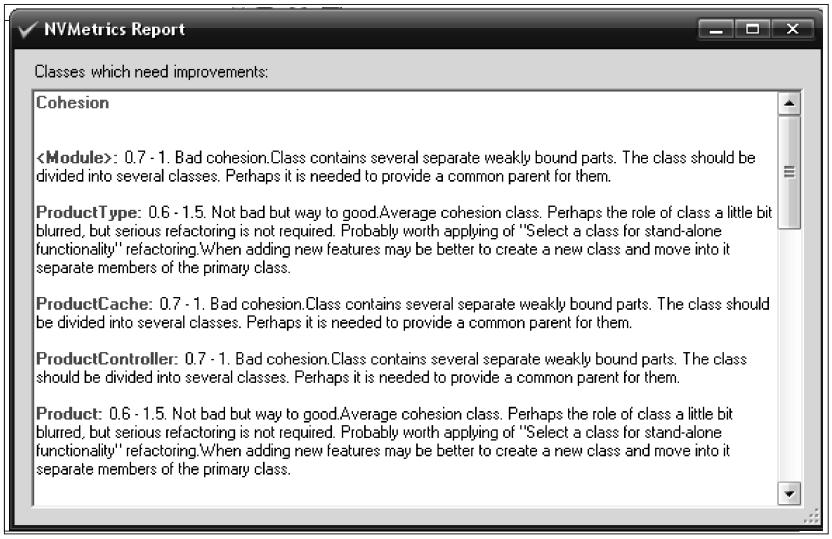


Рис. 4. Рекомендации по модификации исходного кода

Анализ состоит из двух частей. Сначала программа получает полную внутреннюю структуру модуля с помощью функциональности *Reflector* и заполняет свою собственную структуру кода. Затем на основе имеющейся структуры проводится подсчет трех метрик, описанных в п. 2.1.–2.3. Данные метрики считаются независимо друг от друга. Алгоритм подсчета и отображение результатов максимально отделены для уменьшения числа возможных ошибок.

Результаты анализа отображаются на вкладках. Каждая вкладка представляет собой элемент управления, в котором находится список всех классов модуля, а также вычисленные значения метрик в виде списка (*Cohesion* и *Instabilty*) или таблицы (*Coupling*). При нажатии на ячейку со значением метрики в нижней части основного элемента управления появляется текстовая информация о состоянии и качестве кода, а также рекомендации по его улучшению, если таковые необходимы.

При нажатии на кнопку *Get report* происходит генерация отчета, в который входит список классов со значениями метрик, которые требуют улучшения (рис. 4).

Выбор порога, при котором значения метрики ниже граничного не учитываются в создании отчета, осуществляется с помощью специального файла рекомендаций.

Тестирование системы показало ее пригодность к использованию для анализа исходного кода и адекватность выдаваемых рекомендаций по модификации исходного кода. Дальнейшее развитие системы будет происходить в направлении разработки самостоятельного программного продукта, который сможет анализировать исходный код, разработанный на C++, C#, Java и PHP.

СПИСОК ЛИТЕРАТУРЫ

1. **DeMarco T.** Controlling Software Projects: Management, Measurement, and Estimates. Prentice Hall, 1986.

2. **Совершенный код.** Мастер-класс / Пер. с англ. М.: Русская редакция; СПб.: Питер, 2005. 896 с.

3. **Буч Г., Якобсон А., Рамбо Дж.** UML. Классика CS. 2-е изд. / Пер. с англ.; под общей редакцией проф. С. Орлова. СПб.: Питер, 2006. 736 с.

4. **Chidamber S.R., Kemerer C.F.** A Metrics Suite for Object Oriented Design. New Jersey: Prentice-Hall, Inc, 1994.

5. **Martin R.C.** Designing object-oriented C++ applications using the Booch method. New Jersey, Prentice-Hall, Inc, 1995. 530 с.

УДК 004.45

Егоров В.Ю., канд. техн. наук., нач. I отделения, ООО НТП "Криптософт",

e-mail: vec@cryptosoft.ru

Критерии оценки эффективности диспетчеризации задач в многопроцессорной операционной системе

Предлагаются новые критерии оценки эффективности работы диспетчера задач в составе многопроцессорной операционной системы, а также исследуются требования, предъявляемые к алгоритмам диспетчеризации задач.

Ключевые слова: *диспетчеризация задач, критерии диспетчеризации, оценка эффективности диспетчеризации, формулы оценки*

Введение

Ядро современной операционной системы (ОС) можно разбить на несколько достаточно обособленных частей. Одной из них является диспетчер задач (ДЗ) ОС. На ДЗ в составе ОС возлагается работа, связанная с определением последовательности задач, требующих исполнения. От верной реализации ДЗ в значительной степени зависит производительность вычислительной системы (ВС) в целом, поэтому вопросы, связанные с организацией ДЗ, традиционно пользуются повышенным вниманием.

Несмотря на большое число разновидностей ДЗ, вопросы оценки эффективности работы этого компонента в составе ОС изучены недостаточно. Такое состояние дел обусловлено, прежде всего, сложностью алгоритмов, применяемых при выполнении диспетчеризации. Эта сложность еще более возрастает при использовании многопроцессорных ВС.

На сегодняшний день основной тенденцией развития вычислительной техники является увеличение числа процессоров в рамках одной вычислительной системы. Многопроцессорные ВС проникают во все области, требующие повышенной производительности. Поддержка многопроцессорности требуется как в системах общего назначения, так и в системах реального времени.

Алгоритмы диспетчеризации задач однопроцессорной ВС к настоящему моменту исследованы в достаточно большом объеме. Многопроцессорные ВС предъявляют новые требования к алгоритму диспетчеризации, что определяет проведение новых исследований в этой области.

Постановка задачи

В настоящее время довольно большое число публикаций посвящено различным способам реализации ДЗ как в системах общего назначения, так и в системах реального времени. Однако авторы зачастую упускают из виду основное отличие в подходах к реализации ДЗ ОС. Это отличие связано с традиционным восприятием ДЗ как субъекта, выполняющего планирование. На самом деле в вычислительной системе субъектом всегда является процессор, с определенной периодичностью исполняющий код ДЗ. В этом случае вся совокупность данных, хранимых в ДЗ, становится объектом, изменяющим свое состояние при исполнении.

В многопроцессорной ВС число субъектов, исполняющих, возможно одновременно, код ДЗ возрастает. Это означает, что алгоритм диспетчеризации многопроцессорной ВС должен быть адаптирован к параллелизму.

Основной задачей, требующей решения, для создания эффективного алгоритма диспетчеризации является задача определения верных критериев, характеризующих работу ДЗ ОС [1]. Существующие на данный момент критерии оценки диспетчеризации разработаны еще в 60-х годах прошлого века и серьезно устарели. В этой статье сделана попытка определения совокупности критериев, позволяющих адекватно оценить эффективность работы алгоритма ДЗ многопроцессорной системы.

Наиболее информативную оценку эффективности того или иного процесса предлагают критерии в виде отношений, поскольку они лучше всего воспринимаются человеком и могут быть представлены в процен-

тах. Кроме того, такие оценки можно использовать в самой ВС, например, при динамической адаптации ДЗ ОС к нагрузке. Поэтому, при определении критериев оценки диспетчеризации основное внимание следует уделить поиску именно таких критериев.

Подход к оценке эффективности диспетчеризации, принятый в настоящее время

Наиболее часто используемой оценкой работы ДЗ ОС в настоящее время является оценка загрузки процессора ВС [1]. Она вычисляется по следующей формуле:

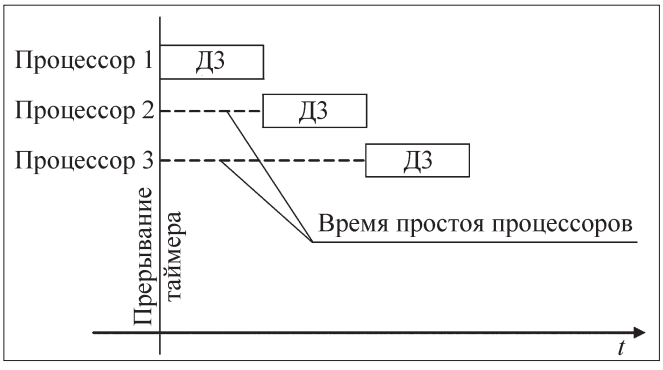
$$U_P = \frac{T_{work}}{T_{idle} + T_{work}}, \qquad (1)$$

где T_{work} – суммарное время исполнения процессором задач в составе системы за единицу времени; T_{idle} – суммарное время простоя процессора за то же время; U_P – загрузка процессора (от словосочетания *processor utilization*).

Однако оценка по формуле (1) показывает лишь отношение общей трудоемкости исполняемых задач к производительности процессора и не является напрямую характеристикой ДЗ. Более того, далее будет показано, что неверная реализации ДЗ может приводить к простою процессоров при наличии в системе задач, ожидающих исполнения. При этом загрузка процессора, вычисленная по формуле (1), будет снижаться.

По сути, алгоритм ДЗ многопроцессорной ВС реализует собой недетерминированный автомат. Производительность многопроцессорной ВС напрямую зависит от реализации этого недетерминированного автомата.

К сожалению, до настоящего времени в современных ОС алгоритм ДЗ реализуется иначе, а именно представляется в виде обычного детерминированного конечного автомата, доступ к которому ограничивается путем организации критического ресурса в составе ОС. Это приводит к тому, что процессоры в многопроцессорных ВС некоторую часть времени простаивают в ожидании доступа к диспетчеру задач. Время такого простоя возрастает пропорционально числу процессоров в многопроцессорной вычислительной системе и является существенным фактором в снижении производительности системы в целом [2, 3].



Простой процессоров по доступу к ДЗ при широковещательной рассылке прерывания таймера

Как известно, при реализации вытесняющей многозадачности, инициатором переключения процессора с одной задачи на другую является прерывание таймера. Если это прерывание будет приходить всем процессорам системы одновременно, то это приведет к длительному простою процессоров по доступу к ДЗ, как показано на рисунке.

Суммарное время простоя всех процессоров будет равно:

$$T_{idle} = \sum^{N} (T_{tma}(n-1)),$$

где T_{tma} – среднее время однократного выполнения алгоритма диспетчеризации; N – число процессоров в системе; T_{idle} – время простоя процессоров в ожидании диспетчеризации, n – шаг итерации.

В таком режиме процессоры системы простаивают, несмотря на то, что в ДЗ в это же самое время имеются задачи, ожидающие исполнения. Если исследовать такую систему только на основании информации о загрузке процессоров в системе, то можно получить неверное представление о работе ДЗ и ВС в целом.

Для минимизации времени простоя можно сделать так, чтобы прерывание диспетчеризации от таймера приходило только на первый процессор в системе. После завершения работы алгоритма диспетчеризации, но перед началом выполнения задачи, процессор должен выслать прерывание диспетчеризации процессору со следующим порядковым номером.

Такой подход сокращает время простоя процессоров, однако содержит в себе ряд недостатков. Во-первых, поскольку алгоритм диспетчеризации является критическим ресурсом и выполняется за определенное время T_{tma}, то при увеличении числа процессоров время $T_{tma} \cdot N$ может достичь времени одного кванта системы. Это приведет к краху диспетчеризации.

Во-вторых, до сих пор мы исходили из предпосылки, что переключение процессора с одной задачи на другую происходит принудительно, однако на деле задача может отдать управление добровольно в любой момент времени. Освобожденный процессор в этом случае поступает в очередь процессоров, ожидающих диспетчеризации, что приводит к увеличению времени простоя процессора.

Из приведенных рассуждений можно сделать следующие выводы:

- время T_{tma} работы алгоритма ДЗ должно быть сведено к минимуму;
- необходимо разработать такие алгоритмы диспетчеризации, при которых процессоры могли бы выполнять поиск задач на исполнение независимо друг от друга.

Выполнение этих требований ведет к повышению производительности системы в целом. Оценку их выполнения можно получить простым подсчетом числа задач, переключаемых процессорами системы за единицу времени. Такую оценку принято называть пропускной способностью системы [1]. Применительно к современным ВС можно получить оценку пропускной способности для каждого процессора системы в отдельности. Обозначим пропускную способность процессора как *C* (от словосочетания *throughput capacity*).

Переключение адресных пространств

Следующим аспектом работы ДЗ, требующим рассмотрения, является необходимость переключения адресных пространств при переходе от одной задачи к другой. Переключение адресных пространств необходимо выполнять при переключении процесса, исполняемого процессором. Оно заключается в сбросе буферов TLB (*Translation lookaside buffer*) процессора, который, во-первых, сам занимает довольно длительное время, а во-вторых, заставляет процессор повторно обращаться к оперативной памяти при выполнении новой задачи. Таким образом, переключение адресных пространств является фактором, снижающим производительность многопроцессорной ВС. При разработке ДЗ необходимо стремиться к минимизации переключений адресных пространств.

Наиболее простой способ добиться минимизации переключений адресных пространств заключается в последовательном выполнении процессором нитей одного и того же процесса. Однако это противоречит самому принципу параллельного программирования, при котором максимальная скорость исполнения программы достигается благодаря использованию нескольких процессоров, исполняющих нити процесса одновременно.

Для ускорения работы ВС в алгоритм ДЗ необходимо внести изменения, которые бы позволили процессору при поиске следующей задачи, ждущей исполнения, при прочих равных условиях выбирать ту нить, чье адресное пространство совпадает с адресным пространством, загруженным в процессор на настоящий момент.

Для оценки эффективности подобной оптимизации необходимо ввести новый критерий, показывающий, насколько часто процессору приходится переключать адресное пространство. Назовем такую оценку эластичностью (*flexibility*) загрузки процессора. Она вычисляется по следующей формуле:

$$F = 1 - \frac{N_P}{N_T}, \qquad (2)$$

где F – эластичность загрузки процессора; N_P – число переключений адресных пространств процессов за единицу времени; N_T – число переключений нитей за то же самое время.

Эластичность загрузки, вычисленная по формуле (2) и равная 1, показывает, что процессор за время измерения не выполнял переключений адресных пространств. Эластичность загрузки, равная 0 показывает, что каждое переключение нити процессором выполнялось с переключением адресного пространства.

Особенный интерес представляет одновременное отслеживание значений загрузки процессора и эластичности его загрузки. Так, если процессор загружен полностью, но эластичность загрузки высока, это означает, что диспетчеризация процессора работает в оптимальном режиме, максимально используя процессорное время для вычислений. Наихудшим сценарием для ДЗ представляется состояние, когда загрузка процессора равна 1 (100 % по относительной шкале), а эластичность загрузки стремится к нулю. В этом случае возникают наибольшие потери времени на исполнение процессором кода ДЗ.

Исходя из вышесказанного, можно предложить следующую относительную (*relational*) оценку загрузки процессора по формуле:

$$R = \begin{cases} 0, \text{ если } U_P - F \le 0, \\ U_P - F, \text{ если } U_P - F > 0, \end{cases} \qquad (3)$$

где R – относительная оценка загрузки процессора; U_P – загрузка процессора; F – эластичность загрузки процессора.

Смысл относительной оценки загрузки процессора по формуле (3) можно описать как способность процессора к исполнению дополнительных нитей без существенных потерь в производительности. Разработчики ДЗ ОС должны стремиться к минимизации относительной оценки загрузки процессора в целях повышения производительности системы.

Ожидание обслуживания

Нити, исполняющиеся в составе ОС, в любой момент времени находятся в одном из своих состояний. Если исключить состояния, обусловленные переходными процессами при старте и останове нити, то остается три состояния: "работа", когда нить исполняет свой алгоритм, "ожидание", когда нить простаивает в ожидании объекта синхронизации и "готовность к исполнению", когда нить простаивает в ожидании обслуживания [4].

С точки зрения повышения производительности ДЗ требуется свести к минимуму время ожидания нитью обслуживания процессором системы. Задержка, вносимая ожиданием нити в очередях диспетчера задач, не может быть соотнесена с каким-либо из процессоров в составе системы, поскольку является характеристикой оценки эффективности ДЗ в целом. Поэтому для оценки задержки вычислений, вызванных простоем нити в очередях диспетчера задач, необходимо соотнести общее время работы нити со временем ее простоя, исключив при этом время простоя нити из-за ожидания объектов синхронизации [5]. Для каждой нити оценку времени ожидания (*waiting*) можно посчитать по следующей формуле:

$$W = \frac{T_{wait}}{T_{wait} + T_{work}}, \quad (4)$$

где W – индекс ожидания нити; T_{wait} – время, проведенное нитью в очередях ДЗ; T_{work} – время исполнения нити.

Среднее арифметическое индекса ожидания нити, вычисленного по формуле (4) для всех нитей, функционирующих в системе, дает индекс ожидания всего диспетчера задач ОС. Разработчики ДЗ любой ОС должны стремиться к минимизации значения индекса ожидания ДЗ.

Распределение задач по приоритетам

В настоящее время общепринятым подходом к диспетчеризации является разделение задач по приоритетам [4, 5]. Каждой нити в системе присваивается определенное начальное значение приоритета, которое в ходе работы системы может изменяться как в большую, так и в меньшую сторону. Эти изменения приоритета нити вносятся в алгоритм работы ДЗ в целях решения проблем гарантии и качества обслуживания нити.

Требование гарантии обслуживания нити утверждает, что какой бы ни был приоритет нити, она должна получить управление за конечное и достаточно короткое время. В противном случае нить может вообще не получить управление.

Требование качества обслуживания нити утверждает, что нить должна получать управление соответственно справедливому распределению процессорного времени между нитями в составе системы с учетом их приоритетов. В противном случае может возникнуть дискриминация по процессорному времени со стороны наиболее приоритетных нитей в составе системы.

Современные дисциплины диспетчеризации предполагают хранение задач, ждущих обработки, в специальных очередях. Каждая очередь соответствует определенному уровню приоритета. Процессор в ходе выполнения алгоритма ДЗ просматривает очереди в целях поиска задачи на исполнение в направлении понижения приоритета. Такая организация ДЗ и приводит к тому, что алгоритм диспетчеризации весьма трудно распараллелить.

Другие подходы к построению ДЗ и организации списков задач, ждущих исполнения, заключаются либо в использовании направленных ациклических графов (DAG), либо сбалансированных деревьев [7], либо генетических алгоритмов [8]. Следует также упомянуть ограничения, накладываемые на алгоритм работы ДЗ при функционировании виртуальных машин [9]. Но какой бы ни был выбран подход к организации ДЗ, требования гарантии и качества обслуживания нити должны удовлетворяться.

При разработке ДЗ любой ОС необходимо учитывать требования гарантии обслуживания нити и качества обслуживания [10]. Необходимо также выработать критерий оценки гарантий и качества обслуживания нити как характеристику алгоритма ДЗ. Таким критерием может служить коэффициент обслуживания (*service*) нити, формула вычисления которого приведена ниже:

$$S = \frac{C_{th}}{C_1 + C_2 + \ldots + C_N}, \quad (5)$$

где C_{th} – число квантов нити за единицу времени; C_1, C_2, ..., C_N – пропускная способность процессоров системы за единицу времени; S – коэффициент обслуживания нити.

Коэффициент S, вычисленный по формуле (5), не должен изменяться при равномерной монотонной загрузке U_P процессоров системы. Если же коэффициент S будет изменяться, это будет свидетельствовать о низком качестве обслуживания нитей со стороны диспетчера задач.

Влияние объектов синхронизации на эффективность диспетчеризации задач

Как известно, объекты синхронизации в операционных системах в общем случае бывают двух типов: уведомительные и синхронизирующие. Основное отличие между этими двумя типами объектов синхронизации заключается в следующем: после перевода объекта синхронизации в открытое ("сигнальное") состояние уведомительный объект синхронизации возвращает в очередь диспетчера задач все нити, которые его ожидали, а синхронизирующий объект синхронизации возвращает в очередь диспетчера задач только одну ожидающую нить. Одновременно с этим синхронизирующий объект синхронизации вновь переходит в закрытое ("несигнальное") состояние.

В операционной системе Windows примером синхронизирующего объекта синхронизации может служить событие с автоматическим сбросом, а уведомительного объекта синхронизации – событие с ручным сбросом [11].

Для повышения быстродействия ВС необходимо, чтобы время перехода нитей от состояния ожидания объекта синхронизации в рабочее состояние было минимальным. Это условие наиболее важно для нитей, ожидающих синхронизирующих объектов синхронизации, поскольку именно с ними связаны такие события, как, например, нажатие клавиши на клавиатуре, приход пакета данных по сети и т.д. Чем быстрее нить обработает системное событие, тем выше будет производительность системы в целом.

Отсюда можно заключить, что еще одной оценкой эффективности алгоритма диспетчеризации задач должно служить среднее время перехода нити в рабочее состояние после срабатывания объекта синхронизации. Причем уведомительные объекты синхронизации из этой оценки можно отбросить, как не оказывающие существенного влияния на эффективность диспетчеризации. В оценке должны остаться только синхронизирующие объекты синхронизации. Среднее

время выхода нитей из ожидания объектов синхронизации можно рассчитать по формуле:

$$T_S = \frac{T_{E1} + T_{E2} + \ldots + T_{EM}}{M}, \qquad (6)$$

где T_S – среднее время выхода нитей из ожидания объектов синхронизации; M – число срабатываний объектов синхронизации за единицу времени; T_{E1}, T_{E2}, ..., T_{EM} – время, в течение которого соответствующая нить переходила в рабочее состояние.

Среднее время выхода нитей из ожидания объектов синхронизации, вычисленное по формуле (6), напрямую зависит от производительности процессоров, входящих в состав ВС. Поэтому оно измеряется в долях секунды. Чем меньше величина T_S при использовании одной и той же аппаратной платформы, тем эффективнее ДЗ ОС.

Заключение

Автором предлагается ввести следующие критерии оценки алгоритма ДЗ. Они значительно отличаются от классических критериев оценки, поскольку ориентированы прежде всего на анализ работы многопроцессорных ВС.

- загрузка процессора системы U_P в процентах как отношение времени полезной работы к общему времени работы процессора;
- пропускная способность процессора C как число переключений задач в единицу времени;
- эластичность загрузки процессора F в процентах как отношение числа переключений задач в единицу времени к числу переключений задач со сменой процесса;
- относительная оценка загрузки процессора R как неотрицательная разница между загрузкой процессора и его эластичностью загрузки;
- индекс ожидания диспетчера задач W в процентах как среднее значение отношения времени ожидания нити в очередях ДЗ к времени исполнения;
- коэффициент обслуживания нити S в процентах как отношение числа квантов нити к общей пропускной способности системы за единицу времени;
- среднее время выхода нити из ожидания объекта синхронизации T_S, т.е. время, прошедшее между моментом освобождения объекта синхронизации и моментом перехода ждущей его нити в рабочее состояние.

Совокупность приведенных критериев оценки касается только работы алгоритма ДЗ и не учитывает другие характеристики, также влияющие на производительность ВС. Автором намеренно исключены из рассмотрения вопросы, связанные с быстродействием таких частей ОС, как подсистема ввода-вывода, подсистема управления файлами, сетевая подсистема, графическая подсистема и т.д.

Использование представленных критериев позволит получать более объективную оценку эффективности функционирования ДЗ в составе ОС. Однако их применение в силу объективных причин возможно только в тех ОС, у которых можно получить доступ к исходным кодам системы. Использование внешних драйверов для получения данных о работе системы представляется нецелесообразным.

СПИСОК ЛИТЕРАТУРЫ

1. **Гордеев А.В., Молчанов А.Ю.** Системное программное обеспечение. СПб.: Питер, 2001. 736 с.

2. **Егоров В.Ю.** Задачи взаимодействия процессоров в многопроцессорной операционной системе // Вопросы радиоэлектроники, серия ЭВТ. 2010. Вып. 5. С. 5–13.

3. **Егоров В.Ю.** Механизмы взаимодействия процессоров в многопроцессорной операционной системе // Вопросы радиоэлектроники, серия ЭВТ. 2010. Вып. 5. С. 14–21.

4. **Соломон Д., Руссинович М.** Внутреннее устройство Microsoft Windows 2000. Мастер-класс. / Пер. с англ. СПб.: Питер. М.: Русская редакция, 2001. 752 с.

5. **Егоров В.Ю.** Методика тестирования времени ожидания задачи в современных операционных системах. Новые информационные технологии и системы // Тр. VIII Международной научно-технической конференции. Ч. 2. Пенза, ПГУ. 2008. С. 65–69.

6. **Робачевский А.М.** Операционная система UNIX. СПб.: БХВ-Санкт-Петербург, 1999. 528 с.

7. **Sinnen O., Kozlov A.V., Shahul A.Z.S.** Optimal scheduling of task graphs on parallel systems // Proc. of 25th IASTED International Multi-conf. Parallel and Distributed Computing and Networks. 2007. P. 170–175.

8. **Hou E.S.H., Ansari N. and Ren H.** A Genetic Algorithm for Multiprocessor Scheduling, IEEE trans. on parallel and distributed systems. 1994. Feb. Vol. 5. No. 2. P. 113–120.

9. **Егоров В.Ю.** Особенности диспетчеризации процессов при функционировании виртуальных машин // Системы и средства информатики. Дополнительный вып. М.: Наука, 2009. С. 58–67.

10. **Егоров В.Ю.** Новые подходы к диспетчеризации задач в операционных системах // Известия высших учебных заведений. Поволжский регион. Технические науки. 2008. № 2. С. 56–63.

11. **Рихтер Дж.** Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows. 4-е изд. СПб: Питер; М.: Русская редакция, 2001. 752 с.

УДК 004.4

В.В. Корнеев, д-р техн. наук, проф., зам. дир. по научн. раб., ФГУП "Научно-исследовательский институт "Квант",

e-mail: korv@rdi-kvant.ru

Безопасность облачных вычислений: совместное использование ресурсов на базе грид

Актуальным на настоящее время направлением в области разработки программного обеспечения высокопроизводительных вычислений и распределенной обработки больших массивов данных является подход, основанный на технологиях грид и облачных вычислений. Однако "узким местом" в рамках программной реализации и практического применения этих технологий является информационная безопасность. В настоящей статье представлен базирующийся на грид-технологии подход к построению безопасных гибридных "облаков".

Ключевые слова: *облачные вычисления, грид, информационная безопасность*

Введение

Острая потребность в совершенствовании технологий распределенной обработки больших массивов данных побуждает исследователей формулировать концепции, являющиеся следствием возникающих на этом пути проблем в разных их проявлениях. Так появились подходы, основанные на сервисно-ориентированных архитектурах, на методологиях аутсорсинга, грид, облачных вычислениях и др. Представляется, что многообразие подходов обусловлено объективной сложностью решения задачи в комплексе, так как достаточно трудно совместить универсальность подхода с его эффективной реализацией. Так, в случае применения предприятием технологии облачных вычислений, возникают опасения, вызванные возможностью несанкционированного доступа к данным при их передаче и обработке в публичном облаке. В настоящей статье для обеспечения безопасности облачных вычислений предлагается использовать подход, развитый в грид и прошедший этапы апробации и усовершенствований в ряде практически значимых проектов.

1. Грид и облачные вычисления

Среди современных, основанных на новейших сетевых технологиях подходов к распределенной обработке данных заметное место занимают грид [1] и облачные вычисления [2]. Оба этих подхода возникли на базе идеи предоставления компьютерных ресурсов по запросам пользователей в условиях неравномерного характера запросов на ресурсы, когда у разных пользователей в разное время появляются бóльшие или меньшие потребности в ресурсах, а суммарных ресурсов в каждый момент времени хватает всем. Конечно, этими двумя не исчерпывается весь спектр технологий распределенной обработки. Наряду с ними существуют и другие: вычисления по запросу, эластичные вычисления, предоставление сервисов по модели коммунальных услуг, распределенные вычисления – все это попытки представить, в основном, одну и ту же сущность, но с различных сторон. Само обилие терминов служит следствием того, что ни один из подходов не исчерпывает разнообразия особенностей технологий распределенной обработки.

Исторически, первой возникла технология грид как подход к совместному доступу к ресурсам пользователей, входящих в виртуальные организации (ВО). Технология гарантирует отсутствие несанкционированного доступа к ресурсам, как между виртуальными организациями, так и внутри одной организации. Однако грид никак не конкретизирует способ использования ресурсов пользователями. Это свидетельствует о том, что эта технология представляет со-

бой, скорее, инструмент, чем продукт. Например, установка являющегося де-факто стандартом программного обеспечения (ПО) грид на основе middleware Globus Toolkit [1] на компьютеры сети, как правило, ведет к созданию грид-среды, однако дает пользователю по сравнению с обычной сетью ограничение: пользователь может получить доступ к вычислительному ресурсу лишь при наличии сертификата и будучи зарегистрированным пользователем на этом ресурсе. Для использования ресурсов грид пользователям необходимо установить существующую или создать собственную систему управления, распределяющую ресурсы между пользователями [3], а также определить способ использования ресурсов, например, установка библиотеки MPI (Message Passing Interface) для выполнения параллельных вычислений или установка пакета для порождения и поддержки функционирования виртуальных машин.

Представляется, что отмеченная выше особенность технологии грид способствовала возникновению более адекватной потребностям пользователей технологии облачных вычислений. Облачные вычисления (*cloud computing*) – технология распределенной обработки данных, в которой компьютерные ресурсы предоставляются пользователю как Интернет-сервисы определенной зафиксированной функциональности, включающие, например, следующие:

- IaaS – инфраструктура как услуга – предоставление пользователю инфраструктуры из виртуальных машин, сети между ними и дискового пространства;
- PaaS – платформа как услуга – предоставление пользователю возможности запустить свое ПО на предоставляемой облаком программной платформе;
- SaaS – программное обеспечение как услуга – предоставление пользователю доступа к ПО, работающему в облаке.

Таким образом, в облачных вычислениях пользователю, в частности сети Интернет, через web-интерфейс предоставляется ресурс конкретной, необходимой ему структуры из числа упомянутых выше.

Для предоставления услуг облачных вычислений используется специализированное ПО, обобщенно именуемое "*middleware control*". Реализация облачных сервисов требует создания самоконтролируемой масштабируемой и устойчивой к аппаратно-программным сбоям инфраструктуры, выполняющей балансировку нагрузки физических вычислительных ресурсов. Естественно существуют как коммерческое, так и свободно распространяемое ПО.

К числу наиболее известных коммерческих реализаций IaaS относится Amazon Web Services (AWS). Соответственно Eucalyptus (http://open.eucalyptus.com) служит примером свободно распространяемого ПО, имеющего те же команды и прикладные программные интерфейсы, что и AWS. Среди другого свободно распространяемого ПО можно отметить Enomaly (www.enomaly.com) и OpenNebula (www.opennebula.org), а также OpenStack Nova (nova. openstack.org) и OpenStack Swift (swift.openstack.org) [4].

Платформы Google AppEngine и Microsoft Azure служат примерами решений класса PaaS. Платформа Google AppEngine адаптирована для масштабируемых приложений на Java или Python, а Azure построена специально для .net и для .net-разработчиков.

Облака называют публичными, если их ресурсы доступны любым пользователям сети, частными (корпоративными), если ресурсы принадлежат предоставляющим их предприятиям (организациям) и доступ к ним имеет только ограниченный круг определенных ими пользователей, и гибридными, если они состоят из публичных и частных облаков. Собственно, само появление гибридных облаков обусловлено тем обстоятельством, что предприятие определяет, какие данные и программы недопустимо предоставлять за пределами предприятия, а какие можно поместить в публичное облако, доверившись его средствам защиты от несанкционированного доступа и сэкономив за счет аренды облачных ресурсов вместо приобретения собственных.

Для того, чтобы создать гибридную облачную инфраструктуру, состоящую из совокупности частных и публичных облаков, каждое из которых имеет свои данные и программы, необходимо организовать безопасное распределенное управление доступом, что представляется возможным в рамках технологии грид. В следующем разделе представлен подход к реализации защиты от несанкционированного доступа, прошедший апробацию в Globus Toolkit, а также продемонстрировано развитие этого подхода сообществом пользователей грид, имеющих разные политики информационной безопасности.

2. Архитектура гибридных облаков на базе грид

Грид, создаваемая на базе middleware Globus Toolkit (GT) [1], объединяет множество вычислительных систем (ВС) в целях предоставления пользователям возможности доступа из любой ВС к ресурсам всех ВС. Структура грид показана на рис. 1.

Каждая ВС имеет управляющую машину (УМ), в качестве которой может выступать сама ВС, как например $ВС_3$, показанная на рис. 1. Остальные машины ВС, называемые вычислительными модулями (ВМ), служат для выполнения заданий пользователей. На УМ всех ВС устанавливается ПО GT, на базе которого функционирует ПО управления ресурсами и заданиями грид [3].

Каждая ВС, входящая в состав грид, имеет собственную локальную систему управления, которая получает задания как непосредственно от пользователей данной ВС, так и через сеть от пользователей других ВС. Исходно GT поддерживал интерфейс к локальным системам планирования заданий (LSF, PBS, PRUN, NQE, CODINE и др.), которые поддерживают очередь параллельных заданий, предназначенных для исполнения на этой ВС, и распределяют вычислительные модули ВС между заданиями в соответствии с числом необходимых заданию ВМ. Например, в ходе создания ин-

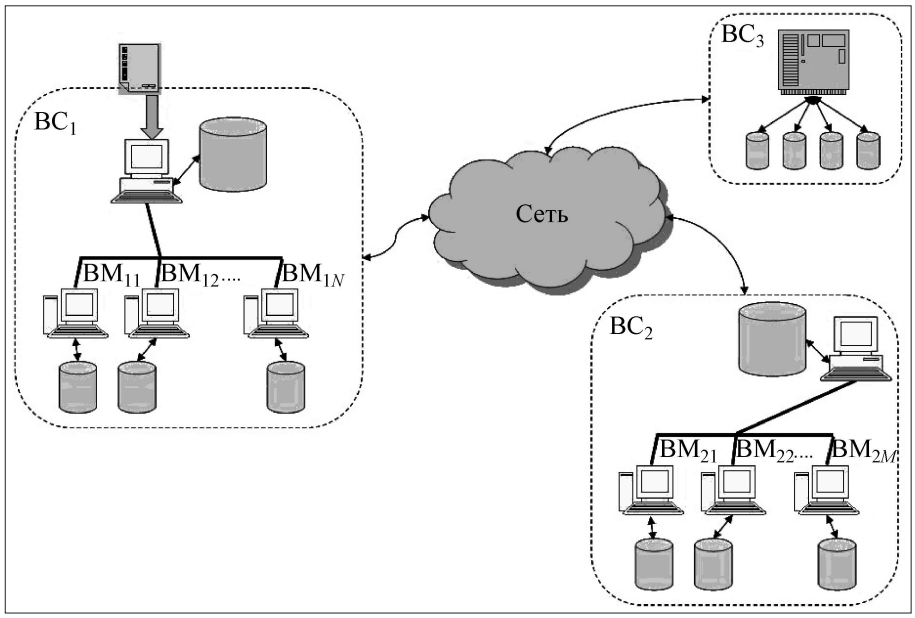


Рис. 1. Архитектура гибридного облака

фраструктуры распределенных вычислений МСЦ РАН был разработан интерфейс GT с системой СУПЗ (система управления параллельными задачами), широко используемой на отечественных суперкомпьютерах семейства МВС-1000 [3]. Аналогичным образом могут быть реализованы интерфейсы с управляющими центрами (контроллерами) облаков, например с API (Application Programming Interface)-сервером, реализующим web-интерфейс с контроллером (*cloud controller*) облака OpenStack Nova [4].

В гибридном облаке, построенном с использованием GT, безопасная пересылка файлов между ВС и запросов пользователей выполняется средствами GT. Собственно, при этом каждое облако выступает как ресурс грид с собственной локальной системой управления. Может быть создана инфраструктура, включающая произвольное число публичных и частных облаков, а также других ресурсов, например существующих "до облачных" параллельных суперкомпьютеров. Следует отметить, что последнее решение можно трактовать как PaaS, предоставляющую платформу MPI для выполнения параллельных программ, созданных на базе этой библиотеки.

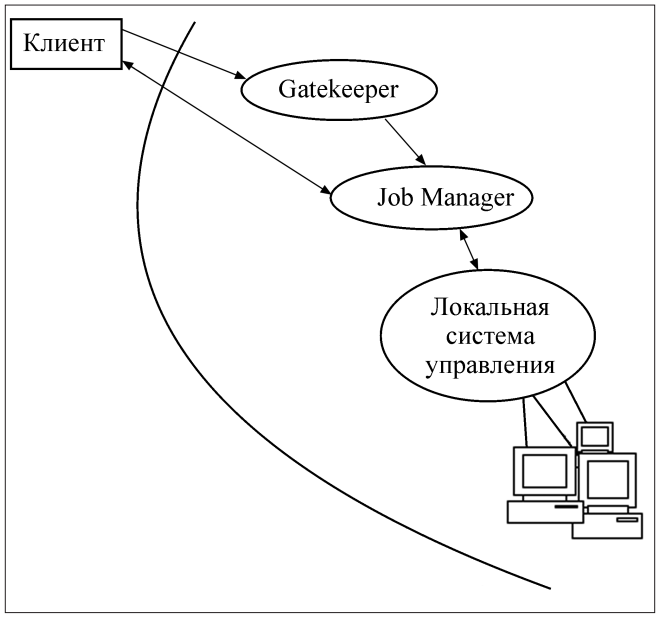


Рис. 2. Архитектура вычислительного ресурса с установленной GRAM

Безопасное управление ресурсами осуществляется в GT системой Globus Resource Allocation Manager (GRAM). Задание пользователя посредством GRAM передается локальной системе управления. Архитектура вычислительного ресурса с установленной GRAM представлена на рис. 2.

Система GRAM состоит из двух компонентов Gatekeeper и Job Manager. Процесс Gatekeeper выполняется на управляющей машине ВС и запускается с правами суперпользователя (*root*). Обращение пользователя грид к процессу Gatekeeper осуществляется с помощью программы *globusrun*, которая является клиентом GRAM. При обращении пользователя грид к процессу Gatekeeper выполняется взаимная аутентификация пользователя и вычислительного ресурса. Для взаимной аутентификации вычислительный ресурс и пользователь должны обладать сертификатами X.509. Взаимная аутентификация пользователя и процесса Gatekeeper осуществляется в соответствии с протоколом SSL.

В случае успешной аутентификации происходит авторизация пользователя. Основная схема авторизации в GT заключается в том, что клиент авторизуется на некотором ресурсе, если в файле авторизации (*grid-mapfile*) извлекаемому из сертификата Gatekeeper идентификатору пользователя поставлено в соответствие имя локального пользователя данного ресурса. Если идентификатор клиента в файле авторизации отсутствует, то клиент признается неавторизованным, и запрос отвергается. В результате авторизации определяется локальная учетная запись, от имени которой пользователь грид получает доступ к ресурсу. После авторизации пользователя, процесс Gatekeeper ограничивает свои права правами этого (непривилегированного) пользователя и

запускает экземпляр процесса Job Manager Instance (JMI), осуществляющий дальнейшее управление заданием в течение всего времени его выполнения, реализуя запросы на остановку задания, получения статуса задания и др. Процесс JMI получает от клиента параметры запуска задания (исполняемый файл, входные, выходные файлы, аргументы) и предает его на выполнение. Для каждого задания запускается собственный экземпляр JMI.

Устанавливаемая по умолчанию конфигурация GRAM осуществляет запуск задания путем порождения нового процесса Unix. Предусмотрена возможность написания скриптов для расширения функциональности процесса Job Manager. Этим путем реализуются интерфейсы с различными локальными системами управления.

С точки зрения реализации политик безопасности GT имеет ряд недостатков [5]:

- авторизация пользователя при запуске задания носит слишком дискретный характер, а именно – пользователь либо имеет локальную учетную запись на ресурсе, и его задание запускается, либо такой записи он не имеет, и тогда пользователь не получает доступа к ресурсу;
- авторизация при управлении выполняемым заданием статична – только пользователь, запустивший задание, имеет возможность управления им;
- нет возможности реализовать динамическую политику авторизации на ресурсе, задаваемую извне, например, со стороны виртуальной организации (ВО), в которую входит пользователь;
- права пользователя на локальном ресурсе определяются его учетной записью и не могут зависеть от вида выполняемого запроса к ресурсу;
- использование локальных учетных записей делает затруднительным совместное использование сервисов разными ВО и сообществами пользователей.

Последние два недостатка могут быть частично преодолены за счет создания учетных записей с разными правами и разделяемых учетных записей, что, однако, порождает другие трудности, вызванные этим разделением.

3. Расширение средств авторизации GT

3.1. Реализация заданных политик безопасности

В работе [5] предлагается расширение GRAM, направленное на частичное преодоление перечисленных недостатков GT. Вводится возможность задавать условия авторизации для оценки обращений пользовательских заданий и запросов к исполняемым заданиям в контексте политик доступа к ресурсу, формулируемых владельцем ресурса и ВО, к которым принадлежит пользователь. С этой целью модифицируется компонент Job Manager, в который вводятся фильтры для проверки задаваемых условий и наборы разрешенных действий по каждому условию.

Для описания заданий в GRAM используется язык RSL (Resource Specification Language), на котором связываются пары атрибутов со значениями, определяющими имя задания, размещение файлов задания, а также требования к ресурсам: число процессоров; объем памяти; максимальное время выполнения и др. В работе [5] сконструирован язык задания политик в терминах RSL, на котором формулируются правила, включающие условия, выполнение которых обязательно для выполнения обращений пользовательских заданий и запросов к исполняемым заданиям. Правила содержат идентификатор пользователя или группы пользователей, отделенный запятой от условий правила.

Принуждение к выполнению политик выполняется путем модификации компонента Job Manager: в него вставляется блок оценки исполнения политик (БП) при внешнем запросе к заданию и запросах задания к внешним ресурсам. Модуль БП разбирает поступивший запрос и определяет возможность его исполнения. При отсутствии подтверждения исполнения запроса выдается соответствующее сообщение об ошибке.

Недостатком предлагаемого решения является то обстоятельство, что JMI выполняется с привилегиями задания и, следовательно, задание имеет возможность модифицировать JMI.

Модификацией Gatekeeper можно позволить администраторам ресурсов влиять на авторизацию, например, ограничить удаленный доступ в периоды пиковой нагрузки локальных пользователей.

3.2. Ролевое расширение авторизации на базе X.509

Упомянутые выше недостатки, присущие авторизации с использованием прокси-сертификатов, могут потребовать применения других способов авторизации. Так как используемые для авторизации в GT сертификаты X.509 стандартизованы, то предлагаемое в работе [6] решение не предполагает их изменения. Оно просто добавляет дополнительный процесс аутентификации и авторизации на базе вводимых дополнительных свидетельств, приписываемых пользователям. Эти свидетельства размещаются в отдельном файле, сопровождающем сертификат X.509. У пользователя может появиться потребность иметь не одну, а несколько ролей в этом файле. Ясно, что пользователь при этом должен быть лишен возможности напрямую писать в этот свой файл по соображениям обеспечения безопасности. Для получения записи, соответствующей его роли, пользователь должен запрашивать специальный доверенный централизованный сервер в компании или в ВО. Этот сервер возвращает пользователю после соответствующей его проверки сформированную запись роли этого пользователя и высылает эту запись пользователю. Для предотвращения подделки записи применяется цифровая подпись. По этой причине, если сервер подписывает данные своим закрытым ключом, то пользователь может проверить достоверность этих данных, используя открытый ключ сервера.

Предлагаемое решение похоже на то, что используется в Attribute Certificates Akenti [7]. Отличие состоит в том, что в Akenti продолжительность использования Attribute Certificates предполагается короткой, подобно билету Kerberos, а цифровые подписи дополнительного файла должны существовать в течение значительно более длительного периода.

Сначала пользователь посылает запрос, сопровождаемый его сертификатом X.509, серверу организации. Сервер проверяет запрос на достоверность. Если запрос принимается, то пользователь получает ролевую запись, которая защищается цифровой подписью. Затем подпись и ролевая запись, зашифрованные открытым ключом пользователя, высылаются ему. При этом передается сначала тэг, затем время, до которого роль существует, затем ее название, например, "программист НИИ "Квант".

Пользователь посылает запрос к ресурсу, сопровождаемому ролевым файлом. Компонент Gatekeeper проверяет тэги в ролевом файле на предмет их признания в этом узле. Если такой тэг обнаруживается, то ролевая запись расшифровывается с помощью открытого ключа компании. Далее осуществляется авторизация в соответствии с ролью из ролевого файла. При этом проверяется срок действия роли. Весь процесс авторизации протоколируется, поэтому можно установить причину отказа в доступе.

3.3. Механизмы аутентификации и авторизации в проекте EGEE

Предлагаемое в проекте EGEE [11, 12] расширение механизма аутентификации и авторизации имеет целью учет, с одной стороны, требований владельцев ресурсов, а с другой стороны, VO, к которым принадлежат пользователи. Основное направление – наделение прокси-сертификатов возможностями, необходимыми для выполнения заданных действий и не бóльшими.

При управлении доступом к ресурсу должны приниматься к рассмотрению как ограничения глобальной политики безопасности, например, членство в VO, так и локальные ограничения, задаваемые, например, списком пользователей, которым запрещен доступ. При этом доступ в грид-систему возможен только для пользователей, входящих хотя бы в одну VO, которую эта система обслуживает.

Для решения задачи масштабируемого доступа в работе [8] предложен сервис членства в виртуальных организациях – VOMS (Virtual Organization Membership Service). Этот сервис содержит базу данных со сведениями о членстве каждого пользователя в определенных VO, его ролях и т. д. Сервис VOMS позволяет авторизованным администраторам менять содержимое своей базы, а пользователям – обращаться с запросами о своем членстве в VO, роли и других атрибутах. Исполняя запрос *voms-proxy-init* с параметрами, соответствующими намерениям пользователя, он получает кратковременно существующий Attribute Certificate [9]. Этот сертификат пользователь в дальнейшем может использовать для доступа к ресурсу грид-системы.

Attribute Certificate имеет форму не влияющего на функциональность расширения прокси-сертификата формата X.509. Расширенный прокси-сертификат воспринимается базовым сервисом безопасности GT как обычный прокси-сертификат.

Прокси-сертификат VOMS может содержать данные из более чем одной VO. Рассмотрим, как изменяется функционирование GRAM в рассматриваемом окружении. Сначала задание прибывает с расширенным прокси-сертификатом на вычислительный ресурс. Пока задание ждет начала исполнения и исполняется, оно должно иметь состоятельный прокси-сертификат. Для хранения прокси-сертификатов используется специальный сервер MyProxy [10], в базу данных которого пользователь помещает выпущенный им прокси-сертификат. Доступ к серверу MyProxy ограничен только кругом доверенных клиентов, среди которых сервис продления сертификатов (Credential Renewal service). Этот сервис позволяет продлевать срок действия прокси-сертификатов заданий, длительность выполнения которых превышает исходный срок действия прокси-сертификата. Для получения сертификата с новым сроком годности сервис продления сертификатов предъявляет старый сертификат, срок годности которого еще не истек. Для каждого исполняемого задания продление сертификатов осуществляется периодически.

Если зарегистрированный прокси-сертификат содержит данные из VOMS, то сервис продления сертификатов опрашивает соответствующие VOMS для внесения изменений, если такие имеются.

При приеме запроса Gatekeeper, наряду с обычной проверкой прокси-сертификата запроса, выполняет дополнительную проверку с помощью сервиса локальной авторизации LCAS (Local Centre Authorization Service [11]). Сервис LCAS состоит из совокупности независимых встраиваемых модулей авторизации, принимающих совместное решение о предоставлении доступа к ресурсу или отказа в доступе. Решение о предоставлении доступа принимается на основе запроса с требованиями к ресурсам, выраженному на языке RSL, пользователя, запрашивающего доступ, а также полномочий пользователя, предоставляемых его сертификатом. В текущей версии LCAS используются следующие встраиваемые модули:

- модуль, проверяющий присутствие идентификатора пользователя в списках пользователей, которым разрешен и запрещен доступ соответственно;
- модуль, проверяющий интервал астрономического времени, допустимый для доступа пользователя;
- модуль, сравнивающий значения атрибутов VOMS и значения в локальном списке управления доступом к ресурсу.

Пользователи могут добавлять свои встраиваемые модули без перекомпиляции.

Для отображения идентификатора пользователя в локальную учетную запись используется сервис локального отображения полномочий LCMAPS (Local Credential MAPping Service [12]). Этот сервис использует два типа встраиваемых модулей:

– сбора информации о полномочиях, необходимых для выполнения запросов;

– принуждения к выполнению заданных политик.

Сервис локального отображения полномочий способен выполнять следующие действия:

- отображение в локальную учетную запись и группу грид идентификатора пользователя на основании grid-mapfile (статическое отображение);
- отображение в пул учетных записей;
- отображение с реализацией всех ограничений политик безопасности VOMS, включая роли;
- принуждение к исполнению требуемой политики безопасности.

Хранилище работ (Job Repository – JR) поддерживает в актуальном состоянии записи о полномочиях всех исполняемых работ. С помощью JR можно отследить прошлое состояние работ.

Заключение

Представленный в настоящей публикации подход к построению гибридных облаков позволяет организовать распределенное управление доступом к ресурсам, отвечающее требованиям распределенной структуры обрабатываемых данных и используемых для этого программ. Такой подход предоставляет возможность использовать все преимущества облачных вычислений, принуждая при этом выполнять заданные требования политики безопасности.

СПИСОК ЛИТЕРАТУРЫ.

1. **Foster I., Kesselman C.** Globus: A Metacomputing Infrastructure Toolkit. URL: http://www.globus.org.

2. **Лоридас П.** Вознесение: приложения для облаков // Открытые системы. 2010. № 6. URL: http://www.osp.ru/os/2010/06/13003733/.

3. **Корнеев В.В., Семенов Д.В.** Распределенный метапланировщик грид. // Вычислительные методы и программирование. 2010. Том 11. С. 69–76.

4. **Зобнин Е.** Облако, открытое для всех. Открытая cloud-инфраструктура OpenStack: обзор и первые впечатления // Хакер. 2011. № 2. С. 112–115.

5. **Keahey K. et al.** Fine-Grain Authorization Policies in the GRID: Design and Implementation // Middleware Workshops'2003. P. 170–177.

6. **Lock R., Sommerville I.** Grid Security and its use of X.509 Certificates. Department of Computer Science Lancaster University. Funded by EPSRC project studentship associated with the UK EPSRC DIRC project grant GR/N13999.

7. **URL:** http://www-itg.lbl.gov/Akenti/.

8. **Cornwall L.A. et al.** Authentication and Authorization Mechanisms for Multi-Domain Grid Environments // Journal of Grid Computing. 2004. № 2. P. 301–311.

9. **Farrell S. and Housley R.** An Internet Attribute Certificate Profile for Authorization // RFC3281. 2002. April. URL: http://www.rfc-editor.org/rfc/rfc3281.txt.

10. **Novotny J., Tuecke S. and Welch V.** An Online Credential Repository for the Grid: MyProxy // Proc. the 10th International Symposium on High Performance Distributed Computing (HPDC-10). IEEE Press. 2001. August.

11. **Steenbakkers M.** Guide to LCAS, Version 1.1.16 / Documentation of the European DataGrid Project. 2003. 15 September. URL: http://hep-proj-grid-fabric.web.cern.ch/help-proj-grid-fabric/documentation/1cas.pdf.

12. **Steenbakkers M.** Guide to LCMAPS, Version 0.0.16/ Documentation of the European DataGrid Project. 2003.15 September. URL: http://hep-proj-grid-fabric.web.cern.ch/hep-proj-grid-fabric/documentation/lcmaps.pdf.

УДК 004.01

Е.А. Степалина, магистрант ГУ–ВШЭ, ведущий технический писатель ООО "СмартЛабс",

e-mail: estepalina@mail.ru

Использование арендуемых систем для документирования программного обеспечения

Рассматриваются виды документации, сопровождающей программное обеспечение, жизненный цикл документации. Сформулированы требования к инструментам документирования для различных видов документов. Рассматриваются системы документирования двух основных типов – CMS и Wiki, приводится сравнительная оценка стоимости использования коробочных и арендуемых версий этих систем.

Ключевые слова: *документирование ПО, документация, Wiki, XML CMS, SaaS*

Введение

В настоящее время разработчики систем управления проектами и задачами начинают предлагать свои решения по схеме SaaS (*Software-as-a-Service*), согласно которой программное обеспечение (ПО) можно взять в аренду с периодической платой за ее использование. Инструментарий разработчика ПО перемещается в "облако", позволяющее минимизировать затраты на старте проекта и гибко масштабировать ресурсы по мере необходимости.

С увеличением сложности программного обеспечения увеличивается потребность в эффективном сопровождении, документировании архитектуры, взаимодействии с пользователями [1]. Средства документирования ПО также становятся доступными для аренды, что является интересным предложением по снижению издержек на сопровождение разрабатываемого продукта. В данной статье рассматриваются основные типы документов, используемые в разработке и поддержке ПО и существующие средства документирования, доступные для использования по SaaS-модели.

Виды документов

В процессе разработки и сопровождения ПО используются следующие виды документов [6]:

• Проектные документы – это технические задания или спецификации требований, методика приемосдаточных испытаний. В ТЗ могут входить UML-диаграммы системы, схемы бизнес-процессов, взаимосвязи компонентов и т.п., архитектура проектируемой системы. По большей части это закрытые, строго конфиденциальные документы, используемые системными архитекторами, топ-менеджерами, руководителями проекта.

• Технические документы – спецификации ПО, спецификации стандартов, принятых в разработке, регламенты кодирования. Например, регламенты именования переменных. (Узкоспециализированные документы, используемые системными инженерами, программистами.)

• Документация кода – описание классов, их свойств и методов, интерфейсов, недокументированных возможностей платформы разработки, узких мест реализации. Документация к коду чаще всего составляется в виде комментариев к исходному коду программы. (Узкоспециализированные документы, используемые программистами.)

• Пользовательская документация (эксплуатационная документация) – руководства по эксплуатации, инструкции, контекстные справки по опциям программы. Это могут быть сведения по установке, настройке системы, советы по устранению неполадок, часто задаваемые вопросы и др. (Документы для широкого круга пользователей продукта.)

С создания проектных документов начинается подготовка и планирование разработки ПО. Процесс раз-

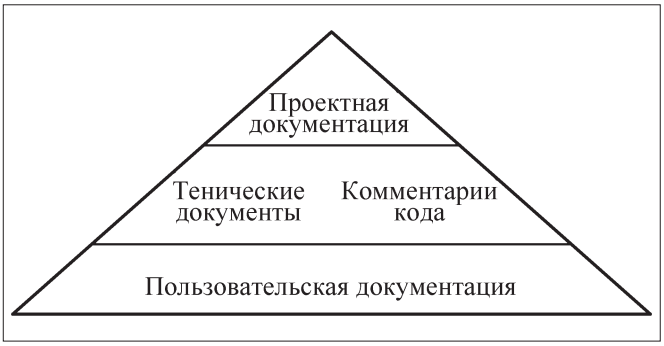


Рис. 1. Состав документации ПО

работки использует технические документы, порождает документацию кода. Когда часть функциональности продукта готова, технические писатели включаются в работу и создают пользовательскую документацию, которую сопровождают на протяжении остальной части жизненного цикла продукта. Наибольший объем документации составляют технические и пользовательские документы (рис. 1).

Требования к инструментам документирования

Разработка и поддержка документов каждого типа предъявляют специальные требования к инструменту. Можно сформировать общий набор требований. Каждое из требований имеет разный приоритет для разных видов документов. Приоритет может быть низким, средним и высоким. В табл. 1 указаны приоритеты требований к средству документирования, предъявляемые разными типами документации. Средства документирования, доступные для аренды, удовлетворяют всем перечисленным требованиям. Таким образом, можно проанализировать эффективность применения средств документирования, предлагаемых по SaaS-модели, для разных типов документации. Из табл. 2 видно, что SaaS-инструменты удовлетворяют большинству требований пользовательской и технической документации. Это не означает, что для разработки проектной документации не следует использовать SaaS-средства. Полученный результат говорит о том, что для пользовательской и технической документации использование таких инструментов будет наиболее эффективным. Для проектных документов можно не менее эффективно применять локальные средства, например, Microsoft Word, простые редакторы UML-диаграмм. Для документирования кода не используются специальные средства, комментарии являются частью текста программы и хранятся в репозитории. При необходимости комментарии кода экспортируются в справочную систему с помощью специальных утилит (Doxygen, JavaDoc).

Таблица 2

Результаты анализа требований к инструменту документирования

Вид документации	Результаты анализа	
	Число требований с приоритетами Высокий/Средний/Низкий	Эффективность применения SaaS-средств
Проектная	1/3/4	Низкая
Техническая	1/6/1	Средняя
Комментарии кода	1/3/4	Низкая
Пользовательская	5/2/0	Высокая

Жизненный цикл документации

На рис. 2 изображены ключевые стадии жизненного цикла документа. На каждой стадии требуются от-

Таблица 1

Приоритет требований к инструменту документирования

Требования	Документация			
	Проектная	Техническая	Комментарии кода	Пользовательская
Функциональные требования				
Поддержка многопользовательского режима редактирования	Низкий	Высокий	Высокий	Высокий
Повторное использование контента	Средний	Средний	Низкий	Высокий
Удаленный доступ к источнику (через Интернет)	Средний	Средний	Средний	Средний
Публикация из единого источника в различные форматы	Низкий	Средний	Низкий	Высокий
Автоматизированная локализация	Низкий	Низкий	Низкий	Высокий
Нефункциональные требования				
Защита конфиденциальности информации	Высокий	Средний	Низкий	Низкий
Масштабируемость системы	Низкий	Средний	Средний	Высокий
Отказоустойчивость	Средний	Средний	Средний	Средний

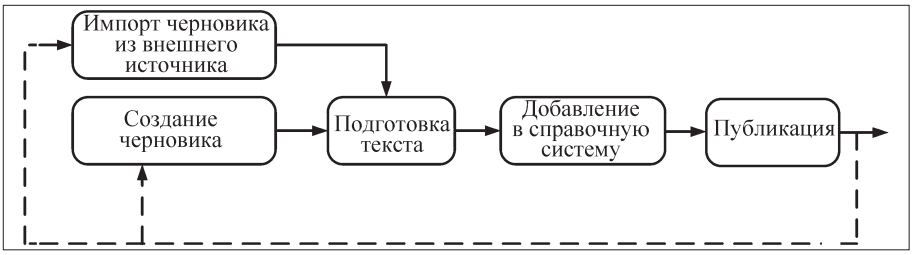


Рис. 2. Жизненный цикл документации

дельные инструменты обработки документа [2]: на стадии разработки черновика – текстовый или XML-редактор, инструмент импорта в систему управления документацией; на стадии подготовки текста – инструменты форматирования, дизайна; на этапе добавления в справочную систему – инструменты индексирования, проверки целостности документа; для публикации требуются инструменты компиляции справочной системы в печатные и веб-форматы.

Все инструменты в том или ином виде интегрированы в систему документирования, которая представляет собой мощную среду разработки технической и пользовательской документации.

Системы документирования

Современные системы документирования можно разделить на два класса:

- Wiki-системы;
- Системы управления документами XML CMS, работающие на основе стандарта DITA.

Эти системы позволяют работать с большими объемами документации. Первая Wiki-система появилась в Интернет в 1995 г. Wiki обеспечивает свободное размещение материала в базе знаний, где навигация по разделам осуществляется в основном с помощью поиска по ключевым словам. Wiki – это веб-ориентированный формат справочной системы, нацеленный на совместную работу над документом. Wiki обеспечивает максимально эффективное взаимодействие с пользователями, дает возможности обратной связи клиентов с разработчиками документируемого продукта. Примером крупнейших Wiki-проектов является ресурс Wikipedia, построенный на основе системы MediaWiki. В настоящее время Wiki-системы активно используются как внутренние корпоративные ресурсы (в компаниях Oracle, VMWare, Яндекс) и развиваются как среды разработки и публикации технической документации. Большое число Wiki-систем разрабатываются в виде open source проектов.

DITA (*Darwin Information Typing Architecture*) – это семантическая модель организации информации [2], согласно которой единица справочной системы – это элемент определенного информационного типа (концепт, задача). DITA требует четкой иерархии справочной информации, что дает возможность эффективного поиска по структуре документа. Для DITA разработаны инструменты публикации в печатные форматы, что делает ее более универсальной. Модель DITA была разработана компанией IBM и представлена в 2001 г. DITA используется в первую очередь в системах, где требуются универсальная публикация документов, поддержка автоматизированной локализации, работа с большими объемами и сложной версионностью контента. Вопросы совместной работы и обратной связи с пользователями документации отодвигаются на второй план. Все инструменты для разработки документации по DITA распространяются по коммерческой лицензии, за исключением open source проекта DITA Open Toolkit. DITA OT разрабатывается при поддержке Geeknet Inc. (SourceForge).

Wiki – это интегрированная, готовая среда создания и публикации документов в Интернет, в то время как DITA-систему нужно собирать из отдельных компонентов: редактора (XML editor); системы управления контентом (XML CMS); публикатора (DITA publisher).

Большинство компонентов для DITA, разрабатываемых разными компаниями, предоставляют программный интерфейс API для интеграции друг с другом. Доступные для аренды XML CMS также имеют встроенный публикатор. Однако необходимость интеграции компонентов делает внедрение DITA более сложным. Запуск DITA-системы оказывается более трудоемким, чем установка Wiki.

Аренда систем документирования

Разработчики Wiki и DITA-ориентированных систем предлагают использовать свои решения по SaaS-схеме.

Wiki, поддерживающие хостинг [3]:

- open source – BusinessWiki, Metadot Wiki, MindTouch, Wagn, Wikidot;
- коммерческие – Confluence, CentralDesktop, EditMe, Incentive, Neticipia, PBWiki, Wikia, Wikispaces.

Из перечисленных систем наиболее полную функциональность, необходимую для разработки документации, предоставляет Confluence компании Atlassian.

Кроме этого, Confluence предоставляется как компонент интегрированной среды JIRA Studio, включающий популярную систему управления задачами JIRA, инструмент непрерывной интеграции Bamboo и др. JIRA Studio распространяется как Google-приложение, на которое можно подписать корпоративный домен Google.

Таблица 3

Затраты на аренду систем документирования, $

Система	Число пользователей									
	В месяц					В год				
	5	10	25	100	500	5	10	25	100	500
Wiki-системы										
Confluence Hosted* (с плагинами Gliffy и Balsamiq)	–	200	400	650	950	–	2 000	4 000	6 500	9 500
Подписка на JIRA Studio для разработчиков документации (включает Confluence)	50	100	250	400	500	500	1 000	2 500	4 000	5 000
CentralDesktop	От 15	От 30	От 75	От 300	От 1 500	От 180	От 360	От 900	От 3 600	От 6 000
EditMe**	От 4,95 за одну учетную запись									
XML CMS для DITA										
Astoria On Demand**	От 12 000					От 144 000				
SiberLogic SiberSafe**	От 250					От 3 000				
DITA Exchange**	От 500					От 6000				

*Требует подписки на JIRA Studio не менее, чем на пять программистов.
**Цена зависит от числа учетных записей, используемых модулей и является предметом индивидуального соглашения.

Из DITA-систем, поддерживающих возможность аренды, доступны следующие XML CMS (все коммерческие) [4]:

- с полной поддержкой SaaS – Astoria On Demand, DITA Exchange, DocZone;
- с опциональной поддержкой – Bluestream XDocs, Siberlogic SiberSafe, Trisoft Infoshare, Vasont, X-Hive Docato.

Опциональная поддержка означает, что данная XML CMS обеспечивает ограниченную функциональность по SaaS-подписке, либо организация хостинга XML CMS обсуждается в каждом случае индивидуально и представляет собой целый проект.

При этом XML CMS – это один из компонентов DITA-системы разработки документации, который требует интеграции с редактором и публикатором контента.

Встроенная поддержка популярных средств разработки и публикации DITA-контента, Arbotext, XMetal, DITA Open Toolkit, реализована во всех CMS, за исключением DITA Exchange.

Для документирования небольших проектов с использованием удаленной системы также применяют сервис Google Docs. Затраты на внедрение инструмента минимальны, Google Docs обеспечивает необходимый на первых этапах функционал для совместной работы над документами, поддерживает популярный формат текстовых документов и электронных таблиц (Microsoft Word и Microsoft Excel).

Стоимость аренды систем документирования

Для сравнения затрат на использование систем документирования по SaaS-схеме можно рассмотреть предложения аренды Wiki-системы Confluence и DITA-ориентированной системы Astoria On Demand.

В табл. 3 представлена стоимость подписки на наиболее мощные и популярные системы документирования для разного числа пользователей на месяц и на год [4]. Нужно учитывать, что XML CMS предназначена для большого числа пользователей (100 и более человек), что входит в оценку стоимости подписки. Из таблицы видно, что Wiki-системы, безусловно, выгоднее использовать в небольших проектах. Как видно, стоимость аренды мощных Wiki-систем при увеличении числа пользователей до 500 человек приближается к затратам на аренду XML CMS, тогда как стоимость лицензий для установки CMS на собственной инфраструктуре значительно выше стоимости лицензий Wiki-систем. Кроме того, среди Wiki-систем существует большое число open source проектов, в то время как свободно распространяемых версий XML CMS не существует.

Заключение

В настоящее время для аренды и использования удаленно через Интернет доступны следующие средства документирования ПО:

- общедоступные Google Apps – Google Docs;
- Wiki-системы – BusinessWiki, Confluence, EditMe;
- Wiki-системы, интегрированные в систему управления задачами: Confluence по подписке на JIRA Studio;
- DITA XML CMS (может потребоваться редактор и публикатор для DITA) – Astoria On Demand, DocZone, DITA Exchange.

Использование средств документирования по SaaS-модели дает следующие преимущества:

- отсутствие затрат на установку системы документирования;
- высокая доступность системы, независимость от платформы;
- улучшение качества документации за счет использования мощных средств с поддержкой контроля качества;
- повышение эффективности документирования (повторное использование контента, единый источник документации, автоматизированная локализация);
- организация масштабируемого процесса документирования.

Благодаря распространению по принципу SaaS, мощные средства документирования становятся доступными для небольших компаний. Однако остаются открытыми вопросы защиты конфиденциальности информации, безопасности работы с документами, отказоустойчивости удаленных систем документирования.

СПИСОК ЛИТЕРАТУРЫ

1. **Hackos J.T.** Information Development: Managing Your Documentation Projects, Portfolio, and People. Indianapolis, Wiley Publishing, Inc. 2007.
2. **Cowan C.** XML in Technical Communication, Institute of Scientific and Technical Communicators (ISTC). Croydon, 2008.
3. **WikiMatrix.** Compare them all. URL: http://wikimatrix.org/ (дата обращения: 4.10.2010)
4. **DITA Newsletter.** The latest news on Darwin Information Typing Architecture XML. URL: http://www.ditanewsletter.com/ (дата обращения: 4.10.2010).
5. **JIRA** Studio Hosted Development Suite. URL: http://www.atlassian.com/hosted/studio/ (дата обращения: 4.10.2010).
6. **Глаголев В.** Разработка технической документации. Руководство для технических писателей и локализаторов ПО. Питер, 2008. 192 с.

УДК 004.056

В.А. Галатенко, д-р физ.-мат. наук, заведующий сектором автоматизации программирования НИИСИ РАН,

e-mail: galat@niisi.msk.ru

К проблеме автоматизации анализа и обработки информационного наполнения безопасности

Рассмотрены основные аспекты Протокола автоматизации анализа и обработки информационного наполнения безопасности (Security Content Automation Protocol, SCAP). Дан обзор протокола SCAP, а также краткая характеристика SCAP-компонентов и рекомендации по использованию протокола SCAP. Также приводится краткий перечень SCAP-совместимых продуктов.

Ключевые слова: *компьютерная безопасность, информационное наполнение безопасности, протокол автоматизации*

Введение

Обеспечение информационной безопасности разнородных, распределенных (как правило, корпоративных) систем – сложная, трудоемкая задача. Ее сложность является, по крайней мере, следствием следующих причин.

- Большое число и разнообразие элементов, информационную безопасность которых требуется обеспечить. Имеется в виду разнообразие аппаратно-программных платформ, большое число и разнообразие программных приложений, разные требования безопасности в различных организациях.
- Необходимость минимизации времени реакции на изменения в информационных системах и их окружении, на появление новых угроз и выявление новых уязвимостей. Согласно работе [1] в 2009 г. в Национальную базу данных об уязвимостях (США) было добавлено более 5700 новых элементов.
- Отсутствие взаимной совместимости. Многие инструментальные средства информационной безопасности (инструментарий для управления программными коррекциями и сканирования уязвимостей) используют собственные форматы, перечни, метрики, терминологию и информационное наполнение.

Необходимыми условиями решения упомянутой выше задачи являются:

- стандартизация;
- автоматизация таких функций, как:
  - реализация и верификация базовых требований безопасности;
  - контроль конфигурационных параметров, влияющих на информационную безопасность;
  - управление программными коррекциями;
  - непрерывный мониторинг состояния информационной безопасности;
  - выявление нарушений информационной безопасности, обнаружение свидетельств компрометации систем.

Для выполнения этих условий и поддержки перечисленных функций был предложен Протокол автоматизации анализа и обработки информационного наполнения безопасности (*Security Content Automation Protocol,* SCAP).

Обзор протокола SCAP

В работе [2] дается следующее техническое определение протокола SCAP: Протокол автоматизации анализа и обработки информационного наполнения безопасности (SCAP) представляет собой совокупность спецификаций, направленных на стандартизацию формата и номенклатуры данных, посредством которых средства информационной безопасности передают сведения об уязвимостях в программном обеспечении и о конфигурации собственно средств обеспечения информационной безопасности (для краткости – конфигурации безопасности).

Протокол SCAP включает две основные составляющие. Во-первых, как протокол, он, как уже отмечалось ранее, содержит набор открытых спецификаций, стандартизующих формат и номенклатуру данных, с использованием которых передаются сведения об уязвимостях в программном обеспечении и о конфигурации безопасности. Каждая спецификация из набора называется SCAP-компонентом.

Во-вторых, SCAP включает в себя стандартизованные эталонные данные об уязвимостях в программном обеспечении и о конфигурации безопасности. Эти данные называются информационным наполнением SCAP.

Компоненты SCAP объединяются в следующие группы:

- перечисление;
- измерение и ранжирование уязвимостей;
- языки выражения и проверки.

Группа "перечисление" включает в себя перечень номенклатур и словарей для представления информации, влияющей на уровень безопасности подконтрольной системы и ее аппаратно-программных компонентов. В эту группу входят:

– общее конфигурационное перечисление (*Common Configuration Enumeration,* CCE) – номенклатура и словарь конфигурационных аспектов систем;

– общее платформное перечисление (*Common Platform Enumeration*, CPE) – номенклатура и словарь имен и версий аппаратно-программных продуктов;

– общие уязвимости (*Common Vulnerabilities and Exposure*s, CVE) – номенклатура и словарь уязвимостей.

Группа "измерение и ранжирование уязвимостей" содержит спецификации для измерения характеристик уязвимостей и генерации на этой основе количественной меры потенциальной опасности (серьезности) выявленных уязвимостей. В версии SCAP 1.0 такая спецификация одна. Это "общая система оценки серьезности уязвимостей" (*Common Vulnerability Scoring System* – CVSS), регламентирующая измерение относительной серьезности (ранжирование) уязвимостей.

Группа "языки выражения и проверки" состоит из XML-схем для спецификации контрольных перечней, генерации отчетов по этим перечням и спецификации низкоуровневых процедур тестирования, используемых в контрольных перечнях.

Таких схем две:

- расширяемый формат описания конфигурационных контрольных перечней (*eXtensible Configuration Checklist Description Format,* XCCDF) – язык для спецификации контрольных перечней и генерации отчетов их применения [3];
- открытый язык уязвимостей и оценки (*Open Vulnerability and Assessment Language,* OVAL) – язык для спецификации низкоуровневых процедур тестирования, используемых в контрольных перечнях.

Почти все перечисленные спецификации поддерживает и развивает компания MITRE Corporation. Лишь за CVSS отвечает Форум групп реагирования на нарушения информационной безопасности (*Forum of Incident Response and Security Teams*, FIRST), а за XCCDF – Агентство национальной безопасности США (*National Security Agency,* NSA) и Национальный институт стандартов и технологий США (*National Institute of Standards and Technology,* NIST).

Эталонные данные – информационное наполнение SCAP – можно получить из нескольких источников. Например, Национальная база данных об уязвимостях (США) (*National Vulnerability Database,* NVD, [4]) хранит словарь элементов CPE и информацию об элементах CVE; MITRE Corporation хранит базу данных OVAL и поддерживает элементы CCE [5].

Краткая характеристика SCAP-компонентов

Категорирование аппаратно-программных ресурсов – необходимое условие формирования режима информационной безопасности и оперативной оценки состояния безопасности. Для выполнения этого условия нужно располагать единой схемой именования ресурсов. Данной цели служат спецификация и словарь "Общее платформное перечисление" (CPE, http://cpe.mitre.org и http://nvd.nist.gov/cpe.cfm).

Универсальный идентификатор ресурсов (*Uniform Resource Identifier*, URI), согласно CPE, имеет следующую структуру:

cpe:/ часть : производитель : продукт : версия : модификация : редакция : язык

Первое содержательное поле "часть" указывает природу ресурса – является ли он аппаратным, базовым или прикладным программным. Последние четыре поля позволяют специфицировать версии и варианты программных продуктов.

Детальную техническую информацию о CPE можно почерпнуть из источников [6–10].

Число конфигурационных параметров современных информационных систем измеряется сотнями, если не тысячами. Многие из них влияют на информационную безопасность. CCE (http://cce.mitre.org) предоставляет стандартный способ описания каждого элемента конфигурации, допустимых диапазонов значений, рекомендуемых установок, а также механизмы для реализации безопасных установок и оценки безопасности имеющейся конфигурации.

Определение CCE-элемента может выглядеть следующим образом [11]:

CCE-3177-3

Определение: Параметр "порог блокирования учетной записи пользователя" должен быть установлен корректно

Параметры: число попыток.

CCE-идентификаторы следует присваивать конфигурационным установкам, допускающим автоматическую интерпретацию и контроль. Например, требование "использовать только сильные пароли" может быть реализовано с помощью последовательности CCE-элементов вида

- параметр "минимальный возраст пароля" должен удовлетворять минимальным требованиям (CCE-2439-8);
- параметр "минимальная длина пароля" должен удовлетворять минимальным требованиям (CCE-2981-9);
- параметр "минимальная сложность пароля" должен быть установлен корректно (CCE-2735-9) и т.п. [11].

CVE – это формат описания известных уязвимостей, с помощью которого создаются новые CVE-идентификаторы, присваивающиеся уязвимостям и на которые затем по мере необходимости ссылаются. Элемент базы данных об уязвимостях может, например, содержать следующую информацию:

CVE-2011-0026

Краткая характеристика: ошибка целочисленной знаковости в функции SQLConnectW в прикладном программном интерфейсе ODBC (odbc32.dll) в Microsoft Data Access Components (MDAC) 2.8 SP1 и SP2 и компонентах Windows Data Access (WDAC) 6.0, позволяющая удаленным злоумышленникам выполнить произвольный код, используя длинную цепочку в имени источника данных (DSN) и специально устроенный аргумент szDSN, минующий знаковое сравнение, ведущая к переполнению буфера (уязвимость "DSN Overflow").

Дата публикации: 12/01/2011

CVSS-серьезность: 9.3 (высокая).

Общая система оценки серьезности уязвимостей [12] предусматривает три составляющие оценки:

- базовую (0 – 10), инвариантную относительно окружения и времени [4];
- зависящую от окружения (0 – 10), определяемую рисками в заданном физическом и логическом окружении информационной системы;
- зависящую от времени (0 – 10), определяемую этапом жизненного цикла уязвимости:
  - уязвимость выявлена;
  - уязвимость обнародована;
  - разработан способ эксплуатации уязвимости;
  - выявлены следы использования уязвимости в информационной системе;
  - выпущены программные коррекции;
  - программные коррекции установлены и т.д.

Комбинация трех составляющих позволяет получить реалистичную оценку серьезности уязвимости в данном месте в данное время.

Для вычисления базовой оценки используются две вспомогательные величины: степень воздействия и сложность эксплуатации.

При вычислении степени воздействия учитывается влияние уязвимости на три основные составляющие информационной безопасности: доступность, целостность, конфиденциальность.

Сложность эксплуатации уязвимости зависит от таких факторов, как способ доступа к атакуемой системе (локальный/удаленный), необходимость аутентификации и т.п.

Открытый язык уязвимостей и оценки (OVAL, http://oval.mitre.org) – это формальный язык на основе XML, который служит для выражения того, как элемент конфигурации, уязвимость и/или программная коррекция могут быть проверены на данной аппаратно-программной платформе.

Спецификация OVAL включает следующие XML-схемы:

- системных характеристик, описывающую объекты и состояния на данной аппаратно-программной платформе;
- OVAL-определений, описывающую, как объекты и состояния системы могут быть оценены;
- OVAL-результатов, используемую при генерации отчетов о результатах применения определений к системным характеристикам.

Например, необходимыми условиями безопасности конфигурации могут быть номер версии библиотеки больший, чем заданная величина, а также определенное значение заданного конфигурационного параметра.

Взаимная совместимость различных инструментов оценки уровня информационной безопасности систем может быть обеспечена путем использования общего информационного наполнения в формате OVAL.

Расширяемый формат описания конфигурационных контрольных перечней (XCCDF, http://scap.nist.gov/specifications/xccdf) – это стандарт, группирующий политику безопасности и конфигурационные установки в единый документ – контрольный перечень для конкретного аппаратно-программного продукта.

Контрольный перечень включает в себя базовые критерии для укрепления информационной безопасности систем посредством надлежащего конфигурирования.

Спецификации XCCDF поддерживают проверку соответствия конфигурации выбранной политике безопасности и генерацию отчетов. В частности, поддерживается отображение политики на отдельные проверки конфигурационных установок системного уровня. Это основа автоматизации при проверке выполнения требований нормативных документов в области информационной безопасности.

Рекомендации по использованию протокола SCAP

Каждый из шести компонентов SCAP обладает своей функциональностью и может использоваться независимо, но наилучших результатов можно добиться путем их совместного применения. Например, в SCAP-совместимых контрольных перечнях используется стандартизованный язык (XCCDF) для описания целевой платформы (CPE) и целевых конфигурационных параметров (CCE).

Верификация конфигурации с точки зрения информационной безопасности – наиболее естественное применение протокола SCAP. Национальная программа контрольных перечней для продуктов информационных технологий (США) [13], несомненно, полезная сама по себе, даст наибольший эффект при внедрении автоматизированных средств верификации и коррекции конфигураций.

Важным на практике частным случаем верификации конфигурации является контроль установленных программных коррекций. Данные о коррекциях в SCAP-формате могли бы поступать от производителей программного обеспечения и использоваться в автоматическом режиме.

Еще один важный частный случай – автоматическое выявление следов успешной компрометации систем. По сути здесь имеется в виду перевод сигнатур атак в SCAP-формат.

Каждая организация, каждая информационная система должна выполнять требования законов и нормативных актов, а также содержательные требования безопасности. Посредством информационного наполнения SCAP-формата можно формировать отображения высокоуровневых требований на низкоуровневые конфигурационные параметры. Например, специалисты NIST разработали SCAP-отображение между низкоуровневыми конфигурационными установками для Windows XP и Windows Vista высокоуровневыми регуляторами безопасности нормативного документа – "*Recommended Security Controls for Federal Information Systems. NIST Special Publication SP 800-53*". Подобная прослеживаемость требований повышает обоснованность выбора конфигурации и служит свидетельством выполнения требований безопасности.

Стандартизация обозначений для относящихся к информационной безопасности сущностей способствует проведению категорирования систем и взаимной совместимости инструментов безопасности. От производителей аппаратно-программных продуктов требуется поддержка этих стандартов.

В условиях ограниченности ресурсов и большого числа выявляемых уязвимостей важную роль играет ранжирование уязвимостей, позволяющее сконцентрировать силы и средства на наиболее острых, критичных проблемах. Наличие количественных, воспроизводимых мер уязвимостей – одно из основных достоинств SCAP.

Наконец, формальное описание статуса информационной безопасности организации и ее систем открывает практически неограниченный простор для аналитических изысканий в области безопасности. В частности, может быть построена и поддерживаться в актуальном состоянии карта информационных ресурсов с оценкой состояния компонентов.

SCAP-совместимые продукты

Параллельно с разработкой и развитием спецификация SCAP организована процедура сертификации аппаратно-программных продуктов на предмет SCAP-совместимости и аккредитация испытательных лабораторий. Перечень SCAP-совместимых продуктов можно найти по адресу [14]. В этом перечне представлены такие компании, как Bigfix, HP, LANDesk, McAfee, Symantec, Tripwire и целый ряд других. В частности, компания Bigfix реализовала инструментарий для конфигурирования безопасности и управления уязвимостями; HP – SCAP-сканер; LANDesk – средства управления программными коррекциями; McAfee – аудитор политик безопасности; Symantec – средства автоматизации анализа рисков; Tripwire – систему Tripwire Enterprise. Это весьма солидный задел. Кроме того, поддержка SCAP заявлена в Linux Fedora Core 14, а контрольные перечни с поддержкой SCAP реализованы для многих вариантов операционной системы MS Windows.

Заключение

Протокол SCAP открывает реальную возможность оценки и обеспечения безопасности состояния корпоративных информационных систем, если будет реализована его поддержка производителями аппаратно-программных продуктов и владельцами систем. Пока SCAP находится на начальном этапе развития, однако перспективы его использования на практике представляются обнадеживающими.

СПИСОК ЛИТЕРАТУРЫ

1. **Quinn S., Scarfone K., Barrett M., Johnson C.** Guide to Adoption and Using the Security Content Automation Protocol (SCAP) Version 1.0. Recommendations of the National Institute of Standards and Technology. U.S. Department of Commerce, National Institute of Standards and Technology, Special Publication 800-117. NIST, July 2010.

2. **Quinn S., Waltermire D., Johnson C., Scarfone K., Banghart J.** The Technical Specification for the Security Content Automation Protocol (SCAP): SCAP Version 1.1 (Draft). Recommendations of the National Institute of Standards and Technology. U.S. Department of Commerce, National Insttute of Standards and Technology, Special Publication 800-126 Revision 1 (Second Public Draft). NIST, May 2010.

3. **Ziring N., Waltermire D.** Specification for the Extensible Configuration Checklist Description Format (XCCDF) Version 1.2 (Draft). U.S. Department of Commerce, National Institute of Standards and Technology, NIST Interagency Report 7275 Revision 4 (Draft). NIST, July 2010.

4. **Национальная** база данных об уязвимостях (США) (National Vulnerability DatabaseNVD). URL:http://nvd.nist.gov.

5. http://cce.mitre.org/lists/cce_list_references.html.

6. **Wunder J., Halbadier A., Waltermire D.** Specification for Asset Identification 1.1 (Draft). U.S. Department of Commerce, National Institute of Standards and Technology, NIST Interagency Report 7693 (Draft). NIST, December 2010.

7. **Halbadier A., Johnson M., Waltermire D.** Specification for the Asset Reporting Format 1.1 (Draft). U.S. Department of Commerce, National Institute of Standards and Technology, NIST Interagency Report 7694 (Draft). NIST, December 2010.

8. **Cheikes B.A., Waltermire D.** Common Platform Enumeration: Naming Specification Version 2.3 (Draft). U.S. Department of Commerce, National Institute of Standards and Technology, NIST Interagency Report 7695 (Draft). NIST, August 2010.

9. **Parmelee M.C., Booth H., Waltermire D.** Common Platform Enumeration: Name Matching Specification Version 2.3 (Draft). U.S. Department of Commerce, National Institute of Standards and Technology, NIST Interagency Report 7696 (Draft). NIST, August 2010.

10. **Cichonski P., Waltermire D.** Common Platform Enumeration: Dictionary Specification Version 2.3 (Draft). U.S. Department of Commerce, National Institute of Standards and Technology, NIST Interagency Report 7697 (Draft). NIST, August 2010.

11. **Mann D.** An introduction to the Common Configuration Enumeration Version 1.7. Mitre Corporation, July, 2008.

12. **Mell P., Scarfone K., Romanosky S.** The Common Vulnerability Scoring System (CVSS) and Its Applicability to Federal Agency Systems. U.S. Department of Commerce, National Institute of Standards and Technology, NIST Interagency Report 7435 (Draft). NIST, August 2007.

13. **Национальная** программа контрольных перечней для продуктов информационных технологий (США) URL: htt://checklists.nist.gov.

14. **Перечень** SCAP-совместимых продуктов. URL: http://nvd.nist.gov/scapproducts.cfm.

CONTENTS

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

Дизайнер *Т.Н. Погорелова*. Технический редактор *Т.И. Андреева*. Корректоры *Л.И. Сажина, Л.Е. Сонюшкина*

Сдано в набор 12.04.11 г. Подписано в печать 24.05.11 г. Формат 60×88 1/8. Бумага офсетная. Печать офсетная. Усл. печ. л. 5,88. Уч.-изд. л. 7,09. Цена свободная.

Отпечатано в ООО "Белый ветер", 115407, г. Москва, Нагатинская наб., д. 54, пом. 4.