

Программная инженерия

Пр
2
2011
ин

Учредитель: Издательство "НОВЫЕ ТЕХНОЛОГИИ"

Издаётся с сентября 2010 г.

Главный редактор
ГУРИЕВ М.А.

Редакционная коллегия:

АВДОШИН С.М.
АНТОНОВ Б.И.
БОСОВ А.В.
ВАСЕНИН В.А.
ГАВРИЛОВ А.В.
ДЗЕГЕЛЁНОК И.И.
ЖУКОВ И.Ю.
КОРНЕЕВ В.В.
КОСТЮХИН К.А.
ЛИПАЕВ В.В.
ЛОКАЕВ А.С.
МАХОРТОВ С.Д.
НАЗИРОВ Р.Р.
НЕЧАЕВ В.В.
НОВИКОВ Е.С.
НОРЕНКОВ И.П.
НУРМИНСКИЙ Е.А.
ПАВЛОВ В.Л.
ПАЛЬЧУНОВ Д.Е.
ПОЗИН Б.А.
РУСАКОВ С.Г.
РЯБОВ Г.Г.
СОРОКИН А.В.
ТЕРЕХОВ А.Н.
ТРУСОВ Б.Г.
ФИЛИМОНОВ Н.Б.
ШУНДЕЕВ А.С.
ЯЗОВ Ю.К.

Редакция:

ЛЫСЕНКО А.В.
ЧУГУНОВА А.В.

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.

СОДЕРЖАНИЕ

Гуриев М.А. Очерк мировой индустрии программного обеспечения	2
Орлик С.В. Программная инженерия и жизненный цикл программных проектов с позиций SWEBOK. Часть 1	6
Казьмин О.О. Преобразование исходного кода в системах динамического распараллеливания программ на основе Т-подхода	10
Шелехов В.И. Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования	14
Крючкова Е.Н., Старолетов С.М. Динамическое тестирование распределенных программных систем на основе автоматных моделей	22
Яблонский С.А. Введение в экосистему "облачных вычислений"	27
Костюк В.В. История с программированием. Его величество Код, вокруг да около него	39

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](http://novtex.ru) E-mail: prin@novtex.ru

© Издательство "Новые технологии", "Программная инженерия", 2011

М.А. Гуриев, д-р техн. наук, проф., директор государственных программ,
IBM-Восточная Европа/Азия

E-mail: marat_guriev@ru.ibm.com

Очерк мировой индустрии программного обеспечения

Проведен анализ текущего состояния мировой индустрии программного обеспечения. Выделены и рассмотрены основные тенденции развития отрасли, такие как движение рабочих мест в сторону локальной экспертизы по актуальным профессиям, передача критического ИТ-функционала на аутсорсинг специализированным сервисным компаниям, рост интереса к свободному программному обеспечению и рост приобретений ведущими компаниями наиболее успешных конкурентов из второго эшелона. Ввиду заметного движения в сторону возрастания требований к качеству как самих программных продуктов, так и к квалификации программистов, а также организации производственных процессов и используемых технологий программирования, делается вывод о возможностях кардинальных изменений в практике программирования.

Ключевые слова: мировая индустрия программного обеспечения (МИПО), тенденции развития МИПО, рейтинг SW-500, рейтинг топ100

Программная инженерия по общепринятому определению [1] тяготеет к индустриальному производству программных продуктов, и основная цель настоящего очерка – бросить взгляд на мировую индустрию программного обеспечения (МИПО) с точки зрения перспектив развития приложений программной инженерии и динамики развития мест приложения труда специалистов в этой области. Автор не претендует на точность сделанных количественных оценок в силу противоречивости находящихся в публичном доступе сведений об оборотах и основных направлениях работ составляющих МИПО компаний, в свою очередь эта противоречивость обусловлена маркетинговыми целями многих публикаций такого рода. Вместе с тем возможные неточности не могут повлиять на качественный анализ структуры и трендов МИПО в интересах программной инженерии.

Разумеется, динамичное многообразие МИПО невозможно даже контурно оценить в одной статье в полной мере, но, поскольку повышение продуктивности человеческой деятельности именно в этом многообразии и составляет предмет программной инженерии, возможно, в будущем журнал будет обращаться к этой теме, в первую очередь, в части российского сегмента мировой индустрии программного обеспечения.

Существующая конфигурация МИПО

Первые десятилетия существования МИПО (50–70-е гг. XX в.) значительную часть индустрии в большинстве лидирующих в этой области стран составляли государственные исследовательские институты и научно-производственные объединения различного типа и формы. Однако затем разработка и производство программ превратились в привлекательную для предпринимательства отрасль.

Действующая конфигурация МИПО сложилась в середине 90-х гг. прошлого века и несет в себе черты иерархической структуры. При этом уровень в иерархии определяется позиционированием и фактическим тиражом производимых продуктов/сервисов в составе практически реализуемых в экономиках и социальной жизни системных проектов внедрения информационных технологий.

Ядром этой структуры принято считать (с некоторыми исключениями) первую десятку производителей ПО и связанных сервисов из рейтинга *The Software 500* (SW-500), публикуемого журналом *Software Magazine* [2] с 1983 г. Поскольку не все из этих производителей заняты исключительно производством ПО/сервисов, рейтинг выделяет только профильные объемы реализации. Последний такой рейтинг датирован ноябрем 2010 г. и содержит данные об объемах продаж ПО за 2009 г. (см. таблицу).

Первые 10 компаний в рейтинге SW-500 2010 г.			
Компания	Доход от продажи ПО/сервисов, млн долл.	Общий доход, млн долл.	Доход от продажи ПО/сервисов, %
IBM	74 934	95 758	78
Microsoft Corporation	50 820	58 573	87
Hewlett-Packard Company	38 265	114 552	33
Oracle Corporation	23 252	23 252	100
Accenture	21 576	23 171	93
Computer Sciences Corporation	16 739	16 739	100
SAP AG	15 295	15 295	100
EMC Corporation	14 025	14 025	100
Hitachi	12 253	96 436	13
Lockheed Martin Corporation	12 130	45 198	27

В этих компаниях к сфере МИПО можно отнести около миллиона рабочих мест. Всего этими производителями реализовано ПО и сервисов на 280 млрд долл., при том что продажи остальных 490 компаний, входящих в рейтинговый список, составили только около 200 млрд долл. Сразу же следует отметить, что очень близкие цифры характеризуют такие же объемы продаж в предыдущем рейтинге (октябрь 2009 г.), т.е. в кризисном 2008 г.

Этот мощный маховик выпуска, а также непрерывной доработки и обновлений достаточно сложных и самых тиражируемых по странам мира в своем классе программ/сервисов, опирающийся на выращенный и устойчивый спрос, охватывающий все экономически активные страны, представляет собой главную гарантию стабильности и устойчивого развития МИПО.

Компании, входящие в ядро, как правило, являются конкурентами по целому ряду сегментов МИПО, но при этом распространена практика, при которой ниже расположенные в рейтинге компании могут одновременно продвигать решения от всех участников первой десятки. Работает и обратная практика, когда одновременно все вендоры первой десятки или любые другие компании одновременно используют со своими технологиями удачное нишевое ПО или сервис от некоторого произвольного стартапа. Результатом такой свободы партнерств и коалиций является истинная свобода конкуренции, без которой МИПО как сложная система просто не смогла бы развиваться. Практика показала, что когда все же возникала монополия в том или ином сегменте индустрии, системная реакция индустрии проявлялась в виде одного или нескольких конкурентов соответствующего масштаба и влияния.

Нельзя не отметить, что в списке первой десятки только SAP и Hitachi выпадают с расположением сво-

их штаб-квартир из географии США. Кстати, это правило 80 % доли американских компаний характерно почти для всех последующих десятков рейтинга, т.е. из 500 компаний около 400 базируются в США. Эта историко-экономическая данность является, как представляется, самой главной технологической и интеллектуальной доминантой Северной Америки в современной цивилизации. Странам БРИК¹ потребуется по крайней мере несколько десятилетий на попытку избавиться от этой доминанты, если таковая попытка возникнет. С другой стороны, географическое размещение подавляющего большинства международных компаний-производителей ПО настолько рассредоточено по странам и континентам, что вопрос размещения штаб-квартир постепенно, но неуклонно теряет свою значимость. Особенно важно отметить рассредоточенность по многим странам коллективов разработчиков, технических писателей и тестеров программных комплексов. Наличие скоростного интернета и средств программной инженерии позволяет этим коллективам продуктивно взаимодействовать с корпоративными центрами разработок и сервисными центрами в рамках хорошо защищенных интранет сетей.

Такой подход к формированию территориально распределенных систем МИПО на ранних этапах корпоративного программирования (в период 1995...2005 гг.) диктовался, в первую очередь, целесообразностью офшорного программирования вследствие значительной разницы в оплате труда разработчиков, имеющей место в разных странах. Однако в предкризисный, кризисный и ранний послекризисный периоды (2006...2010 гг.) руководителям компаний – членов МИПО стало очевидно, что офшорные разработки диктуются в основном производственной целесообразностью, а именно – подключением по всему земному шару дополнительных ресурсов специалистов, способных помочь в непрерывной гонке с конкурентами. Фактор экономии на зарплате при этом быстро отошел в сторону прочих учитываемых факторов.

Некоторые важнейшие тренды развития МИПО

Идея наращивания локального потенциала программной индустрии остается популярной среди правительств многих десятков стран мира со сколь-нибудь заметным ресурсом для поддержки такого роста. Однако заметные результаты удается получить только тем странам, которые взаимодействуют с лидерами МИПО. Результатом такого взаимодействия является определенный компромисс между национальной идеей и общечеловеческой идеей информатизации: глобальная идея реализуется легче, потому что затраты на любую технологию компенсируются за счет глобальных продаж. Локальные технологии, как правило, не окупаются и постоянно нуждаются в дотациях от

¹БРИК – устоявшийся акроним от названия четырех быстроразвивающихся стран: Бразилия, Россия, Индия, Китай (прим. ред.).

государства. Компании МИПО приходят на помощь государственным структурам в контексте сотрудничества в глобально продаваемых технологиях. Следствием стал важный тренд МИПО:

• **Движение рабочих мест в сторону локальной экспертизы по актуальным профессиям.** Этот тренд является следствием продолжающегося усложнения технологий и соответствующего роста требований к уровню образования и навыков сотрудников компаний – членов МИПО. Можно без преувеличения сравнить методы подбора разработчиков лидерами МИПО с приемами сканирования ведущими футбольными и баскетбольными лигами перспективных игроков по всему земному шару. Что еще более важно – указанный тренд, по-видимому, приведет к формированию на основе существующих крупных мультинациональных корпораций подлинных глобально интегрированных предприятий с абсолютно равными правами и возможностями для занятых в них профессионалов независимо от стран расположения штаб-квартир и стран физического проживания этих профессионалов. А это, в свою очередь, снимает остроту тезиса создания национальных ИПО (за исключением неизбежных специализированных работ по доводке рыночных систем под специфические требования оборонных ведомств и спецслужб) и создает препосылки для дальнейшей унификации средств и методов программной инженерии.

Одним из подтверждений рассмотренного тезиса является пример сервисного гиганта Capgemini, открывшего с 11 млрд долл. вторую десятку рассматриваемого рейтинга SW-500 2010. Capgemini имеет штаб-квартиру в Париже, но предоставляет всю палитру связанных с ПО сервисов в 30 странах мира и ее 106 тыс. сотрудников размещены (по убыванию контингентов) в Северной Америке, Европе, Южной Америке и Азиатско-Тихоокеанском регионе. При этом Capgemini партнерствует с SAP, Oracle, HP, IBM, Microsoft.

Очевидно, что Capgemini может быть одним из главных бенефициаров развития межплатформенной интероперабельности решений ее партнеров в программной инженерии, поскольку это позволит унифицировать сервисные схемы на этапе эксплуатации ПО.

Такими же бенефициарами станут замыкающие вторую десятку рассматриваемого рейтинга с пяти миллиардными оборотами крупнейшие индийские сервисные компании Wipro и TATA, работающие с неменьшим географическим покрытием.

Wipro в новостях сообщает об открытии исследовательской лаборатории в Атланте при своем сервисном центре в штате Джорджия, обслуживающем сотни клиентов в США.

TATA в информации о себе подчеркивает, что из ее 160 тыс. консультантов, работающих в 42 странах, 14 тыс. работают в США и более 4 тыс. – в Великобритании.

И еще одна компания из второго десятка заслуживает специального упоминания как самая крупная частная компания в МИПО. Это SunGard Data Systems, Inc, имеющая 20 тыс. сотрудников и оборот более 5 млрд долл. в 70 странах мира и, кроме того, являющаяся доминирующим в США провайдером сис-

тем управления университетами (компания сообщает об охвате ее системами 1600 университетов и колледжей с общим числом студентов 10 млн). Кроме того, решения SunGard признаны на финансовом рынке. Созданная в 1982 г. путем выделения ИТ-подразделения нефтяной компании в самостоятельный бизнес, SunGard стала публичной в 1986 г., однако затем в 2005 г. была выкуплена частными инвесторами.

В целом соотношение между публичными и частными компаниями в составе топ 500 определяется как 54 % публичных к 46 % частных.

Несколько существующих трендов обращения с ИТ в современном предпринимательстве получили свое подтверждение в рейтинге SW-500 2010:

• **Передача критического ИТ-функционала на аутсорсинг специализированным сервисным компаниям продолжилась.** Это подтверждается анализом соответствующих категорий бизнеса. Системные интеграторы и ИТ-консультанты нарастили свои обороты (31 компания); аутсорсеры (17 компаний), объединяющие фирмы в категориях ИТ-сервисы / консалтинг и аутсорсинг развития программных продуктов/тестинг, увеличились в количестве и нарастили оборот.

• **Растет внимание к свободному программному обеспечению.** Устойчивость этого тренда наглядно демонстрирует номер 83 рейтинга – лидирующая на рынке СПО компания Red Hat, Inc., увеличившая свой оборот на 24,8 %.

• **Сохранился заметный тренд приобретений в МИПО:**

- ◆ Adobe приобрела компанию Omniture; CA приобрела NetQoS;
- ◆ Dell приобрела Perot Systems Corp.; IBM приобрела SPSS and ILOG;
- ◆ JDA Software приобрела i2 Technologies;
- ◆ Micro Focus приобрела Borland Software; Open Text приобрела Vignette Corp.

Важно также отметить, что, несмотря на кризис, удержали свои достижения компании – рекордсмены выработки на одного сотрудника. Впереди Innodata Isogen, Inc., № 218 (1,3 млн долл. на человека). На втором месте № 61, Check Point Software Technologies, одна из лидирующих компаний по Объединенному управлению угрозами – Unified Threat Management (1,2 млн долл. на человека) и на третьем – ePlus Inc., № 78 (1,1 млн долл. на человека). Этой компании удалось сформировать эффективные партнерства с ведущими поставщиками технологий (в том числе HP, Cisco, VMware, NetApp, Microsoft, Symantec, IBM, Sun и Lenovo). Одновременно ePlus организовала эффективные партнерства с производителями оборудования (F5, APC, Apple, Citrix, Dell, Emerson, Kingston, Lexmark, OKI и Xerox). Таким образом, получилась одна из самых эффективных схем предпринимательства в МИПО.

Еще один важный тренд не был ликвидирован кризисом.

• **Подавляющее большинство компаний, вошедших в топ SW-500 в 2009 и 2010 гг., инвестировали в поисковые НИОКР.** При этом в 2009 г. в среднем инвестировали 10 % от оборота, т.е. 45 млрд долл. А уже в 2010 г. инвестиции достигли в среднем 15 %, что уже оказа-

лось с учетом выросшего оборота более 73 млрд долл. Эти НИОКР дорогостоят, поскольку нацелены на действительно новые прорывы в агрессивно-конкурентной борьбе за рынок.

При всей значимости и безусловной полезности рассмотренного рейтинга в нем, как правило, обходится вниманием значительное число неамериканских компаний. Например, из российских компаний в рейтинг попала только Лаборатория Касперского и, как представляется, организаторы рейтинга могут оказаться в непростом положении, будучи спрошенными про такие компании, как 1С, АВВЫ, Luxsoft и ряд других вполне результативных российских компаний, безусловно, существующих быть учтенными в рейтинге не в меньшей степени, чем полсотни американских компаний с годовым объемом от 6 до 2 млн долл., занявших в рейтинге места с 451 по 500.

Учитывая такую тенденцию, европейцы практикуют альтернативные рейтинги, с тем чтобы внести свои корректиры и, естественно, поддержать своих производителей. Последний такой рейтинг *Global Top 100 software vendors* (далее – топ100) опубликован в январе 2011 г. [3]. Он подготовлен французской консалтинговой компанией PAC на основе традиционной оценки объемов поставок ПО без сервисной составляющей. Сопоставление результатов с рейтингом SW-500 показало, что первые 8 компаний в обоих рейтингах совпадают, но за ними в рейтинге топ100 в первую десятку попали компании Intuit и Adobe, а во вторую добавились телекоммуникационные компании Cisco и NEC. Однако значимые расхождения состоят в том, что более 20 компаний из рейтинга топ100 с объемами продаж в диапазоне от 449 до 178 млн евро – европейские, американские, японские вообще не попали в рейтинг SW-500, хотя по объективным показателям они должны были располагаться в диапазоне мест с 89 по 143 этого рейтинга. При этом нужно признать, что среди пропущенных компаний европейских оказалось даже меньше, чем американских. Отмеченные нестыковки рейтингования не меняют главного заключения, которое нельзя не сделать по итогам анализа работы МИПО.

В целом рассматриваемый 2009 г. подтвердил более высокую устойчивость МИПО к кризису, чем предполагалось аналитиками. Общий оборот составил 491,7 млрд долл. – с небольшим ростом по сравнению с предыдущим годом (451,2 млрд долл.). Общее число занятых снизилось на 3,92 % до 3 563 407 с 3 707 957 чел. Эти данные позволяют отметить по-прежнему существующую в обществе и деловых кругах недооценку непрерывно растущего значения информационных технологий и, в том числе, МИПО в развитии инфраструктуры экономики и социальной жизни цивилизации. Отрасль остается высококонкурентной с рекордными темпами обновления и модернизации выпускаемой продукции.

Пока трудно определить сроки завершения периода заметного роста МИПО и перехода к более стабильному развитию. Однако все более заметным фактором сдерживания роста индустрии становятся дефицит кадров и низкая производительность труда, при том, что очевидными ускорителями роста являются продолжающийся рост скоростного интернета, расширение сервисных сегментов национальных экономик и разви-

тие социальных сетей. Желательным ответом на складывающуюся ситуацию со стороны программной инженерии могли бы стать структурирование и ранжирование основных проблем (по мнению автора, наиболее полный перечень проблем программной инженерии изложен в работе [4]) и осуществление на этой основе серии технологических прогнозов по соответствующим направлениям, которые в свою очередь могли бы позволить формировать интегрированные программы и планы комплексного перевода программной инженерии на новые уровни продуктивности.

При подготовке этого сжатого обзора были изучены сайты более полутора сотен компаний из рейтингов SW-500 и Top100 SW vendors in Europe. Это позволило почувствовать пульс непрерывной конкурентной гонки "на выживание" внутри программной индустрии. За последнее десятилетие ушли с рынка как самостоятельные вендоры более 100 участников рейтинга SW-500 и их место заняли новые вендоры и сервисные стартапы. Подобные процессы не менее динамично происходят и на локальных рынках по крайней мере 70–80 крупнейших экономик мира. Динамичность определяется самой ролью ПО, функционал которого меняется в среднем на 10...15 процентов каждый год. А пока аналитики крупнейших ИТ-вендоров прогнозируют новый рост, проекции на будущее не дают надежды на снижение темпов изменений.

Главные внутренние составляющие успеха каждой компании из программной индустрии – качество профессионалов и качество менеджмента. Так, в институте программной инженерии Университета Карнеги-Меллона профессора изучают оптимальные способы сочетания непрерывного отслеживания требований архитектуры и командной работы программистов [5]. Самая долгоживущая в составе МИПО компания IBM приурочила к своему столетию завершение поискового исследования на построение действующего на основе массового параллелизма интеллектуального игрока реального времени в телевизионной игре Jeopardi (российский аналог известен под названием "Своя игра") и, тем самым, обозначила новый этап суперкомпьютерной медицинской диагностики в реальном времени с высокоразвитым голосовым интерфейсом [6], который одновременно открывает новую эру интерактивной аналитики в самых разных областях человеческой деятельности. А следовательно, осмыслиенный спрос на программную инженерию, также как на сверхскоростной интернет и настольные суперкомпьютеры, уже обеспечен на два десятилетия вперед. И еще – за эти десятилетия весьма возможны кардинальные изменения в практике программирования, повышение роли прототипирования, построение специализированных интеллектуальных вопрос-ответных систем для нужд программистов и систем тестирования.

СПИСОК ЛИТЕРАТУРЫ

1. http://ru.wikipedia.org/wiki/программная_инженерия
2. <http://www.softwaremag.com/SW500/>
3. <http://www.cnews.ru/news/top/index.shtml?2011/01/21/424097>
4. Липаев В.В. Проблемы программной инженерии: качество, безопасность, риски, экономика // Программная инженерия. 2010. № 1. С. 7–20.
5. <http://www.sei.cmu.edu/library/abstracts/reports/10tr031.cfm>
6. <http://www.computerra.ru/terralab/platform/541734/>

С.В. Орлик, эксперт по архитектуре, Microsoft Россия

E-mail: sorlik@gmail.com

Программная инженерия и жизненный цикл программных проектов с позиций SWEBOk. часть 1

Рассмотрены история и структура свода знаний по программной инженерии SWEBOk, ставшего результатом консенсуса экспертов индустрии программного обеспечения и заложившего основы для использования общего словаря понятий и терминов программной инженерии. Данная работа основана на публично доступном переводе SWEBOk на русский язык, подготовленном автором.

Ключевые слова: SWEBOk, программная инженерия

В конце 90-х гг. прошлого века знания и опыт, которые были накоплены в индустрии программного обеспечения (ПО) за предшествующие 30–35 лет, оформились в дисциплину программной инженерии – *Software Engineering*. В определенной степени, формирование дисциплины программной инженерии на основе развивающихся теоретических разработок и практического опыта напоминает те процессы, которые происходили последние десятилетия в области управления проектами. За это время в индустрии информационных технологий активно возникали и развивались профессиональные ассоциации, специализированные институты, комитеты по стандартизации и другие организации, которые пришли к общему мнению о необходимости сведения профессиональных знаний по соответствующим областям и стандартизации соответствующих программ обучения.

Становление программной инженерии как самостоятельного направления

В 1958 г. всемирно известный статистик Джон Тьюкей (*John Tukey*) впервые ввел термин *software* – программное обеспечение. В 1972 г. IEEE¹ выпустило первый номер *Transactions on Software Engineering* – Труды по Программной Инженерии. Первый целостный взгляд на эту область профессиональной деятельности появился в 1979 г., когда IEEE подготовило стандарт IEEE Std 730 по качеству программного обеспечения. После 7 лет напряженной работы, в 1986 г. IEEE выпустило IEEE Std 1002 "Taxonomy of Software Engineering Standards".

¹IEEE – Computer Society of the Institute for Electrical and Electronic Engineers, IEEE Computer Society (IEEE-CS) – Компьютерное Общество Института инженеров по электротехнике и электронике. <http://www.ieee.org>.

Наконец, в 1990 г. началось планирование всеобъемлющих международных стандартов, в основу которых легли концепции и взгляды стандарта IEEE Std 1074 и результатов работы образованной в 1987 г. совместной комиссии ISO/IEC JTC 1. В 1995 г. группа этой комиссии SC7 "Software Engineering" выпустила первую версию международного стандарта ISO/IEC 12207 "Software Lifecycle Processes". Этот стандарт стал первым опытом создания единого общего взгляда на программную инженерию. Соответствующий национальный стандарт России – ГОСТ Р ИСО/МЭК 12207–99 [1] содержит полный аутентичный перевод текста международного стандарта ISO/IEC 12207–95 (1995 г.).

В свою очередь, IEEE и ACM², начав совместные работы еще в 1993 г. с кодекса этики и профессиональной практики в данной области (*ACM/IEEE-CS Code of Ethics and Professional Practice*), к 2004 г. сформулировали два ключевых описания того, что сегодня мы называем основами программной инженерии – *Software Engineering*:

- *Guide to the Software Engineering Body of Knowledge (SWEBOk), IEEE 2004 Version* – Руководство к Своду Знаний по Программной Инженерии, в дальнейшем просто "SWEBOk" [2].

• *Software Engineering 2004. Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering* – Учебный план для преподавания программной инженерии в вузах (данное название на русском языке представлено вольном смысловом переводе) [3].

Оба стандарта стали результатом консенсуса ведущих представителей индустрии и признанных авторитетов в области программной инженерии – по анало-

²ACM – Association of Computer Machinery – Ассоциация компьютерной техники.

гии с тем, как был создан PMI PMBOK [4]. Так мы пришли к сегодняшнему состоянию программной инженерии как дисциплины.

SWEBOK: Руководство к своду знаний по программной инженерии

С 1993 г. IEEE и ACM координируют свои работы в рамках специального совместного комитета – *Software Engineering Coordinating Committee* (SWECC – <http://www.computer.org/tab/swecc>). Проект SWEBOK был инициирован этим комитетом в 1998 г. Оцененный предположительный объем содержания SWEBOK и другие факторы привели к тому, что было рекомендовано проводить работы по реализации проекта не только силами добровольцев из рядов экспертов индустрии и представителей крупнейших потребителей и производителей программного обеспечения, но и на основе принципа "полной занятости". Базовый комплекс работ в соответствии со специальным контрактом был передан в *Software Engineering Management Research Laboratory* Университета Квебек в Монреале (*Universite du Quebec a Montreal*). Среди компаний, поддержавших этот уникальный проект, были Boeing, MITRE, Raytheon, SAP. В результате проекта, осуществленного при финансовой поддержке этих и других компаний и организаций, а также с учетом его значимости для индустрии, *SWEBOK Advisory Committee* (SWAC) принял решение сделать *SWEBOK 2004 trial edition* общедоступной. В перспективе, в зависимости от финансирования, SWAC считал необходимым законченную версию SWEBOK (изначально планировалось, что она будет готова в 2008 г.) сделать также свободно доступной на сайте проекта – <http://www.swebok.org>. Сегодняшняя "публичность" (общедоступность) результатов проекта стала возможна, в первую очередь, именно благодаря поддержке *SWEBOK Industrial Advisory Board* (IAB) – структуры, объединяющей представителей компаний, поддержавших проект.

Проект SWEBOK планировался в виде трех фаз: *Strawman* ("соломеный человек"), *Stoneman* ("каменный человек") и *Ironman* ("железный человек"). К 2004 г. была выпущена версия Руководства по Своду Знаний третьей фазы – Ironman, т.е. максимально приближенная к окончательному варианту и одобренная IEEE в феврале 2005 г. к публикации в качестве trial-версии. Основная цель текущей "пробной" версии SWEBOK – улучшить представление, целостность и полезность материала руководства на основе сбора и анализа откликов на данную версию с тем, чтобы выпустить финальную редакцию документа в 2008 г. Однако версии 2008 не появилось, хотя ряд дополнений на начало 2010 г. и находится уже в виде драфтов, что не исключает расширения SWEBOK в ближайшей перспективе и, в то же время, не приижает значимости уже выпущенной версии 2004.

По ряду обоснованных причин, "SWEBOK является достаточно консервативным" [2]. После 6 лет непосредственных работ над документом SWEBOK включал "лишь" 10 областей знаний (*knowledge areas*, KA).

При этом, что справедливо и для PMBOK, добавление новых областей знаний в SWEBOK достаточно прозрачно. Все, что для этого необходимо, зрелость (или, по крайней мере, явный и быстрый процесс достижения зрелости) и общепринятость соответствующей области знаний, если это не приведет к серьезному усложнению SWEBOK. (Концепция "общепринятости" – *generally accepted* – определена в IEEE Std 1490-1998, *Adoption of PMI Standard – A Guide to the Project Management Body of Knowledge*).

Важно понимать, что программная инженерия является развивающейся дисциплиной. Более того, данная дисциплина не касается вопросов конкретизации применения тех или иных языков программирования, архитектурных решений или, тем более, рекомендаций, касающихся более или менее распространенных или развивающихся с той или иной степенью активности/заметности технологий (например, web-служб). Руководство к своду знаний, каковым является SWEBOK, включает базовое определение и описание областей знаний (например, конфигурационное управление – *configuration management*) и, безусловно, является недостаточным для охвата всех вопросов, относящихся к вопросам создания программного обеспечения, но, в то же время необходимым для их понимания.

Необходимо отметить, что одной из важнейших целей SWEBOK является именно определение и систематизация тех аспектов деятельности, которые составляют суть профессии инженера-программиста.

Структура и содержание SWEBOK

Описание областей знаний в SWEBOK построено по иерархическому принципу, как результат структурной декомпозиции. Такое иерархическое построение обычно насчитывает два-три уровня детализации, принятых для идентификации тех или иных общепризнанных аспектов программной инженерии. При этом структура декомпозиции областей знаний детализирована только до того уровня, который необходим для понимания природы соответствующих тем и возможности нахождения источников компетенции и других справочных данных и материалов. В принципе, считается, что как таковой "свод знаний" по программной инженерии представлен не в обсуждаемом руководстве (SWEBOK), а в первоисточниках (как указанных в нем, так и представленных за его рамками) [2].

SWEBOK описывает 10 областей знаний (рис. 1, 2):

- *Software requirements* – программные требования.
- *Software design* – дизайн (архитектура).
- *Software construction* – конструирование программного обеспечения.
- *Software testing* – тестирование программного обеспечения.
- *Software maintenance* – эксплуатация (поддержка) программного обеспечения.
- *Software configuration management* – конфигурационное управление.
- *Software engineering management* – управление в программной инженерии.
- *Software engineering process* – процессы программной инженерии.

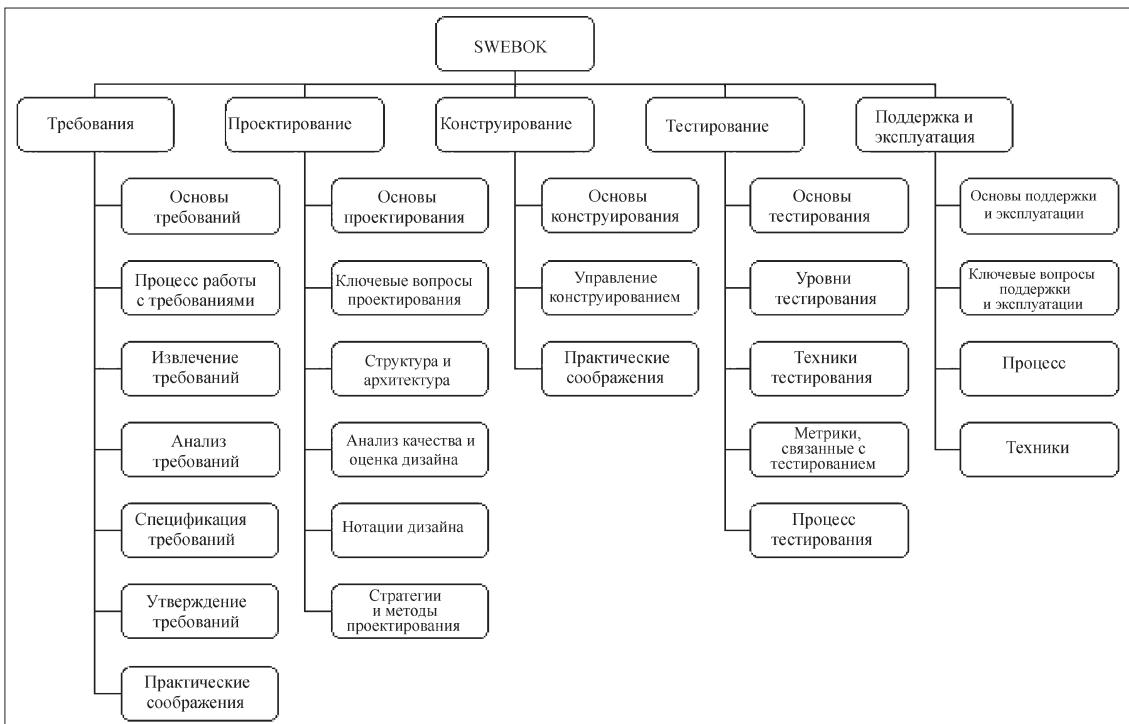


Рис. 1. Первые пять областей знаний SWEBOK (перевод автора на русский язык)

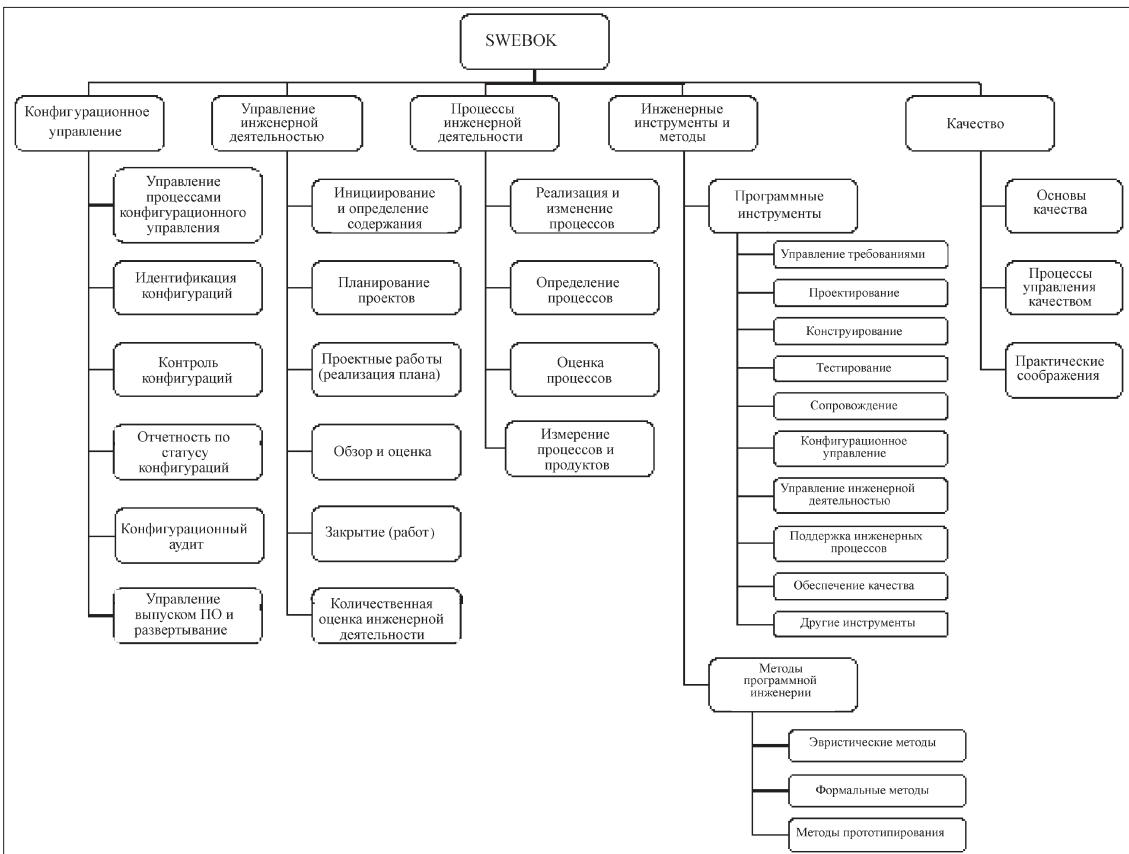


Рис. 2. Вторые пять областей знаний SWEBOK (перевод автора на русский язык)

- *Software engineering tools and methods* – инструменты и методы.
- *Software quality* – качество программного обеспечения.

В дополнение к ним SWEBOK также включает обзор смежных дисциплин, связь с которыми представлена как фундаментальная, важная и обоснованная для программной инженерии (рис. 3):

- Computer engineering.
- Computer science.
- Management.
- Mathematics.
- Project management.
- Quality management.
- Systems engineering.

Стоит отметить, что принятые разграничения между областями знаний, их компонентами (*subareas*) и другими элементами достаточно произвольны. При этом, в отличие от PMBOK, области знаний SWEBOK не включают "входы" и "выходы". В определенной степени такая декомпозиция связана с тем, что SWEBOK не ассоциирован с той или иной моделью (например, жизненного цикла) или методом. Хотя, на первый взгляд, первые пять областей знаний в SWEBOK представлены в традиционной последовательной (каскадной – *waterfall*) модели (см. рис. 1), это не более чем следование принятой последовательности освещения соответствующих тем. Остальные области и структура декомпозиции областей представлены в алфавитном порядке.

Для каждой области знаний SWEBOK описывает ключевые акронимы, представляет область в виде "подобластей" (*subareas*) или, как их часто называют в самом SWEBOK, – "секций" и дает декомпозицию каждой секции в форме списка тем (*topics*) с их описанием.

Перевод SWEBOK на русский язык

К моменту начала работы над сделанным автором переводом SWEBOK в 2004 г. каких-либо даже фрагментарных переводов SWEBOK на русский язык не существовало. В то же время, несмотря на спорность некоторых положений SWEBOK, значимость его для индустрии программного обеспе-

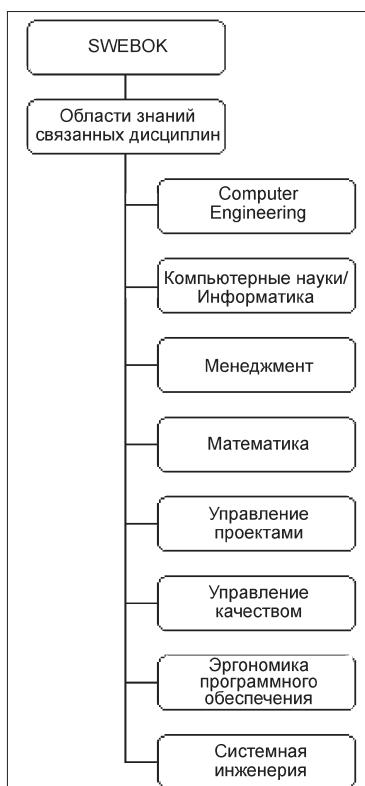


Рис. 3. Область знаний связанных дисциплин (перевод автора на русский язык)

чения как первой коллективной попытки систематизации накопленных знаний просто сложно переоценить. Представленный перевод SWEBOK – Руководство к своду знаний по программной инженерии – необходимо рассматривать как авторский перевод с замечаниями и комментариями ключевых положений SWEBOK. Такой подход ни в коем случае не подменяет оригинального SWEBOK и является всего лишь авторским прочтением последнего.

Представленный автором перевод SWEBOK 2004 сделан по собственной инициативе автора в полном соответствии со SWEBOK Copyright © 2004 by The Institute of Electrical and Electronics Engineers, Inc*. В работе над переводом ряда глав и проработкой комментариев принял участие Юрий Булуй.

Учитывая, что существует ряд неоднозначностей и фактически отсутствует консенсус по соответствующей терминологии на русском языке, автору представлялось полезным использовать в переводе как оригинальные термины на английском языке, так и те их представления на русском языке, которые представляются наиболее адекватными в соответствующем контексте и, в то же время, позволяющие легко сопоставить смысловую нагрузку терминов с первоисточником SWEBOK (см. рисунки).

Также перевод был сопровожден несколькими дополнительными главами, наибольший интерес из которых для профессионалов в области процессной составляющей программной инженерии представляет обзор моделей жизненного цикла разработки программного обеспечения. Автор уделил внимание этой теме в силу ее высокой практической значимости и накопленного в индустрии опыта в части успешно за рекомендовавших процессов разработки ПО, а также соответствующих стандартов, рассматриваемых и активно упоминаемых практически во всех областях знаний SWEBOK.

Обновленная версия перевода, в которую были внесены замечания, поступившие автору после опубликования первой версии, была выпущена зимой 2010 г. в форме специализированного общедоступного авторского web-сайта <http://swebok.sorlik.ru>.

СПИСОК ЛИТЕРАТУРЫ

1. ГОСТ 12207, 1999. Информационная технология. Процессы Жизненного Цикла Программных Средств. ГОСТ Р ИСО/МЭК 12207–99. Государственный Стандарт Российской Федерации. 1999. Госстандарт России, Москва, 2000.
2. Guide to Software Engineering Base of Knowledge (SWEBOK). IEEE Computer Society. 2004. URL: <http://www.swebok.org/>
3. Software Engineering 2004. Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering. A Volume of the Computing Curricula Series. The Joint Task Force on Computing Curricula. IEEE Computer Society, Association for Computing Machinery. 2004. August 23. URL: <http://sites.computer.org/ccse/>
4. A Guide to the Project Management Body of Knowledge. (PMBOK® Guide). Second Edition. Project Management Institute, Inc. 2000. URL: <http://www.pmi.org/>

* "All rights reserved. Copyright and Reprint Permissions: This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy."

О.О. Казьмин, мл. науч. сотр., НИИ механики МГУ им. М.В. Ломоносова
E-mail: nutok@yandex.ru

Преобразование исходного кода в системах динамического распараллеливания программ на основе Т-подхода

Представлены результаты исследований, направленных на создание эффективного препроцессора в рамках разработки системы автоматизированного распараллеливания программ. Дан обзор существующих решений в области статического анализа программ, описаны методы модификации одного из анализаторов для реализации поставленных в работе целей. Рассмотрены методы преобразования программ, применимые в системах распараллеливания на основе Т-подхода.

Ключевые слова: автоматизированное распараллеливание программ, преобразование исходного кода, статический анализ программ, Т-подход, NewTS

Введение

Одним из перспективных подходов к распараллеливанию прикладных программ является автоматизированное динамическое распараллеливание [1]. При его использовании передача данных, синхронизация вычислительных процессов и балансировка вычислительной нагрузки выполняются автоматически, без указаний со стороны пользователя. Автоматизированное динамическое распараллеливание значительно сокращает время разработки приложений, в которых распределить нагрузку между узлами вычислительного поля на этапе реализации кода невозможно или очень сложно.

К настоящему времени разработано несколько средств автоматизированного динамического распараллеливания вычислительных приложений. Большинство из них, например *GUM* [2] и *MultiLisp* [3], работают с программами, написанными на функциональных языках, что позволяет достаточно просто проводить анализ программ, однако ограничивает их производительность. Использование базовых, широко распространенных, императивных языков программирования, таких как С и С++, позволяет добиться высокой эффективности выполнения параллельных программ, но вносит трудности в реализацию механизмов их статического анализа. Рассматриваемый в настоящей работе подход сочетает элементы функционального и императивного програм-

мирования, что позволяет проводить анализ и преобразования кода при высокой эффективности. Описываются методы преобразования программ в рамках системы автоматизированного распараллеливания, основанной на Т-подходе.

Т-подход

Основная идея Т-подхода состоит в попытке соединить преимущества функционального и императивного стиля программирования. В соответствии с положениями функционального стиля программа представляется в виде набора функций, взаимодействующих друг с другом посредством механизмов их вызова и возврата результатов вычислений. Все функции являются чистыми, т.е. не имеют побочных эффектов. Тела функций пишутся на императивных языках программирования, внутри них могут быть использованы любые конструкции таких языков, в том числе – отсутствующие в функциональных языках.

Такое сочетание позволяет, с одной стороны, проводить автоматический анализ и преобразование программ, а с другой – добиться высокой производительности, которая недостижима при использовании только функционального подхода. Сформулируем основные положения Т-подхода более четко.

1. Параллельные программы разрабатываются на широко распространенных языках, таких как FORTRAN, С и С++.

2. Исходные программы пишутся в функциональном стиле.

3. Для описания параллелизма в программу добавляются модификаторы, которые прозрачны для обычного компилятора.

4. Определение того, какие части программы могут работать параллельно, а также их распределение между узлами, выполняется динамически в течение всего времени работы программы.

5. Параллельность достигается путем запуска нескольких функций на разных узлах или процессорах.

Исходя из сформулированных положений, гранулами параллелизма программы являются функции. В рамках рассматриваемого подхода они называются Т-функциями.

Определение 1. *T-функцией* называется функция, написанная и реализованная специальным образом, позволяющим запускать ее отложенно, а именно – параллельно с основной программой.

В ходе исполнения программы может возникнуть ситуация, когда происходит обращение к результату Т-функции в то время, когда она еще не закончила выполнение. В этом случае необходимо указать программе, что такой результат содержит "неготовое" значение. Следовательно, необходимо расширить семантику обычных переменных языка. Для этого引进ится понятие Т-переменной.

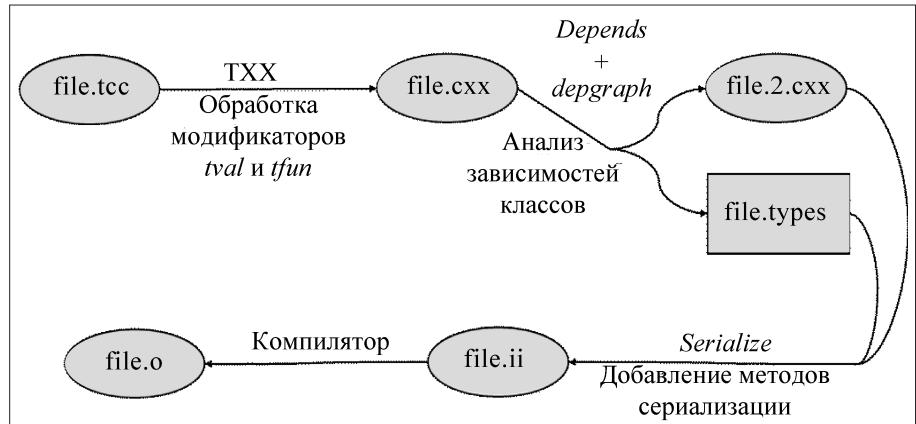
Определение 2. *T-переменной* называется специальная переменная, которая содержит либо готовое, либо неготовое значение.

В программах Т-функции и Т-переменные объявляются с помощью модификаторов *tfun* и *tval*, которые раскрываются при помощи специального препроцессора. Следует отметить, что методы препроцессора, описанные далее, хотя и разрабатывались в рамках последней реализации Т-системы *NewTS* [1], применимы в других версиях систем, основанных на Т-подходе.

Препроцессор Т-системы

Использование Т-подхода предопределяет использование стандартных компиляторов для компиляции пользовательских программ. Однако до компиляции исходный код должен быть обработан специальным препроцессором. Процесс компиляции программы при использовании системы *NewTS* представлен на рисунке.

Компиляция исходного кода происходит в несколько этапов. Первый этап – раскрытие модификаторов Т-системы. Под этим понятием понимается замена объявления переменной с модификатором *tval* или функции с модификатором *tfun* на соответствующий код, написанный на языке C++. Далее выполняется построение графа зависимостей классов и его



Компиляция исходного файла Т-программы

анализ для выявления динамически загружаемых классов и классов, в которые необходимо добавить методы сериализации. Последним этапом работы препроцессора является добавление методов сериализации в выделенные классы. На выходе препроцессора получается код программы, написанный на языке C++, который может быть скомпилирован стандартным компилятором.

На всех этапах работы препроцессора необходима информация о структуре программы, которая выделяется при помощи статического анализа. В силу того, что язык C++ имеет контекстно-зависимую грамматику, реализация эффективного анализатора исходного кода представляет крайне трудоемкую задачу. По этой причине было принято решение модифицировать и применить одно из существующих программных средств статического анализа программ.

Обзор существующих статических анализаторов

Статический анализ исходного кода проходит в три этапа: лексический анализ, синтаксический анализ и семантические действия. В настоящее время существуют эффективные алгоритмы и средства лексического анализа, которые справляются с поставленной задачей. В рамках данной работы эти решения рассматриваться не будут. Подробное их описание можно найти в работах [4, 5]. Алгоритмы синтаксического анализа разработаны уже давно и широко распространены, однако все они накладывают ограничения на грамматику анализируемого языка. Самые распространенные, такие как *LL(k)* и *LR(k)*-алгоритмы, работают только с некоторым подмножеством контекстно-свободных грамматик. Данное условие выполняется в большинстве актуальных задач синтаксического разбора, однако в рассматриваемом случае оно неприемлемо. Грамматика языка C++ является контекстно-зависимой, и несмотря на то что существуют методы адаптации указанных алгоритмов к таким грамматикам, применение их не представляется целесообразным.

Другой подход к синтаксическому анализу состоит в использовании табличных алгоритмов, таких как алгоритм Кока–Янгерса–Касами и алгоритм Эрли [6]. Данные алгоритмы позволяют работать с языками, которые описываются с помощью любых контекстно-свободных и некоторых контекстно-зависимых грамматик. Основным недостатком данных алгоритмов является их сложность – n^3 для алгоритма Кока–Янгерса–Касами и $O(n^3)^*$ для алгоритма Эрли, где n – длина входной строки. Следует отметить, что алгоритм Эрли может быть модифицирован так, чтобы при его исполнении затрачивалось время, линейное для большинства грамматик, которые могут быть проанализированы за линейное время $LL(k)$ и $LR(k)$ -алгоритмами.

В рамках поставленной задачи наиболее эффективным решением является применение алгоритма GLR -анализа (*Generalized LR Parser*) [7]. Данный алгоритм предназначен для разбора по недетерминированым и неоднозначным грамматикам и распознает тот же класс грамматик, что и табличные алгоритмы. Принципы работы данного алгоритма схожи с принципами работы $LR(k)$ -анализа. Главное отличие заключается в том, что вместо обычной стековой памяти алгоритм GLR -анализа использует стековую память графовой структуры и хранит все возможные варианты стеков LR -анализа для данной входной строки. Данное обстоятельство предоставляет широкие возможности по контролю за корректностью разбора при помощи семантического анализа.

Современные средства статического анализа можно разделить на три основные категории: парсеры, генераторы парсеров, библиотеки для их реализации. К последней категории относятся средства, предоставляющие некоторые механизмы лексического и синтаксического анализа. Как правило, они ориентированы на разработку имеющих определенную специфику анализаторов, оптимизаторов компилятора и других языковых средств. Например, библиотека *boost.spirit* нацелена на создание анализаторов непосредственно в коде программы и поэтому оптимизирована для работы с простыми языками. Наиболее подходящее под выдвинутые выше требования средство OpenC++ является библиотекой для создания различных языковых средств для программ на C++. Оно предоставляет основные механизмы статического анализа и поддерживает работу с инстанцированием шаблонов. Однако OpenC++ не разбирает некоторые конструкции языка, такие как пространства имен и переопределение типов, и ориентирован на объектно-ориентированный стиль программирования, что повышает трудоемкость реализации парсера при помощи данного средства. По этим причинам его использование в рамках поставленной задачи не является целесообразным.

* В данном контексте $O(n^3)$ означает то, что при достаточно больших n найдется такая константа C , что время выполнения алгоритма не будет превышать $C \cdot n^3$.

Среди генераторов парсеров подавляющее большинство рассчитано на работу с простыми грамматиками, зачастую разработанными пользователем. Преимущественно в данных средствах используются алгоритмы $LL(k)$ и $LR(k)$ -анализа, что значительно ограничивает возможность их применения для создания анализатора программ на языке C++. Однако существует небольшое число генераторов, основанных на трудоемких, но охватывающих больший класс грамматик, алгоритмах. В рамках данной задачи наибольший интерес представляют два средства: *GNU Bison* и *Elkhound* [8]. Средство *Bison* использует два алгоритма синтаксического анализа, а именно – $LALR(1)$ и GLR . Благодаря наличию GLR -алгоритма разработка необходимого в контексте целей настоящей работы парсера существенно сокращается. При этом совместимость входных данных генератора с языком YACC позволяет сократить разработку грамматики. Средство *Elkhound* основано на гибридном алгоритме $LALR(1)$ и GLR -анализа, что увеличивает производительность генерируемого парсера, с одной стороны, и обеспечивает широкий класс разбираемых грамматик – с другой.

Готовых решений в области синтаксического разбора C++ на сегодняшний день существует немного. В большей степени целям, поставленным в данной работе, соответствуют парсер *Elsa*, реализованный разработчиками генератора *Elkhound*. Данное средство ориентировано на разбор относительно широкого класса конструкций языка с небольшой потерей производительности относительно классических алгоритмов, таких как $LALR$ -анализ. Кроме гибридного алгоритма в данном средстве присутствует развитый семантический анализ, использующийся в автоматическом разрешении неоднозначностей, и разбор инстанцирования шаблонов. Единственным недостатком этого средства является отсутствие поддержки разбора последних версий стандартных библиотек C++. С учетом изложенных обстоятельств, после некоторых модификаций парсер *Elsa* был выбран в качестве основы препроцессора *NewTS*.

Статический анализ в Т-системе

Для раскрытия модификаторов Т-системы необходима семантическая информация о Т-переменной или Т-функции. Подобные данные выявляются при помощи модифицированного парсера *Elsa* [8]. Грамматика разбираемого парсером языка (C++) была расширена новыми ключевыми словами *tfun* и *tval*. Следует отметить, что ключевое слово *tval* имеет синтаксис, отличный от синтаксиса любых конструкций языка C++. Этот модификатор используется только в объявлении переменных и может повторяться неограниченное число раз. Наиболее схожий синтаксис имеет использование указателя (*). Отличие состоит в том, что указатель может применяться вне объявления переменной. В свою очередь, ключевое слово *tfun* синтаксически применяется как флаг объявления функции, примером которых могут служить флаги *virtual* и *inline*.

Соответствующие изменения также были внесены в лексический анализатор, в структуру синтаксического дерева и в механизмы его обхода.

Для генерации эквивалентного кода, заменяющего объявления Т-переменных и Т-функций, необходимо знать тип и имя Т-переменной или тип возвращаемого значения, имена и типы аргументов и имя Т-функции. Благодаря тому, что пользовательские типы данных раскрываются в *Elsa* при построении Абстрактного Синтаксического Дерева (АСД), эта информация может быть получена из абстрактного синтаксического поддерева, соответствующего объявлению переменной или функции. Данный факт предоставляет возможность анализа и замены модификаторов во время трансляции АСД в код на языке C++, что уменьшает накладные расходы на дополнительные обходы дерева.

На этапе анализа зависимостей классов строится граф зависимостей, вершинами которого являются классы и инстанцированные шаблоны, а ребрами – зависимости трех видов: наследование (*inheritance*), включение (*membership*), использование (*usage*). Основной сложностью при построении графа является обработка шаблонов. Шаблонные классы, параметризованные разными аргументами, могут содержать различные данные и, как следствие, требуют отдельной сериализации. Известно, что система шаблонов C++ является алгоритмически полной [9]. По этой причине определение всех инстанцирований шаблонов представляет собой алгоритмически неразрешимую задачу. Принимая во внимание этот факт, было введено несколько ограничений на использование шаблонов. Данные ограничения с одной стороны подходят для широкого класса прикладных приложений, а с другой – позволяют достаточно просто определить список используемых в программе классов. Наиболее важное из этих ограничений состоит в том, что в качестве параметров шаблонов разрешено использовать лишь типы. Другое ограничение состоит в том, что использование внутренних типов не допускается (например, класс $A<T>$ может зависеть от $B<C<T>>$, но не от $B<T::val_type>$).

Для построения графа зависимостей проводится обход АСД слева направо. В процессе обхода выявляются все использования классов, а также определяется, в объявлении каких классов они находятся. Для этого введены несколько значений, которые сохраняются во время обхода: имя рассматриваемого класса; флаг, указывающий на то, что рассматривается объявление поля класса; списки зависимостей.

При определении динамически загружаемых и сериализуемых классов совершаются несколько обходов графа в ширину. Во время первого прохода создается список всех классов, которые требуют динамической загрузки. Поскольку все классы, объекты которых могут быть посланы по сети, в *NewTS* должны наследоваться от некоторого выделенного класса, задача первого прохода состоит в определении наследников этого класса. При выполнении второго прохода опреде-

ляется набор классов, для которых необходимо автоматически сгенерировать методы. Каждый шаг выполняется итеративно. В процессе каждой итерации текущее множество расширяется путем добавления классов-потомков (для списка динамической загрузки) или классов-членов (для определения сериализуемых классов).

Генерация методов сериализации происходит во время трансляции абстрактного синтаксического дерева в код. Во время трансляции происходит обход дерева слева направо сверху вниз. В процессе такого обхода хранится список классов, для которых генерируются методы, а также список полей класса, рассматриваемого в данный момент.

Заключение

В результате исследований в рамках системы *NewTS* были разработаны методы анализа и преобразования исходного кода программ, которые позволяют эффективно использовать преимущества императивного подхода при разработке параллельных программ. Использование императивных языков программирования облегчает перенос существующих последовательных программ на вычислительные комплексы с новой архитектурой и обеспечивает высокую эффективность параллельных программ.

В развитие данной работы предполагается разработка методов препроцессора, оптимизирующих работу пользовательских программ и обеспечивающих их безопасность.

СПИСОК ЛИТЕРАТУРЫ

1. Васенин В.А., Водомеров А.Н., Конев И.М., Степанов Е.А. Т-подход к автоматизированному распараллеливанию программ: идеи, решения, перспективы. М.: Издательство МЦНМО, 2008.
2. Trinder P.W., Hammond K., Mattson J.S. Jr., Partridge A.S., Peyton J.S. GUM: a portable implementation of Haskell // Proc. of conference on Programming Language Design and Implementation. Philadelphia. USA. 1996.
3. Halstead R.H. Implementation of MultiLisp: Lisp on MultiProcessor // Proc. of the 1984 ACM Symposium on LISP and functional programming. Austin. Texas. USA. N.Y.: ACM Press, 1984.
4. Ахо А.В., Сети Р., Ульман Дж.Д. Компиляторы: Принципы, технологии, инструменты. М.: Вильямс, 2003.
5. Appel A.W. Modern Compiler Implementation in ML. Cambridge, New York: Cambridge University Press, 1998.
6. Ахо А.В., Ульман Дж.Д. Теория синтаксического анализа, перевода и компиляции. Т. 1. Синтаксический анализ. М.: Мир, 1978.
7. Grune D., Jacob C. J. H. Parsing techniques: a practical guide. New York.: Springer Science+Business Media, LLC, 2008.
8. McPeak S. Elkhound: A fast, efficient GLR parser generator / Technical Report CSD-02-1214. University of California, Berkeley, 2002.
9. Александреску А. Современное проектирование на C++. М.: Вильямс, 2004.

В.И. Шелехов, канд. техн. наук, зав. лаб., Институт систем информатики им. А.П. Ершова СО РАН, г. Новосибирск

E-mail: vshel@iis.nsk.su

Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования

Описывается технология построения эффективных программ стандартных функций *isqrt*, *floor* и *ilog2*. Корректность программ обеспечивается дедуктивной верификацией и программным синтезом. Условия корректности программ доказываются с помощью системы автоматического доказательства *PVS*.

Ключевые слова: предикатное программирование, дедуктивная верификация, программный синтез, тотальная корректность программы

Введение

Формальная верификация занимает особое место среди других методов верификации программ: она обеспечивает стопроцентную гарантию правильности программы относительно ее спецификации, что недостижимо применением самых совершенных методов тестирования программ. Формальная верификация подразделяется на дедуктивную верификацию и проверку на модели (*model checking*).

Метод проверки на модели предполагает построение модели для программы и автоматическую верификацию этой модели. Метод применим для ограниченного класса программ, допускающих построение модели с конечным (и не чрезмерно большим) числом состояний. Верификация реализуется эффективным перебором состояний.

Дедуктивная верификация гарантирует корректность программы относительно ее спецификации доказательством истинности формул корректности программы. Доказательство проводится с помощью некоторой системы автоматического доказательства. Формулы корректности программы генерируются автоматически по формулам логики и спецификации программы путем применения системы логических правил. Формулы логики программы извлекаются из нее с помощью формальной (денонационной, операционной, аксиоматической, ...) семантики языка программирования. Для современных императивных языков чрезвычайно трудно построить формальную семанти-

ку, однако для языка С формальная семантика разработана [17, 18]. Проведение дедуктивной верификации реальных программ требует больших затрат времени и высокой квалификации коллектива инженеров-верификаторов¹. Как следствие, применение дедуктивной верификации экономически оправдано лишь для программ с высокой ценой ошибки.

Программный синтез реализует логический вывод программы из ее спецификации в рамках некоторой системы автоматического доказательства. Полученная в результате синтеза программа является корректной по построению. По своей силе программный синтез приравнивается к формальной верификации.

Метод разработки программы на базе модели предполагает построение абстрактной модели программы. Корректность модели обеспечивается проведением валидации и верификации. Код программы генерируется из модели в автоматическом или в автоматизированном режиме.

Перечисленные методы формальной верификации, программного синтеза и разработки программы на базе модели относятся к классу *формальных методов*, применяемых при спецификации, разработке и верификации программ.

Несмотря на серьезные трудности применения существующих методов дедуктивной верификации, на

¹"Математик-верификатор" было бы, возможно, более подходящим переводом для "verification engineer".

настоящее время в мире существует более сотни² проектов дедуктивной верификации реальных производственных программных систем. Большие коллектизы инженеров-верификаторов, крупные исследовательские центры по автоматическому доказательству и формальной верификации, специальные фонды и государственная поддержка – все это есть в США, Англии, Франции, Германии Австралии и других странах; в России ничего этого нет.

До сих пор дедуктивная верификация и автоматическое доказательство теорем считаются в России бесперспективными направлениями исследований, тогда как в США, Германии и Англии эти направления интенсивно развиваются начиная с 1980-х гг. Указанные направления не входят в список критических технологий и отсутствуют в перспективных планах Российской академии наук.

Между тем неумолимо приближается время, когда применение формальной верификации станет обязательным требованием сертификации жизненно важных программ. Ведущие банки США и Европы установили это требование для сертификации банковских программ, работающих с пластиковыми картами клиентов, лет пять назад. Космическое агентство NASA применяет дедуктивную верификацию при разработке программ уже более 15 лет, а компании Boeing и Aerobas – почти 10 лет. В последней версии международного стандарта DO-178C для бортовых программ самолетов узаконено применение формальных методов и уточнены условия их использования при сертификации программ [12, 13]. На очереди принятие аналогичных стандартов для программ, используемых в энергетике, медицине и других областях. Впечатляют достижения компании Microsoft, которой удалось провести дедуктивную верификацию сложнейшего компонента операционной системы Windows размером 300 тыс. строк на языке C [14].

Методы программного синтеза и дедуктивной верификации описываются в настоящей статье на примерах разработки быстрых программ для стандартных функций: *floor* – целой части плавающего числа, представленного в бинарном формате в соответствии со стандартом IEEE 754–2008; *isqrt* – целочисленного квадратного корня и *ilog2* – целочисленного двоичного логарифма. Программы указанных стандартных функций на языке C++ используются в известной отечественной системе спутниковой навигации "Навител Навигатор" для автомобилей.

Дедуктивная верификация и синтез предикатных программ определяются в разделе 1 данной статьи. Предлагаемые методы верификации и синтеза базируются на концепции логической семантики и принципиально отличаются от классических методов Флойда и Хоара [15, 16]. Синтез программ целочисленного

квадратного корня и целой части плавающего числа описывается в разделах 2 и 3. Формулы корректности программ стандартных функций были доказаны с помощью известной системы автоматического доказательства PVS [4]. Обзор смежных работ представлен в разделе 4. Опыт разработки программ указанных стандартных функций в режимах верификации и синтеза суммируется в разделе 5.

1. Дедуктивная верификация и программный синтез предикатных программ

Язык предикатного программирования Р (*Predicate programming language*) [1] находится на границе между функциональными и логическими языками. Простейшим оператором языка является вызов предиката $B(x: y)$, где B – имя предиката; x и y – различающиеся наборы переменных. Переменные набора x являются аргументами вызова, а y – результатами; в качестве аргументов допускаются выражения. Базисными операторами языка Р являются: оператор суперпозиции $B(x: z); C(z: y)$; параллельный оператор $B(x: y) \parallel C(x: z)$; условный оператор $\text{if } b \text{ } B(x: y) \text{ else } C(x: y)$, где b – выражение логического типа; x , y и z – различные наборы переменных.

Логическая семантика LS языка Р [2] определяет для каждого оператора S предикат $LS(S)$, истинный после исполнения оператора S. Логическая семантика операторов определяется следующим образом:

$$\begin{aligned} LS(B(x: y)) &\cong B(x, y); \\ LS(B(x: z); C(z: y)) &\cong \exists z. (B(x, z) \& C(z, y)); \\ LS(B(x: y) \parallel C(x: z)) &\cong B(x, y) \& C(x, z); \\ LS(\text{if } b \text{ } B(x: y) \text{ else } C(x: y)) &\cong (b \Rightarrow B(x, y)) \& (\neg b \Rightarrow C(x, y)). \end{aligned}$$

Предикатная программа состоит из набора определений предикатов вида:

$$A(x: y) \equiv \text{pre } P(x) \{S\} \text{post } Q(x, y), \quad (1)$$

где A – имя определяемого предиката, x – аргументы, а y – результаты предиката, S – оператор, $P(x)$ – предусловие, $Q(x, y)$ – постусловие. Предусловие должно быть истинно перед исполнением оператора S, а постусловие – после исполнения. Спецификация предиката A включает предусловие, постусловие и записывается в виде $[P(x), Q(x, y)]$.

Однозначность оператора S, тотальность и однозначность спецификации $[P(x), Q(x, y)]$ определяются, соответственно, формулами:

$$\begin{aligned} P(x) \& LS(S)(x, y_1) \& LS(S)(x, y_2) \Rightarrow y_1 = y_2; \\ P(x) \Rightarrow \exists y. Q(x, y); \\ P(x) \& Q(x, y_1) \& Q(x, y_2) \Rightarrow y_1 = y_2. \end{aligned}$$

²Экспертная оценка автора.

Корректность³ определения предиката (1) со спецификацией $[P(x), Q(x, y)]$ представляется формулой:

$$P(x) \Rightarrow [LS(S)(x, y) \Rightarrow Q(x, y)] \& \exists y. LS(S)(x, y). \quad (2)$$

Доказательство корректности для однозначных программ с однозначной и тотальной спецификацией можно проводить по более простой формуле:

$$P(x) \& Q(x, y) \Rightarrow LS(S)(x, y). \quad (3)$$

Отметим, что доказывать однозначность спецификации и программы не требуется⁴. Достаточно доказать тотальность спецификации. Формула (3) является достаточным условием формулы корректности (2).

Методы дедуктивной верификации и программного синтеза представим на примере оператора суперпозиции в следующем определении предиката:

$$A(x: y) = \text{pre } P(x) \{ B(x: z); C(z: y) \} \text{post } Q(x, y). \quad (4)$$

Предположим, что операторы B и C корректны относительно спецификаций $[P_B(x), Q_B(x, z)]$ и $[P_C(z), Q_C(z, y)]$ соответственно. Пусть спецификация $[P(x), Q(x, y)]$ тотальна. Нетрудно доказать, что корректность предиката A гарантируется в случае истинности правил⁵:

Правило LS1. $P(x) \vdash P_B(x);$

Правило LS2. $P(x) \& Q(x, y) \& Q_B(x, z) \vdash P_C(z) \& Q_C(z, y).$

На базе формулы (3) построена система правил доказательства корректности различных операторов языка P [3]. Правила для других операторов представлены в разделе 2.

Задача верификации. Пусть имеется определение (4). Операторы B и C корректны относительно своих спецификаций. Для доказательства корректности определения (4) требуется доказать тотальность спецификации $[P(x), Q(x, y)]$ и истинность правил LS1 и LS2.

Задача синтеза. Требуется построить программу для предиката $A(x: y)$, представленного спецификацией $[P(x), Q(x, y)]$. Допустим, доказана тотальность спецификации. Пусть для некоторых предикатов $P_B(x)$, $Q_B(x, z)$, $P_C(z)$ и $Q_C(z, y)$ удалось доказать истинность правил LS1 и LS2. Тогда для предиката A синтезируется определение (4) с включением в программу двух новых предикатов, а именно – $B(x: z)$ со спецификацией $[P_B(x), Q_B(x, z)]$ и $C(z: y)$ со спецификацией $[P_C(z), Q_C(z, y)]$. Дальнейшей целью является синтез определений для B и C .

Верификация и синтез зеркальны. Если для предикатной программы сгенерировать формулы корректности, а

³Здесь и далее термин "корректность" используется в смысле тотальной корректности.

⁴Однозначность спецификации следует из ее тотальности и формулы (3). Доказана однозначность программы при условии, что все используемые базисные операции являются однозначными.

⁵Истинность правил определяется как истинность формул, получающихся заменой \vdash на \Rightarrow .

затем по этим формулам провести синтез, то можно получить эквивалентную программу. Наоборот, если на основе спецификации программы и набора формул синтезировать программу, а затем для программы генерировать формулы корректности, то можно получить эквивалентную систему формул. Однако результатом синтеза будет иная программа, чем при обычном программировании. Причина в том, что в процессе программирования неизбежно проводится оптимизация программы. При оптимизации программы меняется ее логика, как правило, в сторону усложнения. Как следствие, верификация программы оказывается более сложной и трудоемкой задачей, чем ее синтез.

Описываемые методы верификации и синтеза реализуются в системе предикатного программирования. Технология предикатного программирования определяет спецификацию, разработку (возможно, в режиме синтеза), верификацию и оптимизирующую трансформацию программ в классе задач дискретной и вычислительной математики. Набор трансформаций применяется к предикатной программе для получения эффективной программы на императивном расширении языка P с последующей конвертацией на любой из императивных языков, включая C, C++, ФОРТРАН и другие. Базовыми трансформациями являются:

- склеивание переменных, реализующее замену нескольких переменных одной;
- замена хвостовой рекурсии циклом;
- подстановка определения предиката на место его вызова;
- кодирование структурных объектов низкоуровневыми структурами с использованием массивов и указателей.

2. Синтез программ вычисления целочисленного квадратного корня

Целая часть квадратного корня есть функция: $m = \text{isqrt}(x) = \text{floor}(\sqrt{x})$, где $z = \text{floor}(t)$ – целая часть вещественного аргумента t со спецификацией $t \leq z < t+1$. Спецификацией для $t = \sqrt{x}$ является формула $t^2 = x$. Таким образом, функция isqrt имеет спецификацию:

formula $\text{isqrt}(\text{nat } x, m) = m^2 \leq x \& x < (m+1)^2;$
 $\text{isqrt}(\text{nat } x : \text{nat } m) \text{ post } \text{isqrt}(x, m);$

Здесь и далее isqrt – имя постусловия, а isqrt – имя предиката.

Синтез программы isqrt управляется обычным процессом построения программы по технологии предикатного программирования. При этом программа извлекается из доказательства формулы корректности (3), т.е. формулы вида: $\text{isqrt}(x, m) \Rightarrow LS(S(x: m))$ для некоторого оператора $S(x: m)$. Кроме того, необходимо доказать тотальность спецификации, а именно лемму:

$\text{isqrt_total: lemma exists nat } m. \text{isqrt}(x, m).$

Сначала синтезируем простейшую программу, определяющую результат m последовательным перебором, начиная с нуля. Для построения программы применяется метод *обобщения исходной задачи* isqrt . Рассмотрим задачу sq0 , определяемую спецификацией:

formula $P_{\text{sq0}}(\text{nat } x, k) = k^2 \Leftarrow x;$

$\text{sq0}(\text{nat } x, k: m) \text{ pre } P_{\text{sq0}}(x, k) \text{ post } \text{Isqrt}(x, m);$

Задача $\text{isqrt}(x : m)$ сводится к $\text{sq0}(x, 0 : m)$, что определяет программу:

$\text{isqrt}(\text{nat } x: \text{nat } m) \{ \text{sq0}(x, 0: m) \} \text{ post } \text{Isqrt}(x, m); \quad (5)$

Корректность программы (5) доказывается по правилу FB1:

Правило FB1. $R(z, y) \vdash P(z) \& Q(z, y).$

Истинность правила гарантирует истинность формулы $R(z, y) \Rightarrow LS(A(z: y))$ при условии, что предикат $A(z: y)$ является корректным относительно однозначной спецификации $[P(z), Q(z, y)]$. Таким образом, требуется доказать лемму:

$\text{Isqrt_fb1: lemma } \text{Isqrt}(x, m) \Rightarrow$ $P_{\text{sq0}}(x, 0) \& \text{Isqrt}(x, m).$ (6)

Разработчик программы должен записать эту лемму и доказать ее, а система автоматически, используя правило FB1 в качестве шаблона, должна синтезировать программу (5) по формуле (6). Следующей целью является синтез программы sq0 из доказательства формулы

$P_{\text{sq0}}(x, k) \& \text{Isqrt}(x, m) \Rightarrow LS(S1(x, k: m))$

для некоторого оператора $S1(x, k: m)$. Отметим, что предусловие $P_{\text{sq0}}(x, k)$ обеспечивает истинность первого конъюнкта в $\text{Isqrt}(x, m)$, а в случае истинности условия $x < (k + 1)^2$ получим решение задачи $\text{sq0}: m = k$. Задача sq0 делится на две: для истинного значения условия $x < (k + 1)^2$ и для ложного, что определяет поиск решения $S1(x, k: m)$ в виде оператора **if** ($x < (k + 1)^2$) $m = k$ **else** $B(x, k: m)$ для некоторого $B(x, k: m)$. Корректность условного оператора доказывается правилом:

Правило FC. $R(z, y) \vdash (C \Rightarrow LS(A(z: y))) \& (\neg C \Rightarrow LS(B(z: y))).$

Истинность правила гарантирует истинность $R(z, y) \Rightarrow LS(\text{if } (C) A(z: y) \text{ else } B(z: y))$. Заметим, что ложное условие $x < (k + 1)^2$ эквивалентно $P_{\text{sq0}}(x, k+1)$, а поскольку истинно $\text{Isqrt}(x, m)$, то решение $B(x, k: m)$ следует определить в виде рекурсивного вызова $\text{sq0}(x, k + 1: m)$. Доказательство корректности рекурсивного вызова предиката реализуется правилом FB3, аналогичным FB1, с дополнительным условием: аргументы рекурсивного вызова должны быть по некоторой мере строго меньше аргументов определяемого рекурсивного предиката, где мера —

натуральная функция от значений аргументов. В нашем случае мера e определяется формулой:

formula $e(\text{nat } x, k: \text{nat}) = (x < (k + 1)^2) ? 0 : x - k^2;$

Таким образом, корректность условного оператора с рекурсивным вызовом определяется леммой:

$\text{sq0_rec: lemma } P_{\text{sq0}}(x, k) \& \text{Isqrt}(x, m) \Rightarrow$
 $(x < (k + 1)^2) ? m = k :$
 $e(x, k + 1) < e(x, k) \&$
 $P_{\text{sq0}}(x, k + 1) \& \text{Isqrt}(x, m).$

По данной формуле, используя в качестве шаблонов правила FC и FB3, система должна автоматически синтезировать рекурсивную программу:

$\text{sq0}(\text{nat } x, k : \text{nat } m) \text{ pre } P_{\text{sq0}}(x, k)$
 $\{ \text{if } (x < (k + 1)^2) m = k \text{ else } \text{sq0}(x, k + 1: m) \}$
 $\text{post } \text{Isqrt}(x, m);$

Заметным недостатком алгоритма sq0 является вычисление квадрата в выражении $(k + 1)^2$. Поскольку $(k + 1)^2 = p = k^2 + 2 * k + 1$, то можно использовать значение k^2 , вычисленное на предыдущем шаге, введением дополнительного параметра $n = k^2$. Аналогичным образом, введя параметр $d = 2 * k + 1$, можно заменить умножение сложением. Далее, поскольку $x < (k + 1)^2$ эквивалентно $x - k^2 < 2 * k + 1$, то вместо x и n можно использовать один параметр $y = x - k^2$. Таким образом, переходим к более общей задаче, определяемой спецификацией предиката sq1 :

formula $P_{\text{sq1}}(\text{nat } x, y, k, d) =$
 $k^2 \leq x \& d = 2 * k + 1 \& y = x - k^2;$
 $\text{sq1}(\text{nat } x, y, k, d : \text{nat } m)$
 $\text{pre } P_{\text{sq1}}(x, y, k, d) \text{ post } \text{Isqrt}(x, m);$

Результатом синтеза, проводимого по той же схеме, является программа:

$\text{isqrt}(\text{nat } x : \text{nat } m) \{ \text{sq1}(x, x, 0, 1: m) \}$
 $\text{post } \text{Isqrt}(x, m);$
 $\text{sq1}(\text{nat } x, y, k, d : \text{nat } m) \text{ pre } P_{\text{sq1}}(x, y, k, d)$
 $\{ \text{if } (y < d) m = k$
 $\text{else } \text{sq1}(x, y - d, k + 1, d + 2: m)$
 $\} \text{ post } \text{Isqrt}(x, m);$

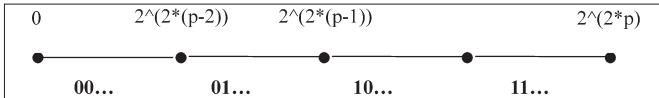
Программа sq1 неэффективна: число шагов равно значению квадратного корня. Более эффективные алгоритмы имеют логарифмическую оценку. Рассмотрим алгоритм на базе двоичного представления аргумента, предполагая, что аргумент n ограничен значением 2^{2^p} для некоторого $p > 0$.

Представим исходную задачу спецификацией:

formula $P_{\text{sqp}}(\text{nat } n, p) = p > 0 \& n < 2^{(2^p)}$
 $\text{isqrt1}(\text{nat } n, p : \text{nat } s) \text{ pre } P_{\text{sqp}}(n, p) \text{ post } \text{Isqrt}(n, s);$

Алгоритм определяет цифры результата, начиная со старшей и далее. Результат s имеет не более p дво-

ичных цифр. Значение старшей цифры равно 0, если $n < 2^{(2*(p-1))}$ и 1 – в противном случае. Следующая цифра есть 0, если $n < 2^{(2*(p-2))}$ и 1 – в противном случае (см. рисунок).



Допустим, для результата s найдено очередное приближение q в виде старших двоичных цифр с номерами от p до k . Тогда должно выполняться условие $q^2 \leq n < (q + 2^k)^2$. Это определяет спецификацию обобщенной задачи:

```
formula P_sq2(nat n, p, k, q) =
    P_sqp(n, p) & k <= p &
    q^2 <= n & n < (q + 2^k)^2;
sq2(nat n, p, k, q : nat s) pre P_sq2(n, p, k, q)
    post Isqrt(n, s);
```

Справедлива следующая лемма:

```
Isqrt1_fb1:lemma P_sqp(n,p)&Isqrt(n, s) =>
    P_sq2(n, p, p, 0) & Isqrt(n, s).
```

По данной формуле, используя правило FB1 в качестве шаблона, система синтезирует:

```
isqrt1(nat n, p: nat s) pre P_sqp(n, p)
{ sq2(n, p, p, 0: s) } post Isqrt(n, s);
```

Заметим, что при $k = 0$ решением задачи $sq2$ будет $s = q$. Если в s цифра с номером $k-1$ есть 1, то следующим приближением для q будет $q + 2^{(k-1)}$. Если $n < (q + 2^{(k-1)})^2$, то цифра с номером $k-1$ есть 0, иначе 1. Данные утверждения суммируем в виде леммы:

```
formula e2(k: nat) = k;
sq2_rec: lemma P_sq2(n, p, k, q) & Isqrt(n, s) =>
    (k = 0)? s = q :
    (n < (q + 2^{(k-1)})^2)?
        e2(k-1) < e2(k) & P_sq2(n, p, k-1, q)
        & Isqrt(n, s) :
    e2(k-1) < e2(k) & P_sq2(n, p, k-1, q + 2^{(k-1)})
        & Isqrt(n, s).
```

Разработчик должен доказать лемму, а система, используя в качестве шаблонов правила FC и FB3, должна автоматически синтезировать программу:

```
sq2(nat n, p, q, k: nat s) pre P_sq2(n, p, k, q)
{ if (k = 0) s = q
  else if (n < (q + 2^{(k-1)})^2 )
    sq2(n, p, k - 1, q : s)
  else sq2(n, p, k - 1, q + 2^{(k-1)} : s)
} post Isqrt(n, s);
```

В выражении $q + 2^{(k-1)}$ можно заменить "+" побитовой операцией "or". Чтобы обеспечить корректность такой замены, необходимо доказать лемму:

```
or_eq_plus: lemma
  k > 0 & k < p & mod(q, 2^k) = 0 & q < 2^p =>
  (q or 2^{(k-1)}) = q + 2^{(k-1)}.
```

Необходимое условие $mod(q, 2^k) = 0$ проще обеспечить включением его в предусловие. Условие $n < (q + 2^{(k-1)})^2$ эквивалентно: $n - q^2 < 2^{((k-1)*2)} +$

$+ q * 2^k$. Используя технику построения алгоритма sq1, обозначим $y = n - q^2$. Далее можно синтезировать программу с данными оптимизациями:

```
formula P_sq3(nat n, p, k, q, y) =
    P_sq2(n, p, k, q) & y = n - q^2 &
    mod(q, 2^k) = 0 ;
isqrt1(nat n, p: nat s) pre P_sqp(n, p)
{ sq3(n, p, p, 0, n: s) }
post Isqrt(n, s);
sq3(nat n, p, k, q, y: nat s)
pre P_sq3(n, p, k, q, y)
{ if (k = 0) s = q
  else { nat t = 2^{((k-1)*2)} + q * 2^k;
    if (y < t) sq3(n, p, k - 1, q, y: s)
    else sq3(n, p, k - 1, q or 2^{(k-1)}, y - t : s)
  }
} post Isqrt(n, s);
```

Для рассматриваемой предикатной программы применяется последовательность трансформаций. На первом этапе реализуются склеивания переменных в $sq3$: $n \leftarrow y$; $s \leftarrow q$. На втором этапе хвостовая рекурсия заменяется циклом. На третьем этапе тело определения $sq3$ поставляется на место вызова в $isqrt1$ и реализуются упрощения. В итоге получим программу на императивном расширении языка P:

```
isqrt1(nat n, p: nat s) {
  s = 0;
  for (nat k = p; k = 0; k = k - 1) {
    nat t = 2^{((k-1)*2)} + s * 2^k;
    if (n >= t) { n = n - t; s = s or 2^{(k-1)}; }
  }
}
```

Далее, умножение на степень двойки заменяется операцией сдвига влево. Для конкретного $p = 16$ проведем развертку цикла, представив тело цикла макросом $sq(k)$. Специализируем первый и последний шаг цикла. Получим окончательную программу:

```
isqrt1(nat n, p: nat s) {
  sq(nat k) =
  { t = 2^{((k-1)*2)} + (s << k);
    if (n >= t) { n = n - t; s = s or (1 <(k-1)) }
  };
  s = 0;
  if (n >= 2^{((p-1)*2)})
    { n = n - 2^{((p-1)*2)}; s = 2^{(p-1)} };
  sq(p - 1); ...; sq(2);
  if (n >= s * 2) s = s or 1;
}
```

По сравнению с программой, представленной на сайте <http://www.finesse.demon.co.uk/steven/sqrt.html>, параметр k смешен на единицу. Однако поскольку тело $sq(k)$ подставляется открыто на место вызовов $sq(p - 1)$, ..., $sq(2)$, то обе программы идентичны.

3. Генерация программы вычисления целой части плавающего числа

Функцию $y = \text{floor}(x)$, вычисляющую наименьшее целое, не превосходящее вещественного аргумента x , можно определить спецификацией:

```

formula flip(real x, int y) = y <= x & x < y + 1;
floor(real x: int y) post flip(x, y);

Требуется построить быстрый алгоритм вычисления floor для чисел в плавающем формате, который определяется стандартом IEEE 754–2008. Плавающее число представляется тройкой (S, E, T), где S = {0, 1} – знак числа, E – сдвинутая экспонента, а T – мантисса в нормализованном представлении без старшего разряда (равнного 1). Значение числа определяется по формуле:

```

$$z = (-1)^S \cdot 2^{E-\text{bias}} \cdot (1 + 2^{1-p} \cdot T).$$

Здесь bias – сдвиг экспоненты, p – число битов в представлении мантиссы, $1 \leq E \leq 2^w - 2$, w – число битов в представлении экспоненты. Значения $E = 0$ и $E = 2^w - 1$ предназначены для кодирования нулей и специальных значений. Значения констант для формата 32 (**float**): bias = 127, w = 8 и p = 24; для формата 64 (**double**): bias = 1023, w = 11 и p = 53. Данные константы определим в программе описаниями:

```

const nat bias;
const subtype (nat i: i >= 2) w; // w >= 2;
const subtype (nat i: i > 0) p; // p > 0;
const nat m = p + w;

```

// число битов в представлении чисел x и y.

Здесь m – число битов в представлении плавающего числа. Определим типы для S, E и T:

```

type bit = {nat a | a = 0 or a = 1};
type nate = {nat e | 1 <= e & e <= 2^w - 2};
type natp = {nat i | i < 2^(p-1)};
type intm = {int i | -2^(m-1) <= i & i < 2^(m-1)};

```

Результат у должен иметь тип intm той же разрядности, что и аргумент x; например, типу **float** должен соответствовать **int32**. Определим спецификацию функции floor:

```

formula val(bit S, nate E, natp T: real) =
    (-1)^S * 2^(E - bias) * (1 + 2^(1-p) * T);
formula Floor(bit S, nate E, natp T, intm y) =
    flip(val(S, E, T), y);
floor(bit S, nate E, natp T: intm y);
post Floor(S, E, T, y)

```

Необходимо доказать тотальность спецификации, т.е. лемму:

Floor_total: lemma exists intm y. Floor(S, E, T, y).

Оказывается, доказать эту лемму невозможно. Причина в том, что для больших E значение y, удовлетворяющее предикату Floor, может выйти за пределы типа intm. Например, для формата m = 32 результат у может не поместиться даже в формате 64. Значение функции val ограничено по модулю значением $2^E \cdot 2^{(p-1)}$, поскольку $1 + 2^{(1-p)} \cdot T < 2$. Отсюда следует правильное определение типа nate, для которого лемма **Floor_total** доказуема:

```
type nate = {nat e | 1 <= e & e < bias + m - 2};
```

Введем функцию val1:

```
formula val1(bit S, int d, nat z: real) = (-1)^S * 2^d * z;
```

Тогда $\text{val}(S, E, T) = \text{val1}(S, E - \text{bias} - p + 1, 2^{(p-1)} + T)$. Вычисление функции floor можно свести к вычислению функции floor1 со спецификацией:

```

formula P_f11(int d, nat z) =
    z >= 2^(p-1) & z < 2^p & d < w;
formula Floor1(bit S, int d, nat z, intm y) =
    flip(val1(S, d, z), y);
floor1(bit S, int d, nat z: intm y)
pre P_f11(d, z) post Floor1(S, d, z, y);

```

Сначала следует доказать тотальность спецификации floor1, а именно – лемму:

Floor1_total: lemma

$P_f11(d, z) \Rightarrow \exists \text{intm } y. \text{Floor1}(S, d, z, y)$.

Сведение floor к floor1 реализуется доказательством леммы:

Floor_fb1: lemma

```

Floor(S, E, T, y) &
    d = E - bias - p + 1 & z = 2^(p-1) + T \Rightarrow
        P_f11(d, z) & Floor1(S, d, z, y).

```

По лемме генерируется следующее определение предиката floor:

```

floor(bit S, nate E, natp T: intm y)
    { floor1(S, E - bias - p + 1, 2^(p-1) + T: y) }
post Floor(S, E, T, y);

```

Алгоритм для floor1 реализуется разбором случаев на базе леммы

```

Floor1_def: lemma P_f11(d, z) & Floor1(S, d, z, y) \Rightarrow
    (S = 0)? ( (d >= 0)? y = 2^d * z :
        y = div(z, 2^{(-d)}) ) :
    (d >= 0)? y = -2^d * z :
        (mod(z, 2^{(-d)}) = 0)?
            y = -div(z, 2^{(-d)}) :
            y = -div(z, 2^{(-d)}) - 1.

```

Здесь div – деление нацело, а mod – деление по модулю. Программа для floor1 генерируется по лемме очевидным образом. Следует отметить, что результат вычисления $y = 2^d * z$ (и $y = -2^d * z$) может оказаться неточным, если неточным было исходное представление числа в плавающем формате. Погрешность может достигать 2^d .

Использование битовых операций над целыми позволяет заменить ряд конструкций программы более эффективными. Применяются следующие замены:

- $2^{(p-1)} + T$ на $2^{(p-1)} \text{ or } T$;
- $z * 2^d$ на $z << d$;
- $\text{div}(z, 2^{(-d)})$ на $z >> (-d)$;
- $\text{mod}(z, 2^{(-d)}) = 0$ на $(z \& (2^{(-d)} - 1)) = 0$.

Значение $2^{(-d)} - 1$ состоит из (-d) единичных битов. Такое значение можно сгенерировать посредством сдвига: $(2^p - 1) >> (p + d)$. Корректность указанных замен обеспечивается доказательством следующих лемм:

```

plus_eq: lemma T < 2^(p-1) \Rightarrow 2^(p-1) + T = 2^(p-1) or T;
sh_left: lemma d >= 0 & d < p \Rightarrow z * 2^d = z << d;
sh_right: lemma g > 0 & g < p \Rightarrow div(z, 2^g) = z >> g;
mod_eq: lemma g > 0 & mod(z, 2^g) = 0 \Rightarrow (z \& (2^g - 1)) = 0;
sh_mask: lemma g > 0 & g < p \Rightarrow 2^g - 1 = (2^p - 1) >> (p - g);

```

С учетом указанных замен следует модифицировать лемму **Floor1_def** и получить по ней новую версию программы floor1. Далее тело floor1 подставляется на место вызова в floor. Приведем окончательную про-

граммой, дополненную вычислением `floor` для нулевого аргумента, но не допускающую других специальных значений в качестве аргумента:

```
floor(bit S, nat E, natp T: intm y)
{ int d = E - bias - p + 1; nat z = 2^(p-1) or T;
  if (E = 0 & T = 0) y = 06
  else if (S = 0)
    {if (d >= 0) y = z << d else
     if (-d >= p) y = 0 else
       y = z >> (-d) }
  else if (d >= 0) y = - (z << d)
  else if (-d >= p) y = - 1
  else if ((z & ((2^p - 1) >> (p + d))) = 0 )
    y = - (z >> (-d))
  else y = - (z >> (-d)) - 1
};
```

4. Обзор работ

Методы верификации и синтеза, предлагаемые в настоящей работе, базируются на концепции логической семантики и принципиально отличаются от классических методов верификации Флойда и Хоара [15, 16], определяющих построение формул частичной корректности императивных программ.

Наиболее популярным методом синтеза является извлечение программы из конструктивного (интуиционистского) доказательства (метод *proofs-as-programs*) [6] с применением правил резолюции и унификации. Синтез предикатных программ проводится в классической логике и, в соответствии с обзором [5], реализуется по шаблонам формул (метод *schema-guided*); в качестве шаблонов используются правила доказательства корректности. Метод шаблонов применяется в основном для синтеза логических программ. В работе [10] метод шаблонов применяется для синтеза императивных программ. В предикативном программировании Э. Хехнера [7] программа синтезируется как результат последовательных уточнений (*refinements*):

$$Q(x, y) \Leftarrow Q_1(x, y) \Leftarrow Q_2(x, y) \Leftarrow \dots \Leftarrow Q_n(x, y) \Leftarrow LS(S)(x: y).$$

Однако вывод $LS(S)(x: y) \Rightarrow Q(x, y)$ гарантирует лишь частичную корректность программы S . Данная формула – часть универсальной формулы корректности (2).

В работе [9] представлен синтез (методом *proofs-as-programs*) трех программ вычисления целочисленного квадратного корня применением трех разных схем доказательства по индукции теоремы тотальности: $\forall x \exists m. lsqrt(x, m)$. В отличие от описанного в работе [9] метод синтеза, предлагаемый в настоящей работе, является универсальным и позволяет синтезировать произвольный алгоритм. Построение эффективного алгоритма вычисления целочисленного квадратного корня, модификации алгоритма `sq3`, описано в работе [8]. Исходная программа на языке *Standard ML* верифицирована автоматическим доказательством условий корректности в системе *Nuprl*. Ко-

⁶Из стандарта не очевидно, что $floor(-0)=0$

нечная программа – результат серии оптимизирующих трансформаций, переводящих программу на язык *HML*, а затем на язык интегральных схем. Корректность трансформаций доказывается в *Nuprl*. Версия алгоритма в разделе 2 сопоставима с реализацией, описанной в работе [8]. Ее можно было бы оттранслировать на язык интегральных схем. Однако процесс построения программы в разделе 2 представлен на одном языковом уровне и значительно проще, чем в работе [8].

5. Опыт дедуктивной верификации и синтеза программ

Первоначально каждая из трех стандартных функций была запрограммирована на языке *P*, причем для `isqrt` и `ilog2` реализовано по три версии алгоритмов разной сложности и эффективности. Наиболее эффективные версии алгоритмов преобразованы на язык *C++* применением оптимизирующих трансформаций. Итоговые программы для функций `floor` и `isqrt` оказались почти тождественными с исходными. Программа для функции `ilog2` существенно быстрее исходной программы на *C++*.

Далее для каждой программы на языке *P* были сгенерированы условия корректности в соответствии с системой правил для однозначных спецификаций. Сгенерированные формулы оттранслированы на язык спецификаций известной системы автоматического доказательства *PVS* [4]. Доказательство на *PVS* оказалось нетривиальным и трудоемким процессом, требующим существенно больше времени по сравнению с программированием без формальной верификации. Трудности возникали не только при доказательстве самих условий корректности, сколько при доказательстве используемых математических свойств, особенно для битовой арифметики.

В процессе верификации обнаружено пять ошибок. Из них четыре – чисто технические. Принципиальной является ошибка, обнаруженная для функции `floor`: результат функции `floor` для аргумента в формате 32 может не поместиться в формате 32. Это опасная ошибка, приводящая к неверным вычислениям.

Позднее в исследовательских целях был проведен программный синтез всех алгоритмов для трех функций. Оказалось, что в режиме синтеза процесс построения соответствующих теорий и логического вывода в системе *PVS* реализуется более естественным образом, более вариативно и с большей согласованностью между теориями. Это дает возможность сократить суммарный объем доказательств примерно в полтора раза по сравнению с режимом верификации. Работа в режиме синтеза оказывается более предпочтительной.

Методы верификации и синтеза, предлагаемые в настоящей статье, опробованы примерно для 20 небольших программ. Разработанный алгоритм генерации формул корректности специализирован для разных видов операторов языка *P*, что обеспечивает построение адекватного набора формул корректности практически идеальным образом. Узким местом дедуктивной верификации остается автоматическое до-

казательство формул корректности. Доказательство на PVS формул корректности оказалось сложным и требовало времени в 7–10 раз больше времени обычной разработки программы без формальной верификации. Разумеется, современные системы автоматического доказательства, такие как *Z3* и *CVC3*, базирующиеся на мощных SMT-решателях, позволят ускорить процесс доказательства.

Представленные методы верификации и синтеза целесообразны для применения в приложениях с высокими требованиями к надежности и безопасности программ.

Пятый год в Новосибирском государственном университете читается курс по предикатному программированию. Издано учебное пособие [11], которое может быть также использовано для подготовки инженеров-верификаторов. Опыт преподавания курса показал, что сопутствующей нетривиальной задачей является обучение студентов методам формальной спецификации программ.

СПИСОК ЛИТЕРАТУРЫ

1. **Карнаухов Н.С., Першин Д.Ю., Шелехов В.И.** Язык предикатного программирования Р. Новосибирск, 2009. 44 с. (Препр. / ИСИ СО РАН; N 153).
2. **Shelekhov V.** The language of calculus of computable predicates as a minimal kernel for functional languages // BULLETIN of the Novosibirsk Computing Center. Series: Computer Science. IIS Special Issue. 2009. 29. P. 107–117.
3. **Шелехов В.И.** Модель корректности программ на языке исчисления вычислимых предикатов. Новосибирск, 2007. 50 с. (Препр. / ИСИ СО РАН; N 145).
4. **PVS Specification and Verification System.** SRI International. URL: <http://pvs.csl.sri.com/>
5. **Basin D., DeVille Y., Flener P., Hamfelt A., and Nilsson J.** Synthesis of programs in computational logic // LNCS. 2004. Vol. 3049. P. 30–65.
6. **Sørensen M.H., Urzyczyn P.** Lectures on the Curry–Howard Isomorphism Amsterdam. Elsevier, 2006. 457 p.
7. **Hehner E.C.R.** A Practical Theory of Programming, second edition. 2004. URL: <http://www.cs.toronto.edu/~hehner/aPToP/>
8. **O’Leary J., Leeser M., Hickey J., Aagaard M.** Non-Restoring Integer Square Root: A Case Study in Design by Principled Optimization. // Theorem Provers in Circuit Design. 1994. P. 52–71.
9. **Creitz C.** Derivation of a fast integer square root algorithm. 2003. URL: <http://www.nuprl.org/Algorithms/03cucs-intsqrt.pdf>.
10. **Srivastava S., Gulwani S. and Foster J.** From Program Verification to Program Synthesis // Proc. of POPL. Madrid. 2010. P. 313–326.
11. **Шелехов В.И.** Предикатное программирование. Учебное пособие. НГУ. Новосибирск, 2009. 109 с.
12. **Souris J., Wiels V. Delmas D. and Delsenry H.** Formal Verification of Avionics Software Products // LNCS. 2009. Vol. 5850. P. 532–546.
13. **O’Halloran C.** Guess and Verify – Back to the Future // LNCS. 2009. Vol. 5850. P. 23–32.
14. **Ball T., Hackett B., Lahiri S., Qadeer S. and Vanegue J.** Towards scalable modular checking of user-defined properties // LNCS. 2010. Vol. 6217. P. 1–24.
15. **Hoare C.A.R.** An axiomatic basis for computer programming // Communications of the ACM, 1969. 12(10). P. 576–580.
16. **Floyd R.** Assigning meanings to programs // Mathematical Aspects of Computer Science. 1967. P. 19–32.
17. **Cohen E., Dahlweid M., Hillebrand M., Leinenbach D., Moskal M., Santen T., Schulte W., Tobies S.** VCC: A Practical System for Verifying Concurrent C // LNCS. 2009. Vol. 5674. P. 1–22.
18. **Ануреев И.С., Марьясов И.В., Непомнящий В.А.** Верификация С-программ на основе смешанной аксиоматической семантики // Моделирование и анализ информационных систем. (Ярославский государственный университет) 2010. Т. 17, № 3. С. 5–28.

9–12 сентября 2011 г. в г. Севастополь, Украина, пройдет Девятый международный симпозиум **IEEE EAST-WEST DESIGN & TEST (EWDTs 2011)**

Цель симпозиума – расширение международного сотрудничества и обмен опытом между ведущими учеными Западной и Восточной Европы, Северной Америки и других стран в области автоматизации проектирования, тестирования и верификации электронных компонентов и систем.

**Контакты: тел.: +380-57-702-13-26,
e-mail: hahanov@kture.kharkov.ua,
<http://www.ewdtest.com/conf/>**

Е.Н. Крючкова, канд. физ.-мат. наук, проф., **С.М. Старолетов**, ассистент,
Алтайский государственный технический университет имени И.И. Ползунова
E-mail: kruchkova_elena@mail.ru

Динамическое тестирование распределенных программных систем на основе автоматных моделей

Представлен процесс тестирования на основе моделей, предложена формальная модель распределенной системы, представлена автоматизированная система для сопровождения процесса разработки, основанная на описании модели и дальнейшего тестирования по этой модели, рассмотрены задачи статического и динамического тестирования.

Ключевые слова: тестирование программ, моделирование, распределенные системы, автоматные модели

Проблема тестирования программ на основе моделей

Резкое увеличение сложности программного обеспечения (ПО), расширение сферы критических применений ПО и повышение требований к его качеству делают все более острой проблему надежности разрабатываемого ПО, а следовательно, и проблему его тестирования. Тестирование программ на основе построения моделей (*Model based testing*) [1] – достаточно новый подход в тестировании, при котором строится математическая модель всей системы, а тестирование проводится по построенной модели. Главным направлением для широкого внедрения в практику программирования методов тестирования по модели является разработка таких систем, которые предоставляют программистам простую нотацию и эффективные инструментальные средства.

В данной работе предлагается способ моделирования сложных программных продуктов на основе автоматных моделей, предназначенный для простого и эффективного тестирования в процессе разработки. Принципиальную возможность практического тестирования на основе моделей можно получить только при условии проектирования системы с одновременным созданием модели, причем создание модели не должно требовать от разработчика больших трудозатрат.

Основная цель работы – простота и естественность описания модели разработчиками в процессе

проектирования ПО, возможность автоматизации процесса построения самой модели [2]. Главный принцип, положенный в основу моделирования: "код и модель – одно целое". Эта парадигма предполагает описание модели на языке описания моделей совместно с исходным кодом на языке программирования в одном и том же файле. Согласно нашему определению автоматной модели, состояния и переходы привязаны к файлам и строкам исходного кода (состояние компонента системы – определенная разработчиком последовательность строк кода, которая, по его мнению, отражает неделимое состояние системы в каждый момент времени). Модель описывается параллельно с разработкой системы "на месте" – в исходном коде, где собственно и определяются состояния, переходы, сообщения и т.д. Таким образом, разработка системы сопровождается описанием ее модели. Практическое применение вышеуказанных принципов демонстрирует работающий прототип, представляющий собой расширение среды *Eclipse*. Пример интерфейса данной системы представлен на рис. 1 (см. третью сторону обложки). Принцип "код и модель – одно целое", как было сказано ранее, при реализации системы тестирования предполагает описание модели на языке описания моделей вместе с исходным кодом в тексте программы. Очевидным решением является внедрение описания модели в псевдокомментариях к исходному коду, которые будут пропущены при компиляции кода приложения, но обрабатываются тестирующей системой. Таким обра-

зом, код модели является метаданными к исходному коду. Модель описывается разработчиком на языке описания модели в виде вложенных XML-сущностей.

Базовые определения. Система

В настоящее время известны различные модели параллельных и распределенных систем. В данной работе предлагается модель распределенной многопоточной системы на основе комбинации расширенных вероятностных многопоточных конечных автоматов. Предложенная модель позволяет описывать различные типы взаимодействия, существующие в реальных системах. В предлагаемой математической модели можно выделить следующие уровни абстракции:

- система – суперавтомат, представляющий собой множество взаимодействующих расширенных конечных автоматов;
- компонент системы – расширенный вероятностный конечный автомат; который определяется набором состояний, переходов и операций;
- состояние автомата, привязанное к коду программы.

Фактически проектирование сложной распределенной системы часто начинают с построения диаграммы развертывания UML, которая представляет собой перечень физических узлов с размещенными на них компонентами и связями между ними. Базируясь на диаграмме развертывания и рассматривая дальнейшее формальное описание модели, на первом уровне абстракции можно дать следующее определение распределенной системы как суперавтомата:

$$\begin{aligned} System = & (Node, A^*, A_{cooper}, BoundNodes, Msg, \\ & Res, CP(A^*)). \end{aligned}$$

Рассмотрим это определение более подробно. $Node$ – множество узлов, на которых работает распределенное приложение. Каждый элемент множества $Node$ содержит информацию об узле: строковое имя узла, адрес протокола IP или DNS-имя, по которому узел доступен в сети, а также множество различных типов систем, на которых могут работать узлы.

Обозначим N множество натуральных чисел. Отображение $BoundNodes$: $Node \rightarrow AN$, $AN = \{(A, n)\}$, $n \in N$, определяет для каждого узла системы множество компонентов, заданных в виде автомата A ($A \in A^*$) следующего уровня абстракции, где A^* – автоматы, описывающие поведение отдельных компонентов. Компоненты (автоматы типа A) выполняются на данном узле с некоторой кратностью n (на узле может работать n одинаковых компонентов системы, каждый из которых описывается автоматом A).

Множества Msg и Res – соответственно глобальные для системы множества сообщений и разделяемых ресурсов, к которым имеют доступ все компоненты системы.

Фактически все вышеперечисленные элементы определяют только диаграмму развертывания UML и,

следовательно, не позволяют описать логику взаимодействия, в отличие от них конструкции A_{cooper} и $CP(A^*)$ задают порядок взаимодействия в распределенной системе и точки сопряжения в ней.

Перейдем к описанию логики взаимодействия. Специального вида автомат A_{cooper} моделирует порядок взаимодействия в распределенной недетерминированной системе. Состояния автомата – это компоненты системы с заданной кратностью, а переходы – это взаимодействия между компонентами как результат отсылки и ожидания сообщения:

$$A_{cooper} = (C_n, C_{n0}, C_{nf}, \delta_{cooper}, In, \Sigma).$$

Здесь C_n – множество состояний автомата A_{cooper} , при этом состояние – это взаимодействующий компонент системы с учетом кратности, т.е. $C_n = \{(A, n_+)\}$, где n_+ – суммарная по всем узлам сети кратность компонента системы, заданного расширенным автоматом A . Множество $C_{n0} \subseteq C_n$ – начальные состояния, с каждого из которых может начаться взаимодействие, а $C_{nf} \subseteq C_n$ – конечные состояния, после отсылки сообщений в которые система заканчивает работу.

Автомат A_{cooper} действует согласно недетерминированной функции переходов δ_{cooper} : $C_n \times \Sigma \times In \rightarrow 2^{C_n}$,

при этом переход $c_1 \rightarrow c_2$ означает, что c_1 инициирует отсылку сообщения, а c_2 ждет сообщения (или сообщений по типу конъюнкции или дизъюнкции). Алфавит Σ автомата A_{cooper} определяет типы действий, элементы Σ представляют собой надписи на переходах.

Отображение In определяет возможность приема сообщений компонентом от нескольких источников, $In : C_n \rightarrow C_n^* \times InType$, $InType = \{\&, \vee\}$, т.е. для состояния из множества C_n может быть указана возможность ожидания сообщений от нескольких состояний по типу конъюнкции (ждать сообщения от всех указанных компонентов) или дизъюнкции (ждать сообщения от одного из указанных компонентов).

В процессе имитационного моделирования распределенной системы на каждом шаге недетерминированно выбирается взаимодействие и осуществляется переход согласно δ_{cooper} . При попадании в заключительное состояние процесс взаимодействия завершается.

Базовые определения. Компонент системы

Программы проектируются на основе предположения о детерминированном выполнении в каждой ситуации, однако сложность взаимодействующей системы может быть такой высокой, что для упрощения моделирования ее логику программу можно считать недетерминированной. Очевидно, что поведение системы в общем случае различно при обработке различных значений данных. Однако моделирование сложной системы на основе зависящего от данных описания

(например, на основе пред- и постусловий) очень сложно и не позволяет описать поведение системы адекватно и, самое главное, в реальных условиях проектирования сложного комплекса. Мы будем применять автомат с вероятностными переходами для моделирования сложной взаимодействующей программы как компонента системы, где элемент случайности – это переход в следующее состояние под воздействием неизвестной природы. Таким образом, предлагается отказаться от логики работы программы в виде контроля пред- и постусловий на переменные системы.

Рассмотрим модель на следующем уровне абстракции – уровне компонентов системы. Каждый компонент моделируется в виде расширенного вероятностного многопоточного конечного автомата с возможностью отправки и приема сообщений между состояниями и блокировкой ресурсов:

$$A = (s_0, S, F, \delta, \gamma, E, msg \subseteq Msg, res \subseteq Res),$$

где S – множество состояний, $s_0 \in S$ – начальное состояние, $F \subseteq S$ – множество заключительных состояний; E – множество событий и исключительных ситуаций; msg – подмножество глобального множества отсылаемых сообщений; res – подмножество глобального множества общих ресурсов. Логика работы автомата A в общем случае определяется двумя функциями переходов δ и γ , а также внешней к A функцией редукции $join$.

Поскольку главный принцип, положенный в основу моделирования системы – практическая простота описания в процессе проектирования системы, в качестве состояния компонента рассматривается определенная разработчиком последовательность $\{p, \dots, r\}$ линий кода $\cup Srcf_i$ в исходном файле $SrcFile$, которая, по его мнению, отражает неделимое состояние системы в каждый момент времени:

$$s = \{\cup Srcf_i | f \in SrcFile, \text{ в файле } f \text{ с линии } p_{i=p \dots r} \text{ до линии } r, \text{ код –логически значимый}\}.$$

Недетерминированная функция переходов δ описывает поведение автомата: находясь в состоянии из S некоторой кратности по действию из D с вероятностью из P в потоке из T по полученному сообщению из Msg' после установки блокировки ресурса на Res' модель компонента системы либо переходит в несколько состояний, создав несколько новых потоков из T , либо в следующее состояние в текущем потоке; при этом возможны отсылка сообщения из Msg' и снятие с блокировки ресурса из Res' :

$$\begin{aligned} \delta: S \times D \times P \times N \times T \times Msg' \times Res' &\rightarrow \\ 2^{\wedge}(((T \times S)^* \cup S) \times Msg'^* \times Res'). \end{aligned}$$

Здесь Msg'^* – итерация множества Msg' .

Функция γ переходов "по ребрам" определяет возможность возникновения некоторого числа событий или исключительных ситуаций из множества $E \times E \times \dots \times E \subseteq E^*$ при переходе из состояния в состояние:

$$\gamma: S \times S \rightarrow E^*.$$

Операция редукции осуществляет сворачивание потоков в один поток: $join: (S \times T)^+ \rightarrow S \times T$. Эту операцию нельзя выразить в функции перехода δ , поскольку мы рассматриваем локальные функции перехода в зависимости от текущего состояния и потока, а редукция осуществляется в зависимости от нескольких состояний и потоков. Еще раз подчеркиваем, что все действия в расширенном автомате связаны с некоторой вероятностью, и здесь мы говорим о моделировании потока управления (*control flow*), но не потока данных (*data flow*).

Базовые определения. Параллелизм

Многопоточной назовем систему, в которой работает несколько потоков или процессов, выполняющих параллельные действия одновременно. Многопоточность в приложении может создаваться за счет явного создания потоков/процессов операционной системы, неявного создания потоков при помощи библиотеки *OpenMP*, работой с потоками на разных узлах с помощью средств MPI или же за счет создания легковесных потоков, встроенных в новое поколение функциональных языков.

Пусть $P(s_{start}, s)$ – вероятность перехода из состояния s_{start} в s согласно функции переходов. Множество состояний $Next(s_{start})$, достижимых из заданного состояния s_{start} на n -м шаге определим рекурсивно:

$$Next^0(s_{start}) = \{s_{start}\}$$

$$\begin{aligned} Next^{n+1}(s_{start}) &= Next^n(s_{start}) \cup \{s \in S \mid P(s', s) \neq 0, \\ s' &\in Next^n(s_{start})\}. \end{aligned}$$

Тогда **параллельными действиями** в системе назовем действия по переходу между состояниями, которые выполняются неупорядочено и одновременно.

Неупорядоченность. Действия подавтоматов $A' = A|_{s=s'}$, $S' \subseteq S$ и $A'' = A|_{s=s''}$, $S'' \subseteq S$ автомата A не упорядочены, если $\forall s_1 \in S'$, $s_2 \in S''$: $\neg(s_1 \in Next(s_1)) \vee s_2 \in Next(s_1)$). Действия множества подавтоматов автомата A не упорядочены, если все они попарно не упорядочены.

Одновременность. Действия неупорядоченных подавтоматов выполняются одновременно, когда в каждый момент времени каждый из них находится в своем состоянии из множества локальных состояний S_{L_i} , $i=1 \dots n$, а множество общих глобальных состояний автомата можно представить как $S = S_{L_1} \times S_{L_2} \times \dots \times S_{L_n}$.

Компоненты системы, каждый из которых задан расширенным автоматом A , могут взаимодействовать

друг с другом через точки сопряжения из множества CP в суперавтомате *System*.

Для того чтобы пояснить точку сопряжения, можно представить множество параллельных плоскостей, в каждой i -й из которых находятся состояния автомата A_i (пример представлен на рис. 2, см. третью сторону обложки). Два *взаимодействующих* автомата находятся каждый в своем состоянии, но каждое такое состояние автомата связано с состоянием другого автомата системы. Множество $CP(A^*)$ точек сопряжения $handshake(s_1, s_2)$ определяет парные связи между моделями компонентов:

$$CP(A^*) = \{CP(A_1, A_2), A_1 \neq A_2, A_1 \in A^*, A_2 \in A^*\}$$

$$CP(A_1, A_2) = \{(s_1, s_2) \mid s_1 \in S_1, s_2 \in S_2, \\ handshake(s_1, s_2)\}.$$

Точки сопряжения можно разделить на явные и неявные. *Явные точки сопряжения* описывают связи, ожидаемые в системе. *Неявные точки сопряжения* возникают в результате использования примитивов синхронизации и не описываются явно. Такие точки существуют в двух случаях:

- при синхронном обмене сообщениями из множества Msg состояние отсылки сообщения соответствует состоянию получения сообщения;
- при блокировке ресурса из множества Res для организации работы критической секции состояние блокировки соответствует состоянию разблокировки в конкурирующем процессе (потоке, компоненте).

Пример графического отображения полученной в процессе разработки модели представлен на рис. 3 (см. третью сторону обложки). Это модель работающей распределенной системы дистанционного тестирования с использованием мобильных устройств.

Внедрение описания модели в программный код

Предлагается стандартные на сегодня промышленные процессы разработки ПО дополнить средствами тестирования на основе моделей, а саму фазу разработки дополнить описанием модели. Рассмотрим более детально технологию описания моделей, практически реализованную в качестве эксперимента под *Eclipse*.

1. Разработка начинается с проектирования модели высокого уровня архитектором системы (автомат A_{cooper}). Модель описывается в графическом виде на основе предлагаемой реализации системы, построенной с помощью подключаемых модулей к среде разработки. Далее по модели генерируются проекты для отдельных компонентов, включающие заглушки кода на целевом языке.

2. Реализация системы представляет собой разработку компонентов системы (автоматы типа A). Предлагаемый графический интерфейс, внедренный в *Eclipse*, предназначен для создания и редактирования

параметров моделей с генерацией представления модели на языке описания в виде сущностей XML. Наличие средств работы с моделью позволяет практически применить моделирование в процессе разработки ПО. Таким образом, к концу фазы разработки должны получиться работоспособный на уровне отдельных методов код, проверенный модульным тестированием, и не противоречащая модели архитектора модель кода, которая далее будет использована для проведения функционального тестирования.

3. Следующий этап – тестирование по модели, которое можно проводить двумя способами – статическим и динамическим. **Статическое (off-line) тестирование** по модели означает полную замену системы ее имитационной моделью. В ходе имитации собирается информация, необходимая для анализа характеристик модели, достижимости состояний и генерации тестовых наборов, наличия *deadlock* – состояний и т.п. **Динамическое (on-line) тестирование** проводится на скомпилированной рабочей системе в целях проверки ее поведения относительно построенной модели. В процессе динамического тестирования проводится тестирование соответствия, а также сбор статистики работы системы: вычисление вероятностей всех возникающих действий ("апостериорные вероятности"), которые потом можно использовать как первоначальные ("априорные") при повторном статическом тестировании, собирается статистика по совершаемым переходам и использованию межкомпонентных связей, вычисляются задержки при взаимодействиях.

Тестирование соответствия в процессе on-line тестирования

Динамическое тестирование реализовано на основе выделенного тестирующего сервера, который получает данные о происходящих в модели событиях через TCP/IP соединение. При этом на сервере создается динамическая модель работающей системы, которую можно сравнить с ожидаемой и найти ошибки. Для обеспечения отсылки данных о проходящих событиях в модели к коду компонента автоматически добавляется генерированный код на целевом языке программирования. Этот код генерируется в соответствии с XML-описанием состояний в программе и взаимодействует с тестирующим сервером, посыпая события на сервер, где они регистрируются в базе данных. Вставка дополнительного кода осуществляется автоматически при сборке проекта в режиме тестирования.

В процессе тестирования после разворачивания тестируемой системы и запуска сервера на нем начинается построение динамической модели компонентов системы. При этом проводится тестирование соответствия (*conformance-testing*):

- *Переход из заданного состояния в одно из заданных*. Переходы из состояния в состояние должны строго соответствовать модели. Фиксирование любого другого перехода говорит об ошибке в потоке управления.
- *Срабатывание событий и исключительных ситуаций*. При переходе из состояния в состояние может

осуществляться генерация событий и исключительных ситуаций, которые должны происходить только в ожидаемых состояниях.

• Соответствие модели верхнего уровня моделям нижнего уровня по сообщениям и кратностям. Модель верхнего уровня (A_{cooper}) описывает возможные взаимодействия через отправку/прием сообщений, модели компонентов более низкого уровня описывают обмен конкретных сообщений между заданными состояниями. Если заданный порядок обмена нарушен – это ошибка во взаимодействии.

• Корректность создания потоков и кратности состояний. При создании потока и моделировании этого действия на нижнем уровне абстракции проверяется соответствие числа ожидаемых и созданных потоков. Превышение заданной кратности сигнализирует или о некорректности потока управления, или о возможных высоких нагрузках.

• Корректность отсылки и приема сообщений. Для любого отправленного сообщения проверяется, дошло ли оно до получателя и правильно ли обрабатывается ситуация его потери.

• Критические секции, связанные с блокировкой внешних ресурсов. Если проводится блокировка ресурса одним потоком/компонентом и выполняется попытка доступа к нему другим потоком/компонентом, то проверяется, что доступ к ресурсу реально имеет только его хозяин.

• Возникающие дедлеки – бесконечные ожидания (*deadlocks*). Граф ожидания ресурсов с циклом означает наличие дедлока.

• Связность компонентов через точки сопряжения. Если определены точки сопряжения, то проверяется, действительно ли выполняется соответствие связанных состояний.

• Возможность отслеживать для каждого действия предысторию или так называемый "state trace" – те состояния и события, которые уже произошли перед данным событием.

СПИСОК ЛИТЕРАТУРЫ

1. Карпов Ю.Г. MODEL CHECKING. Верификация параллельных и распределенных программных систем. СПб.: Питер, 2010. 550 с.

2. Крючкова Е.Н., Старолетов С.М. Тестирование распределенных приложений на основе построения моделей // Прикладная информатика. 2008. № 6 (18). С. 124–134.

3. Gurevich Y. The Sequential ASM Thesis / Microsoft Research. Bulletin of European Association for Theoretical Computer Science. 1999. 32 p. URL: <http://research.microsoft.com/en-us/people/gurevich/Opera/136.pdf>

4. Brown A. An introduction to Model Driven Architecture. URL: <http://www.ibm.com/developerworks/rational/library/3100.html>

The advertisement features a large graphic of a ruler with the year '2011' at the top. Overlaid on the ruler is a white rectangular box containing the conference logo 'CEE-SEC(R)' and the text 'Разработка ПО' (Software Development). Below this, the dates '31 октября 1–3 ноября' (October 31 – November 1–3) are listed. To the right of the ruler, there is a smaller graphic of a speech bubble containing the text 'Конференция «Разработка ПО 2011/CEE-SEC» Москва' (Conference 'Software Development 2011/CEE-SEC' Moscow).

Программный Комитет CEE-SEC 2011 объявляет о начале приема заявок на выступления по следующим направлениям:

- Исследования/Технологии
- Практика разработки ПО
- Человеческий капитал и Образование
- Бизнес и Предпринимательство

*Отбор докладов осуществляется Программным Комитетом Конференции

Телефон орг. комитета: +7 (812) 336 93 44 www.secr.ru

С.А. Яблонский, канд. техн. наук, доц., Высшая школа менеджмента Санкт-Петербургского государственного университета

E-mail: yablonsky.serge@gmail.com

Введение в экосистему "облачных вычислений"

"Облачные вычисления" представляют сегодня динамично формирующуюся экосистему со сложной организацией и структурой. "Облачные вычисления" формируют новую парадигму вычислений, в которую вовлечены различные группы пользователей, посредников, провайдеров, разработчиков и клиентов. Многие термины, как, например, "облачные вычисления", "облачные сервисы", "облака" и многие другие, используются для описания различных, часто противоположных концепций ИТ, что приводит к многочисленным спорам и противоречивым представлениям, усугубляемым различием переводов одних и тех же терминов. Целью настоящей работы является описание технологической экосистемы "облачных вычислений" на основе разработанной онтологии "облачных вычислений". Для уточнения терминологии основные понятия приводятся как на русском, так и на английском языке.

Ключевые слова: облачные вычисления, облачные сервисы, SaaS, PaaS, IaaS, экосистема облачных вычислений

За последние годы информационные технологии (ИТ) и информационные сервисы (ИС) стали существенной статьей российского экспорта и определены как приоритетные сферы модернизации, причем отмечается важность развития экономически эффективного производства с учетом долгосрочной тенденции роста информационных ресурсов, роли технологий и сервисов в глобальном информационном сообществе.

Сегодня практически все федеральные структуры обеспечены современными компьютерами и подключены к сети Интернет. На государственном уровне ставится задача интенсифицировать работы по созданию реальных систем электронного документооборота, систем планирования и финансово-управленческой отчетности в государственных структурах. Осознается важность скорейшего создания единого портала государственных и муниципальных услуг для обеспечения граждан информацией в полном объеме по всем услугам федеральных, региональных и муниципальных структур власти. Его правовая основа гарантирована законом об обеспечении доступа к информации о деятельности государственных органов и органов местного самоуправления.

Внедрение корпоративных информационных систем позволяет достичь запланированных бизнес-результатов только при условии надежной, безопасной и

производительной работы всей ИТ-инфраструктуры. К ней предъявляются все возрастающие требования повышения производительности и надежности при одновременном постоянном увеличении объемов обрабатываемой корпоративной информации и динамичном изменении бизнес-процессов. Вместе с тем выдвигаются требования по сокращению затрат на поддержку и развитие ИТ-инфраструктуры при повышении ее адаптивности к меняющимся потребностям компаний в ИТ-ресурсах.

Эффективным способом удовлетворения этих во многом противоречивых требований в современных условиях является развитие рынка высокопроизводительных Центров Обработки Данных (ЦОД). ЦОД обеспечивают непрерывную работу критических бизнес-приложений, а также снижение стоимости хранения и обработки информации путем более эффективного использования систем хранения данных и консолидации вычислительных мощностей на основе так называемых "облачных вычислений" (*cloud computing*).

Современный ЦОД – это комплексная система управления и консолидированной обработки информации, объединяющая вычислительные, инженерные, электронные и коммуникационные системы и обеспечивающая адаптивность, самоуправление, виртуализацию, предоставление ресурсов по требованию. Создание

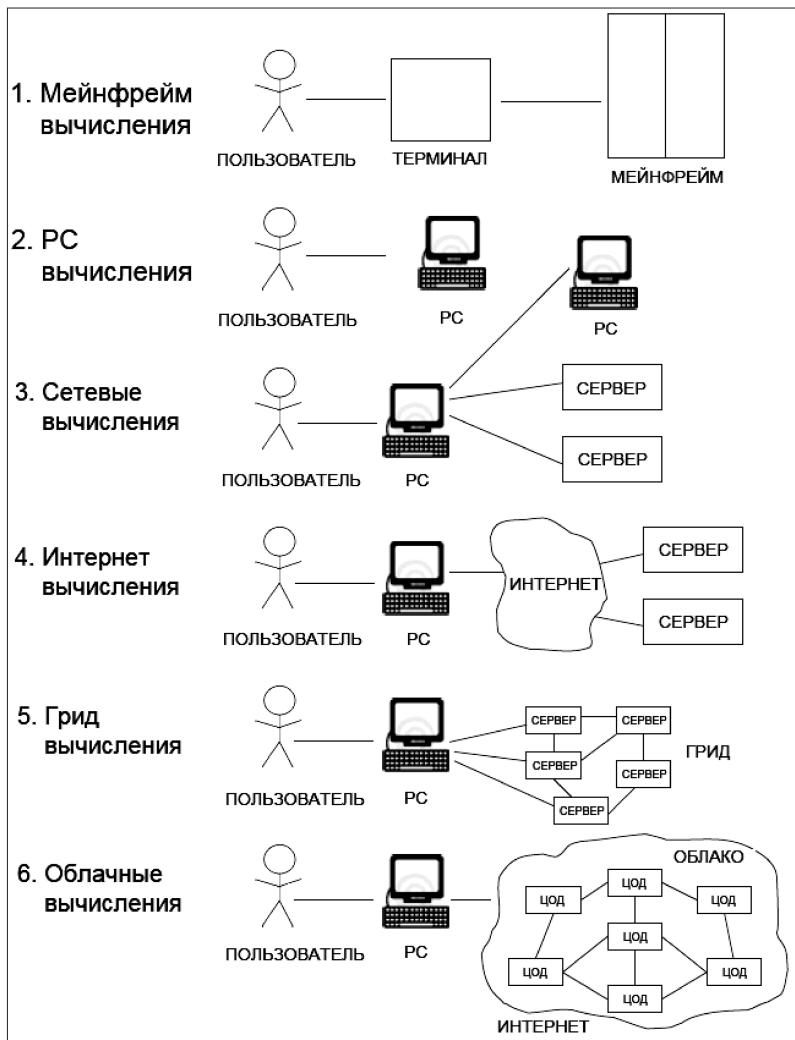


Рис. 1. Историческая смена парадигм вычислений (упрощенная схема)

дание ЦОД оптимизирует затраты на эксплуатацию занимаемых вычислительным оборудованием помещений и на обслуживающий персонал.

Размещение данных и корпоративных информационных систем на мощностях уже существующих ЦОД стало пользоваться устойчивым спросом. Стало возможным не просто арендовать площадь, а заказать ЦОД как сервис, передав ему все риски, связанные с размещением и поддержкой ИТ. Ведутся работы над созданием целых сетей ЦОД, размещаемых по всей стране, подобно глобальным сетям ЦОД компаний *Google*, *Amazon* и др.

Таким образом, идут дальнейшее укрупнение и консолидация ресурсов и сервисов ЦОД в стране, что подтверждается следующими данными [1]:

- Рынок коммерческих ЦОД в России в 2008 г. вырос на 55 %, доходы на нем составили 160 млн долл., а суммарная полезная площадь ЦОД достигла 21 тыс. м², причем 80 % рынка сосредоточено в Моск-

ве и Московской области. Ожидается, что рынок войдет в фазу консолидации начиная с 2011 г.

- Десять ведущих игроков (*Stack Group*, *WideXs*, "Голден Телеком", *ISG*, *KIAEHouse*, *РТКОММ*, "Ростелеком", "Синтэрра", *Masterhost* и *IBS DataFort*) контролируют более половины рынка коммерческих ЦОД – 51,8 %. При этом на долю компании *Stack Group* приходится 15 %, что более чем вдвое превышает показатели ближайших конкурентов – *WideXs* (6,1 %) и "Голден Телеком" (6 %).

- По оценке специалистов, совокупная стоимость всех коммерческих data-центров в Москве и Подмосковье не превышает 250–300 млн долл.

- На строительство и дальнейшую эксплуатацию крупного data-центра (для предприятия уровня *enterprise*) в течение пяти лет может быть затрачено до 15 и более млн долл.

- Один ватт полезной мощности data-центра в среднем обходится в 3 евро. Строительство ЦОД площадью 20 тыс. м² к концу 2008 г. стоило 1 млрд руб. Объем рынка услуг ЦОД на этот же период составлял более 2,3 млрд рублей.

- Суммарная емкость всех data-центров в России, оказывающих услуги внешним компаниям, составляет не более 15 тыс. условных стойко-мест. Это на порядок меньше того, что отечественному рынку нужно на самом деле, – реальный потенциал рынка коммерческих data-центров сейчас составляет около 120 тыс. условных стойко-мест. Причем в ближайшее время эта ситуация сильно не изменится. Даже в том случае, если будут успешно введены в эксплуатацию data-центры, проектирование и строительство которых было публично подтверждено, в условиях кризиса рынок получит не более 1,2–1,5 тыс. стойко-мест. Из них порядка 1,2 тыс. будет сдано в эксплуатацию в Москве и Московской области.

- Однако исследование рынка традиционных ЦОД показывает, что иногда ресурсы ЦОД используются нерационально и простаивают до 85 % времени. В то же время до 60–70 % расходов на ИТ-инфраструктуру приходится на поддержку часто простаивающих мощностей.

Вместе с тем развитие Интернета приводит к лавинообразному увеличению как информации в сети (15 петабайт новых данных ежедневно), так и числа устройств, подключенных к сети (ожидается 1 трлн в 2011 г.).

Поэтому дальнейшее развитие рынка ЦОД, очевидно, нуждается в модернизации рынка ИТ на основе внедрения "облачных вычислений", являющихся одним из наиболее перспективных и долгосрочных

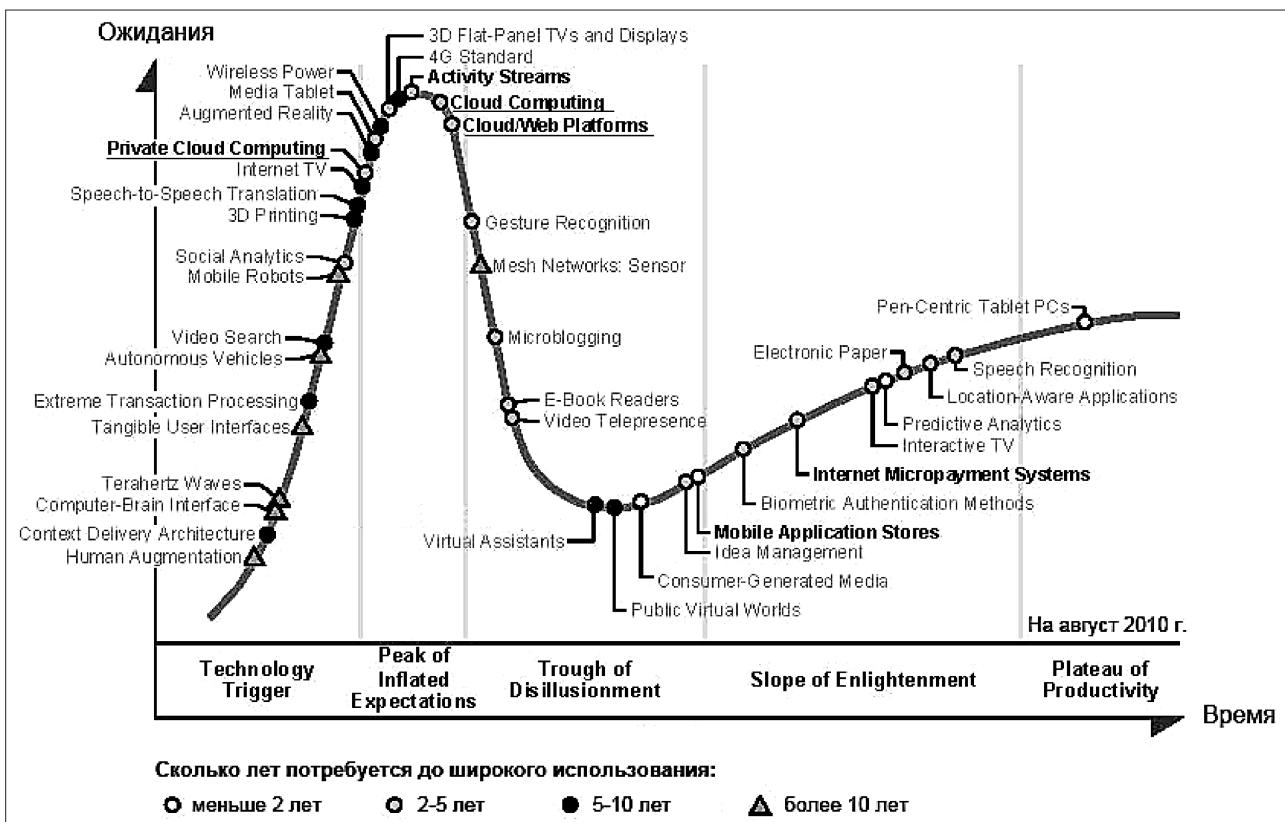


Рис. 2. Место облачных технологий среди технологий, которые в ближайшие 5–10 лет окажут наибольшее влияние на информационно-телекоммуникационную индустрию (источник – Gartner, 2010)

прорывных инновационных направлений развития сервисных ИТ. В свою очередь облачные вычисления позволяют повысить эффективность использования "облачных" ЦОД.

"Облачные вычисления" представляют сегодня не одно узкое направление развития информационных технологий, а, по сути, динамично формирующуюся экосистему со сложной организацией и структурой. Как показано на рис. 1, это новая парадигма вычислений, приходящая на смену уже известным, в которую вовлечены различные группы пользователей, посредников, провайдеров, разработчиков и клиентов. В результате многие термины, как, например, "облачные вычисления", "облачные сервисы", "облачка" и многие другие, используются для описания различных, часто противоположных концепций ИТ, что приводит к многочисленным спорам и противоречивым представлениям, усугубляемым различием переводов одних и тех же терминов.

Под **экосистемой "облачных вычислений"** мы будем понимать семантическую модель технологических (*cloud computing digital ecosystem*) и микро-/макроэкономических аспектов бизнес-экосистемы (*cloud computing business ecosystem*) рассматриваемых информационных технологий [2].

Целью настоящей работы является описание *технологической экосистемы "облачных вычислений"* на основе разработанной онтологии "облачных вычислений" [3, 4] и публикаций [5–17]. Для уточнения терминологии основные понятия приводятся как на русском, так и на английском языке.

В дальнейшем будем придерживаться следующих определений:

- "Облачные сервисы" (*Cloud Services*) – это товары, услуги и решения для потребителей и предпринимателей, которые поставляются и потребляются в режиме реального времени через Интернет.
- "Облачные вычисления" (*Cloud Computing*) – это новая модель разработки, развертывания и доставки "облачных сервисов".

В своем последнем исследовании "*Emerging Technologies Hype Cycle 2010*" аналитики Gartner (<http://www.gartner.com>) показали, что технологии облачных вычислений в настоящее время находятся на пике ожиданий (рис. 2).

По представлению Gartner, каждая новая технология проходит пять различных стадий:

1. *Technology Trigger* (запуск технологии) – объявление о технологии в средствах массовой информации.

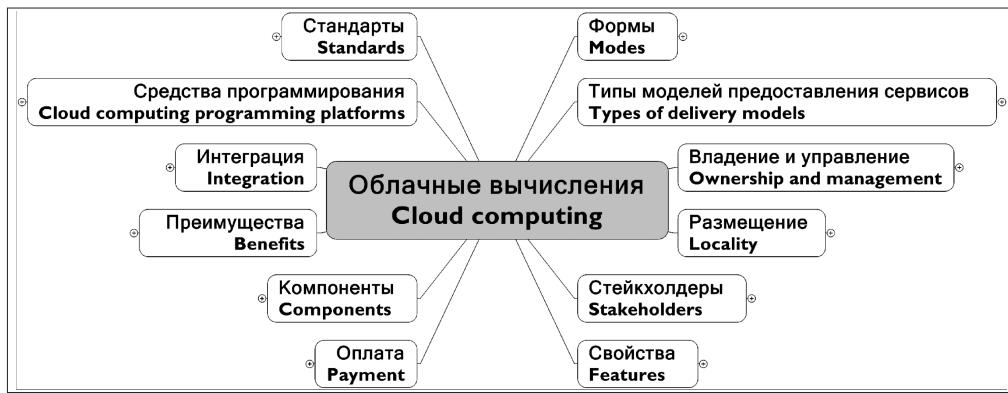


Рис. 3. Таксономия облачных вычислений (первый уровень)

2. *Peak of Inflated Expectations* (пик ожиданий) – на этой стадии на технологию начинают возлагать чрезмерно большие надежды.

3. *Trough of Disillusionment* (впадина разочарований) – данная стадия наступает, когда пользователи выясняют, что надежды на технологию пока что не оправдались, а специалистов, которые умеют доказать преимущества технологии, еще нет, как и положительных примеров ее внедрения. На этой стадии обычно резко сокращаются публикации о технологии, вследствие чего создается впечатление, что она ушла со сцены.

4. *Slope of Enlightenment* (подъем осведомленности) – по мере того как пользователи адаптируются к новой технологии, узнают о практике ее применения, и появляется больше сведущих в ней специалистов, наступает признание ее реальной пользы.

5. *Plateau of Productivity* (плато продуктивности) – на этой стадии технология становится общепризнанной и широко применяемой.

Согласно *Gartner* [14], доход от облачных вычислений по итогам 2009 г. вырос в мире на 18 % и составил 7,5 млрд долл., а к концу 2013 г. мировой рынок облачных вычислений удвоится и превысит 14 млрд долл. Принимаются решения правительства США об использовании облачных вычислений при создании государственных и оборонных информационных систем [17].

Как известно, правительство РФ утвердило государственную программу "Информационное общество (2011–2020 гг.)" [18]. Неудивительно, что одним из основных приоритетов входящей в нее подпрограммы "Российский рынок информационных телекоммуникационных технологий" на период до 2015 г. является создание национальной платформы "облачных вычислений", в том числе [18]:

- разработка интернет-платформы "облачных вычислений", обеспечивающей безопасную работу с типовыми программными приложениями в режиме "программа как услуга";
- разработка на базе национальной программной платформы набора типовых программных сервисов

для использования в органах государственной власти, включая средства коллективной работы с документами, общедоступное сетевое хранилище данных, средства удаленного хостинга программных приложений, средства разработки программного обеспечения;

- обеспечение интеграции национальных сетевых программных сервисов с крупнейшими коммерческими ресурсами, предоставляющими

программное обеспечение в режиме услуги.

В настоящей работе приводится простейшая таксономия "облачных вычислений" (рис. 3), построенная на основе разработанной OWL-S-онтологии "облачных вычислений" [3, 4], состоящей из более 100 различных концептов-сущностей и свыше 300 экземпляров этих сущностей, число которых постоянно увеличивается в связи с развитием "облачных" технологий и рынка "облачных вычислений".

Формы облачных вычислений

Можно выделить четыре основных формы облачных вычислений:

- публичные облака (*Public Cloud*);
- частные облака (*Private Cloud*);
- гибридные или смешанные облака (*Hybrid Cloud*);
- общие облака (*Community Cloud*).

Публичное облако подразумевает развертывание инфраструктуры, предоставление необходимого программного обеспечения и обеспечение механизмов доступа за пределами инфраструктуры организации непосредственно в сети Интернет сторонним клиентам. *Amazon Elastic Compute Cloud* (EC2) является примером одной из первых и наиболее популярных платформ облачных вычислений (<http://aws.amazon.com/ec2/>).

Частное облако создается на основе собственной инфраструктуры для оптимизации ее использования внутри организации за межсетевым экраном (*firewall*). Как правило, крупные организации и провайдеры облачных сервисов развертывают облачные вычисления в ЦОД или в региональной или глобальной сети ЦОД. Пionером в области создания и эксплуатации частных облаков является компания IBM, предлагающая весь спектр моделей частных облаков:

- частное облако (*private cloud*);
- управляемое провайдером частное облако (*managed private cloud*);
- собственное частное облако провайдера (*hosted private cloud*).

IBM Blue Cloud предоставляет клиентам выбор из более распространенного оборудования x86 или аппаратного обеспечения более высокого класса на основе

Таблица 1

Основные различия публичных и частных облаков

Характеристика	Вид облака	
	Публичное облако (Public cloud)	Частное облако (Private cloud)
Владелец Инфраструктуры (<i>Infrastructure Owner</i>)	Внешний поставщик (провайдер) облачных услуг	Организация Enterprise
Масштабируемость (<i>Scalability</i>)	Неограниченная (<i>Unlimited</i>) или по требованию (<i>On-Demand</i>)	Ограниченнaя развернутой инфраструктурой
Контроль и управление (<i>Control and Management</i>)	Возможность управления только виртуальными машинами, что снижает требования, предъявляемые к квалификации ИТ-персонала организации	Высокий уровень контроля и управления данными и процессами осуществляется внутри организации, но необходим высококвалифицированный персонал с большим опытом работы
Безопасность и производительность (<i>Security</i>)	Угрозы безопасности и необходимость нормативного соответствия, которые возникают при использовании публичных облаков посредством открытых сетей общего пользования	Угрозы безопасности отсутствуют или сводятся к минимуму при правильной политике безопасности организации
Стоимость (<i>Cost</i>)	Низкая стоимость с привлекательной моделью "оплата только фактически потребленных ресурсов"	Высокая стоимость за счет оплаты всей ИТ-инфраструктуры 24 ч в сутки в течение всего года, в том числе оплата пространства дата-центра, охлаждения, потребления энергии и аппаратных средств, плюс зарплата персонала
Производительность (<i>Performance</i>)	Ограничение пропускной способности сети и непредсказуемость общедоступных много-пользовательских сетей могут затруднить достижение гарантированной производительности	Гарантированная производительность

POWER. *Blue Cloud* использует программное обеспечение *IBM Tivoli* для автоматического предоставления систем с различными возможностями (процессор/память/диск), что дает организациям возможность за-действовать огромную вычислительную мощность, но платить за нее только по мере необходимости (<http://www.ibm.com/ru/cloud/>).

Очевидно, что в ряде случаев возможно организовать **гибридное облако**, совмещающее публичное и частное облака одной организации. В этом случае организация может хранить критические с точки зрения безопасности данные в частном облаке, а менее кри-тичные – в публичном облаке. Несколько организаций могут создать **общее облако**.

В табл. 1 приводятся основные различия публичных и частных облаков.

Модели предоставления облачных сервисов

Особую роль в облачных вычислениях играют сле-дующие типы моделей предоставления облачных серви-сов (рис. 4):

- "Инфраструктура как сервис" ("Infrastructure as a Service", *IaaS*);
- "Платформа как сервис" ("Platform as a Service", *PaaS*);

- "Программное обеспечение как сервис" ("Software as a Service", *SaaS*).

Инфраструктура как сервис. "Инфраструктура как сервис" (*IaaS*) в науке о сервисах является одной из разновидностей направления "все как сервис" ("Everything as a Service" – *EaaS* или *XaaS*).

"Инфраструктура как сервис" позволяет использовать вычислительные ресурсы в качестве сервиса, что включает виртуальные компьютеры гарантированной вычислительной мощности с каналами определенной пропускной способности для доступа к облаку через Интернет.

Сервис *IaaS* можно считать более общей формой сервиса "Аппаратное обеспечение как сервис" (т.е. вместо построения и обслуживания собственного вычислительного центра небольшая фирма может рас-смотреть вопрос об оплате сервиса аренды аналогичной инфраструктуры в облаке).

В предоставлении таких сервисов участвуют ком-пании *Amazon*, *Google*, *Microsoft*, *IBM*, *Sun/Oracle* и *HP*. Крупномасштабные высокопроизводительные компь-ютерные инфраструктуры (как правило, ЦОД) с высо-кой скоростью сетевого подключения пользователей являются важными компонентами эффективного сер-виса *IaaS*.

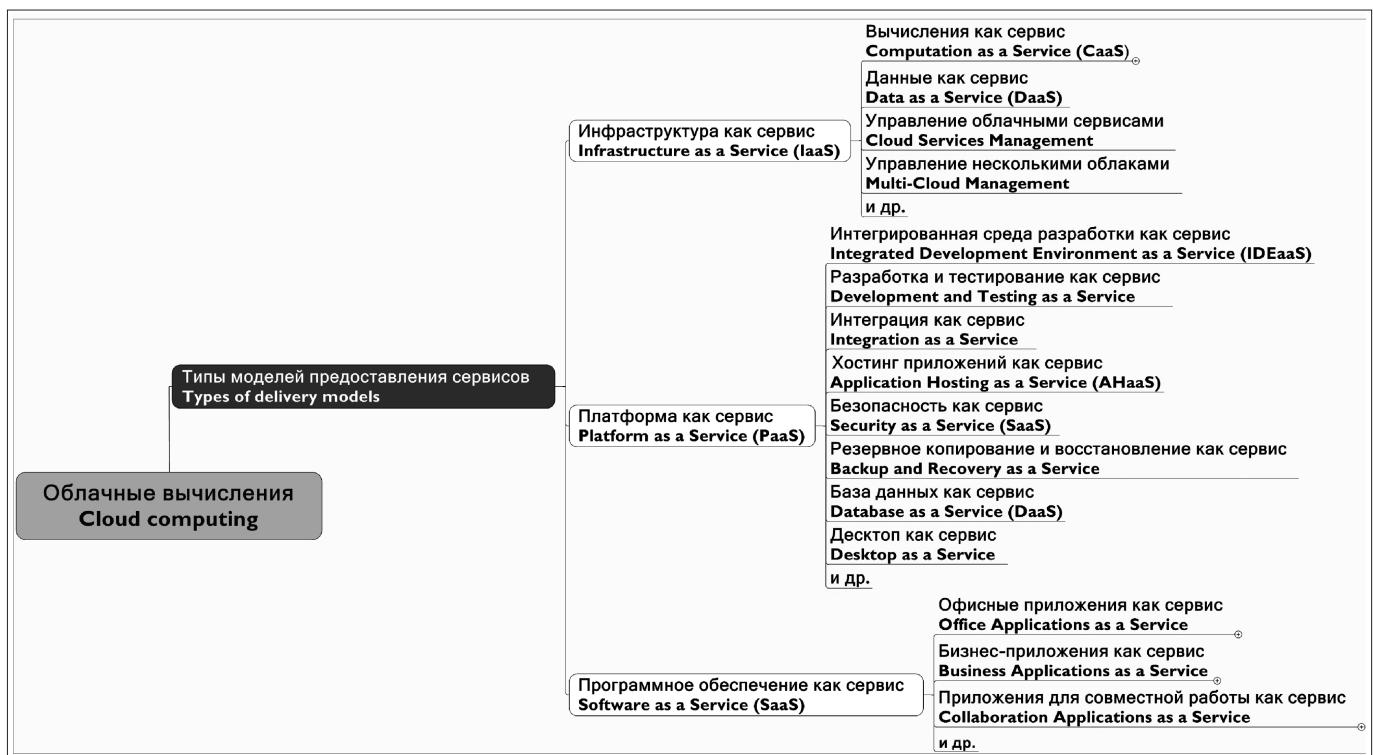


Рис. 4. Типы моделей предоставления облачных сервисов (фрагмент таксономии облачных вычислений)

Примером комплексного решения *IaaS* в частном облаке для крупных организаций может служить *IBM Computing on Demand (CoD)* и конкретно *IaaS*-решение *IBM CloudBurst* (<http://www.ibm.com/ru/cloud/>).

"*Инфраструктура как сервис*" состоит из следующих основных типов сервисов:

- "Вычисление как сервис" ("Computation as a Service", *CaaS*) на базе арендуемых с почасовой схемой оплаты

виртуальных машин-серверов, исходя из производительности виртуальной машины (процессора и объема оперативной памяти), характеристик ОС и развернутого программного обеспечения, пропускной способности канала и объема передаваемых по нему данных (табл. 2);

- "Данные как сервис" ("Data as a Service", *DaaS*) – предоставляется "неограниченное" пространство для

Таблица 2

Краткие характеристики провайдеров сервисов CaaS для публичных облаков

Характеристики	Провайдеры сервисов CaaS		
	GoGrid	Rackspace	Amazon (EC2)
Виртуализация	Xen	VMware	Xen
Операционная система	Windows, Linux	Windows, Linux	Windows, Linux
Выделяемая серверная память	От 0,5 до 8 Гбайт	От 256 Мбайт до 16 Гбайт	От 1,7 до 68,4 Гбайт
Балансировщик нагрузки	Free F5 Load Balancer	Нет	Amazon Elastic Load Balancer
24/7 Поддержка	Да	Да	Нет
Оплата	Почасовая оплата 0,19 долл. за Гбайт оперативной памяти и 60 Гбайт дисковой памяти, \$0,50 за 1 Гбайт передачи выходных данных, а передача входных данных бесплатна	Почасовая оплата 0,06 долл. за Гбайт оперативной памяти и 40 Гбайт дисковой памяти, 0,05 долл. за 1 Гбайт передачи входных данных и 0,22 долл. – за 1 Гбайт выходных данных	Почасовая оплата от 0,085 до 3,18 долл. (в зависимости от различных регионов и инстансов). Оплата передачи данных зависит от того, откуда и куда передаются данные (от 0,00 до 0,15 долл. за 1 Гбайт переданных данных).

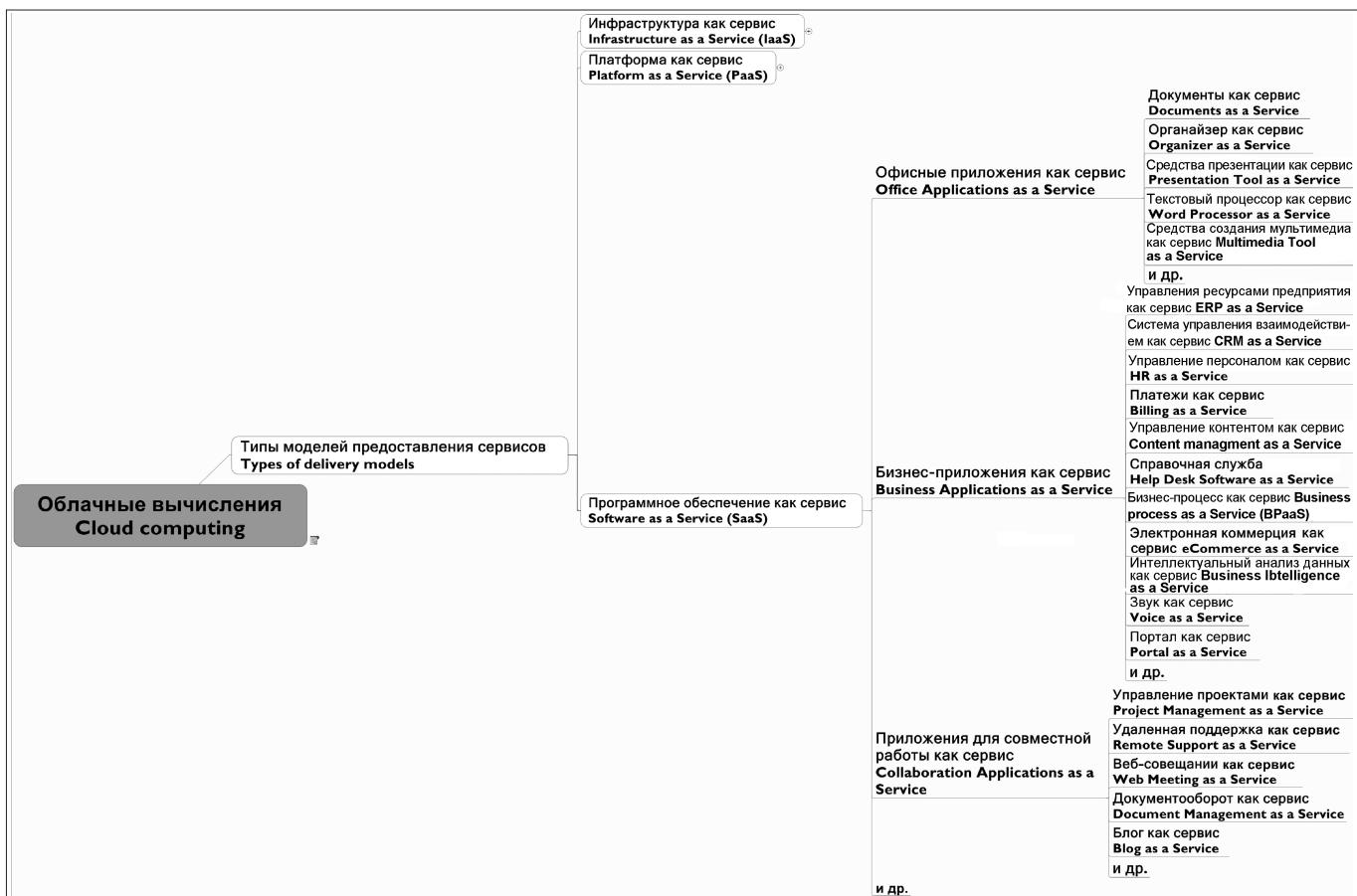


Рис. 5. Типы моделей предоставления *SaaS* сервисов (фрагмент таксономии облачных вычислений)

хранения данных пользователя независимо от их типа, а плата взимается за каждый Гб данных и скорость передачи (импорт/экспорт) данных. Примером могут служить хранение неструктурированных данных с помощью сервиса *Amazon* (*Amazon Simple Storage Service*, *S3*), *Google Storage* или файловое хранилище *IBM* (*IBM Scale out File Service*, *SOFS*). Эти технологии представляют собой распределенные файловые системы. Доступ к *S3* и *Google Storage* осуществляется через интерфейс web-сервиса, тогда как доступ к *SOFS* – с помощью протоколов *NFS* и *FTP*. Как правило, подобные решения позволяют снизить затраты и увеличить эффективность за счет масштабируемых виртуализированных решений по хранению данных, которые используют высокопроизводительную, масштабируемую, кластеризованную сеть хранения данных на основе ЦОД, частное хранилище данных для эффективного управления файлами любых размеров и форматов и защиту данных, включая зеркалирование и репликацию для обеспечения высокой доступности, резервирование, архивирование и восстановление данных.

• "Управление облачными сервисами" ("Cloud Services Management") обеспечивает контроль и управление

всеми и/или некоторым подмножеством сервисов *IaaS* одной облачной платформы.

- "Управление несколькими облаками" ("Multi-Cloud Management") представляет сервисы для управления *IaaS* на нескольких различных платформах, в частности, для управления сервисами в гибридных облаках.

Например, *IBM IaaS* для частных облаков включает в себя предустановленное программное обеспечение для управления инфраструктурой, серверы, систему хранения данных, портал самообслуживания для частного облака и, кроме того, позволяет развертывать две платформы: как для разработки/тестирования, так и для промышленной эксплуатации.

В табл. 2 (данные 2010 г.) приведены краткие характеристики некоторых известных провайдеров сервисов *CaaS* (*GoGrid*, *Rackspace*, *Amazon*) для публичных облаков.

Платформа как сервис. "Платформа как сервис" аналогична "инфраструктуре как сервис", но также включает в себя операционные системы и необходимые сервисы для конкретного приложения. Примером может служить *Microsoft PaaS – Windows Server 2008 R2 Hyper-V and System Center* (<http://www.microsoft.com/windowsserver2008/en/us/hyperv-main.aspx>). Другими

Таблица 3

Краткие характеристики провайдеров сервисов *PaaS* для публичных облаков

Провайдер <i>PaaS</i>	Характеристики		
	Программное обеспечение	Инфраструктура	Примеры размещенных приложений
Azure	Net (Microsoft Visual Studio)	Virtual Machine Based Microsoft Data Centers	Microsoft Pinpoint
Google [Google App Engine]	Python и Java	Google Data Center	Socialwok, Gigapan, LingoSpot
Force.com (http://www.salesforce.com/platform/) 185000 приложений на начало 2011 г.	Apex Programming и Java	Saleforce Data Center	Author Solutions, The Wall Street Journal
Heroku (http://heroku.com/) 116912 приложений на начало 2011 г.	Ruby	Amazon EC2/S3	Übermind, Kukori.ca, Cardinal Blue

словами, "Платформа как сервис" является "инфраструктурой как сервис" с пользовательским программным стеком для данного приложения.

На рис. 5 показаны основные сервисы, входящие в группу сервисов "Платформа как сервис".

Рассмотрим подробнее некоторые из них.

- "Интегрированная среда разработки как сервис" ("Integrated Development Environment as a Service", *IDEaaS*) позволяет организациям расширить внутренние процессы разработки и тестирования программного обеспечения с помощью доступа к облачным сервисам и ресурсам частного или публичного облака. В области *IDEaaS* для частных облаков особую роль играет *IBM*, создавшая различные подмножества подобных сервисов (<http://www.ibm.com/ru/cloud/>). Например, *IBM CloudBurst for Development and Test* – интегрированный набор, включающий аппаратные средства, устройства хранения данных, виртуализованные ресурсы и сетевую инфраструктуру, а также встраиваемую систему управления сервисами, позволяющую клиентам быстро развернуть сервисы в частном облаке. *IBM Smart Business Development and Test Cloud* – аналогичные сервисы для управляемого провайдером (*IBM*) частного облака, обеспечивающие коллективную разработку ПО в облаке с использованием решения *Rational Software Delivery Services for Cloud Computing*. И наконец, *IBM Smart Business Development and Test on the IBM Cloud* – сервисы для разработки и тестирования программного обеспечения, включающие сервисы полного цикла поставки ПО *Rational Software Delivery Services for Cloud Computing* через защищенную и масштабируемую "облачную" среду *IBM*. Использование технологий "интегрированная среда разработки как сервис" для разработки и тестирования приложений позволяет сократить расходы на персонал, повысить качество приложений и ускорить вывод на рынок новых сервисов. В области *IDEaaS* для пуб-

личных облаков разработано множество сервисов компаниями *GoGrid*, *Rackspace*, *Amazon* и др.

- "Разработка и тестирование как сервис" ("Development and Testing as a Service") – частный случай *IDEaaS*.

- "База данных как сервис" (*Database as a Service*, *DaaS*) представлен многими компаниями. Например, *Amazon* (*Relational Database Service*, *RDS* – *MySQL*, *SimpleDB*), *Oracle 11g on Demand*, *IBM DB2*, *Microsoft SQL Cloud* и др.

- "Десктоп как сервис" (*Desktop as a Service*) реализует сервис, обеспечивающий размещение всех рабочих станций организации в качестве виртуальных машин на серверах в ЦОД публичного, но чаще частного облака. Пользователи подключаются к этим виртуальным машинам с помощью тонких клиентов. Все данные хранятся в защищенном облаке. Таким образом, во многом отпадает необходимость контроля за синхронизацией данных, обновлением ОС и программных систем, замены с перенесением данных вышедших из строя рабочих станций, обеспечения защиты данных и т.п. Примером подобного сервиса может служить подход *VDI* (*virtual desktop infrastructure*) *Red Hat*, получивший название *Hosted Desktop Virtualization*. Другой вариант решения представлен в *IBM Smart Business Desktop Cloud*.

В табл. 3 приводятся краткие характеристики основных провайдеров сервисов *PaaS* для публичных облаков.

Программное обеспечение как сервис. Наиболее широко используемые облачные сервисы обычно включают ИТ-сервисы типа "программное обеспечение как сервис" (*SaaS*). Встречается и термин "приложение как сервис" (*Application as a Service*, *AaaS*), в основном использующийся как синоним *SaaS*. Как правило, *SaaS* развертываются в публичных облаках и отличаются большим разнообразием. Лидерами по предоставлению подобных сервисов являются *Google*, *Microsoft*,

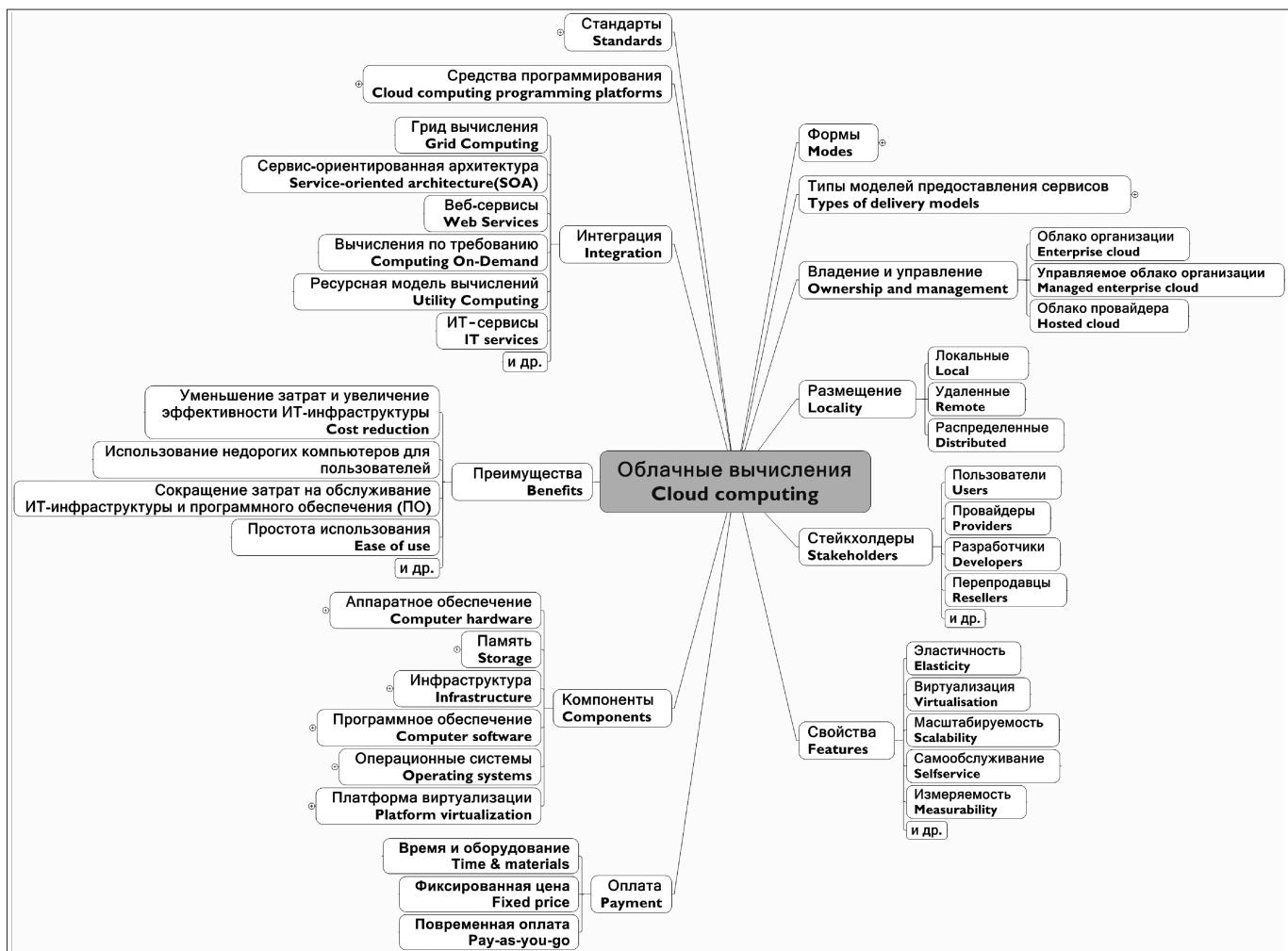


Рис. 6. Фрагмент таксономии облачных вычислений

Salesforce и ряд других компаний. Рамки данной статьи не позволяют представить полный обзор различных вариантов *SaaS* сервисов, наиболее распространенные из которых приведены на рис. 5.

Стейкхолдеры и оплата

Экосистема юридических (организаций) и физических лиц, вовлеченных в работу с облачными вычислениями, показана на рис. 6. Отношения и связи между ними определены на рис. 7.

Оплата использования облачных сервисов, как правило, ассоциируется с известной схемой повременной оплаты (*pay-as-you-go*, PAYG). Однако такая схема в основном используется для оплаты публичных сервисов, а также для оплаты сервисов в управляемых провайдером частных облаках (*managed private cloud*) и собственных частных облаках провайдера (*hosted private cloud*).

В целом, для частных облаков по-прежнему возможны две другие схемы оплаты:

- фиксированная цена (*fixed price*) на основе месячного/квартального/годового тарифа;
- оплата фактического использования времени и оборудования.

Интеграция технологий

Концепция облачных вычислений объединяет в себе такие известные модели и технологии, как вычисления по требованию (*Computing On-Demand*), ресурсная модель вычислений (*Utility Computing*), грид-вычисления (*Grid computing*) и информационные сервисы (см. рис. 6).

Традиционные производители баз данных *Oracle* (*Oracle 11g on-Demand*, *MySQL*), *IBM* (*DB2*), *Microsoft* (*SQL Azure*) и др. осуществляют перенос СУБД в облака. Одновременно компаниями *Google*, *Yahoo!*, *eBay*, *Facebook*, *Amazon* и др. интенсивно развиваются распределенные параллельные файловые системы с защитой от сбоев, образующие основу методов хранения и обработки информации в облаках.

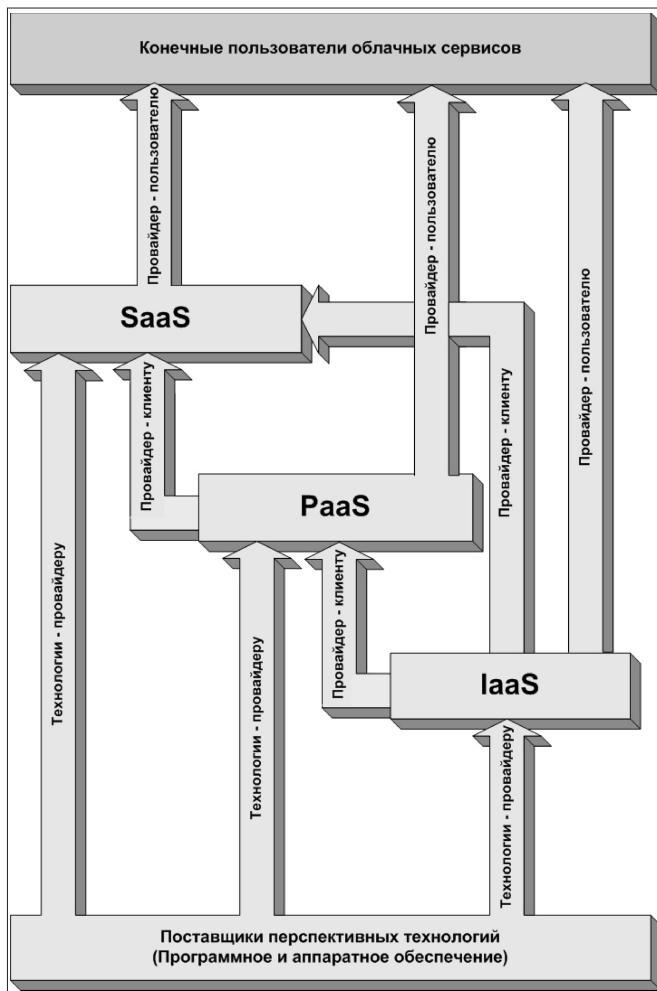


Рис. 7. Стейкхолдеры

Эти новые информационные технологии сегодня образуют технологический стек из *Google File System (GFS)*, *Google big table*, *Apache Hadoop Hadoop Distributed File System (HDFS)* плюс *MapReduce*, p2p вместе с "виантийскими алгоритмами" [19].

Основу "облачных вычислений" составляют облачные сервисы. Развитие науки о сервисах (*Service Science, Management, and Engineering, SSME* [20]) применительно к ИТ привело к возникновению сервисных информационных технологий, например, сервис-ориентированной архитектуры (СОА) с разнообразными языками описания информационных сервисов, которые используются и для описания "облачных" сервисов:

- *Simple Object Access Protocol (SOAP)* (<http://www.w3.org/TR/soap/>);
- *Universal Description Discovery & Integration (UDDI)* (http://www.uddi.org/pubs/uddi_v3.htm);
- *Web Ontology Language for describing web Services (OWL-S)* (<http://www.w3.org/Submission/OWL-S/>);

- *Business Process Execution Language for Web Services (BPEL4WS)* (<http://www.ibm.com/developerworks/library/specification/ws-bpel/>);

- *Web Services Distributed Management (WSDM)* (http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsdm).

Известная концепция управления ИТ-инфраструктурой компании ITSM (*IT Service Management*) начинает использоваться и для мониторинга и управления "облачными" ИТ-ресурсами на основе процессов ITIL (<http://www.itil-officialsite.com/>).

При разработке программного обеспечения облачных вычислений (иногда его называют "облачным программированием") можно выделить варианты, связанные с основными типами моделей предоставления облачных сервисов:

- программирование сервисов *IaaS*;
- программирование сервисов *PaaS*;
- программирование сервисов *SaaS*.

Особенностью "облачного программирования" является широкое использование различных "облачных" программных интерфейсов (*Cloud API*), основанных на приведенных выше языках и стандартах описания ИТ-сервисов. Также решаются задачи взаимодействия с другими облаками (*Interoperability with other clouds*) и использования сервисов различных облаков при реализации нового "облачного" сервиса.

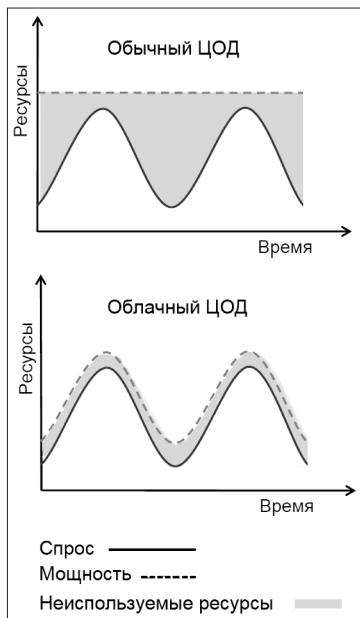
При программировании *SaaS* используются следующие известные технологии: *Java*; *Mobile (Android, iOS)*; *PHP*; *Python*; *Ruby*; *Windows & .NET*.

Преимущества и недостатки

Следует отметить следующие достоинства облачных вычислений [5–17]:

- Ориентированность на самообслуживание (каталог услуг).
- Автоматическая масштабируемость без остановки серверов "облачных" ЦОД и высокая адаптивность.
- Использование недорогих компьютеров для пользователей. Поскольку связь с облаком происходит чаще всего через веб-интерфейс с помощью тонкого клиента (интернет-браузера) по протоколу HTTP, пользователям нет необходимости покупать дорогие компьютеры с большим объемом памяти и дисков.
- Уменьшение затрат и увеличение эффективности ИТ-инфраструктуры. Как уже отмечалось, серверы как средней организации, так и ЦОД, загружены на 10–25 %. Переход на облачные сервисы в Интернете или облачный ЦОД позволяет организации сократить затраты на оборудование и его обслуживание до 50 % (рис. 8).

При этом многократно увеличивается гибкость и адаптивность ИТ-инфраструктуры в постоянно меняющейся экономической обстановке. Если компания не хочет, чтобы стратегическая корпоративная информация хранилась и обрабатывалась в публичном облаке, она может создать свое частное облако и получить все выгоды от виртуализации инфраструктуры.



- Сокращение затрат на обслуживание ИТ-инфраструктуры и программного обеспечения (ПО). Все облачные сервисы устанавливаются, настраиваются и обновляются в облаке. Стоимость ПО, ориентированного на доступ через Интернет, оказывается значительно ниже, чем его аналогов для персональных компьютеров. ПО можно арендовать с почасовой оплатой. Затраты на обслуживание такого ПО на всех рабочих местах практически сводятся к нулю.

- Увеличение доступных вычислительных мощностей и объема хранимых и обрабатываемых данных. Предоставление ресурсов не зависит от местоположения.

- Экономное расходование природных ресурсов. Облачные вычисления позволяют не только экономить на электричестве, вычислительных ресурсах, физическом пространстве, занимаемом серверами, но и разумно использовать природные ресурсы: ЦОД, которые поддерживают облачные сервисы, можно располагать в регионах с более прохладным климатом и более дешевыми источниками энергии.

К недостаткам облачных вычислений на сегодняшний день относятся [5–17]:

- понижение безопасности данных в сети Интернет;
- уменьшение контроля ИТ-инфраструктуры со стороны организации;
- необходимость постоянного высокоскоростного соединения с сетью Интернет;
- ограничение функциональных свойств ПО в Интернете по сравнению с локальными аналогами;
- трудности в оценке совокупной стоимости владения ИТ-ресурсами. При определении суммарных затрат на облачные вычисления, как правило, забывают, что обладание собственным ЦОД влечет оплату электроэнергии, работы персонала, ремонтов и пр.;

- отсутствие отечественных провайдеров облачных сервисов (*Amazon*, *Goggle*, *Saleforce* и др. сосредоточены в США);

- неразвитость отечественной экосистемы облачных вычислений;

- отсутствие отечественных и международных стандартов и законодательной базы облачных вычислений.

Развитие мирового рынка облачных вычислений

В прогнозах развития рынка облачных вычислений нет полного единства. Так компания *IDC* (<http://www.idc.com/>) считает (<http://blogs.idc.com/ie/?=543>), что к 2013 г. 10 % ИТ-бюджетов компаний будут тратиться на облачные сервисы. В отчете *IDC* также дана оценка развития рынка от 17,4 млрд долл. в 2009 г. до 44,2 млрд долл. к 2013 г. Эксперты компании считают, что в денежном эквиваленте рынок будет ежегодно увеличиваться на 26 %, а наиболее популярными в ближайшие 3–5 лет будут *SaaS* в публичных облаках.

Напротив, аналитики компании *Gartner* предполагают, что крупные компании будут активно инвестировать в собственные облачные сервисы в частных облаках.

Согласно данным *IDC*, мировые компании потратили в 2009 г. на "облачные сервисы" 16,5 млрд долл., а в 2014 г. этот показатель вырастет до 55,5 млрд долл.

Российский рынок облачных вычислений

Компания *IDC* впервые провела анализ российского рынка "облачных вычислений" (*Russia Public IT Cloud Services Market 2010–2014 Forecast and 2009 Analysis*). Согласно отчету, этот рынок в 2009 г. составил 4,8 млн долл., а к 2014 г. аналитики предсказывают российскому рынку рост до 161,5 млн долл. В отчете подчеркивается, что российский рынок находится на начальной стадии развития, но наблюдается рост интереса к "облачным сервисам". 94 % объема рынка занимает сегмент *SaaS*, который, однако, назван *AaaS*, что вызывает ряд вопросов по поводу методики *IDC*. Этот сегмент в 2014 г. увеличится до 113,4 млн долл. На сегмент *IaaS* пришлось только 4 %. Ожидается его рост на 35,5 млн долл., а доля *PaaS* может вырасти с 0,1 млн долл. до 12,5 млн долл. Несмотря на спорность ряда конкретных данных, это исследование показывает общую перспективу развития рынка облачных вычислений в России.

Российский рынок облачных вычислений все еще находится в стадии формирования. Несмотря на очевидные достоинства облачных вычислений, их распространению препятствует ряд объективных факторов. Пока большинство отечественных предприятий и организаций с недоверием относятся к аренде абстрактных, виртуальных мощностей, предпочитая работать с конкретным, желательно собственным, оборудо-

дованием. Однако ближайшие годы покажут преимущества таких возможностей и для российского рынка. При этом следует учитывать тот факт, что ведущие западные провайдеры облачных вычислений уже доступны для российских потребителей через Интернет (*SalesForce, Microsoft*).

Список литературы

1. Отчет по рынку Дата-центров 2008–2009. URL: http://www.json.ru/.../rus_market_of_the_commercial_data-centers_final.pdf.
2. Digital Business Ecosystems. Edited by: F. Nachira, P. Dini, A. Nicolai, M. Le Louarn, L. Rivera Léon. European Commission, Luxemburg: Office for Official Publications of the European Communities. 2007. 232 p.
3. Yablonsky S.A. Cloud Service Innovation Ontology Development. – In: Huzingh K.R.E., Conn S., Torkkeli M., Bitran I. (Eds.) // Proc. of XXI ISPIM Conf. The Dynamics of Innovation Bilbao, Spain. 2010. 6–9 June.
4. Яблонский С.А. Модернизация рынка информационных технологий на основе облачных вычислений // Тр. Конф. "Устойчивое развитие российских регионов: инновации, институты и технологические заимствования". Екатеринбург. 2010.
5. Wikipedia. Cloud Computing. URL: http://en.wikipedia.org/wiki/Cloud_computing
6. Velte T. Cloud Computing, A Practical Approach. McGraw-Hill Osborne Media, 2009. 352 p.
7. Rhoton J. Cloud Computing Explained: Implementation Handbook for Enterprises. Recursive Press, 2009. 508 p.
8. Reese G. Cloud Application Architectures: Building Applications and Infrastructure in the Cloud. Theory in Practice. O'Reilly Media 2009. 208 p.
9. Open Cloud Manifesto. URL: <http://www.opencloudmanifesto.org/Open%20Cloud%20Manifesto.pdf>
10. Галпин М. Все об облачных вычислениях с открытым исходным кодом: Часть 1. URL: <http://www.ibm.com/developerworks/ru/library/os-cloud-realities1/>
11. Above the Clouds: A Berkeley View of Cloud Computing. URL: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf>.
12. Catteddu D., Hogben G. eds. Cloud Computing – Benefits, risks and recommendations for information security, European Network and Information Security Agency (ENISA) URL: http://www.enisa.europa.eu/act/rm/files/deliverables/cloud-computing-riskassessment/at_down-load/fullReport.
13. Mell P., Grance T. National Institute of Standards and Technology, Information Technology Laboratory, URL: <http://groups.google.com/group/cloudforum/web/nist-workingdefinition-of-cloud-computing>.
14. Gartner report Five Refining Attributes of Public and Private Cloud Computing. URL: http://www.gartner.com/DisplayDocument?doc_cd=167182&ref=g_fromdoc.
15. Харатишвили Д. Рынок "облачных" услуг в цифрах и фактах // КомпьютерПресс. 2010. № 8.
16. Комов А. // IT News, 2009. № 22 (141), декабрь.
17. NATO and IBM to Use Cloud Technology for Improved Command and Control. Technology Collaboration between IBM and NATO Allied Command Transformation. URL: <http://www.03.ibm.com/press/us/en/pressrelease/33285.wss>
18. Распоряжение от 20 октября 2010 г. № 1815-р "О государственной программе Российской Федерации "Информационное общество (2011–2020 годы)". URL: <http://government.ru/gov/results/12932/>
19. Lynch N.A. Distributed algorithms. Morgan Kaufmann, 1996. 872 p.
20. Hefley B., Murphy W. eds. Service Science, Management and Engineering Education for the 21st Century. Springer, 2008. XXVI. 384 p.

ИНФОРМАЦИЯ

Продолжается подписка на журнал "Программная инженерия" на второе полугодие 2011 г.

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

Подписные индексы по каталогам:

Роспечатать – 22765; "Пресса России" – 39795.

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

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

В.В. Костюк, канд. техн. наук, доц., Российский государственный университет инновационных технологий и предпринимательства

E-mail: Viacheslav.Kostyuk@itbu.ru

История с программированием.

Его величество Код, вокруг да около него.

Рассматриваются вопросы разработки программных кодов, исторические аспекты приемов программирования, начиная с 60-х годов прошлого столетия. Выделяются этапы, начиная с самых первых шагов написания кодов в прямом смысле этого слова в двоичном представлении, до этапов развития языковых систем алгоритмизации и программирования до уровня генерации кода на основе моделей.

Ключевые слова: программирование, генерация кода, моделирование, методы программирования, развитие программирования

Осталось еще немало программистов, которые в 60-х годах прошлого века начали заниматься кодированием программ, реализующих вычислительные процессы. С трепетом в душе они манипулировали ноликами и единичками, кодируя команды и данные, вручную перфорировали на бумажной ленте разработанные алгоритмы и вводили в ЭВМ, чтобы получить важные результаты. Десять лет кодового программирования и пять лет перехода на языковое программирование ознаменовались бумом разработки языковых трансляторов. Многие еще помнят Эйфелеву башню на обложке одного из журналов, целиком увенчанную названиями языков программирования. Контекстно-зависимые искусственные языки общения с ЭВМ приводили в восторг разработчиков тривиальных по нынешним временам программ. Примерно так выглядели наброски алгоритмов в парадигме блок-схем в записных книжках программистов (рис. 1).

Языки программирования первого поколения типа АЛГОЛ заложили основу формализации алгоритмов, которую можно назвать "меточным программированием", когда программист мог из любого места программы передать управление в любую другую точку программы, не задумываясь о рекурсиях.

Это приносило удовлетворение от своего творения, демонстрирующего лаконичность и изящество профессионалов. Искусство умельцев от программиро-

вания позволяло им заплетать алгоритмы настолько изобретательно, что кроме них никто не мог ничего понять. А программы эффективно и даже эффективно работали, требуя минимум памяти ЭВМ и обладая максимальным быстродействием.

Такая виртуальность (в смысле изощренности) профессионалов уж очень дорого обходилась заказчикам и специалистам от программной инженерии, сопровождающим такие продукты. Найти причины возникающих ошибок и исправить их сам автор не всегда был способен. Он сам иногда не мог восстановить в памяти (даже по записям) те задумки, которые ко-

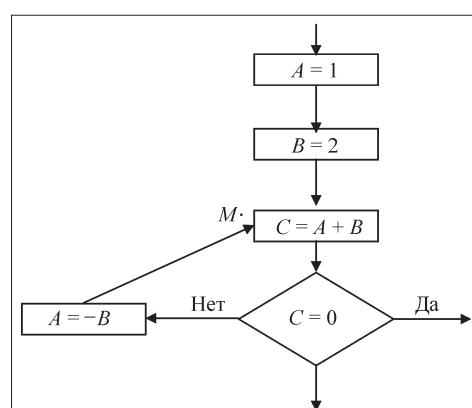


Рис. 1. Блок-схема фрагмента алгоритма

гда-то реализовывал. С тех пор сложилось мнение, что легче заново написать программу, чем дорабатывать чужое.

Компенсировать издержки такого программирования помог принцип процедурной структуризации программ, который можно охарактеризовать как "процедурное программирование". Верхним достижением этого принципа можно считать так называемое "структурное программирование", когда программы стали представлять структурно очерченными процедурами, вход в которые можно было осуществлять только через начало процедур. Структурное программирование называли программированием "без go to".

В стане разработчиков организации хранения данных и доступа к ним в это время развивалось направление "ассоциативного программирования", использовались цепочечные структуры организации данных на основе инверсных списков. На основе оптимизации подходов в области организации данных развивалось также направление под названием "эвристическое программирование", которое искало пути программной реализации процессов искусственного интеллекта.

Можно бесконечно продолжать обсуждение языковых особенностей разной целевой ориентации: это и программирование логических выводов (*Lisp*, *Prolog*), и программирование web-приложений (*Smalltalk*), и т.д. [1, 2, 4]. Это все было существенным вкладом в архив формализации видения окружающего мира. Уровень формализации отображения любой предметной области того времени строился изнутри программного слоя обозрения задач реальности. Обсуждались задачи потребителей информации и предлагались программные реализации этих задач.

Но все это относится к формализации процессов программирования на уровне терминологическом, не говоря уж о понятии экстремального программирования, которое вообще к языкам программирования не имеет никакого отношения и касается чисто методологических аспектов проектирования информационных систем.

Графико-терминологические средства первоначально в основном ограничивались попытками формализации процессов проектирования информационных процессов на уровне моделей, но с появлением CASE-технологий в ряд с задачами модельного представления предметной области ставились цели программной интерпретации этих моделей.

Существенное развитие получили языки с полнословными ключевыми терминами. В основном это англоязычные языки. В качестве примера русскоязычного проблемно-ориентированного языка с элементами объектного подхода можно привести встроенный язык поддержки продуктов фирмы 1С (с англоязыч-

ной интерпретацией для пояснения аналогов русскоязычных ключевых терминов).

Использование полнословных терминов в языках программирования — это тенденция формализации естественно-язычного представления алгоритмизации решения задач на компьютерах. Формализация представления задач компьютеру, можно сказать, захватила и область естественного языка описания предметной области. Но язык слов (терминов) менее эффективен по сравнению с языком образов, графических представлений. Графико-терминологический симбиоз отображения задач убедительно оправдал себя при проектировании баз данных.

Терминологическое описание работы с базами данных выразительно представлено средствами языка SQL. Словарный состав терминов языка SQL, наверное, не уступит ни одному из современных языков программирования. Если бы не графические средства моделирования реляционных баз данных, мало кто бы решился в полной мере пользоваться языком SQL для описания всех требований к сущностям предметной области серьезной разработки, включая описание бизнес-правил валидации и различных триггеров ссылочной целостности и целостности базы данных в общем, обработки транзакций.

Пример, приведенный в Приложении, демонстрирует основной состав вышеупомянутых требований к реляционной базе данных, представленных в виде анкеты, описывающей разные аспекты организации данных этого типа.

Но эта формализация представления организации данных стала возможной благодаря реляционному подходу. Реляционный взгляд на организацию данных позволяет формировать описание и иерархических, и сетевых, и агрегированных подходов, что, возможно, можно будет наблюдать в будущем. Примером упомянутой тенденции может служить объектно-реляционный подход или вариации на объектно-иерархической основе — СУБД Cache [5].

Автоматизация алгоритмизации проектирования организации и манипулирования данными оказалась достаточно просто реализуемой. Автоматизация программирования информационных процессов пока вышла только на уровень визуального программирования.

Визуальное программирование — это отдушина для программистов, насытившихся языками программирования. Визуальный подход вывел массу рядовых программистов (без гениальных способностей) из затруднительного положения. Ведь без этого объектно-ориентированные языки — это такая бездна знаний, в которой мог бы утонуть не только начинающий программист. Во времена языкового бума эффективными считались языки с набором инструкций в преде-

лах 200 команд. В настоящее время этот набор составляет в среднем 500...700 команд, включая функции и не считая системных переменных. Объектно-ориентированные языки добавляют сюда библиотеки классов с большим числом методов и свойств объектов.

Визуальное программирование слаживает эту проблему, но только в плане типовых визуальных интерфейсов. Однако оно не может выполнить эту задачу по отношению к типовым бизнес-процессам. В такой ситуации деваться некуда, как только не придумать что-то кардинальное. Пришло время выводить программирование на уровень моделирования.

Большие изменения в плане формализации подходов программирования наступили, когда накопилась критическая масса типовых разработок, типовых программных реализаций, типовых модулей, которые собирались в большие библиотеки, когда было замечено, что много алгоритмов функционально являются типовыми. Можно представить готовые модули в новую программу, и она почти готова к использованию.

Зародилось желание подстраивать готовые модули "на лету": задай значения типовых переменных, вызови нужные куски готовых программных модулей, находящихся в известных библиотеках и забудь про уже готовое, думай о новом. Таким образом, пришли понятия о классах объектов, свойствах и методах обработки этих свойств [3]. На смену структурному программированию пришло объектно-ориентированное программирование (ООП).

Тот из программистов школы процедурного программирования, кто еще не ощутил прелестей ООП, пытается набрать опыт путем изучения чужих программ в понятиях объектно-ориентированного мышления. Он долго не может разобраться, в каком порядке выполняются те или иные методы, что за чем следует, откуда и куда передается управление после того, как запущен метод MAIN, какие методы запускаются и какие выполняются следующими. С трудом он приходит к открытию того, что если структурное программирование – это программирование "без go to", то объектно-ориентированное программирование – это программирование "без do".

Все управляется событиями, которые возникают в результате действий пользователя, например, нажатия каких-нибудь клавиш. Теперь программист должен мыслить не категориями модулей и процедур, а понятиями классов и методов, т.е. этот принцип программирования можно назвать "беспрограммным программированием".

Кто-то может сказать, что ничего в этом особо нового нет – всего-то перефразировка процедур на методы, типовых модулей на классы и небольшие преобразования в организационной структуре программ. И с большим допущением можно согласиться с этим на

уровне "чистых" объектно-ориентированных языков, и то в случае, когда надо изначально создавать классы, организовывать библиотеки.

Но парадигма беспрограммного программирования бросается сразу в глаза при переходе к визуальному программированию, где все преимущества ООП становятся очевидными. "Процедуры" автоматически формируются в программах (программисту не надо об этом заботиться) при использовании визуальных компонентов, которые имеют возможности создавать разные события. Например, двойной щелчок левой клавиши мышки – чтобы запрограммировать реакцию на данное событие, программисту не надо создавать процедурный блок в программе. Он будет автоматически создан, когда программист захочет поместить туда команды, т.е. программисту остается только запрограммировать реакцию на создаваемое событие, используя минимальный набор команд и не думая о том, где этот блок будет размещен в программе. Если не считать, что процедуры при визуальном программировании создаются все-таки виртуально, то можно утверждать, что это "беспрограммное программирование". И уж бесспорно беспрограммным программированием является объектно-ориентированное программирование.

Объектно-ориентированное программирование открыло широкую дорогу к созданию и использованию классов визуальных интерфейсов (визуальных компонентов) с глубокой иерархией наследуемых классов. Нельзя не отметить роль ООП в тенденции качественных изменений в области формализации процесса разработки программных продуктов. Визуальное программирование вывело программистов на уровень мышления категориями графических компонентов. Такой подход смело можно определить как "компонентное программирование".

С другой стороны, имея графическую среду визуального программирования, захватывая мышкой компоненты с панели инструментов и перенося их в рабочую визуальную область проектируемой программы, в простых разработках программист может полностью обходиться без языкового инструментария, манипулируя управляющими клавишами (в основном мыши). Принцип "клавишного программирования" начинает занимать свои просторы применения. Можно уже сейчас привести много примеров такого подхода и отметить тенденции все большего стремления пользователей компьютеров именно к такому взаимодействию с ним. К клавишному программированию, например, часто прибегают при разработке информационных систем на этапах прототипирования. Существует множество типовых элементов информационных процессов, которые реализуются с помощью специальных программ конструкторами, дизайнерами и мастерами.

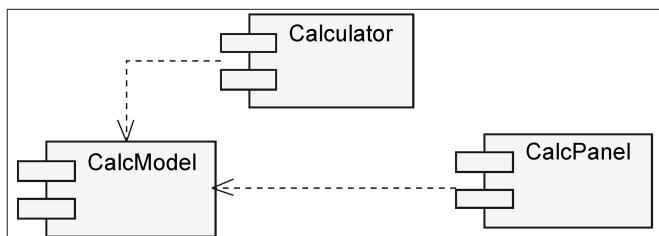


Рис. 2. Диаграмма компонентов программы "Калькулятор" в среде *Rational Rose*

Этих средств иногда вполне достаточно для удовлетворения пользователей. Уже вполне на приемлемом уровне восприятия, таким образом, реализуется проблема порталостроения.

Можно отметить еще одну сторону компонентного программирования. Выбирая компонент и какое-нибудь связанное с ним событие, можно задать этому событию функциональное предназначение относительно предметной области. Это будет событийная функциональность. Программный код реакции на это событие в разных предметных областях может быть типовым, поэтому можно создавать библиотеку классов событийной функциональности. Таким образом, мы приходим к принципу, который можно назвать "событийным программированием". Профессионализм в программировании сместится в сторону знаний событийных функций. Можно выбрать объект событийной функциональности и код готов. Конечно, можно по-

править его немного, если потребуется и при условии владения техникой проведения таких изменений. Но это уже из области "бизнес-программирования", стремление к которому вынуждает развивать графотерминологические средства формализации процесса программирования и выводить его на уровень моделирования предметной области.

Поэтому дружественно был принят общественностью разработчиков информационных систем, включая и программистов, универсальный язык моделирования (UML). Если уж мыслить категориями объектно-ориентированного подхода, то не на стадии программирования, а непосредственно на стадии анализа требований к информационной системе и ее проектирования, закладывая эти требования в визуальную графическую модель с помощью formalizedных графических средств. Ведь получилось генерировать SQL-код средствами моделирования баз данных по стандарту DEF1X. Почему бы не получать код на основе модели бизнес-процессов предметной области?

Весь путь формализации программирования, начиная с двоичного представления кодов на перфолентах и до представления кодов на основе моделей, можно разложить на ряд определяющих вех:

- процедурное программирование;
- объектно-ориентированное программирование;
- визуальное программирование;
- реляционный подход к организации данных;
- CASE-технологии проектирования баз данных;
- универсальный язык моделирования;

• CASE-технологии моделирования предметной области на основе объектно-ориентированного анализа и проектирования (ООАП).

Даже не окончательно отработанная графическая формализация моделирования с помощью пакета *Rational Rose*, дающая скелетный код объектно-ориентированной программной реализации на основании модели какого-либо бизнес-процесса, была принята довольно быстро с одобрением и сразу (имеется в виду без конкурентной вариации) в статусе стандарта.

Детальный анализ потенциальных возможностей UML дает основание полагать развитие этого направления в дальнейшем в сочетании с визуально-клавишным управлением процесса программирования.

Попробуем рассмотреть это на примере Java-приложения, приведенного в <http://lib.jugaru/article/articleview/174/1/0> — программы-калькулятора, состоящей

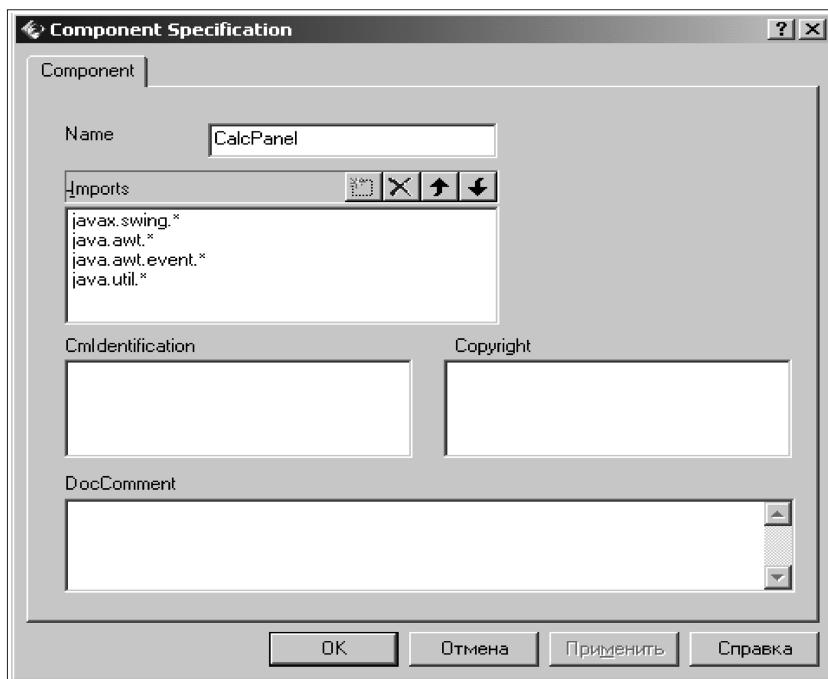


Рис. 3. Конструктор спецификаций компонентов

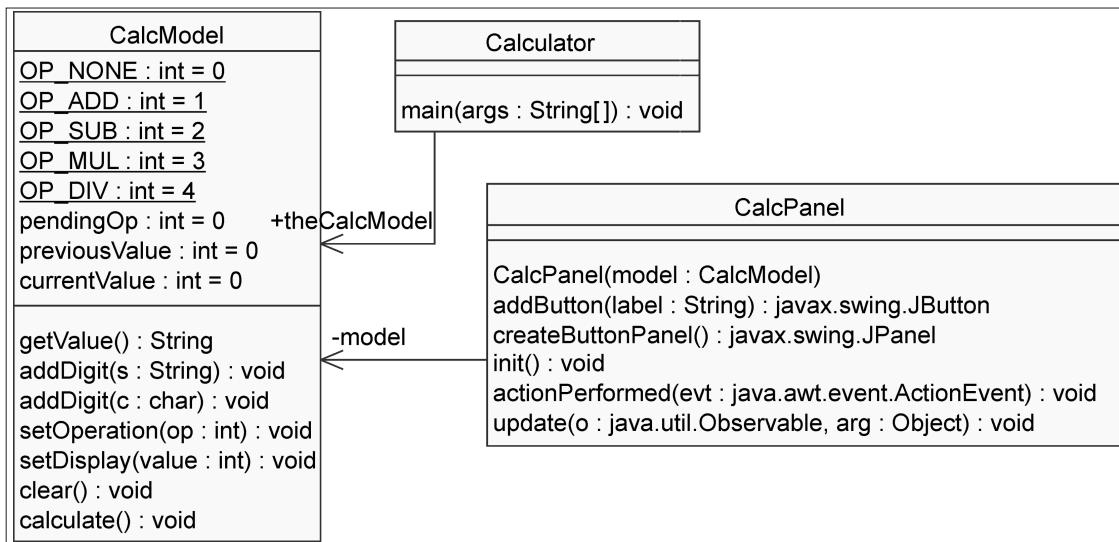


Рис. 4. Диаграмма классов программы "Калькулятор" в среде *Rational Rose*

из трех классов (файлов): CalcModel.java, CalcPanel.java, и Calculator.java.

Чтобы получить такую реализацию с помощью *Rational Rose*, можно непосредственно начать с создания трех компонентов в разделе Component View средствами контекстного меню New/Component. На диаграмме компонентов отношение между этими классами и сами классы можно показать следующим образом (рис. 2). В этой диаграмме модели Калькулятора для кода важно только то, что в результате будет создано три Java-файла с теми именами, которые присвоены компонентам. Они должны совпадать с именами классов. Для большей информативности модели можно воспользоваться более богатыми средствами языка UML, но они не все будут задействованы в формировании кода, как, например, отношения между компонентами, показанные стрелками. Так как нас интересует только получение кода, то на этом останавливаться не будем. Отметим только, что это мощные особенности языка UML.

Теперь рассмотрим, как в модели определить строчки кода, связанные с библиотекой классов JDK. Например, код класса CalcPanel.java начинается с команд, указывающих на использование библиотечных пакетов, в которых должны находиться классы, используемые в приложении CalcPanel.java: import javax.swing.*; import java.awt.*; import java.awt.event.*; import java.util.*.

В *Rational Rose* это формализуется путем заполнения этой информацией окна Imports в диалоговой заставке Component, которая вызывается двумя щелчками мыши на компоненте CalcPanel (это только один из способов, есть ряд других способов войти в этот диалог так же, как в другие, о чем в дальнейшем изложении не будет упоминаться). Само заполнение окна

Imports (рис. 3) также осуществляется клавишным способом, путем просмотра и выбора нужных путей к классам библиотеки JDK.

Далее должно быть установлено соответствие созданных компонент тем классам, которые они представляют (есть такая возможность), а значит, можно создать эти классы, определить их свойства и методы, включая метод MAIN. Все это моделируется и описывается в *Rational Rose*. Это можно описать в диалоговой заставке Class Specification, к которой можно обратиться непосредственно с диаграммы компонентов, либо средствами диаграммы последовательности или кооперации, либо средствами диаграммы классов. Арсенал возможностей *Rational Rose*, не говоря о продукте Rational Software Architect, достаточно широкий.

Диаграмма классов выглядит следующим образом (рис. 4).

Из представленного уже можно утверждать, что это тот минимум моделирования, которого достаточно для того, чтобы получить каркас кода, т.е. генерировать свойства классов и оболочки для кодов методов классов. Приведенные рассуждения мы вели методом "от обратного", т.е. были готовые отлаженные Java-классы, мы их запустили в *Rational Rose* в режиме обратного проектирования (реверса) и получили то, что демонстрировалось выше. И, если эту модель развязать с исходным текстом, то каркас нового Java-файла генерируется независимо от исходного.

Далее встает проблема получения кода методов класса. Для этого в UML можно использовать диаграмму состояний и деятельности объектов, с помощью которых можно смоделировать поведение объекта. Это поведение, которое в объектно-ориентированной программе описывается кодом метода, а на языке

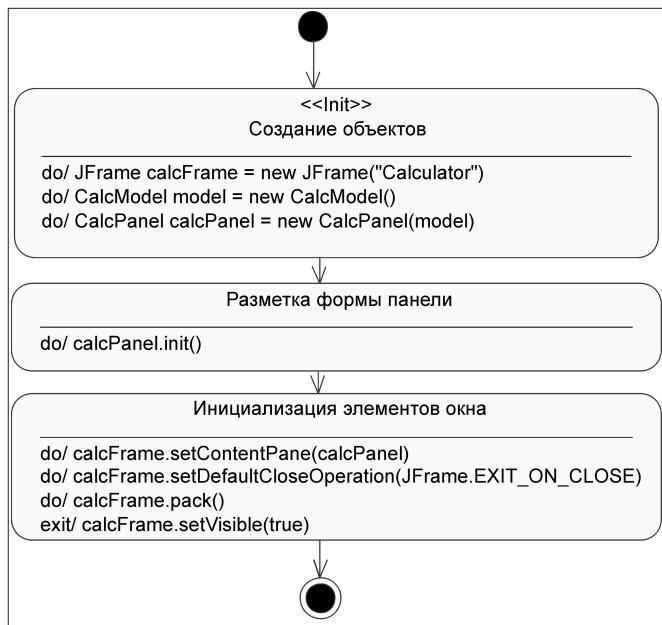


Рис. 5. Диаграмма состояния модуля программы "Калькулятор" в среде *Rational Rose*

UML – состояниями объекта и переходами его из одного состояния в другое. Сигнатура представления поведения объекта, привязываемая к графическим элементам диаграммы модели, состоит из сигнатуры состояния и сигнатуры переходов. Оба эти описания formalизованы следующим образом:

- Сигнатура состояния включает имя состояния (верхняя часть графического обозначения) и список внутренних действий (нижняя часть). В свою очередь

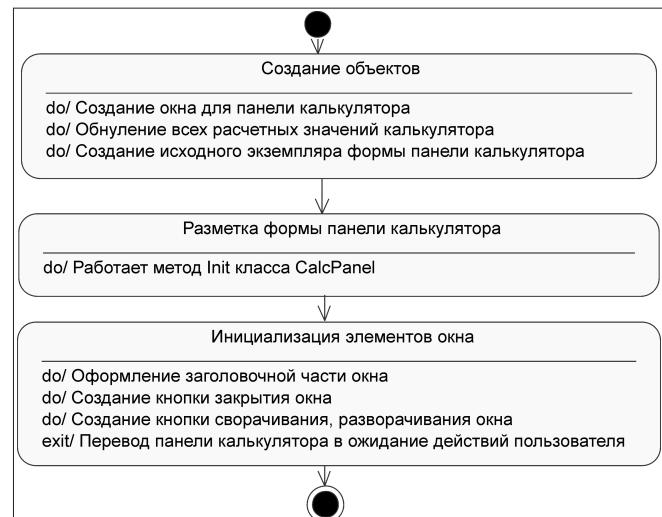


Рис. 6. Модель класса MAIN "Калькулятор" в среде *Rational Rose*

внутренние действия представляются в виде: метка-действия / выражение-действия.

- ◆ Метка действия – это тип действия, а если уточнить – стереотип действия, это ключевые термины: Entry – действия перед входом в состояние, Exit – действие после выхода из состояния перед переходом, Do, Event. Например, Do характеризует действие как изменение свойства объекта, а для кода – это операция присвоения значения переменной, определяющей требуемое свойство объекта. Можно расширить этот список стереотипов. Например, Init – действие по созданию объекта класса.
- ◆ Выражение действия – это название самого действия, а для кода – это операция на языке программирования, что часто рекомендуется использовать в литературе.
- Сигнатура переходов представляется в виде: имя события (список параметров) [сторожевое условие] выражение действия.
- ◆ Выражение действия – то же самое, что и в сигнатуре состояния.
- ◆ Имя события – это его название, сущность. Имя события идентифицирует каждый отдельный переход на диаграмме состояний.
- ◆ Список параметров – это могут быть значения каких-нибудь свойств, которые надо сообщить новому состоянию.
- ◆ Сторожевое условие представляет собой некоторое булевское выражение. Как можно заметить, сигнатура перехода так же, как и сигнатура состояния, приближены к алгоритмическому подходу.

Так, например, можно представить диаграмму состояния для класса MAIN Calculator (рис. 5).

Отсутствие сигнатуры описания переходов говорит о том, что это нетриггерные переходы или их можно назвать безусловными.

Простая модель, но что здесь неприятного для программиста – это не модель, а программа с небольшими пояснениями.

Нижеприведенный вариант этой модели скорее может называться моделью с точки зрения более широкого круга интересующихся тем, как работает метод MAIN класса Calculator.java программы "Калькулятор". Эту модель можно обсуждать с разными специалистами: и с программистами, и с проектировщиками, и с пользователями (рис. 6).

Еще один пример – модель метода actionPerformed класса CalcPanel.java. Это более алгоритмическая модель, но тоже понятная широкому кругу специалистов (рис. 7).

А как же с кодом? А код разработчик формирует клавищным способом. А именно – тип выражения действия подсказывает CASE-системе, какое окно диалога предложить разработчику. Если это тип "Calc" (вычисление), то диалог должен предоставить возмож-

ность выбора операции, свойств объектов, известных по диаграмме классов, методов и классов, которые могли бы участвовать в операции. Если это тип "do" (установить свойства объекта), то доступным должен быть набор элементов, соответствующих операции присвоения или обращения к методу, если это тип "init" (инициализация объекта), то – созданию объекта и т.п. Тип выражения действия должен соответствовать типу операции языка программирования. Не всё в существующих CASE-технологиях на базе UML работает так, как здесь описано, но тенденции развития таковы, что моделирование сближается с клавишными средствами программирования. Клавишное управление диалогом программирования хорошо отработано в визуальных языках. Напрашивается два пути дальнейшего развития CASE-моделирования предметной области.

Первый связан с совершенствованием CASE-технологии в направлении расширения визуальных средств интерпретации модели в программный код, о чем упоминалось выше. На данный момент уже существует пример такого развития. Это *IBM Rational XDE Developer for Java*, *IBM Rational XDE Developer for Microsoft Visual Studio.Net*, *IBM Rational XDE DeveloperPlus* (<http://www.citforum.ru/programming/application/rational/#1>).

Второй – развитие визуальных языков программирования в направлении создания среды моделирования на основе стандарта UML и связки в этой среде модельного уровня с уровнем программирования. Этот путь не исключается, так как в CASE-технологии сложно объединить визуальные средства программирования разных языков, и вряд ли разработчики пой-

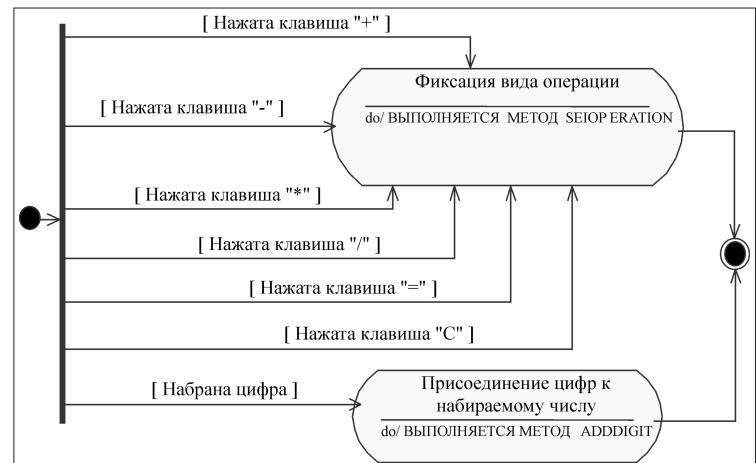


Рис. 7. Модель метода actionPerformed класса CalcPanel.java.

дут на это, даже решившись создавать какую-то единую среду визуального программирования. А для разработчиков языков программирования открываются большие горизонты развития своих продуктов, вплоть до вывода этих продуктов на уровень моделирования предметной области, на орбиту модельного программирования.

Разработчики средств программирования быстро отреагировали на идею визуализации программирования. В большинстве случаев подходы к реализации визуализации похожи друг на друга (почти однотипные), например, в среде разработки программ средствами Delphi, Access, FoxPro. Но для каждого языка своя среда разработки (*IDE* – *Integrated Development Environment*), а тут уже есть готовая единая, стандартная среда моделирования, которую предлагает язык UML и IBM Rational – инструменты разработки, нуж-

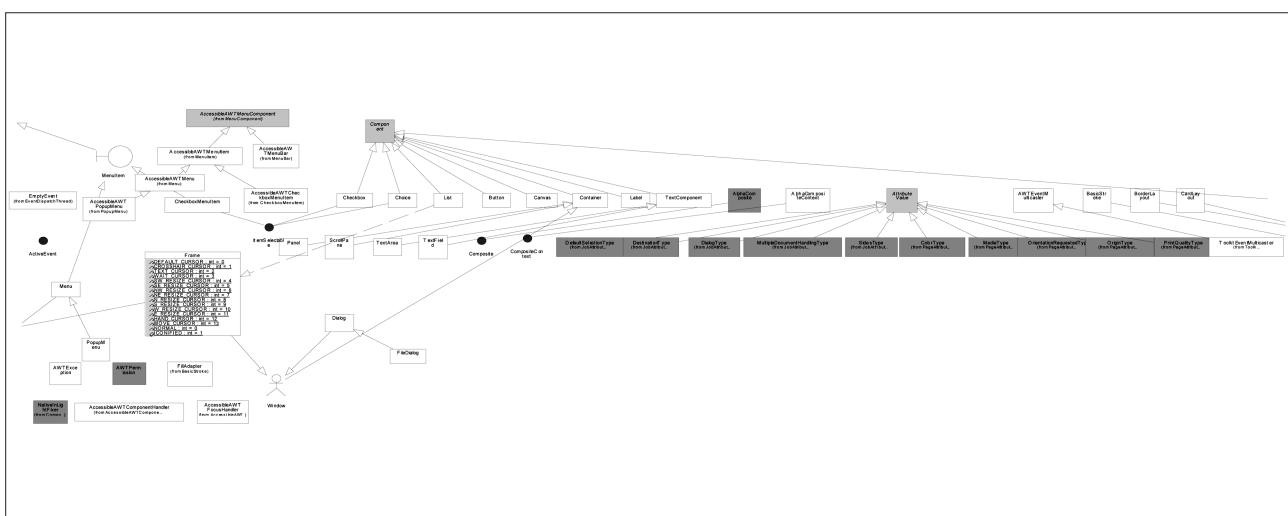


Рис. 8. Образный фрагмент UML-модели библиотеки AWT

на только реализация. Правда надо отметить, что это еще не устоявшаяся для сферы программирования понятийная система. Она намеренно наделена возможностями расширения в любую сторону, но только в целях представления моделей, а не в целях расширения возможностей программной реализации этих моделей.

Решение проблемы модельного программирования постоянно будет наталкиваться на отсутствие еще одного промежуточного уровня программирования по отношению к визуальному — на сборочное программирование, стимулируя развитие и этого направления. Если визуальное программирование, ориентированное на компоненты, требует библиотеки классов масштаба форм, масштаб функциональных классов значительно шире и визуализация работы с функциональными классами потребует своих подходов реализации и развития методов интроспекции.

Это направление развития средств программирования пока трудно сориентировать на кого бы то ни было. Похоже, что здесь свое слово смогут сказать "итологи", продвигающие идею стандартизации программных изделий и функциональных приложений.

В ближайшей перспективе модельное программирование будет ориентировано на определенные языки ООП. Постепенно в нотациях и сигнатурах будет наблюдаваться отход от терминологии языков программирования и должно наступить время "схемного программирования", "безязыкового программирования". Уже сейчас модели все больше выглядят похожими на схемы. На рис. 8 приведен немногого измененный для образного представления фрагмент библиотеки AWT. Не правда ли, похожа на электрическую схему. В таком подобии могут быть представлены диаграммы, на которых можно разместить разные типовые элементы модели предметной области в изображении типовых приложений, актантов, прецедентов, классов, объектов, интерфейсов, контроллеров, сервисов, отношений между ними, состояний, ветвлений деятельности и т.п. К единым диаграммам уже пришли разработчи-

ки в *XDE DeveloperPlus*, где уже не существует строгого разделения диаграмм по типам — теперь можно создать единую диаграмму и определить на ней любые элементы под ответственность проектировщика.

Конечно, разбираться с этими схемами — удел специалистов. Но это так же, как и с радиосхемами — потребители телевизоров не обязаны вникать в суть работы внутренностей бытового прибора, хотя кое в чем некоторые разбираются и могут на схеме определить, где лучевая трубка, а где показан блок питания или развертки. Да и специалисты бывают разного уровня знаний: некоторые понимают схемы на уровне компонентов, а некоторые на уровне сигналов. Аналогично будет и с информационными системами и их кодовыми реализациями в виде схем.

Таким образом, все началось с кодирования программ, а вернее с программирования кодов. В развитии этого процесса о кодах стали только подразумевать, осталось просто программирование. Теперь движение происходит в направлении к модельному программированию, а точнее к моделированию кода. Постепенно о коде будут только подразумевать, останется только моделирование, а программирование уйдет в забвение так же, как Код.

СПИСОК ЛИТЕРАТУРЫ

1. Соммервилл И. Инженерия программного обеспечения / пер. с англ. 6-е издание. М.: Вильямс, 2002. 624 с.
2. Себеста Р.В. Основные концепции языков программирования / пер. с англ. 5-е изд. М.: Вильямс, 2001. 672 с.
3. Трофимов С.А. CASE-технологии: практическая работа в Rational Rose. БИНОМ, 2001.
4. Ахо А.В., Лам М.С., Сети Р., Ульман Д.Д. Компиляторы: принципы, технологии и инструментарий. 2-е изд. М.: Вильямс, 2008.
5. Киорстен В., Ирингер М., Кюн М., Рериг Б. Постреляционная СУБД Cache 5. Объектно-ориентированная разработка приложений. 2-е изд., перераб. и дополн. М.: "Бином-Пресс", 2005. 416 с

ПРИЛОЖЕНИЕ

Анкета, описывающая разные аспекты организации данных реляционного типа

Сущности
№ Сущность " _____ ".
Logical Name: _____
Physical Name: _____
Definition: _____
Note: _____
Note2 (примеры возможных запросов): _____
Note3: _____
Идентификатор _____

ПРИЛОЖЕНИЕ (Продолжение)

Правило валидации

Имя выражения _____ выражение _____

Атрибуты, колонки

№ Атрибут: " _____ " сущности _____

– атрибут:

Logical Name: _____

Physical Name: _____

Definition: _____

Note: _____

Ключи: PK, AK, IE

Тип данных: _____

– домен:

Logical Name: _____

Physical Name: _____

Definition: _____

Note: _____

Domain Parent: _____

Тип данных: _____

Режим нулевых значений: – Not Null, Null

Значение по умолчанию: – Set to empty string, Set to Null, Set to Zero

Имя переменной _____ значение _____

Имя маски: _____ маска: _____

Правило валидации

Имя выражения _____ выражение _____

Имя списка _____

список значений _____

– колонка:

Привязка к домену, имя домена: _____

Связи:

Связь " _____ – _____ "

Verb Phrase со стороны родительской сущности _____

Verb Phrase со стороны дочерней сущности _____

Definition

Тип связи: идентифицирующая,

неидентифицирующая обязательная,

неидентифицирующая необязательная,

неопределенная (M:M)

Кардинальность связи (Cardinality – 0, 1, ∞ ; 1, ∞ (P); 0, 1 (Z); точно N (N);

Правила ссылочной целостности: R – Restrict; C – Cascade; N – None

Имя сущности	Удаление (D)	Вставка (I)	Обновление (U)
Родительская:			
Дочерняя:			

CONTENTS

Guriev M.A. Review of Global Software Industry 2
The article contains an analysis of the current state of the global software industry (GSI). General trends in industry's evolution are determined and analyzed. They include the movement of jobs toward local expertise on relevant professions, the transfer of critical IT-functionality in the outsourcing in specialized service companies, a growing interest in free software and the increase in acquisitions by leading companies the most successful competitors from the second echelon. Significant movements toward increasing quality requirements of both – software and programmers, as well as process engineering and technologies, lead us to conclusion about the possibilities of drastic changes in software engineering in nearest future.

Keywords: global software industry (GSI), tendencies of GSI development, rating of SW-500, rating of top100 changes in GSI

Orlik S.W. Software Engineering and Software Projects Lifecycle Models in the context of SWEBOK. Part I 6

History and structure of SWEBOK are discussed as a general accepted view on software engineering. This work is based on the publically available translation of SWEBOK to the Russian language within notes and comments prepared by author.

Keywords: SWEBOK, software engineering

Kazmin O.O. Source Code Translation in Dynamic Parallelization Systems Based on T-Approach 10

The paper presents the results of research aimed at creating an effective preprocessor within the scope of development of automatic parallelization system.

The paper touches on existing solutions for static code analysis and describes the modification of one of these tools. Also discussed are code translation methods applicable to parallelization systems based on T-approach.

Keywords: automatic parallelization, source code translation, static code analysis, T-approach, NewTS

Shelekhov V.I. Verification and Synthesis of Effective Programs for Standard Functions under Predicate Programming Technology 14

The technology of development of the effective standard functions isqrt, floor, and ilog2 is described. Correctness of the programs is achieved by application of deductive verification and program synthesis methods. The proof of

program correctness conditions is supported by the PVS Verification System.

Keywords: predicate programming, deductive verification, program synthesis, total correctness of program

Kryuchkova E.N., Staroletov S. M. Dynamic Testing of Networking Systems by Using an Automation Model. 22

Model-based testing is presented in the article. Formal model of networking systems is offered and this model is realized as a Eclipse Plugins. The purpose of proposed Automatic Testing System is development process tracing from model description and programming up to software testing.

Keywords: performance model, automaton, formal analyze, networking systems, model-based testing

Yablonsky S.A. Cloud Computing Ecosystem Introduction 27

This paper describes the importance of knowledge management of the cloud computing ecosystem. The methodology, the classification and organization of domain concepts, validation issues, analysis of some development tools, and the first pilot version of cloud computing ecosystem taxonomy is discussed. The cloud computing ecosystem taxonomy adds to current understanding of Cloud Services Innovation Semantic Workflow and Service Model Foundations; organization of Cloud Services domain knowledge in taxonomy form; identification and organization the concepts and instances (or examples) of concepts in a manner that achieves the taxonomy's purpose and the types of clouds and the multilayer models of cloud services.

Keywords: Cloud computing, Cloud Services, SaaS, PaaS, IaaS

Kostyuk V.V. History with Programming. 39

In this article are regarded questions of program code development, historical aspects of programming methods from sixties years of last century. This is selected stages from first steps of code writing in the full sense of the word in double presentation, up to development of program language systems on level of code generation on model basis.

Keywords: programming, code generation, modeling, program methods, development of programming.

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 11.02.11 г. Подписано в печать 14.04.11 г. Формат 60×88 1/8. Бумага офсетная. Печать офсетная.
Усл. печ. л. 5,88. Уч.-изд. л. 6,87. Цена свободная.

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