Подготовлено для II Международной научно-практической конференции «Объектные системы — 2010» (Зимняя сессия), http://objectsystems.ru

УДК 004.4

СОВРЕМЕННОЕ СИСТЕМНОЕ ОБЕСПЕЧЕНИЕ ДЛЯ КОМПОНЕНТНОГО ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

***Ермаков Илья Евгеньевич*,**
преподаватель, Технологический институт ОрёлГТУ,
технический директор, НПО «Тесла»
Россия, г. Орёл
ermakov@metasystems.ru

Создание современных программных систем надлежащего качества невозможно без использования соответствующих системных средств, как инструментального характера, применяемых на этапе разработки, так и платформенного характера, обеспечивающих этап выполнения. Платформы выполнения предоставляют поддержку базовых абстракций, с использованием которых реализована программная система, и всех механизмов, необходимых для её работы. В статье представлен обзор современной проблематики системного ПО и применяемых в этой сфере инженерных подходов.

1. Универсальные языки высокого уровня

Появление новых задач в ИТ традиционно стимулировало внедрение новых языков программирования (концепции которых обычно были разработаны задолго до своего широкого распространения).

Для десятилетия 90-х характерен рост интереса к языкам повышенного уровня — технологиям 4GL, генерационным CASE-системам, функциональному и декларативному программированию. Как обычно бывает, завышенные начальные ожидания сменились этапом реальных применений, когда эти инструменты распространились в тех нишах, где они уместны, и потерпели фиаско там, где их пытались привить безосновательно.

Можно уверенно говорить, что существует оптимальный языковый уровень для широкого класса задач. На этом уровне находится категория универсальных эффективно компилируемых императивных языков программирования. Из промышленных языков это преимущественно два семейства: Паскаль-семейство (Oberon,/-2/-07, Component Pascal, Ada/95/2005, Modula-2/3 и т.д.) и C-семейство — (C, C++, D и т.д.), на верхней границе этого категории — синтетическое семейство Java-C#, частично производное от Оберона.

Наиболее исследованные, ставшие рутинными, задачи постепенно охватываются высокоуровневыми инструментами, т.е. переходят из сферы полноценного программирования в сферу параметризации специализированных генерационных инструментов, каркасов и т.п. Однако неизменным остаётся то, что наиболее актуальные интересные задачи требуют полноценного программирования ровно в том же режиме, как и «бывшие» интересные задачи, а любое полноценное программирование при росте размеров и сроков жизни системы требует полноценного, универсального, качественно спроектированного ЯВУ («камень в огород» сценарных и макроязыков). Точно так же при масштабировании системы быстро приобретает остроту проблема эффективности, что требует применения компилируемых языков из названного выше класса. Разбор аргументов в пользу класса универсальных компилируемых языков и требований, исходящих от задач системного программирования, выполнен, в частности, Б. Страуструпом в книге «Дизайн и эволюция языка C++» [1].

Направление с популярным названием «Cloud Computing», которое инициировано такими лидерами отрасли, как Google и Microsoft, вывело на первый план категорию задач сложного серверного ПО. Оказывается, что распространённые языковые инструменты недостаточно хороши для этой сферы: C++ небезопасен и крайне сложен, Java и C# не так эффективны для системного программирования, как хотелось бы, сценарные языки не

рассматриваются всерьёз для ответственных высоконагруженных систем. Безопасность на уровне современных компонентных языков, эффективность на уровне C/C++, простота и компактность, характерная для классических языков, — вот требования, которые постепенно осознаются лидерами отрасли. Исходя из этих требований, в Google разработан Go — новый язык системного серверного программирования (соавторы которого — создатели ОС Unix Роб Пайк и Кен Томпсон) [2]. Язык создан по модели Oberon-2 и некоторых других языков [3]. Синтаксические решения и ряд других деталей Go нельзя назвать удачными, однако большой интерес представляют средства безопасного системного программирования. Google уже представил новый язык в качестве основного, которому компания отводит будущее серверного программирования (вместо Java и C#). Аналогичная разработка имеется у Mozilla Foundation — язык Rust [4], нацеленный в ту же нишу, что и Go (однако Rust по статусу — всего лишь личная разработка одного из членов Mozilla и потому имеет туманные перспективы).

Оптимальным балансом качеств для ниши системного серверного программирования обладает Оберон-семейство и Компонентный Паскаль, в частности [5]. По качеству инженерного дизайна эти языки принципиально опережают упомянутые выше. Как показывает опыт автора статьи, на Компонентном Паскале возможна продуктивная разработка серверного ПО высокой надёжности и эффективности. Это выполняется на основе дополнительных библиотечных решений, которые предоставляют программисту нужную инфраструктуру, в первую очередь — возможность написания системного кода без нарушения безопасности и герметичности языка (без импорта псевдомодуля SYSTEM). Компонентный Паскаль, оснащённый нужными библиотеками, представляется хорошим ответом на инициативу Google.

Суммируя тенденции в области языков системно-серверного программирования, можно сказать, что рост сложности массовых задач вынуждает индустрию «открывать Америку» и признавать давно известные принципы построения качественных языков программирования [6]. В связи с этим можно ожидать дальнейшего укрепления позиций новых языков семейства Н. Вирта (в частности, Компонентного Паскаля) и их подобий типа Go. Кроме того, может расти интерес к языку Ada, в силу того, что в сфере Ada-систем накоплена мощная инфраструктура создания ответственного распределённого ПО.

2. Компонентно-ориентированное программирование и промежуточное ПО

Как нами было указано в работе [7], современный этап ООП — это компонентно-ориентированное программирование (КОП). КОП подразумевает разработку динамически расширяемых каркасов, на основе которых строятся эволюционирующие системы. Эволюция систем организуется посредством разработки компонентов и подключения их к точкам расширения через объектно-ориентированные интерфейсы. Ключевыми свойствами КОП является возможность связывать компоненты динамически во время выполнения и обеспечение полной безопасности программирования, т.е. невозможности для отказавшего компонента нарушить работу других, которые с ним не связаны функционально.

КОП требует адекватной языковой поддержки. В первую очередь, это касается поддержки модульности и системы типов, которая обеспечивает одновременно расширяемость и безопасность (в частности, решение проблемы хрупкого базового класса, ограничения наследования реализации в пользу композиции и т.п.), что воплощено в языке Компонентный Паскаль главным идеологом КОП К. Шиперски [8, 9] . Принцип «модуль — это всего лишь объект-синглтон», пропагандируемый некоторыми специалистами по ООП (например, Б. Мейером), оказывается непрактичен в КОП. Такое смешение понятий препятствует созданию удачной архитектуры и надёжно расширяемых систем. Признание этого факта проявилось во вводе в Java 7 понятия модуля, в той форме и роли, в какой это принято в языках Modula и Oberon-семейств. Язык Go изначально спроектирован с системой модульности и импорта, в точности идентичной Oberon-2.

Современная система модульности и ООП требует адекватной динамической поддержки. Системы времени выполнения берут на себя сборку мусора, динамическую

загрузку и связывание модулей, абстракцию от нижележащей платформы. Наконец, что сегодня приобретает огромное значение, они предоставляют исполняемым компонентам независимые от платформы средства коммуникации в распределённых сетевых средах. Любую систему времени выполнения можно условно разделить на ядро поддержки языковых средств и системные библиотеки. Первая часть отличается от второй тем, что её функции отражены в синтаксических конструкциях языка. Значение и функционал систем времени выполнения возросли настолько, что уже достаточно давно выделяют особый класс системного ПО — промежуточное, англ. «middleware».

3. Абстракция от платформы и промежуточные представления

Начиная с эпохи Алгола, необходимым требованием к языкам высокого уровня является возможность написания компонентов, полностью переносимых между различными платформами.

Линия переносимости может пролегать выше или ниже, в зависимости от того, какая форма программы является пригодной для межплатформенного распространения: исходный текст, или семантическое дерево, или промежуточный код (для языковоориентированной виртуальной машины или для низкоуровневой кодогенерации).

Степень переносимости исходного кода бывает различной: от 100%-гарантии успешной компиляции программы на любом компиляторе под любую платформу (как обстоит дело с языком Ada) до «условной переносимости» с возможностью «доработать напильником», и то если автор программы придерживался специальной дисциплины (как это принято в мире C/C++ и экосистеме GNU/Linux).

Часто переносимым является не только исходный код, но и некоторый продукт его компиляции (который получается на выходе переднего плана — frontend — компилятора).

Видимо, исторически первым распространился промежуточный код, ориентированный на низкоуровневую кодогенерацию. Такой код поступает на вход заднего плана — backend — компилятора, на выходе которого получается исполняемый машинный код для целевой платформы. Ранним примером является известная сибирская система БЕТА с промежуточным языком ВЯЗ, созданные в Вычислительном центре СО РАН под руководством А.П. Ершова [10]. Современный пример — широко применяемый каркас компиляторов GCC. Однако промежуточное представление этого уровня практически не используется в качестве формы распространения компонентов, а играет роль только для облегчения создания многоязыковых и многоплатформенных систем трансляции. На наш взгляд, нет причин не использовать более широко такие промежуточные формы, тем более, что у них есть определённое преимущество перед рассматриваемыми далее кодами виртуальных машин — они дают эффективную и прозрачную трансляцию в машинный код.

Широкое распространение концепции так называемого байт-кода для виртуальной машины началось с известной разработки Н. Вирта — стекового P-кода, применённого для быстрого распространения языка Паскаль в 1970-х гг. P-код был взят за образец компанией Sun для разработки байт-кода и виртуальной машины языка Java. Интерпретирующие реализации виртуальной машины сегодня уступили место технологиям генерации в машинный код, которая выполняется в момент загрузки компонента в систему выполнения (Just-In-Time, JIT-кодогенерация). Нужно заметить, что для систем динамической кодогенерации большую роль сыграл опыт реализаций интерпретируемого языка Smalltalk, которые выполняют такую генерацию для отдельных фрагментов программ прямо во время выполнения (компания Sun располагала опытом реализации Smalltalk, который и был использован при развитии Java Platform). В настоящее время JIT-системы для Java Platform расширены функциями профилирования выполняющихся программ и последующей оптимизации на основе полученных профилей. Такая оптимизация может заключаться в прямых подстановках и сокращениях кода и иногда приводить к достижению Java-приложениями большего быстродействия, чем C/C++-приложениями [11].

Ветка Паскаль-языков развивалась Н. Виртом по пути упрощения и минимализации. Вполне разумно использовать исходный текст программы в качестве основной формы

распространения компонентов, если этот исходный текст написан на языке с 16-страничным описанием, допускающем мгновенную однопроходную компиляцию, которая выполняется раздельно для каждого модуля (ср. с объёмом исходного текста, поступающего на вход компилятору C/C++ после подстановок всех инструкций #include препроцессора).

В 1994 г., одновременно с выходом в свет языка Java, М. Франц (аспирант Н. Вирта) предложил свою схему распространения исполняемых компонентов в сети Internet [12]. Исполняющая система получила позднее название Juice. Две ключевые проблемы — безопасность и кроссплатформенность — были решены за счёт введения промежуточного семантического представления и динамической кодогенерации машинного кода в момент загрузки компонента. Фактически, представление Франца — это упакованное особым образом семантическое дерево Оберон-программы, что можно рассматривать как вариант, близкий к идее распространения исходного текста программы. В настоящее время опыт динамической кодогенерации для Оберон-систем используется М. Францем в разработках для Java и JavaScript. Недавно выпущенный в Mozilla Foundation оптимизирующий интерпретатор JavaScript TraceMonkey разработан коллективом Франца и базируется на динамической кодогенерации.

Наконец, вопрос платформы и промежуточного представления для современных объектных систем тесно связан с перспективами оборудования, имеющего высокоуровневую систему команд. Фактически, опережающий аналог современных компонентных безопасных платформ был воплощён ещё в 1970-х гг. в известных многопроцессорных вычислительных комплексах «Эльбрус» и их высокоуровневом машинном языке Эль-76 [13]. Создание оборудования исходя из требований системного языка высокого уровня выполнялось Н. Виртом в проектах ПК Lililith (Модула-2) и ПК Ceres (Оберон), в Новосибирском Академгородке создавались ориентированные на Модулу-2 и Оберон ПК «Кронос» и параллельные вычислительные комплексы 5-го поколения МАРС (модульные асинхронные развиваемые системы) [14]; коллективом А.Н. Терехова в конце 80-х для систем правительственной связи была разработана спец-ЭВМ «Самсон» [15], ориентированная на класс статических компилируемых ЯВУ. Идея оборудования, ориентированного на ЯВУ, получила сегодня «второе дыхание» в виде различных экспериментальных процессоров языка Java (например, процессорного ядра picoJava). Таким экспериментам сегодня способствует доступность микросхем с программируемой логикой (ПЛИС/FPGA).

4. Проблемы управления памятью

В ИТ неумолимо действует принцип — как правило, чем сложнее задача, тем более сложные динамические структуры данных (графы связей) она использует и тем большую нагрузку она создаёт на динамическую память. Собственно, развитие методов управления динамической памятью не в последнюю очередь форсировались языками, подобными LISP и Smalltalk. Переход к объектной структуре программ, с наличием множества объектов, ссылающихся друг на друга, требует управления временем жизни этих объектов. Синтетические объектно-ориентированные языки — Object (Borland) Pascal, C++, появившиеся в 80-х гг. как комбинация структурных языков с Simula-подобными средствами, не предоставили программистам никаких средств управления жизнью объектов, кроме традиционных операций NEW и DISPOSE. Разработчики и пользователи в полной мере ощутили и продолжают по сей день ощущать лавину ошибок, связанных с разрушением памяти приложений из-за неверного освобождения памяти динамических объектов.

При разработке языка и ОС Oberon, в которых был гладко интегрирован экстракт ООП (в виде расширяемых записей и их полиморфизма) со структурно-модульными средствами, была осознана необходимость автоматического управления памятью для расширяемых компонентных систем; Oberon стал первым компилируемым языком со сборкой мусора. Герметичность указателей и запрет явного освобождения памяти гарантируют её целостность (инвариант о том, что любой указатель указывает на существующий объект). Это резко повышает надёжность системы, собранной из многочисленных компонентов. Кроме того, принципиально меняется структура связей в объектной системе. Разработчики на языках без

сборки мусора не могут себе позволить применять сложные структуры с большим числом перекрёстных и обратных связей, тем более — между компонентами от различных поставщиков, поскольку будет невозможно обеспечить корректное освобождение объектов. Сборка мусора влияет даже на алгоритмические решения, поскольку позволяет свободно работать со сложными графовыми структурами данных.

Именно названные свойства и возможности являются основными следствиями автоматического управления памятью. К сожалению, большинство разработчиков имеет слабое представление о них, а вместо этого рассматривает сборку мусора как дополнительное удобство, позволяющее не думать о работе с памятью вообще («если есть сборщик — да здравствует мусор»). Этому же способствует популярная во многих языках парадигма «любые данные есть объекты», «любая переменная есть ссылка». Кроме концептуальных проблем (подобных обнаруженному на новой Java эффекту с типом Integer: две переменных этого типа, хранящие одинаковые значения, большие 127, не равны, поскольку сравнение выполняется для указателей; но в случае значений, меньших 127, оказываются равны, поскольку Java-реализация имеет предварительно созданные объекты для чисел -128..127 и совместно использует их для всех констант в программе), это ведёт к катастрофическому возрастанию нагрузки на сборщик мусора. Особенно это характерно для алгоритмов, работающих с потоковыми данными (строками, двоичными последовательностями). Особой разновидностью потоковых данных сегодня становится XML, который требует для своей обработки древовидной структуры (например, реализации интерфейса DOM). Бездумная реализация работы с XML в высоконагруженном приложении способна привести к потоку мусора. Как отмечают специалисты, работающие в области телекоммуникационных систем, для систем, написанных в таком стиле (на C# или Java), существует определённый порог нагрузки, за которым происходит отказ в обслуживании по причине того, что вся вычислительная мощность расходуется на сбор мусора.

Дефекты языковых решений и дефекты разработки компании-поставщики инструментария стараются компенсировать совершенствованием алгоритмов сбора мусора, которых изобретено и внедрено большое количество [16]. За последние два года Java-машины даже научились автоматически распознавать объекты локального назначения и размещать их на стеке. Однако любые оптимизации оказываются ориентированы на определённый режим эксплуатации и проигрывают в других режимах. Например, при большом количестве мелких короткоживущих объектов (характерных для режима «всё есть объект») выгоден сборщик с поколениями, в то же время существуют режимы, в которых он сильно проигрывает примитивному классическому алгоритму «маркировка и сборка». В любом случае, современная сборка мусора — очень непредсказуемая и зыбкая основа для программиста, к тому же имеющая сложное внутреннее устройство, что ведёт к низкой надёжности (фактически, качественные реализации оказываются по силам только крупным коммерческим компаний; у свободных реализаций языков со сложными исполняющими системами, подобных Python, Haskell и др., ахиллесовой пятой являются ошибки в алгоритмах управления памятью, приводящие в определённых случаях к её утечкам). Характеризуя направление развития современных систем сборки мусора, особенно хочется вспомнить Принцип Калашникова от «Информатики-21»: «Избыточная сложность — всегда уязвимость».

Особую остроту задаче сборки мусора придают требования, связанные с многопоточностью и поддержкой реального времени. Недетерминированность времени сборки, недетерминированность момента уничтожения объекта — «дамоклов меч» для разработчиков системного ПО этих режимов. Тем не менее, сборка мусора уже давно находит своё место даже в системах жёсткого РВ, не говоря про распространение в мягком РВ. Управление на основе подсчёта ссылок подкупает эффективностью и предсказуемостью освобождения, однако в базовом виде неприменимо из-за невозможности освобождать структуры, в которых присутствуют циклические связи. Однако делаются попытки создания смешанных стратегий сбора мусора, задействующих достоинства подсчёта ссылок [17].

Тем не менее, эйфория от сборки мусора сменяется признанием факта, что это — не панацея. Что же возможно сделать? Во-первых, применять те языки, в которых в полной мере доступны структуры данных, размещаемые на стеке и в другой непрерывной памяти. Простой пример: в Обероне/Компонентном Паскале стандартный приём — шина сообщений, который заключается в передаче в процедуру или метод полиморфного параметра (сообщения), имеющего вид VAR param: ANYREC. Через такой параметр может быть передана любая запись, затем динамически распознан её тип. Поскольку записи-сообщения размещены на стеке, то стоимость такого полиморфного вызова равна стоимости вызова обычной процедуры и вообще не требует нагрузки на динамическую память. Во-вторых, в случае разработки высоконагруженных систем (например, серверных и телекоммуникационных приложений) нужно организовывать ручное управление жизненным циклом объектов — возврат в пулы для повторного использования и т.п. Не является ли нонсенсом ручное управление памятью поверх встроенного в язык автоматического? Ответ — нет, поскольку, как мы рассмотрели выше, главная функция автоматического управления — сделать невозможным разрушение памяти компонентов приложения, и эта функция выполняется. В случае ошибок с повторным использованием объектов возможна порча памяти в более «культурной» форме и сложные недетерминированные ошибки, однако это будут ошибки в логическом поведении программы, а не ошибки её системного отказа. С точки зрения лёгкости разработки и диагностики это — далеко не одно и то же. В целом же, в вопросе баланса между ручным и автоматическим управлением памятью наблюдаем обычную спираль научно-технического развития с «отрицанием отрицания», на качественно новом уровне.

Можно ли вернуться к явному освобождению памяти, при этом обеспечив безопасность программирования? Оказывается, это возможно. Основное требование — при доступе по указателю на несуществующий объект должен быть гарантирован отказ, вместо непредсказуемого поведения. Иначе говоря, адреса распределяемых динамических объектов не должны повторяться. Именно такой подход применяется на некоторых «больших машинах», имеющих широкое адресное слово (обычно 128 байт и более). В системах "Эльбрус" [13] и IBM/AS400 [18] (для которой характерно единое адресное пространство оперативной памяти и дисковых накопителей) адреса, выделяемые в виртуальном пространстве приложения, непрерывно возрастают, в случае же переполнения ёмкости адреса инициируется полный сбор мусора с уплотнением объектов и перенастройкой ссылок. Если учитывать переход массовых систем на 64-разрядные процессорные архитектуры, такой подход становится привлекательным.

Наконец, отдельный вопрос в работе с памятью — возможность обработки двоичных потоков данных, преобразований между ними и типами данных языка, без нарушения безопасности программирования. Эта проблема, остававшаяся на периферии внимания в Java и C#-системах, прибретает остроту в связи с актуальными задачами серверного ПО. В языках Google Go и Mozilla Rust введены в той или иной форме понятия двоичных областей и способов манипуляции с ними без возможности выйти за их границы. В Обероне/Компонентном Паскале эти задачи решаются библиотечными средствами. Кроме того, на базе безопасных двоичных областей можно решить и некоторые проблемы ручного управления памятью — фактически, можно сделать «виртуальную память в песочнице».

5. Адресные пространства и защита приложений

Традиционная функция операционных систем, поддержанная оборудованием (MMU — memory-management unit) — обеспечение виртуальной памяти и изолированных адресных пространств для приложений. Фактически, для небезопасных языков программирования граница адресного пространства — основной рубеж надёжности.

Многое изменилось с распространением безопасных языков. В операционных системах Оберон-семейства было показано, как безопасный язык делает возможным работу ОС и приложений в едином адресном пространстве. Для объектных систем здесь принципиально то, что решается проблема взаимодействия компонентов — ведь все объекты существуют в

едином адресном пространстве. Кроме того, резко повышается производительность параллельного выполнения (переключение между контекстами процессов — очень дорогостоящая операция) Современная ОС A2 (бывшая BlueBottle) поддерживает десятки тысяч параллельно исполняемых объектов языка Active Oberon и очень быстрый запуск новых [5].

Однако, как всегда, имеется и оборотная сторона медали: размываются границы между зонами ответственности отдельных приложений, затрудняется выгрузка и перезапуск отдельных компонентов (из-за плотной интеграции в «паутину» указателей). Наконец, практически невозможно обеспечить контроль прав компонентов и защиту от злонамеренных действий, либо это ведёт к такому усложнению языка и объектной системы, которое выходит за всякие рамки инженерной целесообразности. Таким образом, наличие раздельных объектных пространств остаётся востребованным для современных систем, за исключением некоторых применений (например, встраиваемых, мультимедийных — как раз тех, в нишу которых направлена ОС A2).

Если несколько изолированных объектных пространств существуют на базе единого адресного, то такой подход называют программной изоляцией памяти, в противовес аппаратной. Её преимущества в том, что между объектными пространствами возможна эффективная передача и разделение статических структур данных, в частности — реализация потоков сообщений.

Подразделение Microsoft Research в 2008 г. представило проект OS Singularity [19] , идея которого — разработка ОС на базе безопасного языка Sing#, кодогенерация из промежуточного представления, развитые механизмы статического контроля и верификации, различные режимы изоляции компонентов (единое пространство, SIP — Software-Isolated Proccesses, HIP — Hardware-Isolated Proccesses). Система опубликована в открытых исходных текстах, в настоящее время проект закрыт. Его опыт в полной мере задействован в разработке новых ОС семейства Windows [20]. К сожалению, приходится констатировать, что сообщество сводобного ПО значительно отстаёт от Microsoft в технологиях системного программирования — технические проблемы, которые успешно решаются этой корпорацией, в Linux-сообществе пока даже не осознаны, культура разработки и технологического оснащения C/C++ программистов в принципе этому не способствует.

7. Выводы

В статье рассмотрены основные функции системного и промежуточного ПО, необходимые для существования современных объектно-ориентированных компонентных систем. Как и в целом в экономике, для успешного развития отрасли важен хорошо известный принцип «опережающего развития производства средств производства». В настоящее время эти средства производства объектных систем — безопасные производительные языки программирования, поддерживающие компонентно-ориентированный стиль, исполняющие системы (runtimes) для этих языков, промежуточное программное обеспечение (middleware).

Следует также понимать, что ни одно отдельное техническое решение, предоставляемое системным ПО, не является безусловным — и разработчик сложных, ответственных, высоконагруженных систем обязательно столкнётся с необходимостью решать задачу инженерного баланса.

Литература

1. Страуструп Б. Дизайн и эволюция языка C++. — М.: Питер, 2006.
2. Официальный сайт языка Go. — Web: http://golang.org
3. What are Go's ancestors? — Web: http://golang.org/doc/go_faq.html#ancestors
4. Официальный сайт языка Rust. — Web: http://github.com/graydon/rust/
5. Проект OberonCore. — Web: http://oberoncore.ru
6. Янг С. Алгоритмические языки реального времени. — Пер. с англ. — М.: Мир, 1986.
7. Ермаков, И.Е. Объектно-ориентированное программирование: прояснение принципов? // Объектные системы - 2010: Материалы I Международной научно-практической конференции. —

Россия, Ростов-на-Дону, 10-12 мая 2010 г. / под общ. ред. П.П. Олейника. — Ростов-на-Дону, 2010. С. 130-135.

8. Szyperski C. Component Software. Beyond Object-Oriented Programming. — Addison Wesley Longman, 1998.

9. Szyperski C. Insight EthOS — On Object Orientation In Operating Systems. — ETH Diss. No 9884. — Zurich, Switzerland, 1992. — ISBN 3-7281-1948-2. Web: http://research.microsoft.com/en-us/um/people/cszypers/books/insight-ethos.htm

10. Документация проекта БЕТА. // Архив академика А.П. Ершова. — Web: http://www.ershov.ras.ru/archive/eaindex.asp?lang=1&gid=210

11. Завалишин Д. Видеолекция о разработке высоконагруженных Java-приложений. — Web: http://video.yandex.ru/users/ysirotkin/view/2/

12. Franz M. Code-Generation On-the-Fly: A Key for Portable Software. — Zurich, Dissertation No.10497, 1994.

13. Литература по системам «Эльбрус». — Web: http://oberoncore.ru/library/start

14. Материалы по проектам «Кронос» и МАРС. — Web: http://kronos.ru/

15. Терехов, А.Н. Технология программирования встроенных систем реального времени. — Докторская диссертация, 1991. — Web: http://ant.tepkom.ru/publications/doc/Terekhov_Doct_thesis.pdf

16. Jones R. Lins R. Garbage Collection: Algorithms for Authomatic Dynamic Memory Management. — John Willey & Sons, 1997.

17. Levanony Y., Petrank E. An On-the-Fly Reference-Counting Garbage Collector for Java. // ACM Transactions on Programming Languages and Systems (TOPLAS) archive, Volume 28, Issue 1, 2006.

18. Солтис Ф. Основы AS/400. — М.: Русская редакция, 1998.

19. Официальная страница open-source проекта Microsoft Singularity. — Web: http://singularity.codeplex.com/

20. Беседа с менеджером по стратегии платформ Microsoft Russia В. Шершульским об итогах проекта Singularity. / Проект OberonCore. — http://forum.oberoncore.ru/viewtopic.php?p=48426